Handbook of Statistics, Vol. 24
ISSN: 0169-7161
© 2005 Elsevier B.V. All rights reserved.
DOI 10.1016/S0169-7161(04)24011-1

11

Classification and Regression Trees, Bagging, and Boosting

Clifton D. Sutton

1. Introduction

Tree-structured classification and regression are alternative approaches to classification and regression that are not based on assumptions of normality and user-specified model statements, as are some older methods such as discriminant analysis and ordinary least squares (OLS) regression. Yet, unlike the case for some other nonparametric methods for classification and regression, such as kernel-based methods and nearest neighbors methods, the resulting tree-structured predictors can be relatively simple functions of the input variables which are easy to use.

Bagging and boosting are general techniques for improving prediction rules. Both are examples of what Breiman (1998) refers to as *perturb and combine* (P&C) methods, for which a classification or regression method is applied to various perturbations of the original data set, and the results are combined to obtain a single classifier or regression model. Bagging and boosting can be applied to tree-based methods to increase the accuracy of the resulting predictions, although it should be emphasized that they can be used with methods other than tree-based methods, such as neural networks.

This chapter will cover tree-based classification and regression, as well as bagging and boosting. This introductory section provides some general information, and briefly describes the origin and development of each of the methods. Subsequent sections describe how the methods work, and provide further details.

1.1. *Classification and regression trees*

Tree-structured classification and regression are nonparametric computationally intensive methods that have greatly increased in popularity during the past dozen years. They can be applied to data sets having both a large number of cases and a large number of variables, and they are extremely resistant to outliers (see Steinberg and Colla, 1995, p. 24).

Classification and regression trees can be good choices for analysts who want fairly accurate results quickly, but may not have the time and skill required to obtain them using traditional methods. If more conventional methods are called for, trees can still be helpful if there are a lot of variables, as they can be used to identify important variables and interactions. Classification and regression trees have become widely used among members of the data mining community, but they can also be used for relatively simple tasks, such as the imputation of missing values (see Harrell, 2001).

Regression trees originated in the 1960s with the development of AID (Automatic Interaction Detection) by Morgan and Sonquist (1963). Then in the 1970s, Morgan and Messenger (1973) created THAID (Theta AID) to produce classification trees. AID and THAID were developed at the Institute for Social Research at the University of Michigan.

In the 1980s, statisticians Breiman et al. (1984) developed CART (Classification And Regression Trees), which is a sophisticated program for fitting trees to data. Since the original version, CART has been improved and given new features, and it is now produced, sold, and documented by Salford Systems. Statisticians have also developed other tree-based methods; for example, the QUEST (Quick Unbiased Efficient Statistical Tree) method of Loh and Shih (1997).

Classification and regression trees can now be produced using many different software packages, some of which are relatively expensive and are marketed as being commercial data mining tools. Some software, such as S-Plus, use algorithms that are very similar to those underlying the CART program. In addition to creating trees using `tree` (see Clark and Pregibon, 1992), users of S-Plus can also use `rpart` (see Therneau and Atkinson, 1997), which can be used for Poisson regression and survival analysis in addition to being used for the creation of ordinary classification and regression trees. Martinez and Martinez (2002) provide Matlab code for creating trees, which are similar to those that can be created by CART, although their routines do not handle nominal predictors having three or more categories or splits involving more than one variable. The MATLAB Statistics Toolbox also has functions for creating trees. Other software, including some ensembles aimed at data miners, create trees using variants of CHAID (CHi-squared Automatic Interaction Detector), which is a descendant of THAID developed by Kass (1980). SPSS sells products that allow users to create trees using more than one major method.

The machine learning community has produced a large number of programs to create decision trees for classification and, to a lesser extent, regression. Very notable among these is Quinlan's extremely popular C4.5 (see Quinlan, 1993), which is a descendant of his earlier program, ID3 (see Quinlan, 1979). A newer product of Quinlan is his system See5/C5.0. Quinlan (1986) provides further information about the development of tree-structured classifiers by the machine learning community.

Because there are so many methods available for tree-structured classification and regression, this chapter will focus on one of them, CART, and only briefly indicate how some of the others differ from CART. For a fuller comparison of tree-structured classifiers, the reader is referred to Ripley (1996, Chapter 7). Gentle (2002) gives a shorter overview of classification and regression trees, and includes some more recent references. It can also be noted that the internet is a source for a large number of descriptions of various methods for tree-structured classification and regression.

1.2. Bagging and boosting

Bootstrap aggregation, or bagging, is a technique proposed by Breiman (1996a) that can be used with many classification methods and regression methods to reduce the variance associated with prediction, and thereby improve the prediction process. It is a relatively simple idea: many bootstrap samples are drawn from the available data, some prediction method is applied to each bootstrap sample, and then the results are combined, by averaging for regression and simple voting for classification, to obtain the overall prediction, with the variance being reduced due to the averaging.

Boosting, like bagging, is a committee-based approach that can be used to improve the accuracy of classification or regression methods. Unlike bagging, which uses a simple averaging of results to obtain an overall prediction, boosting uses a weighted average of results obtained from applying a prediction method to various samples. Also, with boosting, the samples used at each step are not all drawn in the same way from the same population, but rather the incorrectly predicted cases from a given step are given increased weight during the next step. Thus boosting is an iterative procedure, incorporating weights, as opposed to being based on a simple averaging of predictions, as is the case with bagging. In addition, boosting is often applied to *weak learners* (e.g., a simple classifier such as a two node decision tree), whereas this is not the case with bagging.

Schapire (1990) developed the predecessor to later boosting algorithms developed by him and others. His original method pertained to two-class classifiers, and combined the results of three classifiers, created from different learning samples, by simple majority voting. Freund (1995) extended Schapire's original method by combining the results of a larger number of weak learners. Then Freund and Schapire (1996) developed the *AdaBoost algorithm*, which quickly became very popular. Breiman (1998) generalized the overall strategy of boosting, and considered Freund and Schapire's algorithm as a special case of the class of *arcing* algorithms, with the term arcing being suggested by *adaptive resampling and combining*. But in the interest of brevity, and due to the popularity of Freund and Schapire's algorithm, this chapter will focus on AdaBoost and only briefly refer to related approaches. (See the bibliographic notes at the end of Chapter 10 of Hastie et al. (2001) for additional information about the development of boosting, and for a simple description of Schapire's original method.)

Some (see, for example, Hastie et al., 2001, p. 299) have indicated that boosting is one of the most powerful machine/statistical learning ideas to have been introduced during the 1990s, and it has been suggested (see, for example, Breiman (1998) or Breiman's statement in Olshen (2001, p. 194)) that the application of boosting to classification trees results in classifiers which generally are competitive with any other classifier. A particularly nice thing about boosting and bagging is that they can be used very successfully with simple "off-the-shelf" classifiers (as opposed to needing to carefully tune and tweak the classifiers). This fact serves to somewhat offset the criticism that with both bagging and boosting the improved performance comes at the cost of increased computation time. It can also be noted that Breiman (1998) indicates that applying boosting to CART to create a classifier can actually be much quicker than fitting a neural net classifier. However, one potential drawback to perturb and combine methods is that the final prediction rule can be appreciably more complex than what can be obtained using a method that does not combine predictors. Of course, it is often the case that simplicity has to be sacrificed to obtain increased accuracy.

2. Using CART to create a classification tree

In the general classification problem, it is known that each case in a sample belongs to one of a finite number of possible classes, and given a set of measurements for a case, it is desired to correctly predict to which class the case belongs. A classifier is a rule that assigns a predicted class membership based on a set of related measurements, $x_1, x_2, \ldots, x_{K-1}$, and x_K. Taking the measurement space $\mathcal{X}$ to be the set of all possible values of $(x_1, \ldots, x_K)$, and letting $\mathcal{C} = \{c_1, c_2, \ldots, c_J\}$ be the set of possible classes, a classifier is just a function with domain $\mathcal{X}$ and range $\mathcal{C}$, and it corresponds to a partition of $\mathcal{X}$ into disjoint sets, $B_1, B_2, \ldots, B_J$, such that the predicted class is j if $\mathbf{x} \in B_j$, where $\mathbf{x} = (x_1, \ldots, x_K)$.

It is normally desirable to use past experience as a basis for making new predictions, and so classifiers are usually constructed from a *learning sample* consisting of cases for which the correct class membership is known in addition to the associated values of $(x_1, \ldots, x_K)$. Thus statistical classification is similar to regression, only the response variable is nominal. Various methods for classification differ in how they use the data (the learning sample) to partition $\mathcal{X}$ into the sets $B_1, B_2, \ldots, B_J$.

2.1. *Classification trees*

Tree-structured classifiers are constructed by making repetitive splits of $\mathcal{X}$ and the subsequently created subsets of $\mathcal{X}$, so that a hierarchical structure is formed. For example, $\mathcal{X}$ could first be divided into $\{\mathbf{x} \mid x_3 \leqslant 53.5\}$ and $\{\mathbf{x} \mid x_3 > 53.5\}$. Then the first of these sets could be further divided into $A_1 = \{\mathbf{x} \mid x_3 \leqslant 53.5, x_1 \leqslant 29.5\}$ and $A_2 = \{\mathbf{x} \mid x_3 \leqslant 53.5, x_1 > 29.5\}$, and the other set could be split into $A_3 = \{\mathbf{x} \mid x_3 > 53.5, x_1 \leqslant 74.5\}$ and $A_4 = \{\mathbf{x} \mid x_3 > 53.5, x_1 > 74.5\}$. If there are just two classes ($J = 2$), it could be that cases having unknown class for which $\mathbf{x}$ belongs to A_1 or A_3 should be classified as c_1 (predicted to be of the class c_1), and cases for which $\mathbf{x}$ belongs to A_2 or A_4 should be classified as c_2. (Making use of the notation established above, we would have $B_1 = A_1 \cup A_3$ and $B_2 = A_2 \cup A_4$.) Figure 1 shows the partitioning of $\mathcal{X}$ and Figure 2 shows the corresponding representation as a tree.

While it can be hard to draw the partitioning of $\mathcal{X}$ when more than two predictor variables are used, one can easily create a tree representation of the classifier, which is easy to use no matter how many variables are used and how many sets make up the partition. Although it is generally harder to understand how the predictor variables relate to the various classes when the tree-structured classifier is rather complicated, the classifier can easily be used, without needing a computer, to classify a new observation based on the input values, whereas this is not usually the case for nearest neighbors classifiers and kernel-based classifiers.

It should be noted that when $\mathcal{X}$ is divided into two subsets, these subsets do not both have to be subsequently divided using the same variable. For example, one subset

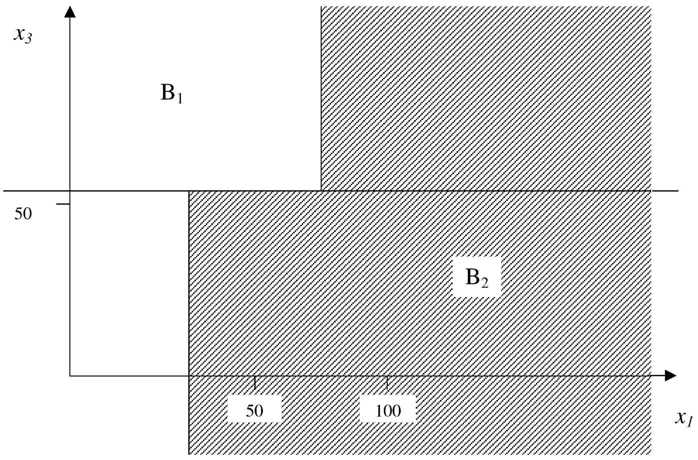

Fig. 1. A partition of $\mathcal{X}$ formed by orthogonal splits.

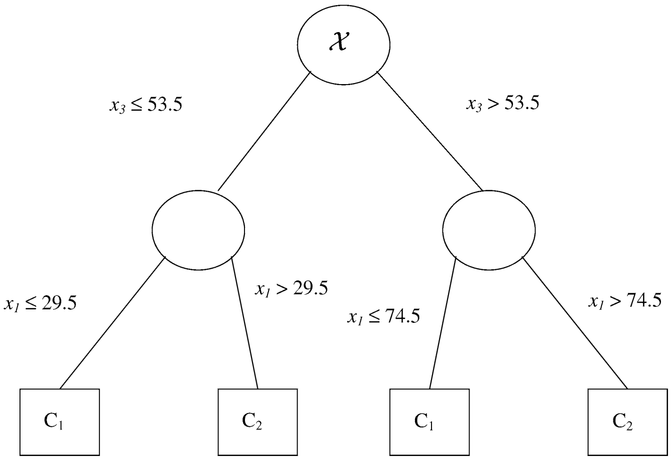

Fig. 2. Binary tree representation of the classifier corresponding to the partition shown in Figure 1.

could be split using x_1 and the other subset could be split using x_4. This allows us to model a nonhomogeneous response: the classification rules for different regions of $\mathcal{X}$ can use different variables, or be different functions of the same variables. Furthermore, a classification tree does not have to be symmetric in its pattern of nodes. That is, when $\mathcal{X}$ is split, it may be that one of the subsets produced is not split, while the other subset is split, with each of the new subsets produced being further divided.

2.2. Overview of how CART creates a tree

What needs to be determined is how to best split subsets of $\mathcal{X}$ (starting with $\mathcal{X}$ itself), to produce a tree-structured classifier. CART uses recursive binary partitioning to create a binary tree. (Some other methods for creating classification trees allow for more than two branches to descend from a node; that is, a subset of $\mathcal{X}$ can be split into more than two subsets in a single step. It should be noted that CART is not being restrictive in only allowing binary trees, since the partition of $\mathcal{X}$ created by any other type of tree structure can also be created using a binary tree.) So, with CART, the issues are:

(1) how to make each split (identifying which variable or variables should be used to create the split, and determining the precise rule for the split),
(2) how to determine when a node of the tree is a terminal node (corresponding to a subset of $\mathcal{X}$ which is not further split), and
(3) how to assign a predicted class to each terminal node.

The assignment of predicted classes to the terminal nodes is relatively simple, as is determining how to make the splits, whereas determining the right-sized tree is not so straightforward. In order to explain some of these details, it seems best to start with the easy parts first, and then proceed to the tougher issue; and in fact, one needs to first understand how the simpler parts are done since the right-sized tree is selected from a set of candidate tree-structured classifiers, and to obtain the set of candidates, the splitting and class assignment issues have to be handled.

2.3. *Determining the predicted class for a terminal node*

If the learning sample can be viewed as being a random sample from the same population or distribution from which future cases to be classified will come, and if all types of misclassifications are considered to be equally bad (for example, in the case of $J = 2$, classifying a c_1 case as a c_2 is given the same penalty as classifying a c_2 as a c_1), the class assignment rule for the terminal nodes is the simple *plurality rule*: the class assigned to a terminal node is the class having the largest number of members of the learning sample corresponding to the node. (If two or more classes are tied for having the largest number of cases in the learning sample with $\mathbf{x}$ belonging to the set corresponding to the node, the predicted class can be arbitrarily selected from among these classes.) If the learning sample can be viewed as being a random sample from the same population or distribution from which future cases to be classified will come, but all types of misclassifications are not considered to be equally bad (that is, classifying a c_1 case as a c_2 may be considered to be twice as bad, or ten times as bad, as classifying a c_2 as a c_1), the different misclassification penalties, which are referred to as *misclassification costs*, should be taken into account, and the assigned class for a terminal node should be the one which minimizes the total misclassification cost for all cases of the learning sample corresponding to the node. (Note that this rule reduces to the plurality rule if all types of misclassification have the same cost.) Likewise, if all types of misclassifications are to be penalized the same, but the learning sample has class membership proportions that are different from those that are expected for future cases to be classified with the classifier, weights should be used when assigning predicted classes to the nodes, so that the predictions will hopefully minimize the number of misclassifications for future cases. Finally, if all types of misclassifications are not to be penalized the same *and* the learning sample has class membership proportions that are different from those that are expected for cases to be classified with the classifier, two sets of weights are used to assign the predicted classes. One set of weights is used to account for different misclassification costs, and the other set of weights is used to adjust for the class proportions of future observations being different from the class proportions of the learning sample.

It should be noted that, in some cases, weights are also used in other aspects of creating a tree-structured classifier, and not just for the assignment of the predicted class. But in what follows, the simple case of the learning sample being viewed as a random sample from the same population or distribution from which future cases to be classified will come, and all types of misclassifications being considered equally bad, will be dealt with. In such a setting, weights are not needed. To learn what adjustments should be made for other cases, the interested reader can find some pertinent information in Breiman et al. (1984), Hastie et al. (2001), Ripley (1996).

2.4. Selection of splits to create a partition

In determining how to divide subsets of $\mathcal{X}$ to create two children nodes from a parent node, the general goal is to make the distributions of the class memberships of the cases of the learning sample corresponding to the two descendant nodes different, in such a way as to make, with respect to the response variable, the data corresponding to each of the children nodes purer than the data corresponding to the parent node. For example, in a four class setting, a good first split may nearly separate the c_1 and c_3 cases from the c_2 and c_4 cases. In such a case, uncertainty with regard to class membership of cases having $\mathbf{x}$ belonging to a specific subset of $\mathcal{X}$ is reduced (and further splits may serve to better divide the classes).

There are several different types of splits that can be considered at each step. For a predictor variable, x_k, which is numerical or ordinal (coded using successive integers), a subset of $\mathcal{X}$ can be divided with a plane orthogonal to the x_k axis, such that one of the newly created subsets has $x_k \leqslant s_k$, and the other has $x_k > s_k$. Letting

$$y_{k\,(1)} < y_{k\,(2)} < \cdots < y_{k\,(M)}$$

be the ordered distinct values of x_k observed in the portion of the learning sample belonging to the subset of $\mathcal{X}$ to be divided, the values

$$(y_{k\,(m)} + y_{k\,(m+1)})/2 \quad (m = 1, 2, \ldots, M - 1)$$

can be considered for the split value s_k. So even if there are many different continuous or ordinal predictors, there is only a finite (albeit perhaps rather large) number of possible splits of this form to consider. For a predictor variable, x_k, which is nominal, having class labels belonging to the finite set D_k, a subset of $\mathcal{X}$ can be divided such that one of the newly created subsets has $x_k \in S_k$, and the other has $x_k \notin S_k$, where S_k is a nonempty proper subset of D_k. If D_k contains d members, there are $2^{d-1} - 1$ splits of this form to be considered.

Splits involving more than one variable can also be considered. Two or more continuous or ordinal variables can be involved in a *linear combination split*, with which a subset of $\mathcal{X}$ is divided with a hyperplane which is not perpendicular to one of the axes. For example, one of the created subsets can have points for which $2.7x_3 - 11.9x_7 \leqslant 54.8$, with the other created subset having points for which $2.7x_3 - 11.9x_7 > 54.8$. Similarly, two or more nominal variables can be involved in a *Boolean split*. For example, two nominal variables, gender and race, can be used to create a split with cases corresponding to white males belonging to one subset, and cases corresponding to males who are not white and females belonging to the other subset. (It should be noted that while Breiman et al. (1984) describe such Boolean splits, they do not appear to be included in the Salford Systems implementation.)

In many instances, one may not wish to allow linear combination and Boolean splits, since they can make the resulting tree-structured classifier more difficult to interpret, and can increase the computing time since the number of candidate splits is greatly increased. Also, if just single variable splits are used, the resulting tree is invariant with respect to monotone transformations of the variables, and so one does not have to consider whether say dose, or the log of dose, should be used as a predictor. But if linear combination splits are allowed, transforming the variables can make a difference in the resulting tree, and for the sake of simplicity, one might want to only consider single variable splits. However, it should be noted that sometimes a single linear combination split can be better than many single variable spits, and it may be preferable to have a classifier with fewer terminal nodes, having somewhat complicated boundaries for the sets comprising the partition, than to have one created using only single variable splits and having a large number of terminal nodes.

At each stage in the recursive partitioning, all of the allowable ways of splitting a subset of $\mathcal{X}$ are considered, and the one which leads to the greatest increase in node purity is chosen. This can be accomplished using an *impurity function*, which is a function of the proportions of the learning sample belonging to the possible classes of the response variable. These proportions will be denoted by $p_1, p_2, \ldots, p_{J-1}, p_J$.

The impurity function should be such that it is maximized whenever a subset of $\mathcal{X}$ corresponding to a node in the tree contains an equal number of each of the possible classes. (If there are the same number of c_1 cases as there are c_2 cases and c_3 cases and so on, then we are not able to sensibly associate that node with a particular class, and in fact, we are not able to sensibly favor any class over any other class, giving us that uncertainty is maximized.) The impurity function should assume its minimum value for a node that is completely pure, having all cases from the learning sample corresponding to the node belonging to the same class. Two such functions that can serve as the impurity function are the *Gini index of diversity*,

$$g(p_1, \ldots, p_J) = 2\sum_{j=1}^{J-1} \sum_{j'=j+1}^{J} p_j p_{j'} = 1 - \sum_{j=1}^{J} p_j^2,$$

and the *entropy function*,

$$h(p_1, \ldots, p_J) = -\sum_{j=1}^{J} p_j \log p_j,$$

provided that $0 \log 0$ is taken to be $\lim_{p \downarrow 0} p \log p = 0$.

To assess the quality of a potential split, one can compute the value of the impurity function using the cases in the learning sample corresponding to the parent node (the node to be split), and subtract from this the weighted average of the impurity for the two children nodes, with the weights proportional to the number of cases of the learning sample corresponding to each of the two children nodes, to get the decrease in overall impurity that would result from the split. To select the way to split a subset of $\mathcal{X}$ in the tree growing process, all allowable ways of splitting can be considered, and the one which will result in the greatest decrease in node impurity (greatest increase in node purity) can be chosen.

2.5. Estimating the misclassification rate and selecting the right-sized tree

The trickiest part of creating a good tree-structured classifier is determining how complex the tree should be. If nodes continue to be created until no two distinct values of $\mathbf{x}$ for the cases in the learning sample belong to the same node, the tree may be overfitting the learning sample and not be a good classifier of future cases. On the other hand, if a tree has only a few terminal nodes, then it may be that it is not making enough use of information in the learning sample, and classification accuracy for future cases will suffer. Initially, in the tree-growing process, the predictive accuracy typically improves as more nodes are created and the partition gets finer. But it is usually the case that at some point the misclassification rate for future cases will start to get worse as the tree becomes more complex.

In order to compare the prediction accuracy of various tree-structured classifiers, there needs to be a way to estimate a given tree's misclassification rate for future observations, which is sometimes referred to as the *generalization error*. What does not work well is to use the *resubstitution estimate* of the misclassification rate (also known as the *training error*), which is obtained by using the tree to classify the members of the learning sample (that were used to create the tree), and observing the proportion that are misclassified. If no two members of the learning sample have the same value of $\mathbf{x}$, then a tree having a resubstitution misclassification rate of zero can be obtained by continuing to make splits until each case in the learning sample is by itself in a terminal node (since the class associated with a terminal node will be that of the learning sample case corresponding to the node, and when the learning sample is then classified using the tree, each case in the learning sample will drop down to the terminal node that it created in the tree-growing process, and will have its class match the predicted class for the node). Thus the resubstitution estimate can be a very poor estimate of the tree's misclassification rate for future observations, since it can decrease as more nodes are created, even if the selection of splits is just responding to "noise" in the data, and not real structure. This phenomenon is similar to R^2 increasing as more terms are added to a multiple regression model, with the possibility of R^2 equaling one if enough terms are added, even though more complex regression models can be much worse predictors than simpler ones involving fewer variables and terms.

A better estimate of a tree's misclassification rate can be obtained using an independent *test sample*, which is a collection of cases coming from the same population or distribution as the learning sample. Like the learning sample, for the test sample the true class of each case is known in addition to the values for the predictor variables. The *test sample estimate* of the misclassification rate is just the proportion of the cases in the test sample that are misclassified when predicted classes are obtained using the tree created from the learning sample. If the cases to be classified in the future will also come from the same distribution that produced the test sample cases, the test sample estimation procedure is unbiased.

Since the test sample and the learning sample are both composed of cases for which the true class is known in addition to the values for the predictor variables, choosing to make the test sample larger will result in a smaller learning sample. Often it is thought that about one third of the available cases should be set aside to serve as a test sample, and the rest of the cases should be used as the learning sample. But sometimes a smaller fraction, such as one tenth, is used instead.

If one has enough suitable data, using an independent test sample is the best thing to do. Otherwise, obtaining a *cross-validation estimate* of the misclassification rate is preferable. For a V-fold cross-validation, one uses all of the available data for the learning sample, and divides these cases into V parts of approximately the same size. This is usually done randomly, but one may use stratification to help make the V cross-validation groups more similar to one another. V is typically taken to be 5 or 10. In a lot of cases, little is to be gained by using a larger value for V, and the larger V is, the greater is the amount of time required to create the classifier. In some situations, the quality of the estimate is reduced by making V too large.

To obtain a cross-validation estimate of the misclassification rate, each of the V groups is in turn set aside to serve temporarily as an independent test sample and a tree is grown, according to certain criteria, using the other $V - 1$ groups. In all, V trees are grown in this way, and for each tree the set aside portion of the data is used to obtain a test sample estimate of the tree's misclassification rate. Then the V test sample estimates are averaged to obtain the estimate of the misclassification rate for the tree grown from the entire learning sample using the same criteria. (If $V = 10$, the hope is that the average of the misclassification rates of the trees created using 90% of the data will not be too different from the misclassification rate of the tree created using all of the data.) Further details pertaining to how cross-validation is used to select the right-sized tree and estimate its misclassification rate are given below.

Whether one is going to use cross-validation or an independent test sample to estimate misclassification rates, it still needs to be specified how to grow the best tree, or how to create a set of candidate trees from which the best one can be selected based on their estimated misclassification rates. It does not work very well to use some sort of a *stop splitting rule* to determine that a node should be declared a terminal node and the corresponding subset of $\mathcal{X}$ not split any further, because it can be the case that the best split possible at a certain stage may decrease impurity by only a small amount, but if that split is made, each of the subsets of $\mathcal{X}$ corresponding to both of the descendant nodes can be split in ways to produce an appreciable decrease in impurity. Because of this phenomenon, what works better is to first grow a very large tree, splitting subsets in the current partition of $\mathcal{X}$ even if a split does not lead to an appreciable decrease in impurity. For example, splits can be made until 5 or fewer members of the learning sample correspond to each terminal node (or all of the cases corresponding to a node belong to the same class). Then a sequence of smaller trees can be created by *pruning* the initial large tree, where in the pruning process, splits that were made are removed and a tree having a fewer number of nodes is produced. The accuracies of the members of this sequence of subtrees—really a finite sequence of nested subtrees, since the first tree produced by pruning is a subtree of the original tree, and a second pruning step creates a subtree of the first subtree, and so on—are then compared using good estimates of their misclassification rates (either based on a test sample or obtained by cross-validation), and the best performing tree in the sequence is chosen as the classifier.

A specific way to create a useful sequence of different-sized trees is to use *minimum cost–complexity pruning*. In this process, a nested sequence of subtrees of the initial large tree is created by *weakest-link cutting*. With weakest-link cutting (pruning), all of the nodes that arise from a specific nonterminal node are pruned off (leaving that specific node as a terminal node), and the specific node selected is the one for which the corresponding pruned nodes provide the smallest *per node* decrease in the resubstitution misclassification rate. If two or more choices for a cut in the pruning process would produce the same per node decrease in the resubstitution misclassification rate, then pruning off the largest number of nodes is favored. In some cases (minimal pruning cases), just two children terminal nodes are pruned from a parent node, making it a terminal node. But in other cases, a larger group of descendant nodes are pruned all at once from an internal node of the tree. For example, at a given stage in the pruning process, if the increase in the estimated misclassification rate caused by pruning four nodes is no more than twice the increase in the estimated misclassification rate caused by pruning two nodes, pruning four nodes will be favored over pruning two nodes.

Letting $R(T)$ be the resubstitution estimate of the misclassification rate of a tree, T, and $|T|$ be the number of terminal nodes of the tree, for each $\alpha \geqslant 0$ the *cost–complexity measure*, $R_\alpha(T)$, for a tree, T, is given by

$$R_\alpha(T) = R(T) + \alpha|T|.$$

Here, $|T|$ is a measure of tree complexity, and $R(T)$ is related to misclassification cost (even though it is a biased estimate of the cost). α is the contribution to the measure for each terminal node. To minimize this measure, for small values of α, trees having a large number of nodes, and a low resubstitution estimate of misclassification rate, will be favored. For large enough values of α, a one node tree (with $\mathcal{X}$ not split at all, and all cases to be classified given the same predicted class), will minimize the measure.

Since the resubstitution estimate of misclassification rate is generally overoptimistic and becomes unrealistically low as more nodes are added to a tree, it is hoped that there is some value of α that properly penalizes the overfitting of a tree which is too complex, so that the tree which minimizes $R_\alpha(T)$, for the proper value of α, will be a tree of about the right complexity (to minimize the misclassification rate of future cases). Even though the proper value of α is unknown, utilization of the weakest-link cutting procedure described above guarantees that for each value for α (greater than or equal to 0), a subtree of the original tree that minimizes $R_\alpha(T)$ will be a member of the finite nested sequence of subtrees produced.

The sequence of subtrees produced by the pruning serves as the set of candidates for the classifier, and to obtain the classifier, all that remains to be done is to select the one which will hopefully have the smallest misclassification rate for future predictions. The selection is based on estimated misclassification rates, obtained using a test sample or by cross-validation.

If an independent test sample is available, it is used to estimate the error rates of the various trees in the nested sequence of subtrees, and the tree with the smallest estimated misclassification rate can be selected to be used as the tree-structured classifier. A popular alternative is to recognize that since all of the error rates are not accurately known, but only estimated, it could be that a simpler tree with only a slightly higher estimated error rate is really just as good or better than the tree having the smallest estimated error rate, and the least complex tree having an estimated error rate within one standard error of the estimated error rate of the tree having the smallest estimated error rate can be chosen, taking simplicity into account (and maybe not actually harming the prediction accuracy). This is often referred to as the *1 SE rule*.

If cross-validation is being used instead of a test sample, then things are a bit more complicated. First the entire collection of available cases are used to create a large tree, which is then pruned to create a sequence of nested subtrees. (This sequence of trees, from which the classifier will ultimately be selected, contains the subtree which minimizes $R_\alpha(T)$ for every nonnegative α.) The first of the V portions of data is set aside to serve as a test sample for a sequence of trees created from the other $V - 1$ portions of data. These $V - 1$ portions of data are collectively used to grow a large tree, and then the same pruning process is applied to create a sequence of subtrees. Each subtree in the sequence is the optimal subtree in the sequence, according to the $R_\alpha(T)$ criterion, for some range of values for α. Similar sequences of subtrees are created and the corresponding values of α for which each tree is optimal are determined, by setting aside, one at a time, each of the other $V - 1$ portions of the data, resulting in V sequences in all. Then, for various values of α, cross-validation estimates of the misclassification rates of the corresponding trees created using all of the data are determined as follows: for a given value of α the misclassification rate of the corresponding subtree in each of the V sequences is estimated using the associated set aside portion as a test sample, and the V estimates are averaged to arrive at a single error rate estimate corresponding to that value of α, and this estimate serves as the estimate of the true misclassification rate for the tree created using all of the data and pruned using this value of α. Finally, these cross-validation estimates of the error rates for the trees in the original sequence of subtrees are compared, and the subtree having the smallest estimated misclassification rate is selected to be the classifier. (Note that in this final stage of the tree selection process, the role of α is to create, for each value of α considered, matchings of the subtrees from the cross-validation sequences to the subtrees of the original sequence. The trees in each matched set can be viewed as having been created using the same prescription: a large tree is grown, weakest-link pruning is used to create a sequence of subtrees, and the subtree which minimizes $R_\alpha(T)$ is selected.)

2.6. *Alternative approaches*

As indicated previously, several other programs that create classification trees are apparently very similar to CART, but some other programs have key differences. Some can handle only two classes for the response variable, and some only handle categorical predictors. Programs can differ in how splits are selected, how missing values are dealt with, and how the right-sized tree is determined. If pruning is used, the pruning procedures can differ.

Some programs do not restrict splits to be binary splits. Some allow for *soft splits*, which in a sense divide cases that are very close to a split point, having such cases represented in both of the children nodes at a reduced weight. (See Ripley (1996) for additional explanation and some references.) Some programs use resubstitution estimates of misclassification rates to select splits, or use the result of a chi-squared test, instead of using an impurity measure. (The chi-squared approach selects the split that produces the most significant result when the null hypothesis of homogeneity is tested against the general alternative using a chi-squared test on the two-way table of counts which results from cross-tabulating the members of the learning sample in the subset to be split by the class of their response and by which of the children node created by the split they correspond to.) Some use the result of a chi-squared test to determine that a subset of $\mathcal{X}$ should not be further split, with the splitting ceasing if the most significant split is not significant enough. That is, a stopping rule is utilized to determine the complexity of the tree and control overfitting, as opposed to first growing a complex tree, and then using pruning to create a set of candidate trees, and estimated misclassification rates to select the classifier from the set of candidates.

Ciampi et al. (1987), Quinlan (1987, 1993), and Gelfandi et al. (1991) consider alternative methods of pruning. Crawford (1989) examines using bootstrapping to estimate the misclassification rates needed for the selection of the right-sized tree. Buntine (1992) describes a Bayesian approach to tree construction.

In all, there are a large number of methods for the creation of classification trees, and it is safe to state than none of them will work best in every situation. It may be prudent to favor methods, and particular implementations of methods, which have been thoroughly tested, and which allow the user to make some adjustments in order to tune the method to yield improved predictions.

3. Using CART to create a regression tree

CART creates a regression tree from data having a numerical response variable in much the same way as it creates a classification tree from data having a nominal response, but there are some differences. Instead of a predicted class being assigned to each terminal node, a numerical value is assigned to represent the predicted value for cases having values of **x** corresponding to the node. The sample mean or sample median of the response values of the members of the learning sample corresponding to the node may be used for this purpose. Also, in the tree-growing process, the split selected at each stage is the one that leads to the greatest reduction in the sum of the squared differences between the response values for the learning sample cases corresponding to a particular node and their sample mean, or the greatest reduction in the sum of the absolute differences between the response values for the learning sample cases corresponding to a particular node and their sample median. For example, using the squared differences criterion, one seeks the plane which divides a subset of $\mathcal{X}$ into the sets A and B for which

$$\sum_{i:\,\mathbf{x}_i \in A} (y_i - \bar{y}_A)^2 + \sum_{i:\,\mathbf{x}_i \in B} (y_i - \bar{y}_B)^2$$

is minimized, where $\bar{y}_A$ is the sample mean of the response values for cases in the learning sample corresponding to A, and $\bar{y}_B$ is the sample mean of the response values for cases in the learning sample corresponding to B. So split selection is based on making the observed response values collectively close to their corresponding predicted values. As with classification trees, for the sake of simplicity, one may wish to restrict consideration to splits involving only one variable.

The sum of squared or absolute differences is also used to prune the trees. This differs from the classification setting, where minimizing the impurity as measured by the Gini index might be used to grow the tree, and the resubstitution estimate of the misclassification rate is used to prune the tree. For regression trees, the error-complexity measure used is

$$R_\alpha(T) = R(T) + \alpha|T|,$$

where $R(T)$ is either the sum of squared differences or the sum of absolute differences of the response values in the learning sample and the predicted values corresponding to the fitted tree, and once again $|T|$ is the number of terminal nodes and α is the contribution to the measure for each terminal node. So, as is the case with classification, a biased resubstitution measure is used in the pruning process, and the $\alpha|T|$ term serves to penalize the overfitting of trees which partition $\mathcal{X}$ too finely. To select the tree to be used as the regression tree, an independent test sample or cross-validation is used to select the right-sized tree based on the mean squared prediction error or the mean absolute prediction error.

Some do not like the discontinuities in the prediction surface which results from having a single prediction value for each subset in the partition, and may prefer an alternative regression method such as MARS (Multivariate Adaptive Regression Splines (see Friedman, 1991)) or even OLS regression. However, an advantage that regression trees have over traditional linear regression models is the ability to handle a nonhomogeneous response. With traditional regression modeling, one seeks a linear function of the inputs and transformations of the inputs to serve as the response surface for the entire measurement space, $\mathcal{X}$. But with a regression tree, some variables can heavily influence the predicted response on some subsets of $\mathcal{X}$, and not be a factor at all on other subsets of $\mathcal{X}$. Also, regression trees can easily make adjustments for various interactions, whereas discovering the correct interaction terms can be a difficult process in traditional regression modeling. On the other hand, a disadvantage with regression trees is that additive structure is hard to detect and capture. (See Hastie et al. (2001, p. 274) for additional remarks concerning this issue.)

4. Other issues pertaining to CART

4.1. Interpretation

It is often stated that trees are easy to interpret, and with regard to seeing how the input variables in a smallish tree are related to the predictions, this is true. But the instability of trees, meaning that sometimes very small changes in the learning sample values can lead to significant changes in the variables used for the splits, can prevent one from reaching firm conclusions about issues such as overall variable importance by merely examining the tree which has been created.

As is the case with multiple regression, if two variables are highly correlated and one is put into the model at an early stage, there may be little necessity for using the other variable at all. But the omission of a variable in the final fitted prediction rule should not be taken as evidence that the variable is not strongly related to the response.

Similarly, correlations among the predictor variables can make it hard to identify important interactions (even though trees are wonderful for making adjustments for such interactions). For example, consider a regression tree initially split using the variable x_3. If x_1 and x_2 are two highly correlated predictor variables, x_1 may be used for splits in the left portion of a tree, with x_2 not appearing, and x_2 may be used for splits in the right portion of a tree, with x_1 not appearing. A cursory inspection of the tree may suggest the presence of interactions, while a much more careful analysis may allow one to detect that the tree is nearly equivalent to a tree involving only x_1 and x_3, and also nearly equivalent to a tree involving only x_2 and x_3, with both of these alternative trees suggesting an additive structure with no, or at most extremely mild, interactions.

4.2. Nonoptimality

It should be noted that the classification and regression trees produced by CART or any other method of tree-structured classification or regression are not guaranteed to be optimal. With CART, at each stage in the tree growing process, the split selected is the one which will immediately reduce the impurity (for classification) or variation (for regression) the most. That is, CART grows trees using a *greedy algorithm*. It could be that some other split would better set things up for further splitting to be effective. However, a tree-growing program that “looks ahead” would require much more time to create a tree.

CART also makes other sacrifices of optimality for gains in computational efficiency. For example, when working with a test sample, after the large initial tree is grown, more use could be made of the test sample in identifying the best subtree to serve as a classifier. But the minimal cost–complexity pruning procedure, which makes use of inferior resubstitution estimates of misclassification rates to determine a good sequence of subtrees to compare using test sample estimates, is a lot quicker than an exhaustive comparison of all possible subtrees using test sample estimates.

4.3. Missing values

Sometimes it is desired to classify a case when one or more of the predictor values is missing. CART handles such cases using *surrogate splits*. A surrogate split is based on a variable other than the one used for the primary split (which uses the variable that leads to the greatest decrease in node impurity). The surrogate split need not be the second best split based on the impurity criterion, but rather it is a split that mimics as closely as possible the primary split; that is, one which maximizes the frequency of cases in the learning sample being separated in the same way that they are by the primary split.

5. Bagging

Bagging (from bootstrap aggregation) is a technique proposed by Breiman (1996a, 1996b). It can be used to improve both the stability and predictive power of classification and regression trees, but its use is not restricted to improving tree-based predictions. It is a general technique that can be applied in a wide variety of settings to improve predictions.

5.1. Motivation for the method

In order to gain an understanding of why bagging works, and to determine in what situations one can expect appreciable improvement from bagging, it may be helpful to consider the problem of predicting the value of a numerical response variable, $Y_{\mathbf{x}}$, that will result from, or occur with, a given set of inputs, $\mathbf{x}$. Suppose that $\phi(\mathbf{x})$ is the prediction that results from using a particular method, such as CART, or OLS regression with a prescribed method for model selection (e.g., using Mallows' C_p to select a model from the class of all linear models that can be created having only first- and second-order terms constructed from the input variables). Letting μ_ϕ denote $\mathrm{E}(\phi(\mathbf{x}))$, where the expectation is with respect to the distribution underlying the learning sample (since, viewed as a random variable, $\phi(\mathbf{x})$ is a function of the learning sample, which can be viewed as a high-dimensional random variable) and not $\mathbf{x}$ (which is considered to be fixed), we have that

$$\begin{aligned}
&\mathrm{E}\big(\big[Y_{\mathbf{x}} - \phi(\mathbf{x})\big]^2\big) \\
&\quad = \mathrm{E}\big(\big[(Y_{\mathbf{x}} - \mu_\phi) + \big(\mu_\phi - \phi(\mathbf{x})\big)\big]^2\big) \\
&\quad = \mathrm{E}\big([Y_{\mathbf{x}} - \mu_\phi]^2\big) + 2\mathrm{E}(Y_{\mathbf{x}} - \mu_\phi)\mathrm{E}\big(\mu_\phi - \phi(\mathbf{x})\big) + \mathrm{E}\big(\big[\mu_\phi - \phi(\mathbf{x})\big]^2\big) \\
&\quad = \mathrm{E}\big([Y_{\mathbf{x}} - \mu_\phi]^2\big) + \mathrm{E}\big(\big[\mu_\phi - \phi(\mathbf{x})\big]^2\big) \\
&\quad = \mathrm{E}\big([Y_{\mathbf{x}} - \mu_\phi]^2\big) + \mathrm{Var}\big(\phi(\mathbf{x})\big) \\
&\quad \geqslant \mathrm{E}\big([Y_{\mathbf{x}} - \mu_\phi]^2\big).
\end{aligned}$$

(Above, the independence of the future response, $Y_{\mathbf{x}}$, and the predictor based on the learning sample, $\phi(\mathbf{x})$, is used.) Since in nontrivial situations, the variance of the predictor $\phi(\mathbf{x})$ is positive (since typically not all random samples that could be the learning sample yield the sample value for the prediction), so that the inequality above is strict, this result gives us that if $\mu_\phi = \mathrm{E}(\phi(\mathbf{x}))$ could be used as a predictor, it would have a smaller mean squared prediction error than does $\phi(\mathbf{x})$.

Of course, in typical applications, μ_ϕ cannot serve as the predictor, since the information needed to obtain the value of $\mathrm{E}(\phi(\mathbf{x}))$ is not known. To obtain what is sometimes referred to as the *true* bagging estimate of $\mathrm{E}(\phi(\mathbf{x}))$, the expectation is based on the empirical distribution corresponding to the learning sample. In principle, it is possible to obtain this value, but in practice it is typically too difficult to sensibly obtain, and so the bagged prediction of $Y_\mathbf{x}$ is taken to be

$$\frac{1}{B}\sum_{b=1}^{B}\phi_b^*(\mathbf{x}),$$

where $\phi_b^*(\mathbf{x})$ is the prediction obtained when the base regression method (e.g., CART) is applied to the bth bootstrap sample drawn (with replacement) from the original learning sample. That is, to use bagging to obtain a prediction of $Y_\mathbf{x}$ in a regression setting, one chooses a regression method (which is referred to as the *base method*), and applies the method to B bootstrap samples drawn from the learning sample. The B predicted values obtained are then averaged to produce the final prediction.

In the classification setting, B bootstrap samples are drawn from the learning sample, and a specified classification method (e.g., CART) is applied to each bootstrap sample to obtain a predicted class for a given input, $\mathbf{x}$. The final prediction—the one that results from bagging the specified base method—is the class that occurs most frequently in the B predictions.

An alternative scheme is to bag class probability estimates for each class and then let the predicted class be the one with the largest average estimated probability. For example, with classification trees one has a predicted class corresponding to each terminal node, but there is also an estimate of the probability that a case having $\mathbf{x}$ corresponding to a specific terminal node belongs to a particular class. There is such an estimate for each class, and to predict the class for $\mathbf{x}$, these estimated probabilities from the B trees can be averaged, and the class corresponding to the largest average estimated probability chosen. This can yield a different result than what is obtained by simple voting. Neither of the two methods works better in all cases. Hastie et al. (2001) suggest that averaging the probabilities tends to be better for small B, but also includes an example having the voting method doing slightly better with a large value of B.

Some have recommended using 25 or 50 for B, and in a lot of cases going beyond 25 bootstrap samples will lead to little additional improvement. However, Figure 8.10 of Hastie et al. (2001) shows that an appreciable amount of additional improvement can occur if B is increased from 50 to 100 (and that the misclassification rate remained nearly constant for all choices of B greater than or equal to 100), and so taking B to be 100 may be beneficial in some cases. Although making B large means that creating the classifier will take longer, some of the increased time requirement is offset by the fact that with bagging one does not have to use cross-validation to select the right amount of complexity or regularization. When bagging, one can use the original learning sample as a test set. (A test set is supposed to come from the same population the learning sample comes from, and in bagging, the learning samples for the B predictors are randomly drawn from the original learning sample of available cases. A test set can easily be drawn as well, in the same way, although Breiman (1996a) suggests that one may use the original learning sample as a test set (since if a huge test set is randomly selected from the original learning sample of size N, each of the original cases should occur in the huge test set with an observed sample proportion of roughly N^{-1}, and so testing with a huge randomly drawn test set should be nearly equivalent to testing with the original learning sample).) Alternatively, one could use *out-of-bag* estimates, letting the cases not selected for a particular bootstrap sample serve as independent test sample cases to use in the creation of the classifier from the bootstrap sample.

5.2. *When and how bagging works*

Bagging works best when the base regression or classification procedure that is being bagged is not very stable. That is, when small changes in the learning sample can often result in appreciable differences in the predictions obtained using a specified method, bagging can result in an appreciable reduction in average prediction error. (See Breiman (1996b) for additional information about instability.)

That bagging will work well when applied to a regression method which is rather unstable for the situation at hand is suggested by the result shown in the preceding subsection. It can be seen that the difference of the mean squared prediction error for predicting $Y_{\mathbf{x}}$ with a specified method,

$$\mathrm{E}\big(\big[Y_{\mathbf{x}} - \phi(\mathbf{x})\big]^2\big),$$

and the mean squared prediction error for predicting $Y_{\mathbf{x}}$ with the mean value of the predictor, $\mu_\phi = \mathrm{E}(\phi(\mathbf{x}))$,

$$\mathrm{E}\big([Y_{\mathbf{x}} - \mu_\phi]^2\big),$$

is equal to the variance of the predictor, $\mathrm{Var}(\phi(\mathbf{x}))$. The larger this variance is relative to the mean squared prediction error of the predictor, the greater the percentage reduction of the mean squared prediction error will be if the predictor is replaced by its mean. In situations for which the predictor has a relatively large variance, bagging can appreciably reduce the mean squared prediction error, provided that the learning sample is large enough for the bootstrap estimate of μ_ϕ to be sufficiently good.

When predicting $Y_{\mathbf{x}}$ using OLS regression in a case for which the proper form of the model is *known* and all of the variables are included in the learning sample, the error term distribution is approximately normal and the assumptions associated with least squares are closely met, and the sample size is not too small, there is little to be gained from bagging because the prediction method is pretty stable. But in cases for which the correct form of the model is complicated and not known, there are a lot of predictor variables, including some that are just noise not related to the response variable, and the sample size is not really large, bagging may help a lot, if a generally unstable regression method such as CART is used.

Like CART, other specific methods of tree-structured regression, along with MARS and neural networks, are generally unstable and should benefit, perhaps rather greatly, from bagging. In general, when the model is fit to the data adaptively, as is done in the construction of trees, or when using OLS regression with some sort of stepwise procedure for variable selection, bagging tends to be effective. Methods that are much more stable, like nearest neighbors regression and ridge regression, are not expected to greatly benefit from bagging, and in fact they may suffer a degradation in performance. Although the inequality in the preceding subsection indicates that μ_ϕ will not be a worse predictor than $\phi(\mathbf{x})$, since in practice the bagged estimate of μ_ϕ has to be used instead of the true value, it can be that this lack of correspondence results in bagging actually hurting the prediction process in a case for which the regression method is rather stable and the variance of $\phi(\mathbf{x})$ is small (and so there is not a lot to be gained even if μ_ϕ could be used). However, in regression settings for which bagging is harmful, the degradation of performance tends to be slight.

In order to better understand why, and in what situations, bagging works for classification, it may be helpful to understand the concepts of bias and variance for classifiers. A classifier ϕ, viewed as a function of the learning sample, is said to be *unbiased* at $\mathbf{x}$ if when applied to a randomly drawn learning sample from the parent distribution of the actual learning sample, the class which is predicted for $\mathbf{x}$ with the greatest probability is the one which actually occurs with the greatest probability with $\mathbf{x}$; that is, the class predicted by the Bayes classifier. (The Bayes classifier is an ideal classifier that always predicts the class which is most likely to occur with a given set of inputs, $\mathbf{x}$. The Bayes classifier will not always make the correct prediction, but it is the classifier with the smallest possible misclassification rate. For example, suppose that for a given $\mathbf{x}$, *class 1* occurs with probability 0.4, *class 2* occurs with probability 0.35, and *class 3* occurs with probability 0.25. The Bayes classifier makes a prediction of *class 1* for this $\mathbf{x}$. The probability of a misclassification for this $\mathbf{x}$ is 0.6, but any prediction other than *class 1* will result in a greater probability of misclassification. It should be noted that in practice it is rare to have the information necessary to determine the Bayes classifier, in which case the best one can hope for is to be able to create a classifier that will perform similarly to the Bayes classifier.) Letting $\mathcal{U}_\phi$ be the subset of the measurement space on which ϕ is unbiased, and $\mathcal{B}_\phi$ be the subset of the measurement space on which it is not unbiased, if we could apply ϕ to a large set of independently drawn replications from the parent distribution of the learning sample to create an ensemble of classifiers, and let them vote, then with high probability this aggregated classifier would produce the same predicted class that the Bayes classifier will (due to a law of large numbers effect), *provided that the* $\mathbf{x}$ *being classified belongs to* $\mathcal{U}_\phi$.

The *variance* of a classifier ϕ is defined to be the difference of the probability that $\mathbf{X}$ (a random set of inputs from the distribution corresponding to the learning sample) belongs to $\mathcal{U}_\phi$ and is classified correctly by the Bayes classifier, and the probability that $\mathbf{X}$ belongs to $\mathcal{U}_\phi$ and is classified correctly by ϕ. (To elaborate, here we consider a random vector, $\mathbf{X}$, and its associated class, and state that $\mathbf{X}$ is classified correctly by ϕ if $\phi(\mathbf{X})$ is the same as the class associated with $\mathbf{X}$, where $\phi(\mathbf{X})$ is a function of $\mathbf{X}$ *and* the learning sample, which is considered to be random. The situation is similar for $\mathbf{X}$ being classified correctly by the Bayes classifier, except that the Bayes classifier is not a function of the learning sample.) Since an aggregated classifier will behave very similarly to the Bayes classifier on the subset of the measurement space on which the base classifier is unbiased (provided that the ensemble of base classifiers from which the aggregated classifier is constructed is suitably large), the aggregated classifier can have a variance very close to zero, even if the base classifier does not. The key is that the voting greatly increases the probability that the best prediction for **x** (the prediction of the Bayes classifier, which is not necessarily the correct prediction for a particular case) *will be made* provided that **x** belongs to the part of the measurement space for which the base classifier will make the best prediction with a greater probability than it makes any other prediction. (Note that the probabilities for predicting the various classes with the base classifier are due to the random selection of the learning sample as opposed to being due to some sort of random selection given a fixed learning sample.) Even if the base classifier only slightly favors the best prediction, the voting process ensures that the best prediction is made with high probability. When bagging is applied using a particular base classifier and a given learning sample, the hope is that the learning sample is large enough so that bootstrap samples drawn from it are not too different from random replications from the parent distribution of the learning sample (which is assumed to correspond to the distribution of cases to be classified in the future), and that the base classifier is unbiased over a large portion of the measurement space (or more specifically, a subset of the measurement space having a large probability).

The *bias* of a classifier ϕ is defined to be the difference in the probability that $\mathbf{X}$ belongs to $\mathcal{B}_\phi$ and is classified correctly by the Bayes classifier, and the probability that $\mathbf{X}$ belongs to $\mathcal{B}_\phi$ and is classified correctly by ϕ. Thus the bias of ϕ pertains to the difference in the performance of ϕ and that of the Bayes classifier when $\mathbf{X}$ takes a value in the part of the measurement space on which ϕ is more likely to predict a class other than the class corresponding to the best possible prediction. (Note that the sum of the variance of ϕ and the bias of ϕ equals the difference in the misclassification rate of ϕ and the misclassification rate of the Bayes classifier. Breiman (1998, Section 2) gives more information about bias and variance in the classification setting, suggests that these terms are not ideal due to a lack of correspondence with bias and variance in more typical settings, and notes that others have given alternative definitions.) Just as the process of replication and voting makes a bagged classifier, with high probability, give the same predictions as the Bayes classifier on the subset of the measurement space on which the base classifier is unbiased, it makes a bagged classifier, with high probability, give predictions *different* from those given by the Bayes classifier on the subset of the measurement space on which the base classifier is biased. But if this subset of the measurement space is rather small, as measured by its probability, the amount by which the bias can increase due to aggregation is bounded from above by a small number. *So a key to success with bagging is to apply it to a classification method that has a small bias* with the situation at hand (which typically corresponds to a relatively large variance). It does not matter if the classifier has a large variance, perhaps due to instability, since the variance can be greatly reduced by bagging. (It should be noted that this simple explanation of why bagging a classifier having low bias works ignores effects due to the fact that taking bootstrap samples from the original learning sample is not the same as having independent samples from the parent distribution of the learning sample. But if the original learning sample is not too small, such effects can be relatively small, and so the preceding explanation hopefully serves well to get across the main idea. One can also see Breiman (1996a, Section 4.2), which contains an explanation of why bagging works in the classification setting.)

Generally, bagging a good classifier tends to improve it, while bagging a bad classifier can make it worse. (For a simple example of the latter phenomenon, see Hastie et al. 2001, Section 8.7.1.) But bagging a nearest neighbors classifier, which is relatively stable, and may be rather good in some settings, can lead to no appreciable change, for better or for worse, in performance. Bagging the often unstable classifier CART, which is typically decent, but not always great, can often make it close to being ideal (meaning that its misclassification rate is close to the Bayes rate, which is a lower bound which is rarely achieved when creating classifiers from a random sample). This is due to the fact that if a carefully created classification tree is not too small, it will typically have a relatively small bias, but perhaps a large variance, and bagging can greatly decrease the variance without increasing the bias very much. Results in Breiman (1996a) show that bagged trees outperformed nearest neighbors classifiers, which were not improved by bagging. Bagging can also improve the performances of neural networks, which are generally unstable, but like CART, tend to have relatively low bias.

Breiman (1996a) gives some indications of how well bagging can improve predictions, and of the variability in performance when different data sets are considered. With the classification examples examined, bagging reduced CART's misclassification rates by 6% to 77%. When the amount of improvement due to bagging is small, it could be that there is little room for improvement. That is, it could be that both the unbagged and the bagged classifier have misclassification rates close to the Bayes rate, and that bagging actually did a good job of achieving the limited amount of improvement which was possible. When compared with 22 other classification methods used in the Statlog Project (see Michie et al., 1994) on four publicly available data sets from the Statlog Project, Breiman (1996a) shows that bagged trees did the best overall (although it should be noted that boosted classifiers were not considered). With the regression examples considered by Breiman (1996a), bagging reduced CART's mean squared error by 21% to 46%.

6. Boosting

Boosting is a method of combining classifiers, which are iteratively created from weighted versions of the learning sample, with the weights adaptively adjusted at each step to give increased weight to the cases which were misclassified on the previous step. The final predictions are obtained by weighting the results of the iteratively produced predictors.

Boosting was originally developed for classification, and is typically applied to *weak learners*. For a two-class classifier, a weak learner is a classifier that may only be slightly better than random guessing. Since random guessing has an error rate of 0.5, a weak classifier just has to predict correctly, on average, slightly more than 50% of the time. An example of a weak classifier is a *stump*, which is a two node tree. (In some settings, even such a simple classifier can have a fairly small error rate (if the classes can be nearly separated with a single split). But in other settings, a stump can have an error rate of almost 0.5, and so stumps are generally referred to as weak learners.) However, boosting is not limited to being used with weak learners. It can be used with classifiers which are fairly accurate, such as carefully grown and pruned trees, serving as the *base learner*. Hastie et al. (2001) claim that using trees with between four and eight terminal nodes works well in most cases, and that performance is fairly insensitive to the choice, from this range, which is made. Friedman et al. (2000) give an example in which boosting stumps does appreciably worse than just using a single large tree, but boosting eight node trees produced an error rate that is less than 25 percent of the error rate of the single large tree, showing that the choice of what is used as the base classifier can make a large difference, and that the popular choice of boosting stumps can be far from optimal. However, they also give another example for which stumps are superior to larger trees. As a general strategy, one might choose stumps if it is suspected that effects are additive, and choose larger trees if one anticipates interactions for which adjustments should be made.

6.1. AdaBoost

AdaBoost is a boosting algorithm developed by Freund and Schapire (1996) to be used with classifiers. There are two versions of AdaBoost.

AdaBoost.M1 first calls for a classifier to be created using the learning sample (that is, the base learner is fit to the learning sample), with every case being given the same weight. If the learning sample consists of N cases, the initial weights are all $1/N$. The weights used for each subsequent step depend upon the weighted resubstitution error rate of the classifier created in the immediately preceding step, with the cases being misclassified on a given step being given greater weight on the next step.

Specifically, if $I_{m,n}$ is equal to 1 if the nth case is misclassified on the mth step, and equal to 0 otherwise, and $w_{m,n}$ is the weight of the nth case for the mth step, where the weights are positive and sum to 1, for the weighted resubstitution error rate for the mth step, e_m, we have

$$e_m = \sum_{n=1}^{N} w_{m,n} I_{m,n}.$$

The weights for the $(m+1)$th step are obtained from the weights for the mth step by multiplying the weights for the correctly classified cases by $e_m/(1-e_m)$, which should be less than 1, and then multiplying the entire set of values by the number greater than 1 which makes the N values sum to 1. This procedure downweights the cases which were correctly classified, which is equivalent to giving increased weight to the cases which were misclassified. For the second step, there will only be two values used for the weights, but as the iterations continue, the cases which are often correctly classified can have weights much smaller than the cases which are the hardest to classify correctly. If the classification method of the weak learner does not allow weighted cases, then from the second step onwards, a random sample of the original learning sample cases is drawn to serve as the learning sample for that step, with independent selections made using the weights as the selection probabilities.

The fact that the cases misclassified on a given step are given increased weight on the next step typically leads to a different fit of the base learner, which correctly classifies some of the misclassified cases from the previous step, contributing to a low correlation of predictions from one step and the next. Amit et al. (1999) provide some details which indicate that the AdaBoost algorithm attempts to produce low correlation between the predictions of the fitted base classifiers, and some believe that this phenomenon is an important factor in the success of AdaBoost (see Breiman's discussion in Friedman et al. (2000) and Breiman (2001, p. 20)).

Assuming that the weighted error rate for each step is no greater than 0.5, the boosting procedure creates M classifiers using the iteratively adjusted weights. Values such as 10 and 100 have been used for M. But in many cases it will be better to use a larger value, such as 200, since performance often continues to improve as M approaches 200 or 400, and it is rare that performance will suffer if M is made to be too large. The M classifiers are then combined to obtain a single classifier to use. If the error rate exceeds 0.5, which should not occur with only two classes, but could occur if there are more than 2 classes, the iterations are terminated, and only the classifiers having error rates less than 0.5 are combined. In either case, the classifiers are combined by using a weighted voting to assign the predicted class for any input **x**, with the classifier created on the mth step being given weight $\log((1 - e_m)/e_m)$. This gives the greatest weights to the classifiers having the lowest weighted resubstitution error rates. Increasing the number of iterations for AdaBoost.M1 drives the training error rate of the composite classifier to zero. The error rate for an independent test sample need not approach zero, but in many cases it can get very close to the smallest possible error rate.

Breiman's discussion in Friedman et al. (2000) makes the point that after a large number of iterations, if one examines the weighted proportion of times that each of the learning sample cases has been misclassified, the values tend to be all about the same. So instead of stating that AdaBoost concentrates on the hard to classify cases, as some have done, it is more accurate to state that AdaBoost places (nearly) equal importance on correctly classifying each of the cases in the learning sample. Hastie et al. (2001) show that AdaBoost.M1 is equivalent to forward stagewise additive modeling using an exponential loss function.

AdaBoost.M2 is equivalent to AdaBoost.M1 if there are just two classes, but appears to be better than AdaBoost.M1 (see results in Freund and Schapire, 1996) if there are more than two classes, because for AdaBoost.M1 to work well the weak learners should have weighted resubstitution error rates less than 0.5, and this may be rather difficult to achieve if there are many classes. The weak learners used with AdaBoost.M2 assign to each case a set of *plausibility values* for the possible classes. They also make use of a loss function that can assign different penalties to different types of misclassifications. The loss function values used are updated for each iteration to place more emphasis on avoiding the same types of misclassifications that occurred most frequently in the previous step. After the iterations have been completed, for each **x** of interest the sequence of classifiers is used to produce a weighted average of the plausibilities for each class, and the class corresponding to the largest of these values is taken to be the prediction for **x**. See Freund and Schapire (1996, 1997) for details pertaining to AdaBoost.M2.

6.2. *Some related methods*

Breiman (1998) presents an alternative method of creating a classifier based on combining classifiers that are constructed using adaptively resampled versions of the learning sample. He calls his method *arc-x4* and he refers to AdaBoost as *arc-fs* (in honor of Freund and Schapire), since he considers both arc-x4 and arc-fs to be arcing algorithms (and so Breiman considered boosting to be a special case of arcing). When the performances of these two methods were compared in various situations, Breiman (1998) found that there were no large differences, with one working a little better in some situations, and the other working a little better in other situations. He concludes that the *adaptive resampling is the key to success*, and not the particular form of either algorithm, since although both use adaptive resampling, with misclassified cases receiving larger weights for the next step, they are quite different in other ways (for example, they differ in how the weights are established at each step, and they differ in how the voting is done after the iterations have been completed, with arc-x4 using unweighted voting).

Schapire and Singer (1998) improve upon AdaBoost.M2, with their *AdaBoost.MH* algorithm, and Friedman et al. (2000) propose a similar method, called *Real AdaBoost*. Applied to the two class setting, Real AdaBoost fits an additive model for the logit transformation of the probability that the response associated with **x** is a particular one of the classes, and it does this in a stagewise manner using a loss function which is appropriate for fitting logistic regression models. Friedman et al. (2000) also propose *Gentle AdaBoost* and *LogitBoost* algorithms, which in some cases, but not consistently, give improved performance when compared to the other AdaBoost algorithms. Bühlmann and Yu, in their discussion of Friedman et al. (2000), suggest boosting bagged stumps or larger trees, and they give some results pertaining to this method, which they refer to as *bag-boosting*.

The boosting approach can be extended to the regression setting. Friedman (2001) proposes *TreeBoost* methods for function approximation, which create additive models, having regression trees as components, in a stagewise manner using a *gradient boosting* technique. Hastie et al. (2001) refer to this approach as *MART* (Multiple Additive Regression Trees), and they find that incorporating a shrinkage factor to encourage slow learning can improve accuracy.

6.3. *When and how boosting works*

When used with a rather complex base classification method that is unstable, such as carefully grown and pruned trees, boosting, and arcing in general, like bagging, in many cases can dramatically lower the error rate of the classification method. Stated simply, this is due *in part* to the fact that unstable methods have relatively large variances, and boosting decreases the variance without increasing the bias. Breiman (1998) shows that when applied to linear discriminant analysis (LDA), which is a fairly stable method of classification, neither bagging nor boosting has any appreciable effect. In some cases (see, for example, results reported by Freund and Schapire (1996) and Quinlan (1996), where C4.5 was compared to boosted C4.5), boosting *can* make a classifier appreciably worse, but this does not seem to happen in a large percentage of settings which have been investigated, and when it does happen, perhaps a small learning sample size is a contributing factor. Several studies have indicated that boosting can be outperformed when the classes have significant overlap. Nevertheless, it is often the case that some form of boosted classifier can do nearly as good or better than any other classifier.

As to how boosting typically achieves such a low misclassification rate, the opinions are many and varied. Friedman et al. (2000) point out that when used with a rather simple base classifier which is stable but perhaps highly biased, such as trees limited to a rather small number of terminal nodes, which is a common way of applying boosting, the success of boosting is due much more to bias reduction (see Schapire et al., 1998) than it is to variance reduction, and this makes boosting fundamentally different than bagging, despite the fact that both methods combine classifiers based on various perturbations of the learning sample. Schapire et al. (1998) offer the opinion that AdaBoost works because of its ability to produce generally high margins. But Breiman (1999) claims that their explanation is incomplete, and provides some explanation of his own. Some have attempted to explain boosting from a Bayesian point of view. Friedman et al. (2000) give support to their opinion that viewing boosting procedures as stagewise algorithms for fitting additive models greatly helps to explain their performance, and make a connection to maximum likelihood. In the discussion of Friedman et al. (2000), various authors offer additional explanation as to why boosting works. In particular, Breiman finds fault with the explanation provided by Friedman et al. (2000), since it does not explain the strong empirical evidence that boosting is *generally* resistant to overfitting. Friedman et al. (2000) provide one example of boosting leading to an overfit predictor. Also, Rätsch et al. (2001) conclude that overfitting can occur when there is a lot of noise, and they provide some pertinent references.) In his discussion, Breiman indicates that the key to the success of boosting is that it produces classifiers of reasonable strength which have low correlation. Breiman provided additional insight with his 2002 Wald Lecture on Machine Learning (see http://stat-www.berkeley.edu/users/breiman/wald2002-1.pdf).

While the preceding paragraph indicates that it is not fully understood why boosting works as well as it does, there is no question about the fact that it can work very well. In various comparisons with bagging, boosting generally, but not always, does better, and leading researchers in the field of machine learning tend to favor boosting. It should be noted that some of the comparisons of boosting to bagging are a bit misleading because they use stumps as the base classifier for both boosting and bagging, and while stumps, being weak learners, can work very well with boosting, they can be too stable and have too great a bias to work really well with bagging. Trees larger than stumps tend to work better with bagging, and so a fairer comparison would be to compare bagging fairly large trees, boosting both stumps and slightly larger trees, and using an assortment of good classifiers without applying boosting or bagging. Boosting and bagging classifiers other than trees can also be considered. From the results of some such studies, it can be seen that often there is not a lot of difference in the performances of bagged large trees and boosted large trees, although it appears that boosting did better more often than not.

Of course, the main interest should be in an overall winner, and not the comparison of boosting with bagging. However, various studies indicate that no one method works best in all situations. This suggests that it may be wise to be familiar with a large variety of methods, and to try to develop an understanding about the types of situations in which each one works well. Also, one should be aware that there are other types of methods for classification and regression, such as kernel-based methods and support vector machines, that have not been covered in this chapter. Indeed, one other class of methods, of the P&C variety, *random forests* with random feature selection, have been shown by Breiman (2001) to compare very favorably with AdaBoost. Furthermore, despite the fact that random forests use ensembles of classifiers created from independent identically distributed learning samples, as opposed to the adaptively produced learning samples used in boosting, Breiman (2001) claims that AdaBoost is similar to a random forest. So it seems premature to declare that boosting is the clear-cut best of the new methods pertaining to classification developed during the past 15 years, while at the same time it is safe to declare that boosting can be used to create classifiers that are very often among the best.

References

Amit, Y., Blanchard, G., Wilder, K. (1999). Multiple randomized classifiers: MRCL. Technical Report. Department of Statistics, University of Chicago.

Breiman, L. (1996a). Bagging predictors. *Machine Learning* **24**, 123–140.

Breiman, L. (1996b). Heuristics of instability and stabilization in model selection. *Ann. Statist.* **24**, 2350–2383.

Breiman, L. (1998). Arcing classifiers (with discussion). *Ann. Statist.* **26**, 801–849.

Breiman, L. (1999). Prediction games and arcing algorithms. *Neural Comput.* **11**, 1493–1517.

Breiman, L. (2001). Random forests. *Machine Learning* **45**, 5–32.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J. (1984). *Classification and Regression Trees*. Wadsworth, Pacific Grove, CA.

Buntine, W.L. (1992). Learning classification trees. *Statist. Comput.* **2**, 63–73.

Ciampi, A., Chang, C.-H., Hogg, S., McKinney, S. (1987). Recursive partitioning: a versatile method for exploratory data analysis in biostatistics. In: MacNeil, I.B., Umphrey, G.J. (Eds.), *Biostatistics*. Reidel, Dordrecht.

Clark, L.A., Pregibon, D. (1992). Tree based models. In: Chambers, J.M., Hastie, T.J. (Eds.), *Statistical Models in S*. Wadsworth and Brooks/Cole, Pacific Grove, CA.

Crawford, S.L. (1989). Extensions to the CART algorithm. *Int. J. Man–Machine Stud.* **31**, 197–217.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Inform. Comput.* **121**, 256–285.

Freund, Y., Schapire, R. (1996). Experiments with a new boosting algorithm. In: Saitta, L. (Ed.), *Machine Learning: Proceedings of the Thirteenth International Conference*. Morgan Kaufmann, San Francisco, CA.

Freund, Y., Schapire, R. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. System Sci.* **55**, 119–139.

Friedman, J. (1991). Multivariate adaptive regression splines (with discussion). *Ann. Statist.* **19**, 1–141.

Friedman, J. (2001). Greedy function approximation: a gradient boosting machine. *Ann. Statist.* **29**, 1189–1232.

Friedman, J., Hastie, T., Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion). *Ann. Statist.* **28**, 337–407.

Gelfandi, S.B., Ravishankar, C.S., Delp, E.J. (1991). An iterative growing and pruning algorithm for classification tree design. *IEEE Trans. Pattern Anal. Machine Intelligence* **13**, 163–174.

Gentle, J.E. (2002). *Elements of Computational Statistics*. Springer-Verlag, New York.

Harrell, F.E. (2001). *Regression Modeling Strategies: with Applications to Linear Models, Logistic Regression, and Survival Analysis*. Springer-Verlag, New York.

Hastie, T., Tibshirani, R., Friedman, J. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, New York.

Kass, G.V. (1980). An exploratory technique for investigating large quantities of categorical data. *Appl. Statist.* **29**, 119–127.

Loh, W.-Y., Shih, Y.-S. (1997). Split selection methods for classification trees. *Statist. Sin.* **7**, 815–840.

Martinez, W.L., Martinez, A.R. (2002). *Computational Statistics Handbook with MATLAB*. Chapman and Hall/CRC, Boca Raton, FL.

Michie, D., Spiegelhalter, D.J., Taylor, C.C. (1994). *Machine Learning, Neural and Statistical Classification*. Horwood, London.

Morgan, J.N., Messenger, R.C. (1973). THAID: a sequential search program for the analysis of nominal scale dependent variables. Institute for Social Research, University of Michigan, Ann Arbor, MI.

Morgan, J.N., Sonquist, J.A. (1963). Problems in the analysis of survey data, and a proposal. *J. Amer. Statist. Assoc.* **58**, 415–434.

Olshen, R. (2001). A conversation with Leo Breiman. *Statist. Sci.* **16**, 184–198.

Quinlan, J.R. (1979). Discovering rules by induction from large classes of examples. In: Michie, D. (Ed.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press, Edinburgh.

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning* **1**, 81–106.

Quinlan, J.R. (1987). Simplifying decision trees. *Int. J. Man–Machine Stud.* **27**, 221–234.

Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Quinlan, J.R. (1996). Bagging, boosting, and C4.5. In: *Proceedings of AAAI '96 National Conference on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA.

Rätsch, G., Onoda, T., Müller, K.R. (2001). Soft margins for AdaBoost. *Machine Learning* **42**, 287–320.

Ripley, B.D. (1996). *Pattern Recognition and Neural Networks*. Cambridge Univ. Press, Cambridge.

Schapire, R. (1990). The strength of weak learnability. *Machine Learning* **5**, 197–227.

Schapire, R., Singer, Y. (1998). Improved boosting algorithms using confidence-rated predictions. In: *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*. ACM Press.

Schapire, R., Freund, Y., Bartlett, P., Lee, W. (1998). Boosting the margin: a new explanation for the effectiveness of voting methods. *Ann. Statist.* **26**, 1651–1686.

Steinberg, D., Colla, P. (1995). *CART: Tree-Structured Nonparametric Data Analysis*. Salford Systems, San Diego, CA.

Therneau, T.M., Atkinson, E.J. (1997). An introduction to recursive partitioning using the rpart routine. Technical Report. Section of Statistics, Mayo Clinic.