Tree-Based Methods

May 24, 2016

Contents

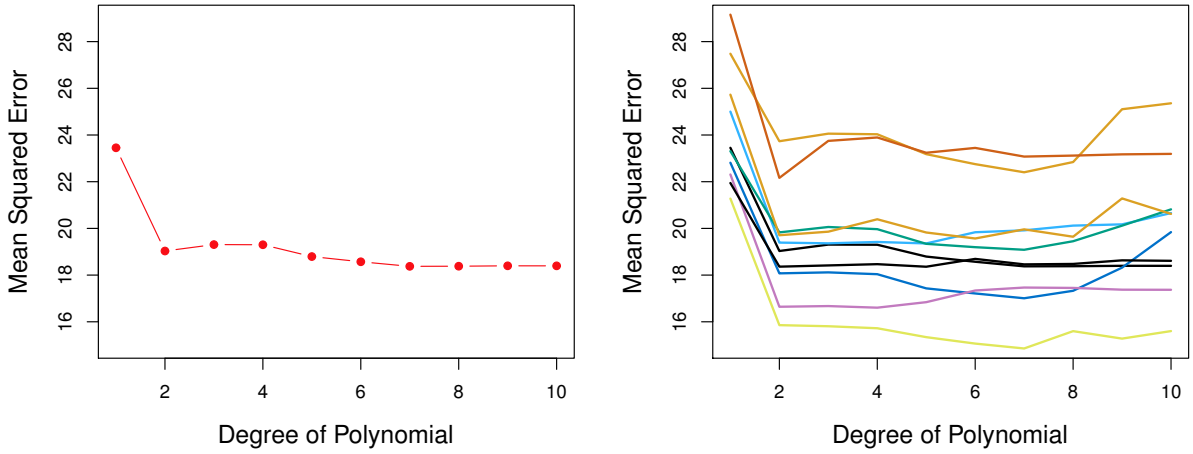
1 Cross-Validation

1.1 The Validation Set Approach

Suppose that we would like to estimate the test error associated with fitting a particular statistical learning method on a set of samples. The validation set approach, is a very simple strategy for this task. It involves randomly dividing the available set of samples into two parts, a training set and a validation set or hold-out set. The model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set. The resulting validation set error rate — typically assessed using MSE in the case of a quantitative response — provides an estimate of the test error rate.

We illustrate the validation set approach on the Auto data set. Assume that there appears to be a non-linear relationship between mpg and horsepower, and that a model that predicts mpg using horsepower and horsepower2 gives better results than a model that uses only a linear term. It is natural to wonder whether a cubic or higher-order fit might provide even better results. We can answer this question by looking at the p-values associated with a cubic term and higher-order polynomial terms in a linear regression. But we could also answer this question using the validation method.

We randomly split the 392 observations into two sets, a training set containing 196 of the data points, and a validation set containing the remaining 196 observations. The validation set error rates that result from fitting various regression models on the training sample and evaluating their performance on the validation sample, using MSE as a measure of validation set error, are shown in the left-hand panel of the following Figure: The validation set MSE for the quadratic fit is considerably smaller than for the linear fit. However, the validation set MSE for the cubic fit is actually slightly larger than for the quadratic fit. This implies that including a cubic term in the regression does not lead to better prediction than simply using a quadratic term.



If we repeat the process of randomly splitting the sample set into two parts, we will get a somewhat different estimate for the test MSE. As an illustration, the right-hand panel of Figure displays ten different validation set MSE curves from the Auto data set, produced using ten different random splits of the observations into training and validation sets. All ten curves indicate that the model with a quadratic term has a dramatically smaller validation set MSE than the model with only a linear term. Furthermore, all ten curves indicate that there is not much benefit in including cubic or higher-order polynomial terms in the model. But it is worth noting that each of the ten curves results in a different

test MSE estimate for each of the ten regression models considered. And there is no consensus among the curves as to which model results in the smallest validation set MSE.

The validation set approach is conceptually simple and is easy to implement. But it has two potential drawbacks:

1. The validation estimate of the test error rate can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the validation set.

2. In the validation approach, only a subset of the observations — those that are included in the training set rather than in the validation set — are used to fit the model. Since statistical methods tend to perform worse when trained on fewer observations, this suggests that the validation set error rate may tend to overestimate the test error rate for the model fit on the entire data set.

1.2 k-Fold Cross-Validation

This approach involves randomly dividing the set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation set, and the method is fit on the remaining $k-1$ folds. The mean squared error, MSE_1, is then computed on the observations in the held-out fold. This procedure is repeated k times; each time, a different group of observations is treated as a validation set. This process results in k estimates of the test error, $MSE_1, MSE_2, \ldots, MSE_k$. The k-fold CV estimate is computed by averaging these values,

$$CV_k = \frac{1}{k}\sum_{i=1}^{k} MSE_i \tag{1}$$

1.3 R Lab

The Validation Set Approach

```
library(ISLR)

## Warning:  package 'ISLR' was built under R version 3.2.4

attach(Auto)
#E1
set.seed(2)
train=sample(392,196)
lm.fit=lm(mpg~horsepower,subset=train)
mean((mpg-predict(lm.fit,Auto))[-train]^2)

## [1] 23.29559

lm.fit2=lm(mpg~poly(horsepower,2,raw=T),data=Auto,subset=train)
mean((mpg-predict(lm.fit2,Auto))[-train]^2)

## [1] 18.90124

lm.fit3=lm(mpg~poly(horsepower,3,raw=T),data=Auto,subset=train)
mean((mpg-predict(lm.fit3,Auto))[-train]^2)

## [1] 19.2574
```

```
#E2
set.seed(1)
train=sample(392,196)
lm.fit=lm(mpg~horsepower,subset=train)
mean((mpg-predict(lm.fit,Auto))[-train]^2)

## [1] 26.14142

lm.fit2=lm(mpg~poly(horsepower,2,raw=T),data=Auto,subset=train)
mean((mpg-predict(lm.fit2,Auto))[-train]^2)

## [1] 19.82259

lm.fit3=lm(mpg~poly(horsepower,3,raw=T),data=Auto,subset=train)
mean((mpg-predict(lm.fit3,Auto))[-train]^2)

## [1] 19.78252

detach(Auto)
```

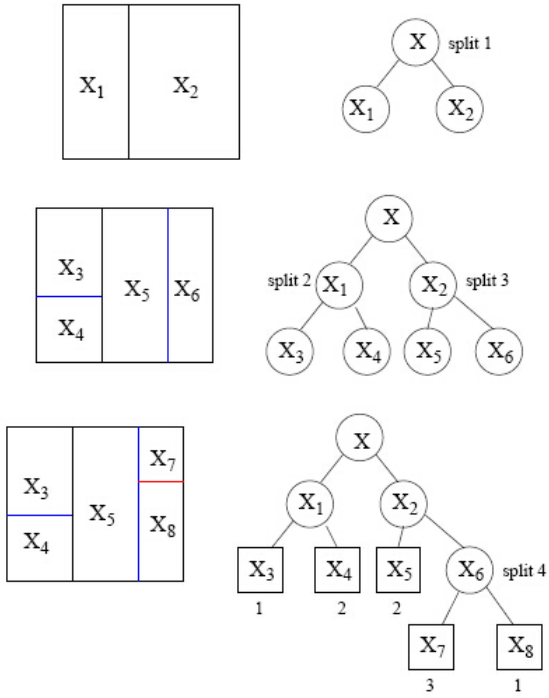
k-Fold Cross-Validation

```
library(boot)
set.seed(17)
cv.error.10=rep(0,10)
for (i in 1:10){
 glm.fit=glm(mpg~poly(horsepower,i),data=Auto)
 cv.error.10[i]=cv.glm(Auto,glm.fit,K=10)$delta[1]
 }
cv.error.10

##  [1] 24.20520 19.18924 19.30662 19.33799 18.87911 19.02103 18.89609
##  [8] 19.71201 18.95140 19.50196
```

2 The Basics of Decision Trees

In this chapter, we describe tree-based methods for regression and classification. These 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.

In this chapter we also introduce bagging, random forests, and boosting. Each of these approaches involves producing multiple trees which are then combined to yield a single consensus prediction. We will see that combining a large number of trees can often result in dramatic improvements in prediction accuracy, at the expense of some loss in interpretation.




```
## Warning:  package 'rpart.plot' was built under R version 3.2.4
```

Decision trees can be applied to both regression and classification problems. We first consider regression problems, and then move on to classification.

2.1 Regression Trees

We use the Hitters data set to predict a baseball player's Salary based on Years (the number of years that he has played in the major leagues) and Hits (the number of hits that he made in the previous year).We first remove observations 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.)

```
## n=263 (59 observations deleted due to missingness)
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 263 53319110 535.9259
##   2) Years< 4.5 90  6769171 225.8315 *
##   3) Years>=4.5 173 33393450 697.2467
##     6) Hits< 117.5 90  5312120 464.9167 *
##     7) Hits>=117.5 83 17955720 949.1708 *
```

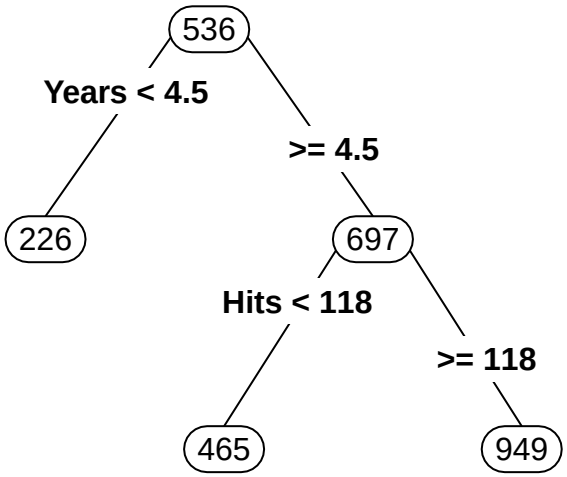


Figure shows a regression tree fit to this data. It consists of a series of splitting rules, starting at the top of the tree. The top split assigns observations having Years<4.5 to the left branch. The predicted salary for these players is given by the mean response value for the players in the data set with Years<4.5. For such players, the mean salary is \$226 thousands of dollars. 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 segments 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 at least 118 hits last year. These three regions can be written as R1 ={X | Years<4.5}, R2 ={X | Years>=4.5, Hits<117.5}, and R3 ={X | Years>=4.5, Hits>=117.5}. The predicted salaries for these three groups are \$226 thousands of dollars, \$465 thousands of dollars, \$949 thousands of dollars, respectively.

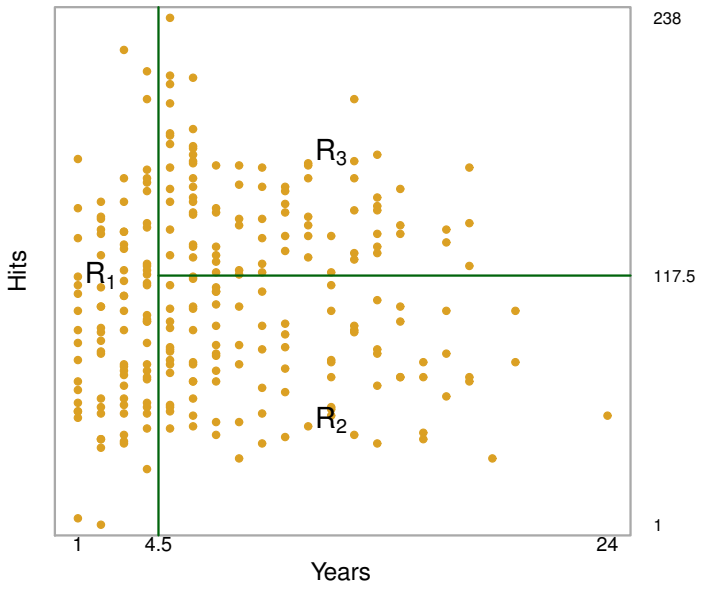
We might interpret the regression tree displayed in Figure 8.1 as follows: Years is the most important factor in determining Salary, and players with less experience earn lower salaries than more experienced players. Given that a player is less experienced, the number of hits that he made in the previous 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. The regression tree shown in Figure is likely an over-simplification of the true relationship between Hits, Years, and Salary. However, it has advantages over other types of regression models: it is easier to interpret, and has a nice graphical representation.

Prediction via Stratification of the Feature Space

We now discuss the process of building a regression tree. Roughly speaking, there are two steps.

1. We divide the predictor space — that is, the set of possible values for $X_1, X_2, \ldots, X_p$ — into J distinct and 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 observations in R_j .




```
Hit<-subset(Hitter,Hitter$Years<=4.5)
mean(Hit$Salary,na.rm=TRUE)

## [1] 225.8315
```

We now elaborate on Step 1 above. How do we construct the regions $R_1, R_2, \ldots, R_J$? 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 $R_1, R_2, \ldots, R_J$ that minimize the RSS, given by

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

where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box. Unfortunately, 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. The approach is top-down because it begins at the top of the tree (at which point all observations belong to a single region) and then successively splits the predictor space; each split is indicated via two new branches further down on the tree. 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 $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS. (The notation $\{X|X_j < s\}$ means the region of predictor space in which X_j takes on a value less than s.) That is, we consider all predictors $X_1, \ldots, X_p$, and all possible values of the cutpoint s for each of the predictors, and then choose the predictor and cutpoint such that the resulting tree has the lowest RSS. In greater detail, for any j and s, we define the pair of half-planes

$$R_1(j, s) = \{X|X_j < s\} \text{ and } R_1(j, s) = \{X|X_j \geq s\} \tag{3}$$

and we seek the value of j and s that minimize the equation

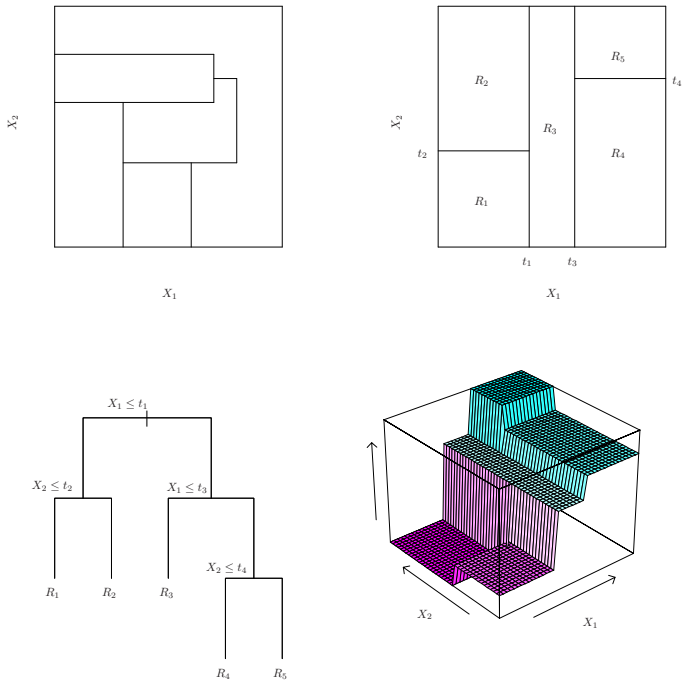
$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2 \tag{4}$$

Finding the values of j and s that minimize (3) 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 and best cutpoint 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 observations.

Once the regions $R_1, R_2, \ldots, R_J$ have been created, we 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 five-region example of this approach is shown in following Figure. Top Left: A partition of two-dimensional feature space that could not result from recursive binary splitting. Top Right: The output of recursive binary splitting on a two-dimensional example. Bottom Left: A tree corresponding to the partition in the top right panel. Bottom Right: A perspective plot of the prediction surface corresponding to that tree.



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 set performance. This is because the resulting tree might be too complex. A smaller tree with fewer splits (that is, fewer regions $R_1, R_2, \ldots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias. One possible alternative to the process described above is to build the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold. This strategy will result in smaller trees, but is too short-sighted since a seemingly

worthless split early on in 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 subtree. How do we determine the best way to prune the 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. Instead, 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

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|, \tag{5}$$

is as small as possible. Here $|T|$ indicates the number of terminal nodes of the tree T , R_m is the rectangle (i.e. the subset of predictor space) corresponding to the m-th terminal node, and $\hat{y}_{R_m}$ is the predicted response associated with R_m — that is, the mean of the training observations in R_m. The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data. When $\alpha = 0$, then the subtree T will simply equal T_0, because then (4) just measures the training error. However, as α increases, there is a price to pay for having a tree with many terminal nodes, and so the quantity (4) will tend to be minimized for a smaller subtree.

Algorithm

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 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. For each $k = 1, ..., K$:

   (a) Repeat Steps 1 and 2 on the $\frac{K-1}{K}$ th fraction of the training data, excluding the kth fold.

   (b) Evaluate the mean squared prediction error on the data in the left-out k-th fold, as a function of α.

   (c) Average the results, and pick α to minimize the average error.

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

R Lab

Baseball Data (Hitters {ISLR})

Description: Major League Baseball Data from the 1986 and 1987 seasons.

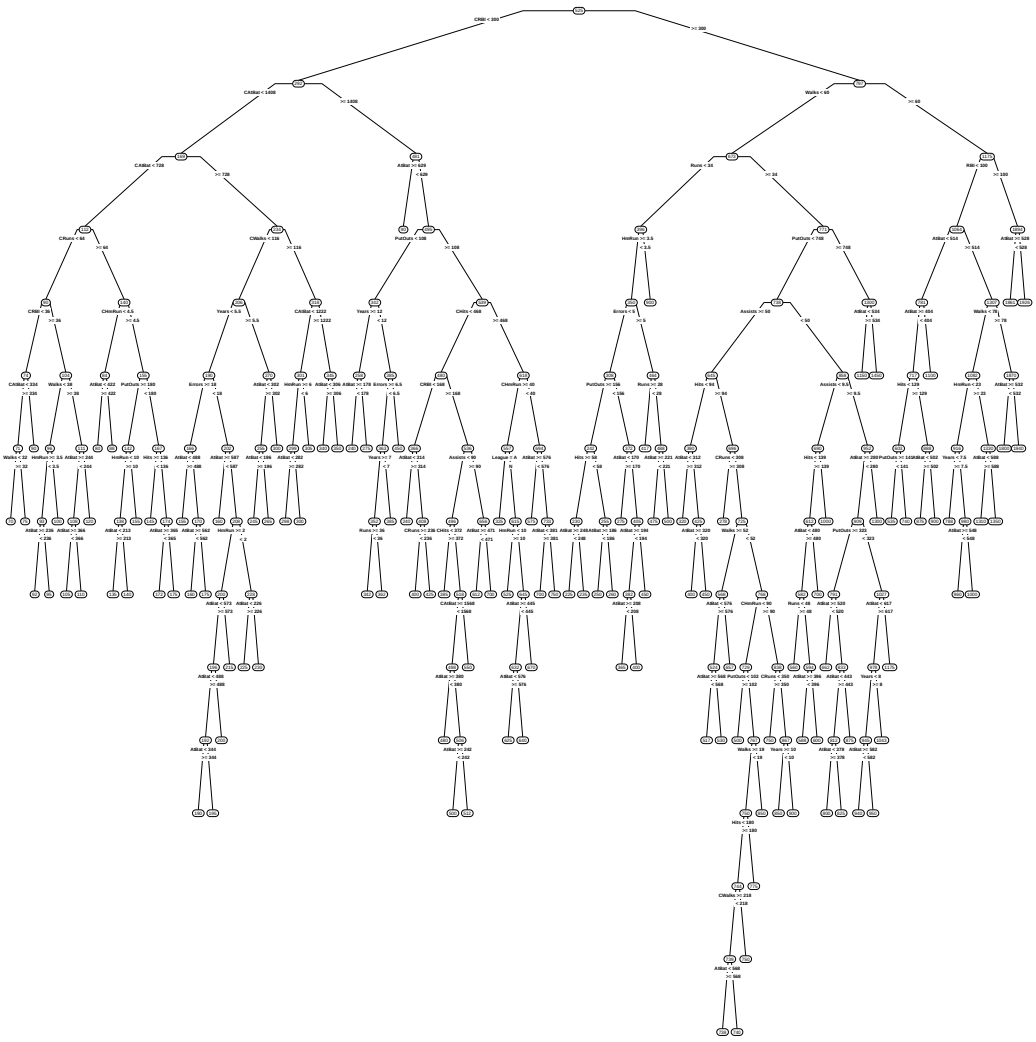
Step 1: Input Data and remove NA

```
#Create the data
library(ISLR)
data("Hitters")
set.seed(12345)
#Split train and test data
apart.data<-function(data,train.data.present=.5){
```

```
        train.index<-sample(c(1:nrow(data)),round(nrow(data)*train.data.present))
        data.train<-data[train.index,]
        data.test<-data[-c(train.index),]
        result<-list(train=data.train,test=data.test)
        result
}
good<-complete.cases(Hitters)
Hitter<-Hitters[good,] #remove NA
data.all<-apart.data(Hitter)
data.train<-data.all$train
data.test<-data.all$test
nrow(data.train)

## [1] 132

nrow(data.test)

## [1] 131

class(data.train)

## [1] "data.frame"
```

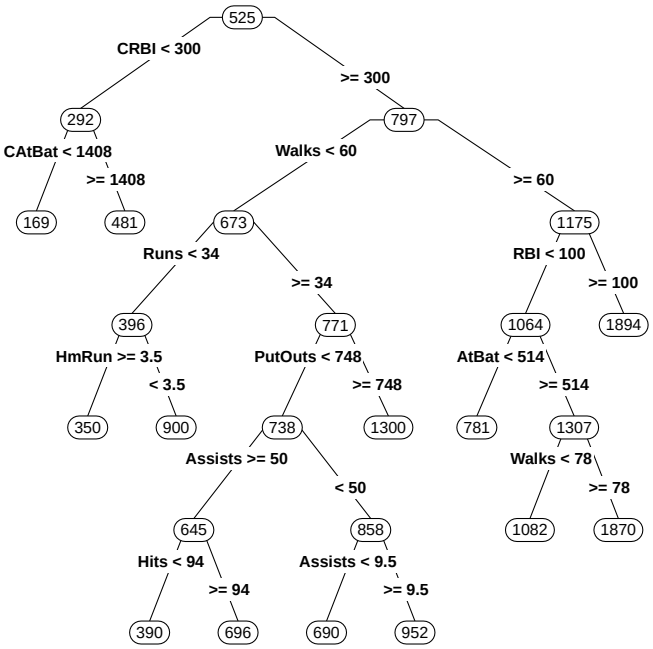
Step 2: Tree-Based analysis (Complete grown tree)

```
#Tree based analysis
#Complete grown tree
Ctl<-rpart.control(minsplit=2,maxcompete=4,xval=6,cp=0)
Fit_train<-rpart(Salary~.,data=data.train,control=Ctl)
rpart.plot(Fit_train,type=4)

## Warning:  labs do not fit even at cex 0.15, there may be some overplotting
```



Step 3: Find a suitable cp and prune tree

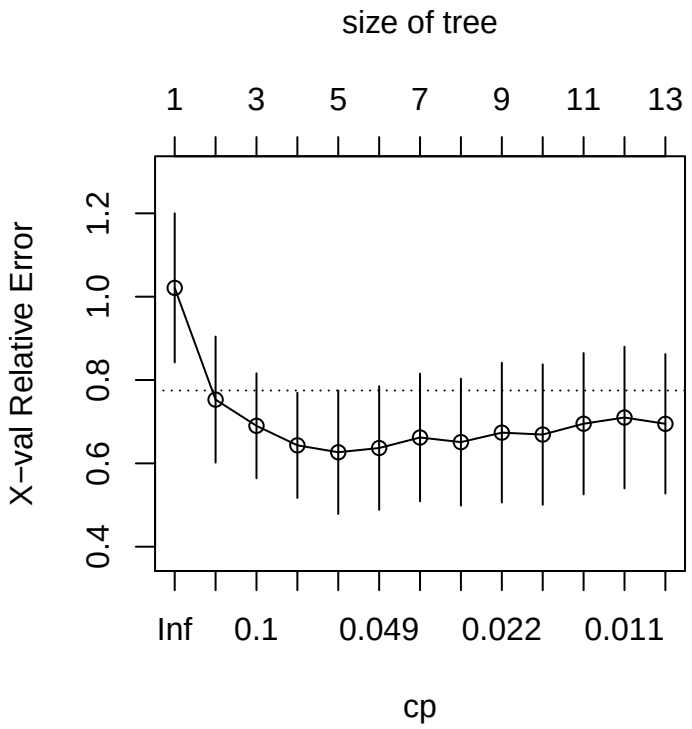
```
#Find a suitable cp
Ctl<-rpart.control(minsplit=2,maxcompete=4,xval=6)
Fit_train1<-rpart(Salary~.,data=data.train,control=Ctl)
rpart.plot(Fit_train1,type=4)
```




```
printcp(Fit_train1)
```

```
##
## Regression tree:
## rpart(formula = Salary ~ ., data = data.train, control = Ctl)
##
## Variables actually used in tree construction:
##  [1] Assists AtBat   CAtBat  CRBI    Hits    HmRun   PutOuts RBI
##  [9] Runs    Walks
##
## Root node error: 21286368/132 = 161260
##
## n= 132
##
##          CP nsplit rel error  xerror    xstd
## 1  0.392703      0   1.00000 1.02112 0.17898
## 2  0.133532      1   0.60730 0.75298 0.15123
## 3  0.077433      2   0.47377 0.69007 0.12595
## 4  0.058682      3   0.39633 0.64326 0.12642
## 5  0.056026      4   0.33765 0.62664 0.14817
## 6  0.041985      5   0.28162 0.63682 0.14834
## 7  0.041726      6   0.23964 0.66209 0.15325
## 8  0.027910      7   0.19791 0.65091 0.15238
## 9  0.016907      8   0.17000 0.67374 0.16777
## 10 0.013020      9   0.15310 0.66911 0.16843
## 11 0.010969     10   0.14008 0.69510 0.16956
## 12 0.010430     11   0.12911 0.71000 0.17005
```

```
## 13 0.010000     12   0.11868 0.69488 0.16730

plotcp(Fit_train1)
```




```
#Fit_train2<-prune(Fit_train1,cp=...)
#rpart.plot(Fit_train2,type=4)
```

2.2 Classification Trees

A classification tree is very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one. Recall that for a regression tree, the predicted response for an observation is given by the mean response of the training observations that belong to the same terminal node. In contrast, for a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations 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 observations that fall into that region.

The task of growing a classification tree is quite similar to the task of growing a regression tree. Just as in the regression setting, we use recursive binary splitting to grow a classification tree. However, in the classification setting, RSS cannot be used as a criterion for making the binary splits.

In practice, two measures are preferable.

The **Gini index** is defined by

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}), \tag{6}$$

a measure of total variance across the K classes. Here $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class. It is not hard to see that 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 — a small value indicates that a node contains predominantly observations from a single class.

An alternative to the Gini index is **cross-entropy**, given by

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log_2 \hat{p}_{mk} \tag{7}$$

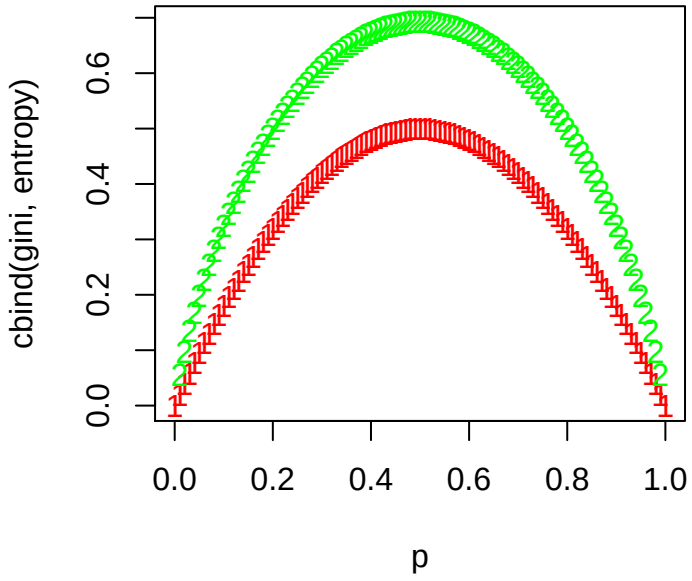
Since $0 \leq \hat{p}_{mk} \leq 1$, it follows that $0 \leq -\hat{p}_{mk} \log_2 \hat{p}_{mk}$. One can show that the cross-entropy will take on a value near zero if the $\hat{p}_{mk}$'s are all near zero or near one. Therefore, like the Gini index, the cross-entropy will take on a small value if the mth node is pure. In fact, it turns out that the Gini index and the cross-entropy are quite similar numerically.

Variable Y	Count
Y=0	0
Y=1	10
Entropy=$-0/10 \times \log_2(0/10) - 10/10 \times \log_2(10/10) = 0$	

Variable Y	Count
Y=0	5
Y=1	5
Entropy=$-5/10 \times \log_2(5/10) - 5/10 \times \log_2(5/10) = 1$	

Variable Y	Count
Y=0	0
Y=1	10
Entropy=$-10/10 \times \log_2(10/10) - 0/10 \times \log_2(0/10) = 0$	

```
p = seq(0, 1, 0.01)
gini = p * (1 - p) * 2
entropy = -(p * log(p) + (1 - p) * log(1 - p))
matplot(p, cbind(gini, entropy), col = c("red", "green"))
```



Information gain

The entropy typically changes when we use a node in a decision tree to partition the training instances into smaller subsets. Information gain is a measure of this change in entropy.

$$\text{Information Gain} = \text{entropy(parent)} - [\text{average entropy(children)}] \tag{8}$$

An Example of Information gain

Consider the following data set. How would you distinguish class I from class II?

X	Y	Z	Class
1	1	1	I
1	1	0	I
0	0	1	II
1	0	0	II

Table 1: Complete dataset

In Complete dataset, the entropy is calculate as

$$\begin{aligned} \text{Entropy}(S) &= -P(\text{Class} = I) \times \log_2 P(\text{Class} = I) - P(\text{Class} = II) \times \log_2 P(\text{Class} = II) \\ &= -1/2 \times \log_2(1/2) - 1/2 \times \log_2(1/2) = 1 \end{aligned}$$

- Split on attribute X

X	1	1	1	0
Class	I	I	II	I

Table 2: Split on attribute X

After split, the entropy is calculate as

$$\begin{aligned}
\text{Entropy}(S|X) &= P(X=1) \times \text{Entropy}(S_{X=1}) + P(X=0) \times \text{Entropy}(S_{X=0}) \\
&= 3/4 \times [-P(\text{Class} = I|X=1) \times \log_2 P(\text{Class} = I|X=1) \\
&\quad -P(\text{Class} = II|X=1) \times \log_2 P(\text{Class} = II|X=1)] \\
&\quad +1/4 \times [-P(\text{Class} = I|X=0) \times \log_2 P(\text{Class} = I|X=0) \\
&\quad -P(\text{Class} = II|X=0) \times \log_2 P(\text{Class} = II|X=0)] \\
&= 3/4 \times [-2/3 \times \log_2(2/3) - 1/3 \times \log_2(1/3)] + 1/4 \times 0 \\
&= 3/4 \times 0.9184 + 1/4 \times 0 = 0.6888
\end{aligned}$$

The Information gain is

$$\text{Information gain} = \text{Entropy}(S) - \text{Entropy}(S|X) = 1 - 0.6888 = 0.3112$$

- Split on attribute Y

Y	1	1	0	0
Class	I	I	II	II

Table 3: Split on attribute Y

After split, the entropy is calculate as

$$\begin{aligned}
\text{Entropy}(S|Y) &= P(Y=1) \times \text{Entropy}(S_{Y=1}) + P(Y=0) \times \text{Entropy}(S_{Y=0}) \\
&= 0
\end{aligned}$$

The Information gain is

$$\text{Information gain} = \text{Entropy}(S) - \text{Entropy}(S|Y) = 1 - 0 = 1$$

- Split on attribute Z

Z	1	1	0	0
Class	I	II	I	II

Table 4: Split on attribute Z

After split, the entropy is calculate as

$$\begin{aligned}
\text{Entropy}(S|Z) &= P(Z=1) \times \text{Entropy}(S_{Z=1}) + P(Z=0) \times \text{Entropy}(S_{Z=0}) \\
&= 1
\end{aligned}$$

The Information gain is

$$\text{Information gain} = \text{Entropy}(S) - \text{Entropy}(S|Z) = 1 - 1 = 0$$

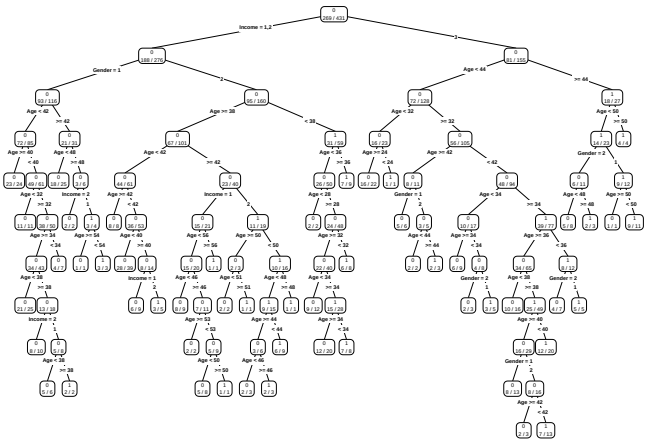
Therefore, the best solution is split on attribute Y. Although information gain is usually a good measure for deciding the relevance of an attribute, it is not perfect. A notable problem occurs when information gain is applied to attributes that can take on a large number of distinct values.

Information gain ratio is sometimes used instead. This biases the decision tree against considering attributes with a large number of distinct values. However, attributes with very low information values then appeared to receive an unfair advantage.

R Lab

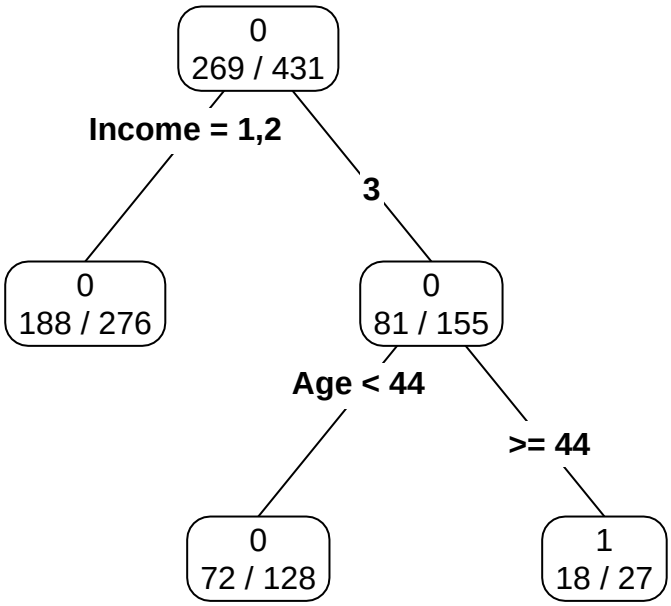
Step 1: Creat Tree

```
BuyOrNot<-read.table(file="BuyOrNot.txt",header=TRUE)
BuyOrNot$Income<-as.factor(BuyOrNot$Income)
BuyOrNot$Gender<-as.factor(BuyOrNot$Gender)
nrow(BuyOrNot)

## [1] 431

set.seed(12345)
Ctl<-rpart.control(minsplit=2,maxcompete=4,xval=10,maxdepth=10,cp=0)
TreeFit<-rpart(Purchase~.,data=BuyOrNot,method="class",parms=list(split="gini"),control=Ctl)
rpart.plot(TreeFit,type=4,branch=0,extra=2)
```



Step 2: Customizing Tree

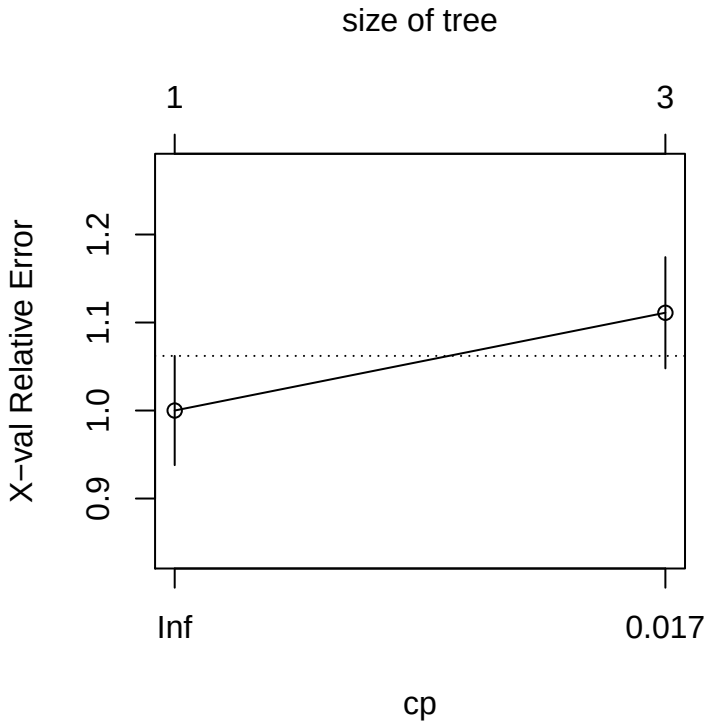
```
Ctl<-rpart.control(minsplit=2,maxcompete=4,xval=10,maxdepth=10)
(TreeFit1<-rpart(Purchase~.,data=BuyOrNot,method="class",parms=list(split="gini"),control=Ctl))

## n= 431
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 431 162 0 (0.6241299 0.3758701)
##   2) Income=1,2 276  88 0 (0.6811594 0.3188406) *
##   3) Income=3 155  74 0 (0.5225806 0.4774194)
##     6) Age< 44.5 128  56 0 (0.5625000 0.4375000) *
##     7) Age>=44.5 27   9 1 (0.3333333 0.6666667) *

rpart.plot(TreeFit1,type=4,branch=0,extra=2)
```



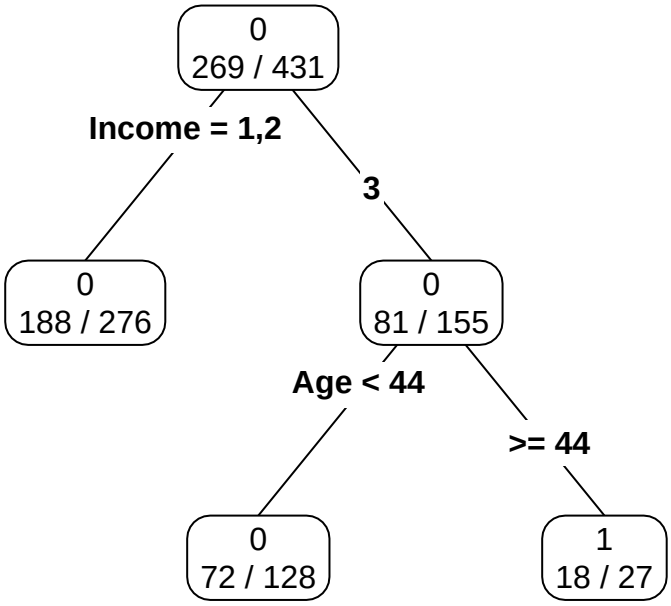
Step 3: Prune tree

```
printcp(TreeFit1)

##
## Classification tree:
## rpart(formula = Purchase ~ ., data = BuyOrNot, method = "class",
##     parms = list(split = "gini"), control = Ctl)
##
## Variables actually used in tree construction:
## [1] Age    Income
##
```

```
## Root node error: 162/431 = 0.37587
##
## n= 431
##
##         CP nsplit rel error xerror    xstd
## 1 0.027778      0   1.00000 1.0000 0.06207
## 2 0.010000      2   0.94444 1.1111 0.06320

plotcp(TreeFit1)
```

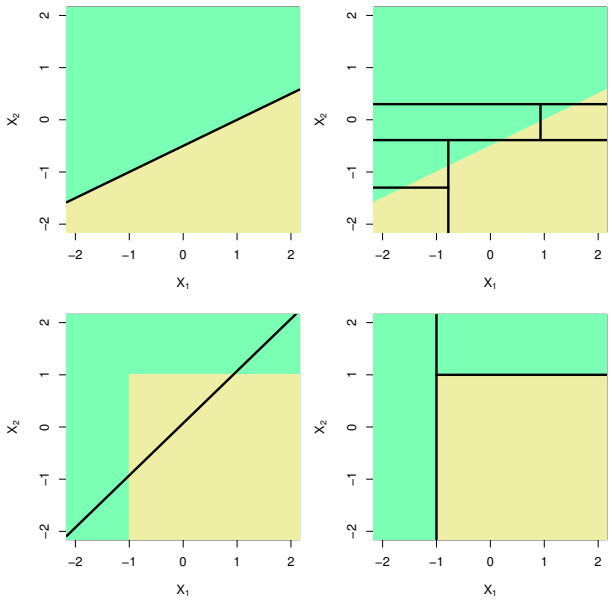



```
TreeFit2<-prune(TreeFit1,cp=0.008)
rpart.plot(TreeFit2,type=4,branch=0,extra=2)
```



Trees Versus Linear Models

Regression and classification trees have a very different flavor from the more classical approaches for regression and classification presented. 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 likely work well, and will outperform a method such as a regression tree that does not exploit this linear structure. If instead there is a highly nonlinear and complex relationship between the features and the response, then decision trees may outperform classical approaches. An illustrative example is displayed in following Figure.



Top Row: A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right). Bottom Row: Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right).

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

Decision trees for regression and classification have a number of advantages over the more classical approaches.

- Trees are very easy to explain to people. In fact, they are even easier to explain 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 easily 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.

2.3 Bagging, Random Forests, Boosting

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

2.3.1 Bagging

Bootstrap aggregation, or bagging, is a general-purpose procedure for reducing the variance of a statistical learning method; we introduce it here because it is particularly useful and frequently used in the context of decision trees.

Recall that given a set of n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^2/n. In other words, averaging a set of observations reduces variance. Hence a natural way to reduce the variance and hence increase the prediction accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions.

Of course, this is not practical because we generally do not have access to multiple training sets. Instead, we can bootstrap, by taking repeated samples from the (single) training data set. In this approach we generate B different bootstrapped training data sets.

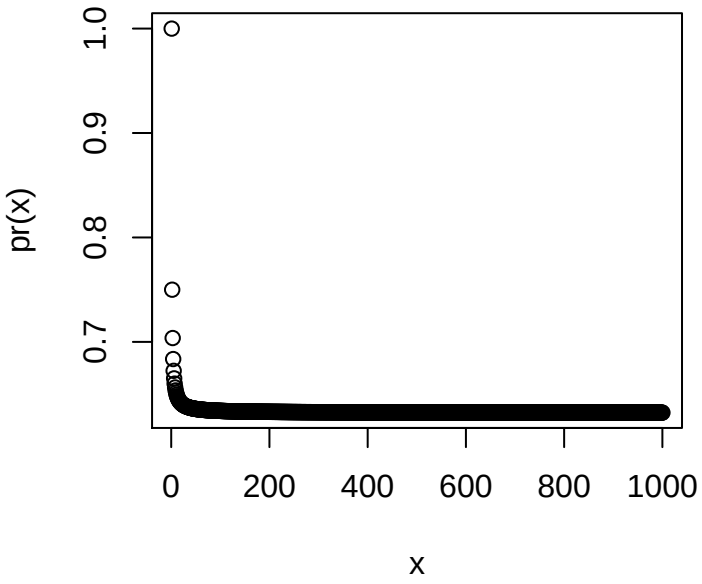
While bagging can improve predictions for many regression methods, it is particularly useful for decision trees. To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions. These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. Bagging has been demonstrated to give impressive improvements in accuracy by combining together hundreds or even thousands of trees into a single procedure.

How can bagging be extended to a classification problem where Y is qualitative? In that situation, there are a few possible approaches, but the simplest is as follows. For a given test observation, 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.

Out-of-Bag Error Estimation

In bootstrap, we sample with replacement so each observation in the bootstrap sample has the same $1/n$ (independent) chance of equaling the jth observation. Applying the product rule for a total of n observations gives us $(1 - 1/n)^n$.

```
pr = function(n) return(1 - (1 - 1/n)^n)
x = 1:1e+03
plot(x, pr(x))
```



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 observations. One can show that on average, each bagged tree makes use of around two-thirds of the observations. The remaining one-third of the observations 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 observation using each of the trees in which that observation was OOB. This will yield around B/3 predictions for the ith observation. In order to obtain a single prediction for the ith observation, we can average these predicted responses (if regression is the goal) or can take a majority vote (if classification is the goal). This leads to a single OOB prediction for the ith observation. An OOB prediction can be obtained in this way for each of the n observations, from which the overall OOB MSE (for a regression problem) or classification error (for a classification problem) can be computed. The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the trees that were not fit using that observation.

It can be shown that with B sufficiently large, OOB error is virtually equivalent to leave-one-out cross-validation error. The OOB approach for estimating the test error is particularly convenient when performing bagging on large data sets for which cross-validation would be computationally onerous.

Variable Importance Measures

As we have discussed, bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, however, it can be difficult to interpret the resulting model. Recall that one of the advantages of decision trees is the attractive and easily interpreted diagram that results, such as the one displayed in Figure 8.1. However, when we bag a large number of trees, it is no longer possible to represent the resulting statistical learning procedure using a single tree, and it is no longer clear which variables are most important to the procedure. Thus, bagging improves prediction accuracy at the expense of interpretability.

Although the collection of bagged trees is much more difficult to interpret than a single tree, one can

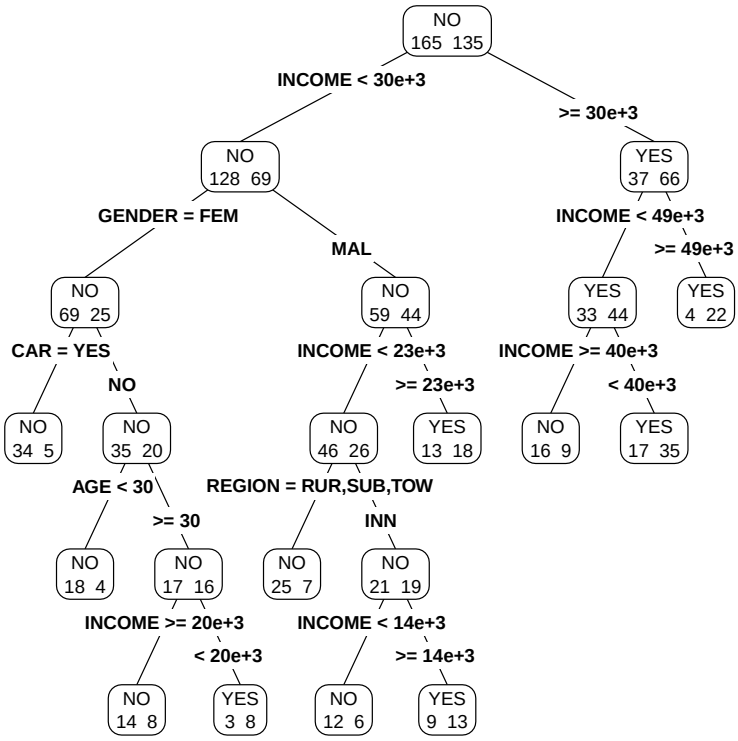
obtain an overall summary of the importance of each predictor using the RSS (for bagging regression trees) or the Gini index (for bagging classification trees). In the case of bagging regression trees, we can record the total amount that the RSS (1) is decreased due to splits over a given predictor, averaged over all B trees. A large value indicates an important predictor. Similarly, in the context of bagging classification trees, we can add up the total amount that the Gini index (6) is decreased by splits over a given predictor, averaged over all B trees.

R Lab

Data: Mailshot

Step 1: Create a single tree.

```
MailShot<-read.table(file="MailShot.txt",header=TRUE)
MailShot<-MailShot[,-1]
Ctl<-rpart.control(minsplit=20,maxcompete=4,maxdepth=30,cp=0.01,xval=10)
set.seed(12345)
TreeFit<-rpart(MAILSHOT~.,data=MailShot,method="class",parms=list(split="gini"))
rpart.plot(TreeFit,type=4,branch=0,extra=1)
```




```
CFit1<-predict(TreeFit,MailShot,type="class")
ConfM1<-table(MailShot$MAILSHOT,CFit1)
(E1<-(sum(ConfM1)-sum(diag(ConfM1)))/sum(ConfM1))

## [1] 0.2833333
```

Step 2-1: Bagging result used ipred package:

```
library("ipred")

## Warning:  package 'ipred' was built under R version 3.2.4

set.seed(12345)
(BagM1<-bagging(MAILSHOT~.,data=MailShot,coob=TRUE,control=Ctl))

##
## Bagging classification trees with 25 bootstrap replications
##
## Call: bagging.data.frame(formula = MAILSHOT ~ ., data = MailShot, coob = TRUE,
##     control = Ctl)
##
## Out-of-bag estimate of misclassification error:  0.4467

CFit2<-predict(BagM1,MailShot,type="class")
ConfM2<-table(MailShot$MAILSHOT,CFit2)
(E2<-(sum(ConfM2)-sum(diag(ConfM2)))/sum(ConfM2))

## [1] 0.2233333
```

Step 2-2: Bagging result used adabag package:

```
detach("package:ipred")
library("adabag")

## Warning:  package 'adabag' was built under R version 3.2.4
## Loading required package:  mlbench
## Warning:  package 'mlbench' was built under R version 3.2.4
## Loading required package:  caret
## Warning:  package 'caret' was built under R version 3.2.4
## Loading required package:  lattice
##
## Attaching package:  'lattice'
## The following object is masked from 'package:boot':
##
##    melanoma
## Loading required package:  ggplot2
## Warning:  package 'ggplot2' was built under R version 3.2.4

MailShot<-read.table(file="MailShot.txt",header=TRUE)
MailShot<-MailShot[,-1]
Ctl<-rpart.control(minsplit=20,maxcompete=4,maxdepth=30,cp=0.01,xval=10)
set.seed(12345)
BagM2<-bagging(MAILSHOT~.,data=MailShot,control=Ctl,mfinal = 25)
BagM2$importance

##       AGE       CAR    GENDER    INCOME   MARRIED  MORTGAGE    REGION
## 18.274203  2.899280  7.094099 50.035502  7.022585  6.118021  6.653972
##      SAVE
##  1.902338
```

```
CFit3<-predict.bagging(BagM2,MailShot)
CFit3$confusion
```

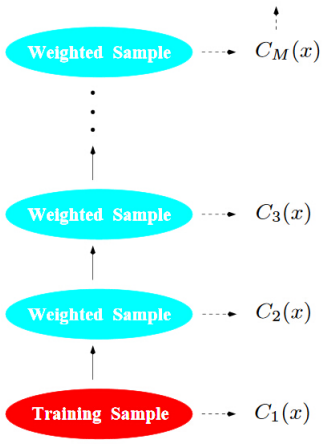
```
##                Observed Class
## Predicted Class  NO YES
##             NO  145  47
##             YES  20  88
```

```
CFit3$error
```

```
## [1] 0.2233333
```

2.3.2 Boosting

We now discuss boosting, yet another approach for improving the predictions resulting from a decision tree. 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

- Average many trees, each grown to re-weighted versions of the training data.
- Final Classifier is weighted average of classifiers:

$$C(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m C_m(x)\right]$$

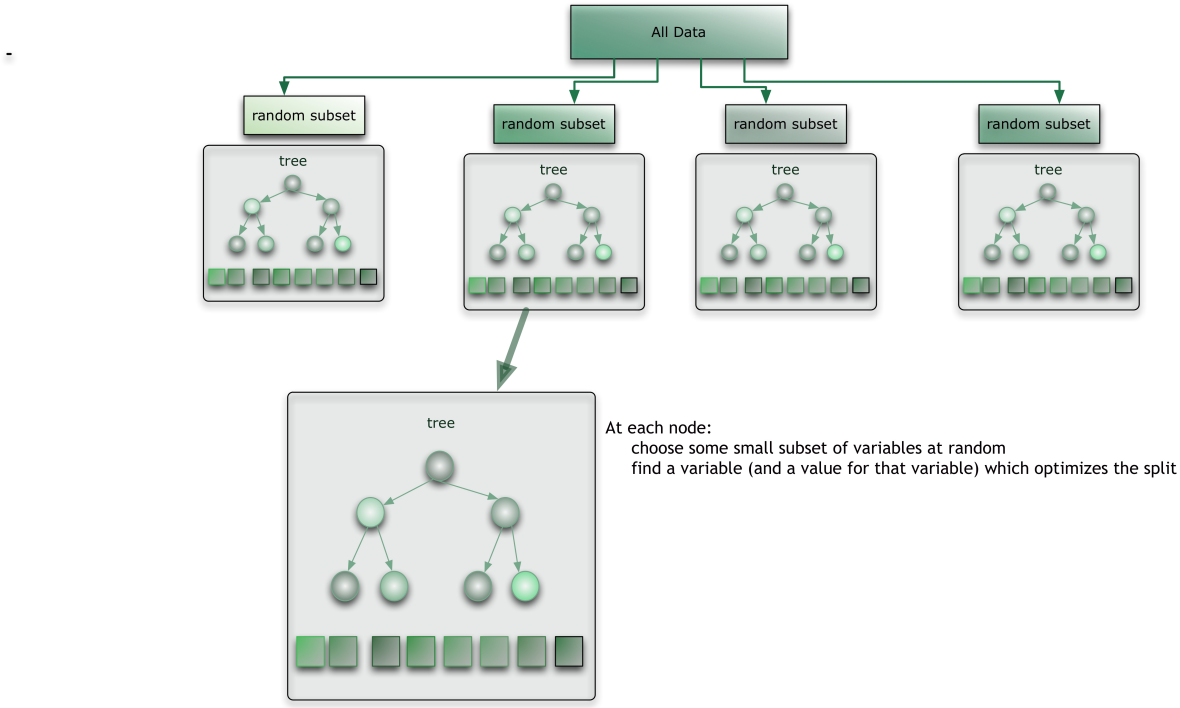
R Lab

```
MailShot<-read.table(file="MailShot.txt",header=TRUE)
MailShot<-MailShot[,-1]
Ctl<-rpart.control(minsplit=20,maxcompete=4,maxdepth=30,cp=0.01,xval=10)
set.seed(12345)
BoostM<-boosting(MAILSHOT~.,data=MailShot,boos=TRUE,mfinal=25,coeflearn="Breiman",control=Ctl)
BoostM$importance
```

```
##       AGE       CAR    GENDER    INCOME   MARRIED  MORTGAGE    REGION
## 24.653433  2.421432  3.444950 44.876762  4.006865  5.772242 11.572626
##      SAVE
##  3.251690
```

```
ConfM4<-table(MailShot$MAILSHOT,BoostM$class)
(E4<-(sum(ConfM4)-sum(diag(ConfM4)))/sum(ConfM4))
```

```
## [1] 0.02666667
```



2.3.3 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 $m \approx \sqrt{p}$ — that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors.

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 predictors. This may sound crazy, but it has a clever rationale. Suppose that 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 as large of a reduction in variance as averaging many uncorrelated quantities. In particular, this means that 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. Therefore, on average $(p - m)/p$ of the splits will not even consider the strong predictor, and so other predictors will have more of a chance. 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 predictor subset size m. For instance, if a random forest is built using $m = p$, then this amounts simply to bagging. Using a small value of m in building a random forest will typically be helpful when we have a large number of correlated predictors.

Algorithm

Here is how such a system is trained; Each tree is grown as follows:

1. Sample N cases at random with replacement to create a subset of the data (see top layer of figure above). The subset should be about 66% of the total set.
2. At each node:
   (a) For some number m (see below), m predictor variables are selected at random from all the predictor variables.
   (b) The predictor variable that provides the best split, according to some objective function, is used to do a binary split on that node.
   (c) At the next node, choose another m variables at random from all predictor variables and do the same.
3. Each tree is grown to the largest extent possible. There is no pruning.

Depending upon the value of m, there are three slightly different systems:

1. Random splitter selection: $m = 1$.
2. Breiman's bagger: $m =$ total number of predictor variables.
3. Random forest: $m \ll$ number of predictor variables. Brieman suggests three possible values for m: $\sqrt{m}, 2\sqrt{m}$.

Strengths and weaknesses

Random forest runtimes are quite fast, and they are able to deal with unbalanced and missing data. Random Forest weaknesses are that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy. Of course, the best test of any algorithm is how well it works upon your own data set.

R Lab

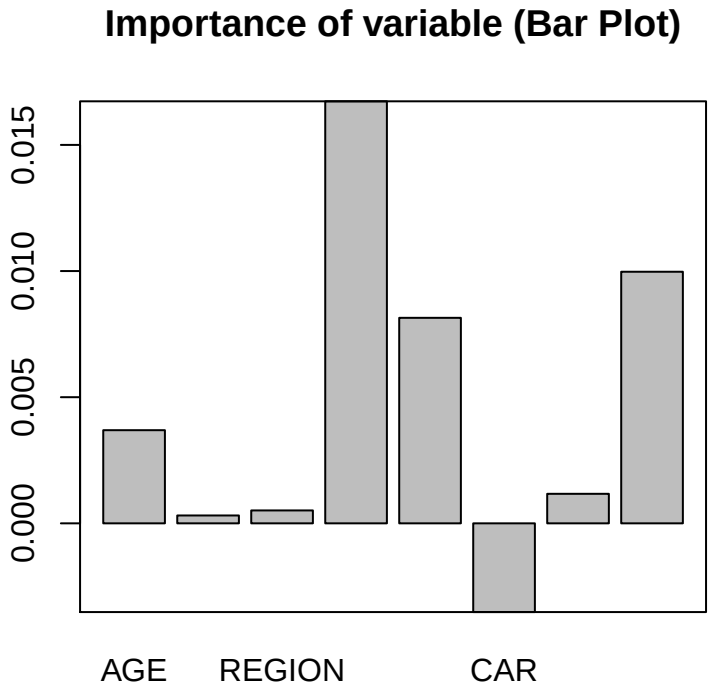
```
library("randomForest")

## Warning:  package 'randomForest' was built under R version 3.2.4
## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.
##
## Attaching package:  'randomForest'
## The following object is masked from 'package:ggplot2':
##
##    margin

MailShot<-read.table(file="MailShot.txt",header=TRUE)
MailShot<-MailShot[,-1]
set.seed(12345)
(rFM<-randomForest(MAILSHOT~.,data=MailShot,importance=TRUE))
```

```
##
## Call:
##  randomForest(formula = MAILSHOT ~ ., data = MailShot, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 44%
## Confusion matrix:
##      NO YES class.error
## NO  101  64   0.3878788
## YES  68  67   0.5037037
```

```
Fit<-predict(rFM,MailShot)
ConfM5<-table(MailShot$MAILSHOT,Fit)
(E5<-(sum(ConfM5)-sum(diag(ConfM5)))/sum(ConfM5))
```

```
## [1] 0.02666667
```

```
barplot(rFM$importance[,3],main="Importance of variable (Bar Plot)")
box()
```




```
importance(rFM,type=1)
```

```
##          MeanDecreaseAccuracy
```

```
## AGE                  1.8484166
## GENDER               0.2103645
## REGION               0.3246826
## INCOME               8.2267037
## MARRIED              4.8779590
## CAR                 -2.7974752
## SAVE                 0.9860112
## MORTGAGE             6.7435090
```

```
varImpPlot(x=rFM, sort=TRUE, n.var=nrow(rFM$importance),main="Importance of variable (scatter Plot)")
```