Tree-Based Methods Vignette

Daniel Halteh

12/13/17

Intro:

Tree-based (or Decision Tree) methods are useful statistical methods in solving either regression or classification problems. Using a stratified or segmented approach, tree-based methods are commonly implemented for their simple nature, as well as their ease of interpretability.

Tree-based methods are often used with other statistical methods, such as bagging, random forests, and boosting. These approaches implement iterating over a number of Decision Trees, and consequently using the created trees to generate some sort of variable consensus.

First, we will focus on creating a Classification Tree using the College data set to develop and implement our Decision Tree. This data set can be found within the ISLR package. This simulated data set contains various statistics from US Colleges found within the 1995 issue of US News and World Report. For our classification, we will look specifically at whether we can use certain variables to classify colleges as either 'Public' or 'Private'.

To address this sort of classification problem, we will implement a Decision Tree, which is built using a process of recursive partitioning. This process of partitioning continues until our algorithm determines the subsetted data is sufficiently homogeneous, or of the same type.

This partitioning process, often deemed "Divide and Conquer", will continue until one of the following conditions is met: - All (or nearly all) of the examples at the node have the same class - There are no remaining features to distinguish among the data - The tree has grown to a predefined size limit

Tree building in general is a considered a greedy approach in that it selects the best split at the current node, rather than looking for a certain split that will create added Purity in some arbitrary future step.

Decision Trees can be somewhat limited in that they can only perform axis-parallel splits. This makes classification problems with diagonal splits much tougher to implement.

In order to choose the best "split" at which to divide our data, we look to do so in a way that the resulting partitions contain a subset primarily of a single class.

Purity

The level of homogeneity of a certain subset can be quantified, and is typically referred to as a measure of "Purity". For example, a particular subset composed entirely of a single class is characterized as "Pure."

A widely accepted measurement of Purity introduced in Chapter 8 of ISLR is the Gini Purity Index. This Gini Purity Index measures the total variance across the target variable's k classes after a particular split.

Before proceeding with Tree-Methods, we will take a further look at the Gini Purity Index formula:

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

Here, $\hat{p}_{mk}$ represents the proportion of training observations in the mth region from the kth class. Thus, the inside portion of the formula is maximized when a particular node's purity is 0.50. In contrast, it will be minimized (thus generating a lower Gini Index coefficient) when the node's purity is 1, which can be denoted as "pure".

Further, we can use this formula to demonstrate how nodes with high levels of purity result in $\hat{p}_{mk}$ values close to either 1 or 0. In other words, a node with a small Gini Index value will contain observational data primarily from a single class k.

An alternate to the Gini Purity Index is cross-entropy, which is given by the formula:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} log\left(\hat{p}_{mk}\right)$$

$\hat{p}_{mk}$ once again represents the proportion of training observations in the mth region from the kth class. Like in formula for the Gini Purity Index, $\hat{p}_{mk}$ values close to either 1 or 0 (which indicate a fully "Pure" node) will give us a Cross-Entropy value close to 0. We can thus conclude that, in similar fashion to the Gini Index, smaller values of Cross-Entropy indicate a "Pure" mth node.

Our Data

Now, we will continue our Tree-Method discussion by exploring our college data set. As mentioned earlier, we will look specifically at whether we can use our predictors to classify colleges as either 'Public' or 'Private'.

We begin by implementing a basic Decision Tree. By focusing on the accuracy of our chosen classification method, we can look to further improve our prediction algorithms by introducing other approaches such as bagging, random forests, and boosting.

```
library(ISLR)
attach(College)
college1 <- College
```

Creating a classification plot before developing and implementing our model will allow us to visualize and understand our data more thoroughly. This can be accomplished by plotting variables of interest along the x and y (in our case, $x = Room.Board$, and $y = Grad.Rate$). Since we plan to further develop our model into a Random Forest later on, we can choose these variables somewhat arbitrarily.

However, it is important to remember to add some visual classification to our scatterplot to help us differentiate the levels of our target variable, which in our case is the 'Private' variable.

```
library(ggplot2)
ggplot(data = college1, aes(Room.Board, Grad.Rate)) + geom_point(aes(color = Private))
```

We quickly notice the increasing, upwards trend of our plot, as well as a rather distinct stratification for the two levels of our 'Private' variable. In other words, we notice that as 'Room.Board' increases, 'Grad.Rate' increases accordingly as well. For schools with higher values of 'Room.Board' and 'Grad.Rate', we see that those schools generally fall into the 'Private' category.

Further, upon looking at the upper portion of our scatterplot, we see a certain case that has a 'Grad.Rate' larger that 100%, which is clearly something that needs to be cleaned. This can be done with some quick manipulation:

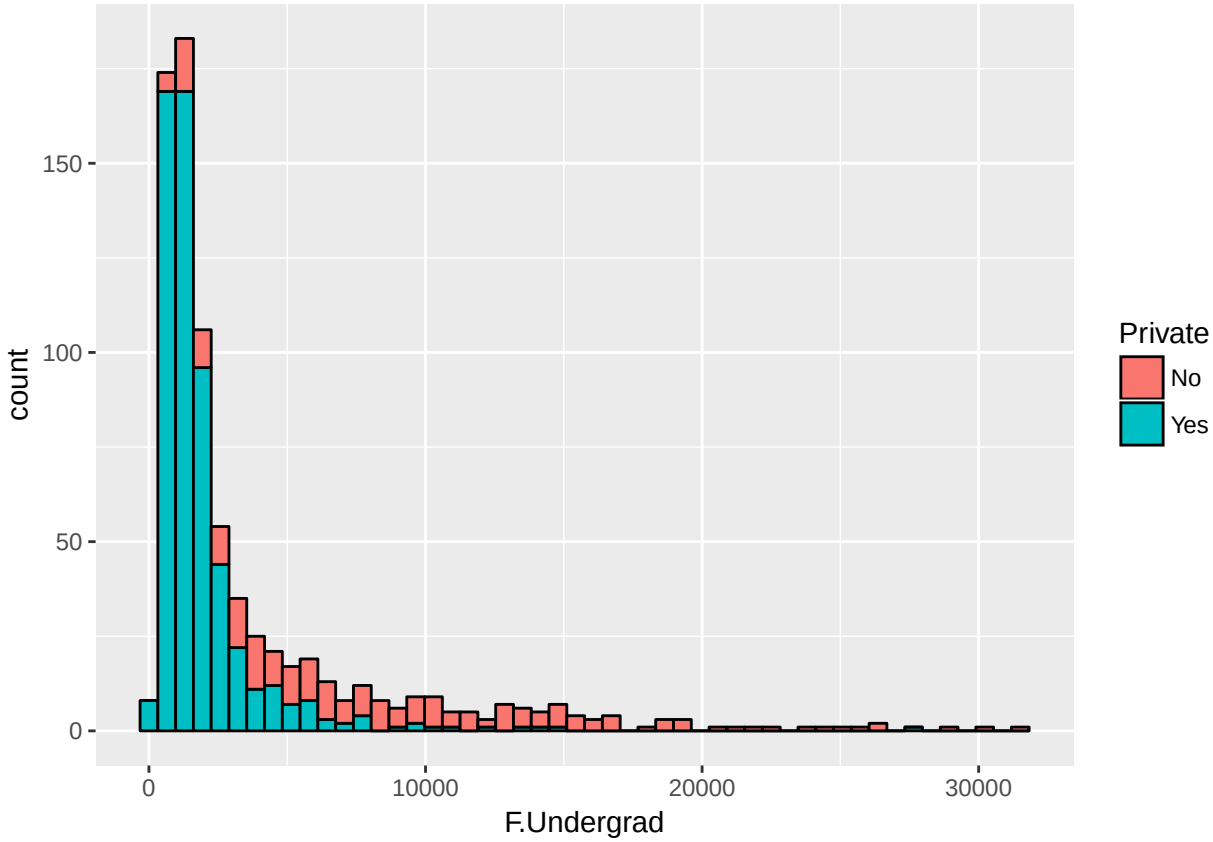
```
College[College$Grad.Rate>100,]
```

```
##                   Private Apps Accept Enroll Top10perc Top25perc
## Cazenovia College     Yes 3847   3433    527         9        35
##                   F.Undergrad P.Undergrad Outstate Room.Board Books
## Cazenovia College        1010          12     9384       4840   600
##                   Personal PhD Terminal S.F.Ratio perc.alumni Expend
## Cazenovia College      500  22       47      14.3          20   7697
##                   Grad.Rate
## Cazenovia College       118
```

```
College["Cazenovia College", "Grad.Rate"] <- 100
```

We next continue with our preliminary exploration by creating a histogram of Full-Time students (the 'F.Undergrad' variable) colored, again, by our target variable 'Private'. .

```
ggplot(data = College, aes(F.Undergrad)) + geom_histogram(aes(fill = Private), color = 'black', bins = 5
```



This plot shows that, in general, schools with a smaller number of Full-Time students tend to be private, while schools with larger Full-Time student populations tend to be public.

After completing some basic exploration, we can proceed with creating our train/test data sets. While there are many methods to accomplishing this, one such method is the sample.split() function, which can be found in the 'caTools' package. This function takes in a vector of data labels (we will use Grad.Rate as our index labels) and a train/test split ratio (we will use 0.70). An optional third argument 'group' can be implemented for vectors or lists with multiple copies of each sample.

Note: We will discuss Cross-Validation, a more effective resampling technique, in later sections. For now, we will use pseudo-random selection to generate our splits.

For reproducibility purposes, we set the seed first:

```
library(caTools)
set.seed(101)
split1 <- sample.split(college1$Grad.Rate, SplitRatio = 0.70)
```

After creating our split (stored as 'split1'), we can use our split indices to create our training and testing data sets.

```
train.college <- college1[split1 == TRUE, ]
test.college <- college1[split1 == FALSE, ]
```

Before proceeding with our Decision Tree, we should check to see if our training and testing data sets were properly created.

```
# Length should still be 18 for both
length(train.college)
```

```
## [1] 18
```

```
length(test.college)
```

```
## [1] 18
```

```
# Ratio of training to full should be approximately 0.70 as specified in the sample.split function
nrow(train.college)/(nrow(train.college) + nrow(test.college))
```

```
## [1] 0.6962677
```

Now, we can proceed by creating our Decision Tree. We will use the 'rpart' library and function to accomplish this.

The 'rpart' function takes in a number of added arguments, but we will limit our use to formula, method, and data.

Tips for using the 'rpart' function: - Rather than limiting our predictor variables as mentioned in our bagging technique, we will use the full model - Since we do have a classification task at hand, our method argument will be noted as 'class' - Lastly, make sure to build our tree based off of the training data set only

```
library(rpart)
tree1 <- rpart(Private ~ . , method = 'class', data = train.college)
```

After creating our Decision Tree model, we can now use R's predict() function to "predict" the Private label on our testing data set.

```
tree1.preds <- predict(tree1, test.college)
```

We can call view(tree1.preds) to observe our predictions. Our first observation, for example, will be classified as 'Yes' (or Private) approximately 0.9935484 % of the time.

```
View(tree1.preds)
```

However, since our aim with Decision Trees is classification, we need to mutate these columns to create a third variable called 'Private' that has two levels of classification: "Yes" and "No." We can accomplish this in multiple ways, but here we choose to implement R's sapply function.

```
tree1.preds <- as.data.frame(tree1.preds) # Convert to data frame first

p1 <- function(x){
  if(x >= 0.5){
    return("Yes")
  } else return("No")
}

tree1.preds$Private <- sapply(tree1.preds$Yes, p1)
View(tree1.preds)
```

With our Decision Tree's predictions in hand, we can now create a confusion matrix to help us better quantify and understand our results. Confusion matrices are particularly useful in comparing our predicted and actual classifications.

```
table(tree1.preds$Private, test.college$Private)
```

```
##
##        No Yes
```

```
##   No   56   9
##   Yes   9 162
```

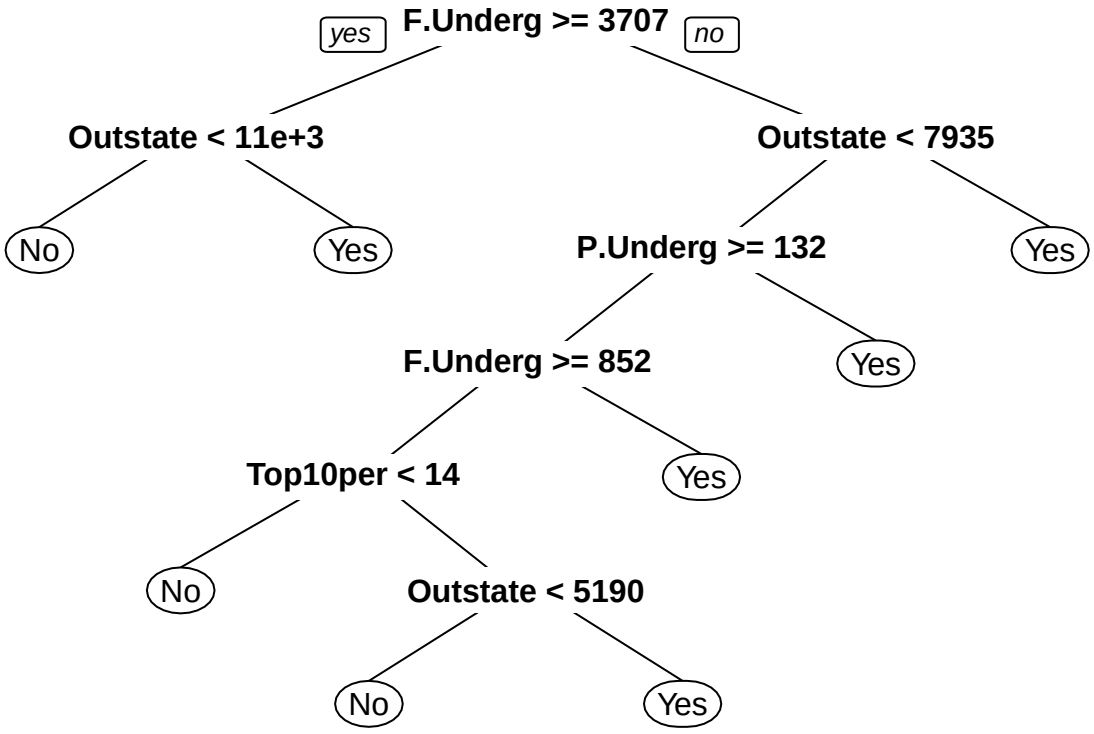
The confusion matrix plots our predicted classifications horizontally, and the actual classifications vertically. In this particular case, our accuracy rate can be calculated by adding our True No's and True Yes's and then dividing by the total number of test cases.

$(TrueNo + TrueYes)/n - - > (56 + 162)/236 = 0.9237288$

Thus, our Decision Tree's accuracy rate for classification is approximately 92.373%, giving us an error rate of about 8.9%.

We can finally plot our decision tree using the 'rpart.plot' function.

```
library(rpart.plot)
prp(tree1)
```



Bagging

After creating our initial Decision Tree, we aim to refine our model to improve our predictive accuracy. One common method used for this very purpose is Bagging, or Bootstrap Aggregation.

Before introducing how Bagging works, we should take note of our basic Decision Tree, specifically its drawbacks.

The main issue with basic trees stems simply from the their typical high variance. After splitting our college data earlier, we were left with two data sets: training and testing. Building a Decision Tree using one sample (more precisely, the entire set) of training data will result in a tree with extremely high variance. As such, applying the generated tree algorithm to our testing data will likely result in a high classification error rate.

Bagging is simply a way for us to bootstrap our training data by taking multiple samples of our single training data set. This is particularly useful when we are limited to the use of one training data set.

The bagging approach to Classification Trees allows us to develop a training model that is extracted from the "average" of each of the predictions, thus giving us the formula:

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

Although bagging is a widely used technique for improving predictive accuracy, it is particularly useful in the realm of Decision Trees. For classification purposes, we will simply implement a "majority vote" across the predicted classes for each observation across ALL of the bagged B trees.

Next, we will consider another commonly used statistical technique that implements and further improves on bagging.

Random Forests

One major issue with bagging as a stand-alone method is that the resulting trees can often be correlated.

Random Forests, unlike bagged Decision Trees, do not consider all of the predictor variables p at each split. Rather, they take a random sample m predictors from the full set p, where m is typically the square root of p, and performs a split based only on that subset of predictors.

In short, Random Forests allow us to generate a large number of "bootstrapped" trees based on random samples of the predictor variables. Note that bagging alone is done on the observational data, while Random Forests take into account the correlation of the predictors as well.

This randomization process is typically highlighted for its ability to decorrelate the generated "bagged" trees. For example, in data sets with a particularly strong predictor present, most (or all) of the trees will typically use the strong predictor in the top split. As a result, the predictions generated from the trees will be highly correlated. Simply averaging a large number of these bagged, correlated trees will not lead to a desired reduction in variance.

Random Forests allow us to circumvent the issue of correlation amongst the bagged trees by recursively randomizing the selection of predictors at each split for each tree.

From a general perspective, our earlier process of creating Decision Trees using basic "bagging" is simply a Random Forest in which our subsetted predictors m is simply our the entire set of predictors p given in the data set.

Our Data (cont):

Now, we can finally build our Random Forest model for predicting 'Private.' We will use the randomForest function from the 'randromForest' library.

We will pass in our formula as we did in our Decision Tree, the training data set, and also set importance = TRUE, which assesses and extract the importance of the predictors.

```
library(randomForest)
```

```
## 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
```

```
rfmod1 <- randomForest(Private ~ . , data = train.college, importance = TRUE)

rfmod1$confusion
```

```
##      No Yes class.error
## No  127  20  0.13605442
## Yes  12 382  0.03045685
```

After creating and implementing our model based off of our training data, we can apply it to our testing data set.

```
rf_pred <- predict(rfmod1, test.college)
table(rf_pred, test.college$Private)
```

```
##
## rf_pred  No Yes
##     No   57   4
##     Yes   8 167
```

```
# Accuracy rate
mean(rf_pred == test.college$Private)
```

```
## [1] 0.9491525
```

```
# Error rate
mean(rf_pred != test.college$Private)
```

```
## [1] 0.05084746
```

As shown above, our testing classification accuracy our Random Forest model is 0.9491525, thus giving us a classification error rate of of 5.085 %. Compared to our "bagged model" error rate of 8.9%, we do see an improvement accuracy when implementing a bootstrapped Random Forest model.

Now, we can look further into our model by analyzing the importance feature mentioned earlier.

```
rfmod1$importance
```

```
##                        No           Yes MeanDecreaseAccuracy
## Apps         0.0165003400  1.369803e-02         1.439403e-02
## Accept       0.0212330450  1.321452e-02         1.544756e-02
## Enroll       0.0483438002  2.762000e-02         3.325670e-02
## Top10perc    0.0085406554  2.980709e-03         4.481890e-03
## Top25perc    0.0018805388  2.048434e-03         1.969897e-03
## F.Undergrad  0.1322445346  6.488955e-02         8.254894e-02
## P.Undergrad  0.0436830645  8.312382e-03         1.780138e-02
## Outstate     0.1516302420  6.445772e-02         8.800382e-02
## Room.Board   0.0139428107  1.266108e-02         1.298149e-02
## Books        0.0000672939 -4.560194e-05        -5.939722e-06
## Personal     0.0019981774  1.377024e-03         1.554554e-03
## PhD          0.0102118463  7.858058e-03         8.527101e-03
## Terminal     0.0074254460  7.597226e-03         7.547733e-03
## S.F.Ratio    0.0460841989  7.524813e-03         1.808045e-02
## perc.alumni  0.0338811664  4.740616e-03         1.260400e-02
## Expend       0.0179667301  1.140183e-02         1.322195e-02
## Grad.Rate    0.0217829463  4.211563e-03         9.018736e-03
##              MeanDecreaseGini
## Apps                9.159870
## Accept             11.243412
```

```
## Enroll            22.935359
## Top10perc          3.864293
## Top25perc          2.841842
## F.Undergrad       37.818450
## P.Undergrad       14.746907
## Outstate          41.989451
## Room.Board        10.416617
## Books              2.249351
## Personal           3.689357
## PhD                5.361085
## Terminal           4.781988
## S.F.Ratio         19.218178
## perc.alumni        5.862984
## Expend             9.229722
## Grad.Rate          7.412737
```

We will look at the 'MeanDecreaseAccuracy' and 'MeanDecreaseGini' columns:

The Mean Decrease in Accuracy is extracted from our OOB, or out of bag, error calculations. Recall that Random Forests are unique in that they implement a subsetted, randomization process for selecting predictors at each split. Leaving out or permuting a particularly strong variable, for example, can decrease the accuracy of our Random Forest plot. In essence, variables with larger importance will consequently have larger 'MeanDecreaseAccuracy' values.

In our data, we can see that our most "important" variable is 'Outstate', as it has the largest 'MeanDecreaseAccuracy' of $8.800382e-02$. The variable 'F.Undergrad' is close behind with 'MeanDecreaseAccuracy' value of $8.254894e-02$.

The 'MeanDecreaseGini' column, on the other hand, measures the mean decrease in the Gini coefficient. Recall that the Gini coefficient is measure of node purity following a certain split, and is measured on a scale from $[0, 1]$, where a coefficient of 0 represents a Pure (homogeneous) node. After each variable split, the Gini coefficient is then calculated for each resulting node, and measured in comparison to coefficient of the previous node. The coefficient changes for each variable are then summed, and ultimately normalized. Variables whose splits result in an increase in Purity (smaller Gini coefficients) will thus have a larger 'MeanDecreaseGini', as their omission will result in a consequent decrease of node Purity.

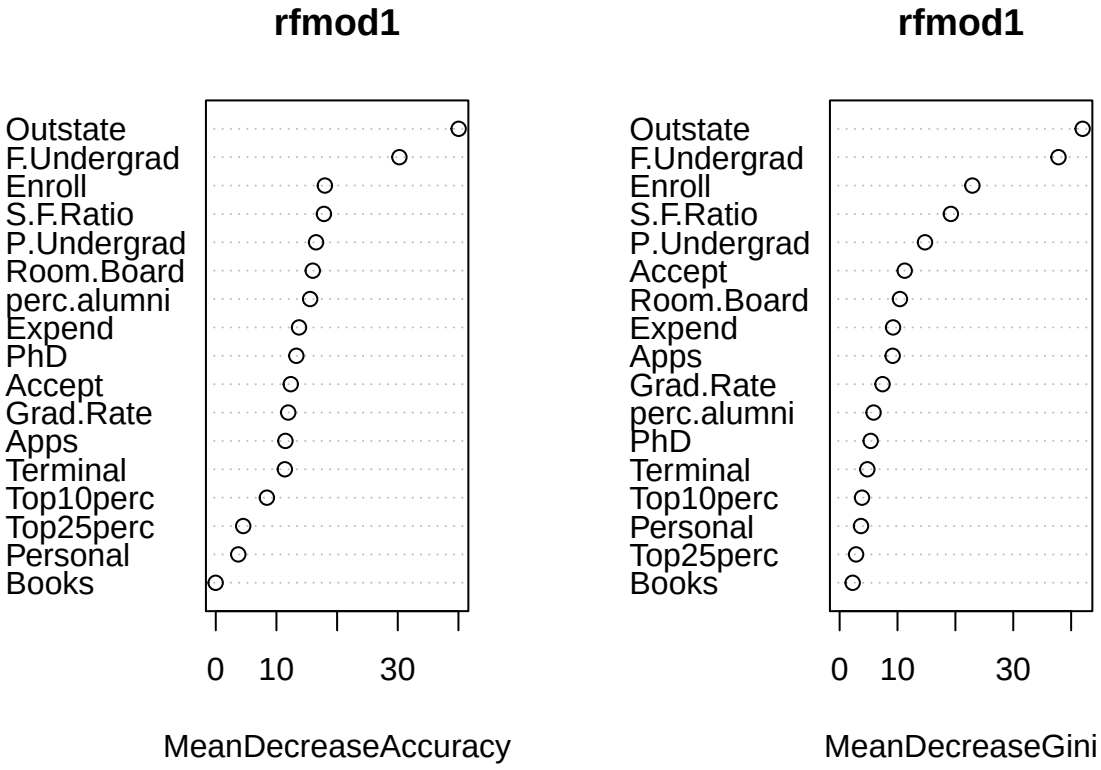
In the data above, we can see that the variable 'Outstate' most contributes to the Purity of our nodes, as it has the highest 'MeanDecreaseGini' value of 41.989451. The variable 'F.Undergrad' has the second highest value of 37.818450.

To help us visualize and assess the consistency of the 'MeanDecreaseAccuracy' and 'MeanDecreaseGini' across our predictors, we can use the 'caret' package to create variance importance plots.

```
library(caret)
```

```
## Loading required package: lattice
```

```
par(mfrow = c(1,2))
varImpPlot(rfmod1, type = 1) # Mean decrease in accuracy
varImpPlot(rfmod1, type = 2) # Mean decrease in node impurity
```



The plots above help us to reaffirm conclusions from earlier that Outstate and F.Undergrad appear the most "important" predictors in our Random Forest.

Regression Trees:

Before continuing our discussion of statistical techniques for improving predictive accuracy, let's introduce the concept of Regression Trees. Although regression problems are often approached through Linear Models, we will explore the use of Regression Trees as an alternative approach.

The process of building a Regression Tree follows similarly to that of our basic Decision Tree from earlier.

We can summarize the process as follows:

1) We begin by first dividing our predictor space, defined as the set of all possible values for $X_1, X_2, ..., X_P$, into J distinct regions, $R_1, R_2, ..., R_J$.

2) For each non-overlapping R_J, we take the mean of the resulting observational data to then generate our "prediction".

Let's assume that our division in step one leaves us with two subspaces: R_1 and R_2

The mean of the response variable for the training data in each respective subspace can be noted as $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$. For simplicity, we will call these values a and b.

The "tricky" part of step 1 is actually locating the divisions needed to create our desired subspaces. Using axis-parallel splits, we look to divide our predictor space in such a manner that minimizes our RSS, or Residual Sum of Squares:

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

As we are limited on a computational level, simply calculating the RSS for every possible partition sequence is not an option. Thus, we choose to once again apply a greedy approach to splitting, which is known as

recursive binary splitting.

Our first split in the "top-down"" process looks to find the predictor X_j and split point such that our resulting subspaces $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ will give us the largest initial decrease in RSS for the predictor.

As this is a "recursive" approach, we continue by repeating the same division process on each of the two nodes we just created. This process will essentially continue until a certain split count or change-in-RSS level is met.

Once our final subset of regions $R_1, ..., R_J$ is created, we can finally generate our predictions for the response variable by finding the mean of the observational training data to which each individual observation belongs.

Pruning a Tree:

One of the issues that may arise with our Regression Tree building is that of over-fitting. Developing an overly complex tree that precisely fits our training data will likely perform poorly on both our testing set and other future data.

Creating a smaller tree with fewer regions $\{R_J\}$ will not only decrease variance, but will give us better interpretation and improved future implementation.

This process of growing a large, complex tree, and then shortening it to obtain an optimal "sub-tree" is known as Pruning. We will implement a version of Pruning called Cost Complexity Pruning, or Weakest Link Pruning. This process allows us to consider a sequence of trees indexed by a non-negative tuning parameter α.

The tuning parameter is useful in that it counterbalances the weight of a certain subtree's complexity and tendency to overfit the training data. Minimizing the sum of the two values gives us the following formula for pruning:

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

We can see that as α increases, so does the penalty for having a large tree with excessive terminal nodes. The optimal alpha can be obtained through a cross-validation.

Boosting

The last statistical approach we will implement is boosting, which is another commonly used technique for improving predictive accuracy.

As described earlier, bagging functions by creating numerous copies of the training data through bootstrapping, fitting unique decision trees to each of the copies, and finally combining all of the bagged trees into one final model for predictions.

Boosting develops trees in similar manner, however also implementing the added feature of learning from previous trees. Unlike bootstrap sampling, boosting does not create an ultimate, variance-reduced model based off aggregating a number of bagged trees. Rather, it undergoes a process of tree modification in which each version is an "improvement" over the previous. Boosting accomplishes this by iteratively fitting our model to our current residuals, as opposed to the response variable Y.

Why modification over aggregation? Well, we can revisit our issue of overfitting to help answer that question. Our earlier processes of aggregation involved fitting large trees to our training data, and ultimately merging them in some form to generate an "ultimate" model. This process will allow for our final model to fit our current training data quite well, but also leaving a huge margin for overfitting.

By learning from the fallacies of the previous trees, boosting is able to undergo a "slow learning" approach in which targets previous cases of misclassification (or observations with high RSS for regression), increases the weights on those particular observations, and creates a modified model with those increased weights.

For regression trees, we can outline boosting as such:

1. Beginning with an empty model, we set our outputs to be the residuals of our training data.

a) $\hat{f}(x) = 0$
b) $r_i = y_i$

2. Use cross-validation to determine the number of trees B. One important aspect of boosting is that a large number of trees B will usually not lead to overfitting, given the slow-learning nature of the algorithm.

After determining our optimal B, we perform the following modification process on each tree iteratively. For $b = 1, 2, ..., B$, we:

a) Fit tree $\hat{f}^b$ with d splits to the training data set (X, r), where the output r represents the residuals of the current tree (as opposed to the response Y).

In decision trees, recall that the number of splits d (or $d + 1$ terminal nodes) determines the complexity of our boosted model. As our aim with boosting is to allow the model to learn "slowly", we choose to generally let $d = 1$, effectively creating a sequence of "stumps."

In this scenario, we are essentially creating an ensemble of trees that form a general additive model. Additive models are characterized as such due to the independence of each predictor's effect on the response. Here, the number of splits in each tree (or stumps when $d = 1$) is effectively determining speed in which each predictor variable is iteratively added to the model.

b) We then update our boosted model $\hat{f}^b$ by adding a "shrunken" version of the new tree. This shrinking process is accomplished by weighing down our current tree using the parameter λ. This shrinking parameter λ is typically assigned a value of 0.01 or 0.001, which simultaneously requires us to use a large number of iterated trees B to achieve the desired modification process.

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

c) After modifying our model using the weighted addition, we can now now use our boosted version to update the residuals. Recall that in boosting, our response variable is the residuals of our training data.

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. After iterating through each of our $b = 1, 2, ..., B$, we can finally output our boosted model.

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

Cross-Validation

In our typical classification and regression problems, we calculate our test error rate from applying a particular statistical model (usually generated from our training data) to an observation that was not included in the training data. Thus, when given a testing data set, calculating test error rate is quite straightforward. However, there are numerous instances in which a testing data set is not available for such use.

Thus, we implement a powerful resampling method know as Cross-Validation. Cross-Validation essentially functions by extracting a subset of the training data prior to fitting, and consequently applying the generated model to that extracted subset.

We will first introduce Cross-Validation for regression problems with a quantitative response variable.

The Validation Set Approach

The Validation Set Approach is an extremely simple strategy involving a random division of the training data into two sets of comparable size:

1) An updated training data set
2) A validation data set. The simple, random nature of this division makes this particular strategy quite easy to implement.

However, with simplicity also comes a pair of potential drawbacks:

1) The test error rate can be highly variable, and is often dependent on which observations were chosen to be included in the training and validation data sets, respectively.
2) Since we are subsetting our original training data to form our new data sets (both updated training and validation), we are effectively decreasing the count of training data. Decreasing the number observations for which our model trains itself will often cause our test error rate to be an overestimate of the rate for the full data set.

Leave-One-Out Cross-Validation

Leave-One-Out Cross-Validation (LOOCV) improves on the Validation Set approach by extracting a single observation (x_1, y_1) from the given training data set, and using that sole observation as the validation set. Thus, the remaining observations $(x_2, y_2), ..., (x_n, y_n)$ will form our training data.

The decision tree model will consequently be fit on the $n - 1$ training observations. Thus, given our x_1, we can predict our $\hat{y}_1$.

Given the fact that observation (x_1, y_1) was not included in our training data, the MSE for that particular observation will provide an unbiased estimate for the desired test error rate. However, basing our test error off of simply one observation will leave us with extremely high variability, given the arbitrary nature of the selected point.

To avoid issues of variability, we can repeat this process by iterating over our training data and extracting a different observation each time to represent our validation set. This approach will thus give us a sequence of n MSE's, which can be represented as such: $MSE_1, ..., MSE_n$.

We can calculate the LOOCV estimate for the test MSE by simply averaging the n MSE's as such:

$$CV_{(n)} = \frac{1}{n} \sum_{i=1}^{n} MSE_i$$

One added advantage for choosing LOOCV over the validation set approach is simply the severe decrease in bias due to the size of our training data being $n - 1$ observations. Recall that the validation set approach required us to create training and testing data sets of similar size, which would give us an updated training data set with approximately $n/2$ observations. As a result, we notice that the LOOCV does not tend to overestimate the test error rate in a similar manner to the validation set approach.

Another advantage is simply due to the lack of randomness in generating our testing/training data splits. Unlike the the validation set approach, which performs a random division on the given training data, fully iterating through our training observations to generate n MSE's will give us a consistent test error rate.

When implementing a resampling technique such as LOOCV, we need to be conscious of the potential computational cost of fitting n models (to generate n MSE's) on larger data sets. As such, performing

LOOCV on regression models such as least squares or polynomial can be similarly accomplished using the following "shortcut":

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n}(\frac{y_i - \hat{y}_i}{1 - h_i})^2$$

This formula resembles the previously depicted method of averaging the iterated MSE's, however we are dividing the residual errors by 1 - (their respective leverage). This division effectively inflates each residual by a factor that reflects the magnitude of its leverage, which is particularly useful for high-leverage points.

k-Fold Cross-Validation

One alternative to LOOCV is known as k-Fold Cross-Validation, which involves randomly dividing the training data into k groups (folds) of approximately equal size. Our first fold will essentially be our validation set, while our $k - 1$ folds will represent our testing data.

From a general perspective, we can see that LOOCV is simply a specific case of k-Fold Cross-Validation in which our k is simply n. In most applications, however, we will typically use $k = 5$ or $k = 10$. These value for k will generally allow us to avoid the computational power needed to perform LOOCV, while still still keeping our bias and variance relatively low.

Below is the formula for computing our k-Fold Cross-Validation estimate of the test error rate:

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

While both LOOCV and k-Fold Cross-Validation are commonly used techniques for estimating our test MSE, there are often instances in which the actual value of the Cross-Validation estimate is either an overestimate or underestimate of our true testing error.

During these instances, however, we will typically choose to locate the minimum point of the estimated test MSE curve. While the value itself may not be of importance, minimum's location can still allow us to compare methods based on their respective test MSE's. Furthermore, we will look to select the method that minimizes the test MSE, regardless of the test MSE's estimated value in that instance.

The Bias-Variance Trade-Off for k-Fold Cross-Validation

As mentioned in the previous section, k-Fold Cross-Validation holds an advantage over LOOCV in terms of computational efficiency and simple feasibility for larger data sets. Disregarding these aforementioned advantages, another added benefit of using k-Fold Cross-Validation is its increased level of accuracy in predicting the actual values of the test error rate. We will address this heightened level of accuracy through our discussion of the bias-variance trade-off.

From a bias-based perspective, we can quickly demonstrate that LOOCV tends to be a preferred approach. Recall that LOOCV improves upon the validation set approach by increasing our testing data from approximately $n/2$ observations for the validation set approach to $n - 1$ observations for LOOCV. Using n-1 observations essentially reflects our entire training data set, and as such, will give us results with little to no bias. k-Fold Cross-Validation greatly improves upon the validation set approach in terms of bias, but not quite to the magnitude of LOOCV.

Before choosing our resampling method of choice, however, we first need to consider the variance of our procedures.

When performing LOOCV, recall that at each iteration, our training data comprises of n-1 observations. Given the similarity of the training data upon which our models are trained, the estimated errors will be highly and positively correlated. As such, the mean of these estimates will have a higher variance.

In contrast, using k-Fold Cross-Validation (when choosing a $k < n$) will give us a set of less correlated test error rates, and consequently an estimated mean with significantly less variance.

Given this bias-variance trade-off, we will typically choose values of $k = 5$ or $k = 10$ for k-Fold Cross-Validation, as these specific values tend to generate test error rates with neither high variance, nor high variance.

Cross-Validation on Classification Problems

We will now briefly apply the Cross-Validation approach to solving classification problems, which follows quite similarly to that for regression cases.

Rather than using MSE to measure our testing error rate, we will instead choose to count the number of misclassified observations. This alternative measurement is demonstrated below:

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n} Err_i$$

, where $Err_i = I(y_i \neq \hat{y}_i)$. We can similarly use our misclassification rate to define our k-Fold Cross-Validation approach to classification problems:

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} Err_i$$

, again, where $Err_i = I(y_i \neq \hat{y}_i)$.

Application

In this section, we will apply the previously covered techniques to an application problem (question number 8 in the "Applied" section at the end of Chapter 8 of ISLR).

In the lab (covered in Chapter 8), a classification tree was applied to the Carseats data set after converting Sales into a qualitative response variable. Now we will seek to predict Sales using regression trees and related approaches, treating the response as a quantitative variable.

(a) Split the data set into a training set and a test set.

We will first load the Carseats data set, which can be found in the 'ISLR' package. We then split our data using the sample.split() funciton from the caTools library.

Upon completing our 70% split, we check to see if our data has been stratified appropriately.

```
library(ISLR)
attach(Carseats)
library(caTools)

set.seed(103)

split1 <- sample.split(Carseats$Sales, SplitRatio = 0.70)
train.Carseats <- Carseats[split1 == TRUE, ]
test.Carseats <- Carseats[split1 == FALSE, ]
```

```
nrow(train.Carseats) + nrow(test.Carseats) # Should be 400
```

```
## [1] 400
```

(b) Fit a regression tree to the training set. Plot the tree, and interpret the results. What test MSE do you obtain?

For cross-validation purposes, we will choose to use the tree library instead of rpart.

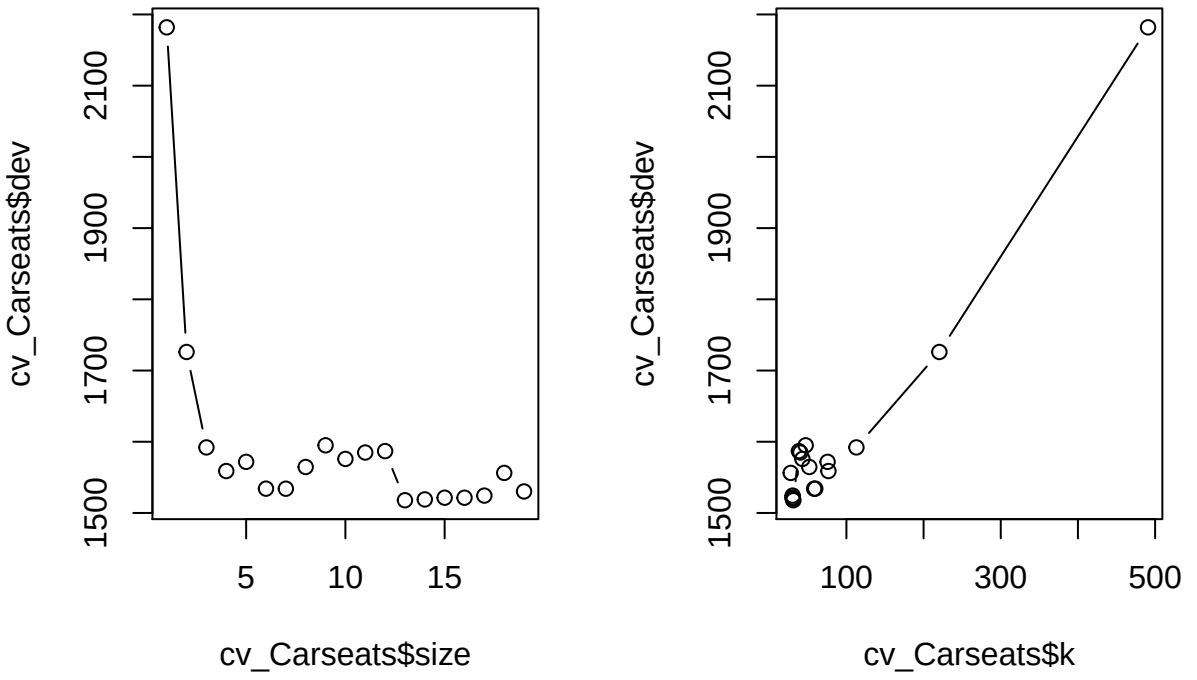
```
library(tree)
set.seed(103)
tree.Carseats <- tree(Sales ~ ., data = train.Carseats)

p1 <- predict(tree.Carseats, test.Carseats)
mean((test.Carseats$Sales - p1)^2)
```

```
## [1] 3.889145
```

Our MSE is calculated to be 3.889145.

(c) Use cross-validation in order to determine the optimal level of tree complexity. Does pruning the tree improve the test MSE?

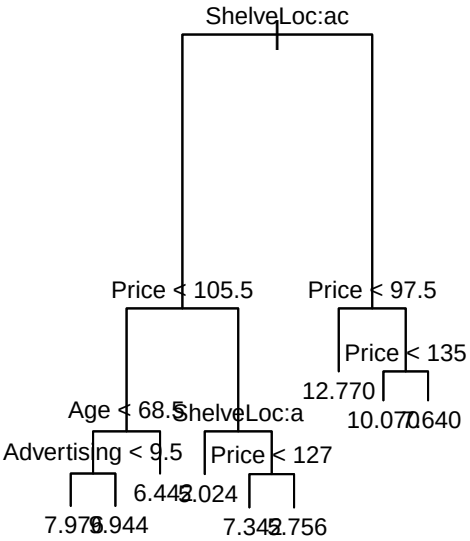
```
set.seed(103)
cv_Carseats <- cv.tree(tree.Carseats, FUN = prune.tree)

c1 <- cbind.data.frame(cv_Carseats$dev, cv_Carseats$size, cv_Carseats$k)

par(mfrow = c(1,2))
plot(cv_Carseats$size, cv_Carseats$dev, type = "b")
plot(cv_Carseats$k, cv_Carseats$dev, type = "b")
```




```
ptree_Carseats <- prune.tree(tree.Carseats, best = 9)
plot(ptree_Carseats)
text(ptree_Carseats, cex = .75)
```

```
pred2 <- predict(ptree_Carseats, test.Carseats)
mean((test.Carseats$Sales - pred2)^2)
```

```
## [1] 3.860015
```



As we can see from the plot above, our deviance is minimized when the size of our tree is 9. Thus, we will use our tree size of 9 to form our optimized "pruned" tree. After running our model using the pruned tree, we can see that our test MSE decreases from 3.889145 to 3.860015. As such, in this particular example, pruning our tree has increased our model's predictive accuracy.

(d) Use the bagging approach in order to analyze this data. What test MSE do you obtain? Use the importance() function to determine which variables are most important.

When creating our model using the bagging approach, we will essentially implement a Random Forest model that uses the entire set of predictors when considering node splits, which in this case is of size 10.

```
library(randomForest)
library(caret)

set.seed(103)
rf_Carseats <- randomForest(Sales ~ ., data = train.Carseats, importance = TRUE, mtry = 10)
# Since we want m = p for bagging
rf_pred <- predict(rf_Carseats, test.Carseats)
mean((test.Carseats$Sales - rf_pred)^2)
```

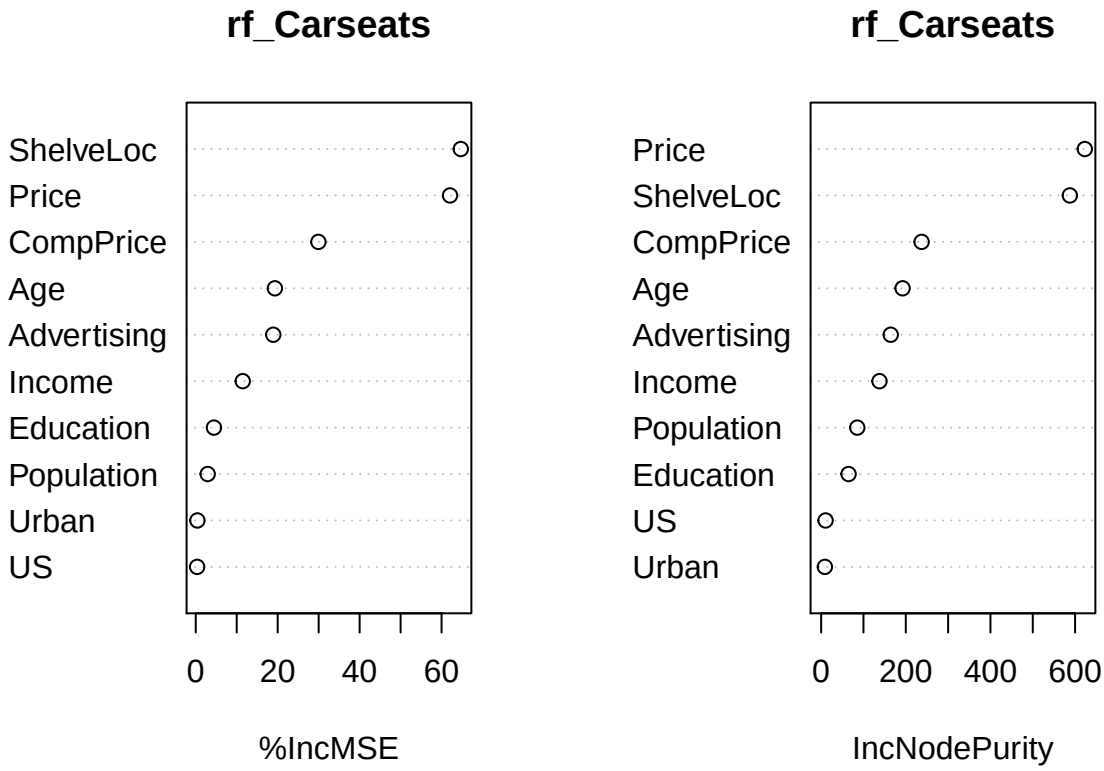
```
## [1] 2.216938
```

```
rf_Carseats$importance
```

```
##                %IncMSE IncNodePurity
## CompPrice   1.352228454    237.422010
## Income      0.279028763    137.839100
## Advertising 0.569872461    164.461714
## Population  0.051885126     85.355559
## Price       4.558166590    622.934484
## ShelveLoc   4.207597437    587.592850
## Age         0.627718696    192.450797
## Education   0.072466742     64.779750
## Urban       0.002604382      8.819821
```

```
## US          0.002815534     10.454958
```

```
par(mfrow = c(1,2))
varImpPlot(rf_Carseats, type = 1) # Mean decrease in accuracy
varImpPlot(rf_Carseats, type = 2) # Mean decrease in node impurity
```



Using the bagging approach lowers the test MSE even further to 2.216938. We can also see from our importance plots above that Price, ShelveLoc, and CompPrice are the three most important predictors of Sales.

(e) Use random forests to analyze this data. What test MSE do you obtain? Use the importance() function to determine which variables are most important. Describe the effect of m, the number of variables considered at each split, on the error rate obtained.

Recall that Random Forests are a specialized form of bagging in which we do not consider the entire set of predictors when determining our node splits. Rather, we will consider a value m, which is the square root of our full set of size p. Here we will use 3 as an integer estimate of the square root of 10.

```
set.seed(105)
rf_Carseats <- randomForest(Sales ~ ., data = train.Carseats, importance = TRUE, mtry = 3)
# Since we want m = srqt(p) for Random Forests
rf_pred <- predict(rf_Carseats, test.Carseats)
mean((test.Carseats$Sales - rf_pred)^2)
```

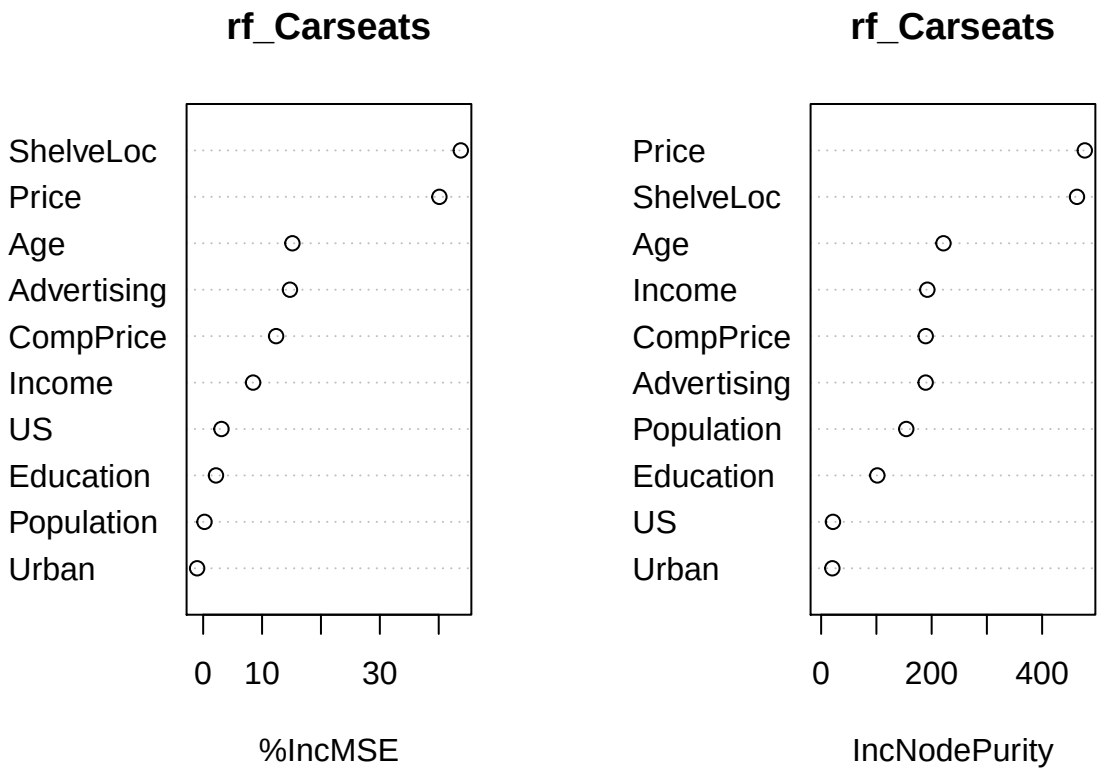
```
## [1] 2.961922
```

```
rf_Carseats$importance
```

```
##                %IncMSE IncNodePurity
## CompPrice    0.52528145     189.25246
## Income       0.31791621     192.24984
## Advertising  0.59988748     189.18118
## Population   0.00760582     154.09350
```

```
## Price        2.89842223     477.22661
## ShelveLoc    3.10934697     463.05493
## Age          0.62845224     221.38799
## Education    0.05486777     101.68655
## Urban       -0.01319342      20.30183
## US           0.04676847      21.37686
```

```r
par(mfrow = c(1,2))
varImpPlot(rf_Carseats, type = 1) # Mean decrease in accuracy
varImpPlot(rf_Carseats, type = 2) # Mean decrease in node impurity
```



Although the Random Forest approach is expected to perform better than bagging due to the former model's trees being uncorrelated, our MSE in this example jumped up to 2.986438. In this particular instance,

Price and ShelveLoc appear to once again be the two most important predictors of Sales. However, the increase in MSE and Node Purity plots have CompPrice and Age as their third most important predictors, respectively.

Citations

Hastie, Trevor, Tibshirani, Robert and Friedman, Jerome. The Elements of Statistical Learning. New York, NY, USA: Springer New York Inc., 2001.