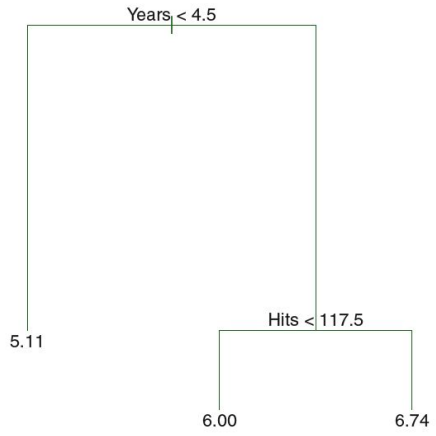
Ch. 8. Tree-Based Methods

Tree-based methods involve stratifying or segmenting the predictor space into a number of simple regions. In order to make a prediction for a given observation, we typically use the mean or the mode of the training observations in the region to which it belongs. Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision tree methods.

Decision trees can be applied to both regression and classification problems. We first consider regression problems.

Regression Trees

We begin with a simple example (pg. 304). We use the Hitters data set to predict a baseball player's Salary (response) based on Years(the number of years he has played in the major leagues) and Hits (the number of hits that he made in previous year).
As data preparation, we remove the observation that are missing Salary values and log-transform Salary so that its distribution has more of a typical bell-shape (Recall that Salary is measured in thousands of dollars).

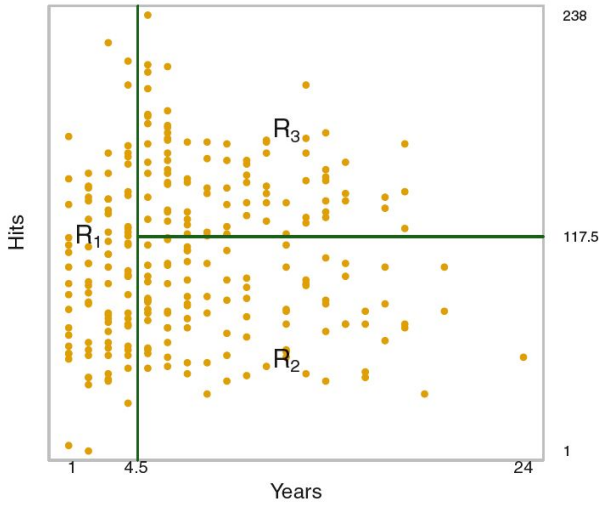


The regression tree fit for this data consists of a series of splitting rules, starting at the top of the tree.
The top split assigns obs having Years<4.5 to the left branch. The predicted Salary for these players is given by the mean response value for the players in the data set with Years<4.5. For such players, the mean log salary is 5.107 and so we make a prediction of $e^{5.107}$ thousands of dollars, i.e. 165,174$.

Players with Years>=4.5 are assigned to the right branch, and then that group is further subdivided by Hits.

Overall, the tree stratifies or segment the players into three regions of predictor space: players who have played for four or fewer years, players who have played for five or more years AND who made fewer than 118 hits last year, and players who have played for five or more years AND who made 118 hits or more last year.

These three regions can be written as R_1 = {X | Years <4.5}, R_2 = {X | Years >=4.5, Hits < 117.5}, and R_1 = {X | Years >=4.5, Hits >= 117.5}. These regions are known as *terminal nodes* or *leaves* of the tree.



The tree is typically drawn upside-down with the leaves at the bottom. The points along the tree where the predictor space is split are referred to as *internal nodes*. The segments of the trees that connect the nodes as *branches*.

We might interpret the regression tree for Hitters as follows:
Years is the most important factor in determining Salary, and players with less experience earn lower salaries than more experienced players.
Given that a player is less experienced, the number of hits that he made in the previous year seems to play little role in his salary.
But among players who have been in the major leagues for five or more years, the number of hits made in the previous year does affect salary, and players who made more hits last year tend to have higher salaries.

Prediction via Stratification of the Feature Space

Roughly speaking, there are two steps for building a regression tree:

1. We divide the predictor space -- that is, the set of possible values for X_1, X_2, …, X_p -- into J distinct and non-overlapping regions, R_1, R_2,...,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 observations in R_j.

How do we construct the regions $R_1,...,R_J$? In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or boxes, for simplicity and for ease of interpretation of the resulting predictive model.

The 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 the training observation within the jth box.

It is computationally infeasible to consider every possible partition of the feature space into J boxes. For this reason, we take a *top-down*, *greedy* approach that is known as **recursive binary splitting**.
It is **greedy** because at each step of the tree-building process, the **best split** is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

In order to perform recursive binary splitting, we first select the predictor X_j and the cutpoint *s* such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j >= s\}$ leads to the greatest possible reduction in RSS.

In greater detail, for any *j* and *s*, we define the pair of half-planes

$R_1(j,s) = \{X|X_j < s\}$ and $R_2(j,s) = \{X|X_j >= s\}$ and we seek the value of *j* and *s* that minimize the equation:
$\sum_{i: x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i: x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$ 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 $R_2(j,s)$.

Finding the values of *j* and *s* that minimize the equation can be done quite quickly, especially when the number of features *p* is not too large.

Next, we repeat the process, looking for the best predictor X_j and the best cutpoint s in order to split the data further so as to minimize the RSS within each of the resulting regions. However, this time, instead of splitting the entire predictor space, we split one of the two previously identified regions.

We now have three regions. Again, we look to split one of these three regions further so as to minimize the RSS.
The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five obs.

Once the regions $R_1,...,R_J$ have been created (the leaves), we predict the response for a given test obs using the mean of the training obs in the region to which that test obs belongs.

Tree Pruning

The process described above may produce good predictions on the training set, but is likely to overfit the data, leading to poor test performance.
This is because the resulting tree might be too complex (low bias, high variance).

A **smaller tree** with fewer splits (that is, fewer regions $R_1,...,R_J$) might lead to lower variance and better interpretability at the cost of little bias.

One way to achieve this is to build the tree only so long as the decrease in RSS due to each split exceed some (high) threshold. This strategy will result in smaller trees, but is too short-sighted since a seemingly worthless split early on the tree might be followed by a very good split (that is a split that leads to a large reduction in RSS) later on.

Therefore, a **better strategy** is to grow a very large tree T_0, and then *prune* it back in order to obtain a smaller *subtree*.
How do we determine the **best way to prune a tree**? Intuitively, our goal is to select a subtree that leads to the lowest test error rate.

Given a subtree, we can estimate its test error using cross-validation or the validation set approach. However, estimating the cross-validation error for every possible subtree would be too cumbersome, since there is an extremely large number of possible subtrees. We **need a way to select a small set of subtrees for consideration**.

Cost complexity pruning -- also known as *weakest link pruning* -- gives us a way to do just this.
Rather than considering every possible subtree, we consider a sequence of trees indexed by a nonnegative tuning parameter **α**.
For each value of **α** there corresponds a subtree $T \subset T_0$ such that:
$\sum_{m=1}^{|T|} \sum_{i:x \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T|$ is as small as possible. Here |T| is indicates the number of terminal nodes (leaves) of the tree T, R_m is the rectangle (i.e. the subset of predictor space) corresponding to the mth terminal node and $\hat{y}_{R_m}$ is the predicted response associated with R_m.

The tuning parameter **α** controls a trade-off between the subtree's complexity and its fit to the training data. As **α** increases, there is a price to pay for having a tree with many terminal nodes, and so the equation above will tend to be minimized for a small tree (similar to lasso).

It turns out that as we increase **α** from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees for a function of **α** is easy.

We can then select a value of **α** using a validation. We then return to the full data set and obtain the subtree corresponding to **α**.

Building a Regression Tree:

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of obs.
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 training obs into K folds. For each k = 1, …, 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**.

Classification Trees

A classification tree is very similar to a regression tree, except that is is used to predict a qualitative response rather than a quantitative one.
Recall that for a regression tree, the predicted response for an obs is given by the mean response of the training obs that belong to the same terminal node (leaf).
In contrast, for a classification tree, we predict that each observation belongs to the most commonly occurring class of training obs in the region to which it belongs.
In interpreting the results of a classification tree, we are often interested not only in the class prediction corresponding to a particular terminal node region, but also in the class proportions among the training obs that fall into that region.

Growing a classification tree is similar to growing a regression tree except that we cannot use RSS for making the binary splits. A natural alternative is the *classification error rate* → simply the fraction of the training obs in that region that do not belong to the most common class (in that region):

$E = 1 - max_k(\hat{p}_{mk})$ Here $\hat{p}_{mk}$ represents the proportion of training obs in the mth region that are from the kth class. However, it turns out that classification error is not sufficiently sensitive for tree-growing, and in practice two other measures are **preferable**.

The *Gini index* is defined by:
$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$. It is a measure of total variance across the K classes. The Gini index takes on a small value if all of the $\hat{p}_{mk}$'s are close to zero or one.
For this reason, the Gini index is referred to as a measure of node purity: small value → node contains predominantly observation from a single class (⇒ it splits the data "well").

An alternative is *entropy* given by:

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

Since 0 ≤ $\hat{p}_{mk}$ ≤ 1 (it's a proportion). it follows that 0 ≤ - $\hat{p}_{mk}$ log $\hat{p}_{mk}$.
One can show that the entropy will take on a value near zero if the $\hat{p}_{mk}$'s are all near to zero or near one. Therefore, like the Gini index, the entropy will take on a small value if the mth node is pure. These two metrics are quite similar numerically.

When building a classification tree, either the Gini index or the entropy are typically used to evaluate the quality of a particular split. Any of these three approaches might be used when *pruning* the tree, but the classification error rate is preferable if prediction accuracy of the final pruned tree is the goal.

Trees Versus Linear Models

Linear regression assumes a model of the form

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j$$

whereas regression trees assume a model of the form

$$f(X) = \beta_0 + \sum_{m=1}^{M} c_m \cdot 1_{(X \in R_m)}$$

Which model is better? It depends on the problem at hand:

- If the relationship between the features and the response is well approximated by a linear model, then an approach such as **linear regression** will work well and will outperform a method that does not exploit this linear structure (like regression trees)
- If instead there is a highly non-linear and complex relationship between the features and the response, then **decision trees** may outperform classical approaches.

Of course other considerations beyond simply test error may come into play in selecting a statistical learning method; for instance, in certain settings, prediction using a tree may be preferred for the sake of interpretability and visualization.

Advantages and Disadvantages of Trees

- ⇧ Trees are very easy to explain to people. Even easier than linear regression.
- ⇧ Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous chapters.
- ⇧ Trees can be displayed graphically, and are easily interpreted even by a non-expert(especially if they are small).
- ⇧ Trees can handle qualitative predictors without the need to create dummy variables.
- ⇩ Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches seen in this book.
- ⇩ Additionally, trees can be very non-robust. In other words, a small change in the data can cause a large change in the final estimated tree (high variance).

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

Bagging, random forests, and boosting use trees as building blocks to construct more powerful prediction models.

Bagging

The bootstrap, introduced as a way to estimate the standard deviation of a quantity of interest in chapter 5, can be used in a completely different context, in order to improve statistical learning methods such as decision trees.

“Normal” decision trees will suffer from high variance. *Boostrap aggregation* or *bagging* is a general-purpose for reducing the variance of a statistical learning method.

Recall that given a set of n independent obs Z_1, ... , Z_n, each with variance σ^2, the variance of the mean of the obs is given by σ^2/n. In other words, averaging a set of observations reduces variance.

Hence a natural way to reduce variance and increase the prediction accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model for each training set, and average the resulting predictions.

Of course, this is not practical as we generally do not have access to multiple training sets. Instead, we can bootstrap, by taking repeated samples from the (single) training set.
In this approach, we generate B different bootstrapped training data sets. We then train our method on the bth bootstrapped training set in order to get $\hat{f}^{*b}(x)$, and finally average all the B predictions to obtain $\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$.

While bagging can improve predictions for many regression methods, it is particularly useful for decision trees. We construct B regression trees using B bootstrapped training sets, and average the resulting predictions.

These trees are grown depp, and are not pruned.
Hence each individual tree has high variance but low bias. Averaging these B trees reduces the variance.

How can bagging be extended to a **classification problem** where the response Y is qualitative? There are a few approaches, the simplest is as follows.

For a given test obs, we can record the class predicted by each of the B trees, and **take a majority vote**: the overall prediction is the most commonly occurring class among the B predictions.

The number of trees B is not a critical parameter with bagging; using a very large value of B will not lead to overfitting. In practice we use a value of B sufficiently large that the error has settled down.

Out-of-Bag Error Estimation

It turns out that there is a very straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation or the validation set approach.

Recall that the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the obs. One can show that, on average, each bagged tree makes use of around ⅔ of the obs. The remaining ⅓ not used to fit a given bagged tree are referred to as the out-of-bag (OOB) observations.

We can predict the response for the ith obs using each of the trees in which that obs was OOB. This will yield around B/3 predictions for the ith obs. In order to obtain a single prediction for the ith obs, we can average these predicted responses (if regression) or can take a majority vote (if classification). This leads to a single OOB prediction for the ith obs. An OOB pred. can be obtained for each of the n obs, from which the overall OOB MSE (regression) or OOB classification error (classification) can be computed.

The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each obs is predicted using only the trees that were not fit using that obs. It cn be shown that with B sufficiently large, OOB error is virtually equivalent to leave-one-out cross-validation error.
This OOB approach for the test error is particularly convenient when performing bagging on large data sets for which cross-validation would be computationally onerous.

Variable Importance Measures

Bagging typically results in improved accuracy compared to a single tree. However, it can be difficult to interpret the resulting model because when we bag a large # of trees it's impossible to represent the resulting procedure with a single tree, and it is no longer clear which variables are most important to the procedure.

One can obtain an overall summary of the importance of each predictor using the RSS (for bagging regression trees) or the Gini index (for bagging class. trees).

In the case of regression we can record the total amount that the RSS is decreased due to splits over a given predictor, averaged over all B trees → large value, important predictor.

In the context of classification, we can add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees → large value, important predictor.

Random Forests

Random forests provide an improvement over bagged trees by way of a small tweak that **decorrelates the trees**.

As in bagging, we build a number of decision trees on bootstrapped training samples.
But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as **split candidates** from the full set of p predictors.

The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$.

In other words, in building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available p predictors.

Rationale: suppose there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split.
Consequently, all of the bagged trees will look quite similar to each other. Hence the predictions from the bagged trees will be highly correlated.

Unfortunately, averaging many highly correlated quantities does not lead to a large of a reduction in variance as averaging many uncorrelated quantities → bagging will not lead to a substantial reduction in variance over a single tree in this setting.

Random forests overcome this problem by forcing each split to consider only a subset of the predictors. We can think of this process as decorrelating the trees, thereby making the average of the resulting trees less variable and hence more reliable.

The **main difference between bagging and random forests** is the choice of the predictor subset size m. Bagging is a special case of random forests with m = p.

Using a small value of m in building a random forest will typically be helpful when we have a large number of correlated predictors.

As with bagging, RFs will not overfit if we increase B, so in practice we use a value of B sufficiently large for the error rate to have settled down.

Boosting

Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification. Here we restrict our discussion of boosting to the context of decision trees.

Boosting works in a similar way to bagging 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 for Regression Trees

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

What is the **idea**?
Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead **learns slowly**.
Given the current model, we fit a decision tree to the residuals from the model. That is, we fit a tree using the current residuals as the response, rather than the real response Y.
We then add this new decision tree into the fitted function in order to update the residuals.

Each of these trees can be rather small, with just a few terminal nodes, determined by the **parameter d** in the algorithm.

By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well. The **shrinkage parameter** λ slows the process down even further, allowing more and different shaped trees to attach the residuals.

In general, statistical learning approaches that learn slowly tend to perform well.

Note that in boosting, unlike bagging, the construction of each tree depends strongly on the trees that have already been grown.

Boosting classification trees proceeds in a similar but slightly more complex way.

Boosting has **three tuning parameters**:

1. B → total number of trees grown. Boosting *can* overfit if B is too large although this occurs slowly if at all. We use cross-validation to select B.
2. λ → shrinkage parameter, a small positive number. Controls the rate at which boosting learns (aka learning rate). Typical values: 0.01 or 0.001, right choice depends on the problem. Very small λ can require a very large value of B in order to achieve good performance (learning too slow).
3. The number of d splits in each tree. Controls the complexity of the boosted ensemble. Often d =1 works well → each tree is called a *stump*, consisting of a single split. In this case the boosted ensemble is fitting an additive model, since each term involves only a single variable. More generally d is the **interaction depth**, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Difference between boosting and RFs:
in boosting, because the growth of a particular tree takes into account the other trees that have already been grown, **smaller trees are typically sufficient**. Using smaller trees can aid in interpretability as well (stumps → additive model).