Chapter 8 Notes: Tree-Based Methods

Statistical Learning with R

Jon Geiger

March 18, 2022

Tree-Based Methods are methods for regression and classification which *stratify* or *segment* the predictors into a number of simple regions. They are useful for interpretation, but typically do not compete well with other supervised learning methods. Tree-based methods are often combined to improve accuracy.

8.1: The Basics of Decision Trees

Decision trees can be used for both regression and classification problems.

A **regression tree** is a decision tree used for regression. In the context of baseball data, we can think of predicting a hitter's salary based on the number of years and number of hits that person has. We might first segment the data according to how long people have played, saying that they have either < 4.5 years or ≥ 4.5 years. Then, for the people who have played for more than 4.5 years, we might arrange the data based on who has more or less than 117.5 hits. We would then make predictions that estimate the salary for these two groups of people to be in the middle of these salaries.

The final regions on the tree are called the *terminal nodes* or *leaves*. The places where the predictors are split are called *internal nodes*. The segments that connect the nodes are called *branches*.

Stratifying the Feature Space is how we build a regression tree. There are two main steps:

1. We divide the predictor space $(X_1, X_2, \ldots, X_p)$ into J distinct, non-overlapping regions, $R_1, R_2, \ldots, R_J$.
2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observatiosn in R_j.

Our goal is to find "boxes" that minimize the RSS, given by:

$$\sum_{j=1}^{j} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean response for a specific region R_j.

Rather than considering every possible combination of regions, we take a *top-down, greedy* approach knwon as **recursive binary splitting**. This consists of the following steps:

1. Select a predictor X_j and a cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS. In particular, if we define the pair of half-planes (or regions) as

$$R_1(j, s) = \{X|X_j < s\} \quad \text{and} \quad R_2(j, s) = \{X|X_j \geq s\},$$

we seek the values of j and s which minimize the RSS for both of the regions, or:

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2,$$

where $\hat{y}_{R_1}$ is the mean response for the training observations in $R_1(j,s)$ and $\hat{y}_{R_2}$ is the mean response for the training observations in $R_2(j,s)$.

2. Repeat step 1 to split the data further, splitting one of the two previously identified regions. This will increase the number of regions by one, and the process can be repeated until we have the regions $R_1, \ldots, R_J$.

3. We then predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.

A good method for constructing trees which don't overfit to data are formed in a process called *tree pruning*, which creates a large tree T_0, and *prunes* it back to obtain a *subtree*. Creating every possible tree and testing it using a test MSE or cross-validation is very time-intensive and not efficient, so we can use a method called **cost complexity pruning**. This is summarized below.

We say that α is a nonnegative tuning parameter which controls a trade-off between a subtree's complexity and its fit to the training data. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{T} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T| \quad \text{or} \quad \alpha|T| + \sum_{m=1}^{T} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2$$

is as small as possible. $|T|$ is the number of terminal nodes of the tree T, R_m is the rectangle (subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the predicted response associated with R_m.

When $\alpha = 0$, $T = T_0$ (no pruning), but as we increase α, there's a penalty for choosing a tree with many nodes. This is similar to the shrinkage methods we saw in Chapter 6, in which λ was used to control the complexity of a linear model. α can be selected using cross-validation. The algorithm for building a regression tree is as follows:

1. Use recursive binary splitting to grow a large tree T_0 on the training data, stopping when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use K-fold cross-validation to choose α. That is, divide the trainign observations into K folds. For each $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.
   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

A **classification tree** is a decision tree used for classification rather than regression. For regression trees, we used the mean response in a region. For classification trees, we use the *most commonly occurring class* for the training observations. We are also interested in the *class proportions* for each leaf (terminal node) of the tree.

The process of constructing a classification tree also uses recursive binary splitting, but since we can't use RSS, we could use the *classification error rate*, which is the fraction of training observations in that region which don't belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk})$$

where $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class. Unfortunately, this is not sensitive enough for tree growing, so we could use two other measures.

The **Gini index** is given by:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

which is a measure of the total variance across all the K classes. This takes on a low value if $\hat{p}_{mk}$ is close to zero or one, and can be thought of as *node purity*.

The **entropy** is an alternative to the Gini index, given by:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk},$$

and it turns out that entropy is very numerically similar to the Gini index. These three values are used as follows:

- When *building* a classification tree, either the Gini index or the entropy is used.
- When *pruning* a classification tree, any of the three approaches can be used.
- When *measuring the accuracy* of a classification tree, the classification error rate is used.

In comparing trees and linear models, we can make the fairly clear claim that when the data can be well-explained by a linear model, a linear model will outperform a tree-based approach. However, if there is a highly non-linear and complex relationship between the features and ther esponse, then decision trees will typically outperform classical approaches. Figure 8.7 on p. 339 sums this up quite nicely.

Pros of decision trees:

- Very easy to explain to people. Arguably easier than linear regression!
- Decision trees can be thought to more closely mirror human decision-making than regression and classification approaches from previous chapters.
- Trees can be displayed graphically, and are easily interpreted by non-experts
- Trees handle qualitative predictors without needing to create dummy variables.

Cons of decision trees:

- Trees don't have the same level of predictive accuracy as some other regrssion and classification approaches seen in this book.
- Trees are fairly non-robust. A small change in the data can cause a large change in the final estimated tree, as changes will cascade throughout the whole tree.

By aggregating many decision trees using methods like *bagging*, *random forests*, and *boosting*, the predictive performance of trees can be improved.

8.2: Bagging, Random Forests, Boosting, and Bayesian Additive Regression Trees

An **ensemble method** is an approach combining many simple "building block" models to obtain a single, potentially very powerful, model. The building blocks are sometimes known as *weak learners*, since they lead to mediocre predictions on their own.

Bagging is a general-purpose prodecure for reducing the variance of a statistical learning method, and is very closely related to bootstrapping. Decision trees suffer from very high variance, so if we are able to break our data down into smaller chunks, we can reduce the variance by *averaging a set of observations*.

Given a set of n independent observations $Z_1, \ldots, Z_n$ each with variance σ^2, the variance of $\bar{Z}$ is given by σ^2/n. We can take advantage of this fact to reduce the variance of a statistical learning method by sampling boostrapped data sets. In particular, we wish to take B boostrapped training data sets, we train our method on the bth boostrapped training set in order to get $\hat{f}^{*b}(x)$, and we average all the predictions to obtain:

$$\hat{f}_{\text{bag}} = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions for a final prediction of $\hat{f}_{\text{bag}}(x)$. These trees are deep but haven't been pruned. Each tree has low bias and high variance, and by taking the average of all of the trees, the variance is scaled down by a factor of $1/n$.

To apply bagging to classification trees, the simplest approach is to, instead of taking the average, use the most commonly occurring class, or a *majority vote*, among the B predictions. If $B/4$ of the bootstrapped samples predict class A for observation X_i, and $3B/4$ of the bootstrapped samples predict class B, then $\hat{f}_{\text{bag}}(X_i)$ would predict class B.

Luckily for bagging, since it's a bootstrapping method and involves averaging, using a high value of B will *not* overfit the data, but rather make the test error settle down quite nicely. Additionally, because a single bootstrap data set will, on average, use two-thirds of the data, wen can easily calculate an estimate for the test MSE, known as the *out-of-bag* error estimate, since it's calculated based on all of the observations *not* in the bag (about 1/3). It turns out that with sufficiently large B, the OOB error is virtually equivalent to LOOCV error, thus it's a very good estimate of the test error.

Because bagging is less interpretable than a simple decision tree (though much more flexible), we can get an overall summary of the importance of each predictor using the RSS (for regression) or the Gini index (for classification), measuring how much each of these values decreases due to splits in each of the variables. This summary is known as a *variable importance* summary, and graphical representations are often used with bagging trees since the trees themselves cannot be displayed.

Random Forests provide an improvement over bagged trees by *decorrelating* the trees. Random forests also use bootstrapped samples, but each time a plit is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors, and the split can only use one of those m predictors. We typically choose $m \approx \sqrt{p}$.

How does this work? What is decorrelation?

- Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. In the collection of bagged trees, most or all of the trees will use this strong predictor in the top split. This makes all the bagged trees *correlated* with one another.
- Averaging many highly correlated quantities does not lead to as large a reduction in variance as averaging many uncorrelated quantities.
- Forcing each split to consider only a subset of each predictors allows only $(p-m)/p$ splits to even consider the strong predictor, making the average of the results less variable and more reliable.

Random forests can be thought of as a more general form of bagging, where bagging is a special case of random forests with $m = p$. Random forests generally lead to a reduction in both test error and OOB error over bagging. Especially with something like genetic data, for which there may be certain genes which have a very large impact, random forests should provide a significant improvement over bagging since many of the bagged trees would be highly correlated.

Boosting, in the decision tree context, also involves combining a large number of decision trees, except that the trees are grown *sequentially*: each tree is grown using information from previously grown trees. Each tree is fit on a modified version of the original data set. Boosting is a *slow learner*, meaning that it takes many iterations to give accurate results.

Boosting has three tuning parameters:

- B: the number of trees. Boosting can overfit if B is too large, though it tends to happen slowly. We use cross-validation to select B.
- λ: the shrinkage parameter, a small positive number. This controls the rate at which boosting learns. Very small λ requires larger B to learn effectively. Essentially a rate of convergence, akin to the learning parameter for something like gradient descent.
- d: the number of splits in each tree. Often $d = 1$ works well, in which case there is one split with just two terminal nodes (called a *stump*). This is known as the *interaction depth*.

The algorithm for boosting for regression trees is as follows:

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set
2. For $b = 1, 2, \ldots, B$, repeat:
   a. Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   b. Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
   c. Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x)$$
3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

This procedure offers an improvement over bagging or random forests because overfitting is very possible since those algorithms fit the data hard. Because boosting is a slow learner, it does not overfit to the training set and can yield lower test errors than either of the procedures above. Statistical learning approaches that tend to learn slowly, also tend to perform very well.

In boosting, because the growth of a particular tree takes into account the other trees that have already been grown, smaller trees ($d = 1$) are usually sufficient. This aids in interpretability (using stumps leads to an additive model).

Bayesian Additive Regression Trees (BART), is an ensemble learning method which combines principles from bagging/random forests, as well as boosting. Bagging and random forests make predictions from an average of regression trees, with separate trees built. Boosting uses a weighted sum of trees, where each tree tries to capture new information not yet accounted for. BART captures both of these methods' benefits in one algorithm.

Some notation:

- K: the number of regression trees
- B: the number of iterations for which the BART algorithm will run.
- $\hat{f}_k^b(x)$: the prediction at x for the kth regression tree used in the bth iteration.
- $\hat{f}^b(x) = \sum_{k=1}^{K} \hat{f}_k^b(x)$ for $b = 1, \ldots, B$.

The BART algorithm is as follows:

1. Let $\hat{f}_1^1(x) = \hat{f}_2^1(x) = \cdots = \hat{f}_K^1(x) = \frac{1}{nK} \sum_{i=1}^{n} y_i$.
2. Compute $\hat{f}^1(x) = \sum_{k=1}^{K} \hat{f}_k^1(x) = \frac{1}{n} \sum_{i=1}^{n} y_i$.
3. For $b = 2, \ldots, B$:
   (a) For $k = 1, 2, \ldots, K$:
      i. For $i = 1, \ldots, n$, compute the current partial resitual
      $$r_i = y_i - \sum_{k'<k} \hat{f}_{k'}^b(x_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(x_i)$$
      ii. Fit a new tree, $\hat{f}_k^b(x)$, to r_i, by randomly perturbing the kth tree from the previous iteration, $\hat{f}_k^{b-1}(x)$. Perturbations that improve the fit are favored.
   (b) Compute $\hat{f}^b(x) = \sum_{k=1}^{K} \hat{f}_k^b(x)$
4. Compute the mean after L burn-in samples,
   $$\hat{f}(x) = \frac{1}{B-L} sum_{b=L+1}^{B} \hat{f}^b(x)$$

.

In essence, within a particular iteration, the BART algorithm looks at potential changes it could make to the previous tree, makes a change, and then once an iteration is done, a sum of all the trees is computed. Once this sum has been computed for each iteration, the mean $\hat{f}(x)$ is calculated with the formula from step 4 of the algorithm.

BART performs extremely well in comparison with other tree-based methods, even if it is less interpretable than the other ones.

As a summary, we have the following:

- *Bagging* involves growing trees independently from random samples, with many similarities. Bagging can get caught in local optima.
- *Random Forests* involves growing trees independently from random samples, but each split is performed using a random subset of the features. Random forests do not get caught as often in local optima.
- *Boosting* uses the original data without taking random samples, and is a slow learning method. Each new tree is fit to the signal left over from earlier trees, and shrunken before being used.
- *BART* uses the original data, and trees are grown successively. Each tree is perturbed to avoid local optima and achieve a more thorough exploration of the model space.