Lecture 7: Classifiers I

Intro to Data Science for Public Policy, Spring 2016

by Jeff Chen & Dan Hammer, Georgetown University McCourt School of Public Policy

Contents

Introduction

According to the American Community Survey (ACS), an annual survey of approximately 3.5% of the US population as conducted by the US Census Bureau, over 16.5% of residents of the U.S. state of Georgia were without healthcare coverage in 2015. To some degree, this is [un]surprising. While the statistic is informative, it is not actionable as more specifics are required. How can a policy or program close the gap?

In order to close the gap, the data needs to enable the prediction and classification of a population into two classes: covered and not covered This binary problem or membership problem is known as a classification problem. By correctly classifying a population as covered and not covered, decision makers and outreach staff can mobilize targeted outreach. From a data science perspective, the real objective is to be able to identify and replicate re-occurring patterns in the training data, then generalize the insights onto a sample or population that is not contained in the sample. In other words, the production of actionable data insights means that the results of data analysis and modeling is worth more than a sound bite, but can be applied and deployed directly to solve a problem.

In most policy environments, a data analyst will typically manually select population characteristics to use in cross tabulations to find statistical patterns; however, this tradtional approach can suffer from human bias that may yield misleading results. Some features may be more important than others, and humans usually do not systematically check all features. For example, the table below compares healthcare coverage and citizenship. Each of the cells are quite interpretable: 49.4% of non-citizens are without coverage, but that sub-population is only 3.5% of the population.

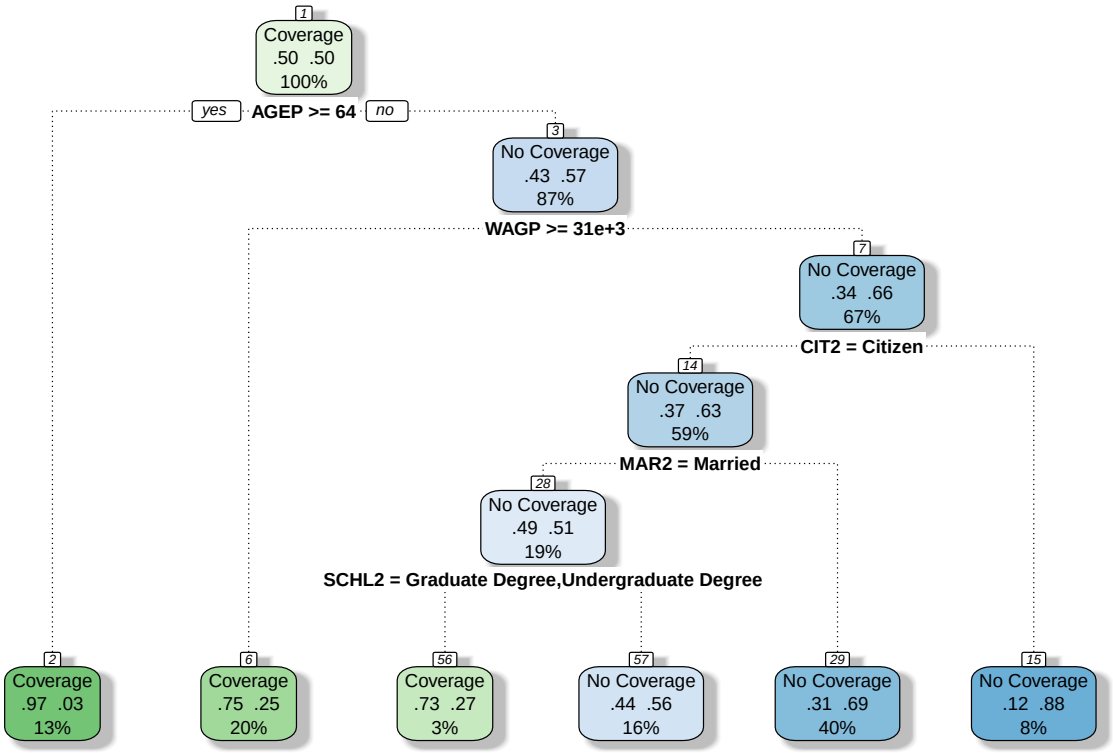
	Coverage	Without coverage	% Without coverage
Citizen	6,387,690	1,051,815	14.1%
Non-citizen	276,192	282,769	49.4%
All	6,663,882	1,334,584	16.6%
% Non-citizen	4.1%	21.1%	3.5%

A cross-tabulation does not, however, provide sufficient predictive power. Expanding the table to include more features such as age, gender, wages, etc. may not improve inference either as this will lead to "analysis paralysis".

Supervised learning offers a number of options to more efficiently identify patterns surface important variables, and predict membership. Given the label $Y(Coverage)$, we will use supervised learning

techniques to determine how to split the survey sample into smaller cells using the following features: Sex, Age, Education, Marital Status, Race, Citizenship).

Below is a visualization of the results, which are interpretable as well as defined by discrete cells of Georgians who have and do not have healthcare coverage. For example, non-citizens under the age of 64 without a college education are 90% likely to not have coverage, and citizens between 16 and 64 who are not married have a 71% chance of not having health coverage.

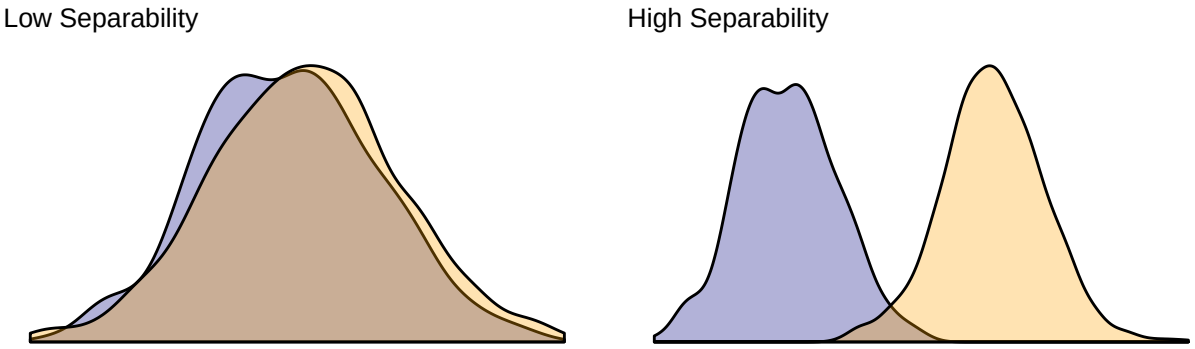


The above diagram was produced using a decision tree algorithm, which is one of many forms of supervised learning known as *classifiers* or *classification algorithms*. Classifiers take on many forms. Some use recursive partitioning to break a population into many, more homogeneous subpopulations. Others estimate a series of equations to fit a line or plane between two or more classes. Others will average the results of an ensemble of models to predict membership. Each class of model is defined with mathematical scenarios in mind. This chapter is organized into three sections. Section 1 describes common considerations in classification models. Section 2 provides an overview of a number of classifiers, including kNN, decision trees, and random forests and ensemble methods.

Section 1 - An Overview

What goes into a classifier?

Classifiers are a type of supervised learning problem that handles target features that contain discrete labels, otherwise known as classes. Using the example above, having and not having health insurance would be two classes. Being part of Generations X, Y, and Z would be three classes. In short, classification algorithms accept a discrete target feature and input features to maximize *separability* between two or more classes – or the ability to clearly distinguish one group from another based on associated characteristics. A low separability scenario, for example, would be one where the distributions of two classes substantially overlap, whereas a high separability case would have little overlap. The output of a classification algorithm is a probability that indicates how likely a given record belongs to a target class given the input features.



The output probability is the key to evaluating the accuracy of a model. Unlike regression, classifiers rely on entirely different measures of accuracy given the nature of the labeled data. All measures, however, rely on metrics that can be derived from a confusion matrix, or a 2×2 table where the rows typically represent actual classes and columns represent predicted classes.

	Predicted (+)	Predicted (-)
Actual (+)	True Positive (TP)	False Negative (FN)
Actual (-)	False Positive (FP)	True Negative (TN)

Each of the cells contains the building blocks of accuracy measures:

- The True Positive (TP) is the count of all cases where the actual positive ($Y = 1$) case is accurately predicted.
- The True Negative (TN) is the count of all cases where the actual positive ($Y = 0$) case is accurately predicted.
- The False Positive (FP) is count of all cases where the actual label was $Y = 0$, but the model classified a record as $\hat{Y} = 1$. This is also known as Type I error.
- The False Negative (FN) is count of all cases where the actual label was $Y = 1$, but the model classified a record as $\hat{Y} = 0$. This is also known as Type II error.

Accuracy. Overall accuracy is measured as the sum of the main diagonal divided by the population (below).

$$TPR = \frac{TP + TN}{TP + FN + FP + TN}$$

True Negative Rate. By combining TN and FP, we can calculate the True Negative Rate (TPR), which is proportion of $Y = 0$ cases that are accurately predicted. TNR is also referred to as the "specificity".

$$TNR = \frac{TN}{TN + FP} = \frac{TN}{Actual(-)}$$

True Positive Rate. By combining TP and FN, we can calculate the True Positive Rate (TPR), which is proportion of $Y = 1$ cases that are accurately predicted. TPR is also referred to as the "sensitivity" or "recall".

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{Actual(+)}$$

Positive Predicted Value. By combining TP and FP, we can calculate the Positive Predicted Value (PPV), which is proportion of predicted $Y = 1$ cases that actually are of tht class. PPV is also referred to as "precision".

$$PPV = \frac{TP}{TP + FP}$$

What does accuracy look like? To illustrate this, the next series of tables provides simulated results of a classifer. Let's assume that a health insurance classifier was tested on a sample of $n = 100$ with actual labels perfectly split between $Y = 1$ and $Y = 0$. A perfect performing model would resemble the following table, where TP = 50 and FP = 50. With perfect predictions with $Accuracy = \frac{50+50}{100} = 100$, the $TPR = \frac{50}{50+0} = 100$ and $PPV = \frac{50}{50+0} = 100$ indicate that model is perfectly balanced and precise.

	Predicted (+)	Predicted (-)
Actual (+)	50	0
Actual (-)	0	50

A model with little discriminant power or ability to distinguish between classes would look like the following. While the $TPR = \frac{35}{35+5} = 87.5$ is high, overall $Accuracy = \frac{35+0}{100} = 45$, which is largely driven by low precision $PPV = \frac{35}{35+60} = 36.8$.

	Predicted (+)	Predicted (-)
Actual (+)	35	5
Actual (-)	60	0

While these calculations are simple and understandable, determining the predicted label is not as simple. In a simple case, given an outcome $Y = 1$, a voting rule would classify a probability of greater than 50% as $Y = 1$. However, it is fairly common that a trained classifier with strong performance may never produce a probability of more than 50%. In order to generalize accuracy, we can rely on one or a combination of the following measures.

Measure	Description	Interpretation
Receiving Operating Characteristic (ROC) Curve	ROC curves plotpairs of TPRs and FPRs that correspond to varied discriminant thresholds between 0 and 1. By systematically testing thresholds. For example, TPRs and FPRs are calculated and plotted given probability thresholds $p = 0.2$, $p = 0.5$, and $p = 0.8$.	Once plotting the curve with TPR as Y and FPR as X, the area under the curve (AUC) represents robustness of the model, ranging from 0.5 (model is as good as a coin toss) to 1.0 (perfectly robust model). In the social sciences, an acceptable AUC is over 0.8.The AUC statistic is sometimes referred to as the "concordance".
F_1 Score	The score is formulated as $F_1 = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{PPV \times TPR}{PPV + TPR}$ where precision or PPV $= \frac{TP}{TP+FP}$ and recall or TPR $= \frac{TP}{TP+FN}$	The measure is bound between 0 and 1, where 1 is the top score indicating a better model.

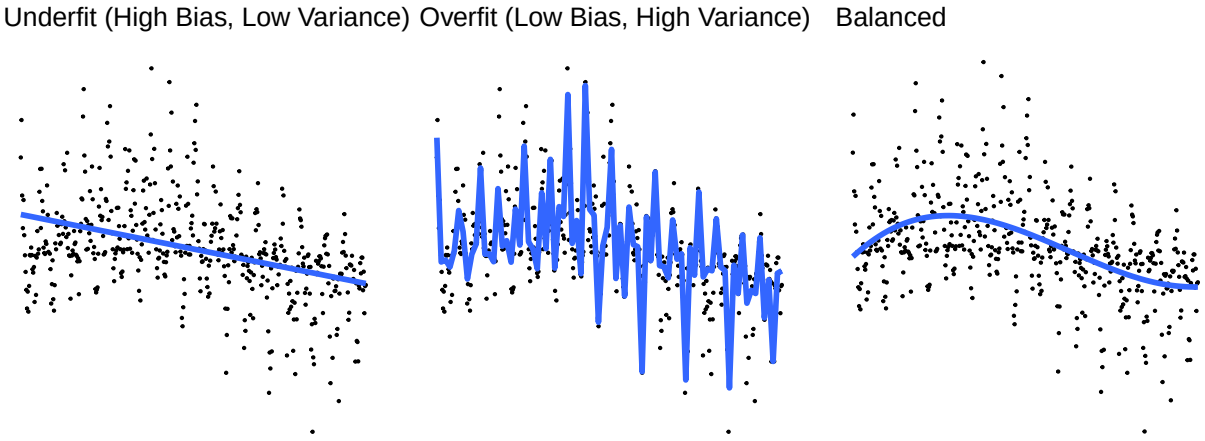
Bias & variance

Humans tend towards simple and seemingly elegant explanations of phenomena that allow for comfortably generalizing insights across populations. The simplicity is favored when generalizing insights as it is more

accessible to more people. Generalizations sometimes do not, however, accurately describe a phenomenon. Thus, as humans seek to improve theories and representations of phenomena, explanations become increasingly complex to account for progressively more intricate patterns based on observation. As patterns become more complex, humans will tend to account for edge cases, but these patterns may simply be noise. This balance between generalizability and accuracy is the basis of the *Bias-Variance Tradeoff*.

To illustrate this tradeoff, the three graphs below exemplify the two extremes and the balanced case.

- **Underfitting**. Starting from the left, an overly simplistic explanation might boil down a relationship too much to the point that the model is "underfitted" – that the true shape of the relationship is not captured and the model is not responsible when generalized to new data. The variance of predictions in the training set is likely to be high and the same tends to be true for the test set. Models that have high bias (overly simplfified) tend to tell a digestable story, but have little predictive value.
- **Overfitting**. The middle graph shows a low bias model, which is often a far more complex model with more features and polynomials to improve the model fit. At first glance, captures much of the variability in the point-spread. While low bias models tend to have low bias in the training step, their predictions in out-of-sample testing tend to have high varaince as the model was calibrated to noise and is thus extra sensitive to noise.
- **Goldilocks**. The last graph shows a good balance between the two extremes: Just enough complexity to capture the curvature, but not so much to mold to every crevice of the data.



As we move through this chapter, the bias-variance tradeoff is a critical consideration. It informs how models are formulated and deployed.

Section 2 - Methods

In the context of healthcare coverage, we will use KNNs to illustrate the process of training a classifier. With the practical aspects in mind, we will explore two types of tree-based learning, namely decision trees and random forests. Then wrap up with logistic regression and a comparison of the performance of each of the four classifiers.

To start, we will need to import data from the healthcare coverage example in Section 1. The data was obtained from the 2015 American Community Survey (ACS), which is available from US Census Bureau website. So that this chapter can focus more on classification methods, data has been lightly pre-processed, and any data wrangling that is shown herein is specific to the method. For more information on pre-processing, review the RMarkdown processing file the Lecture 7 repository.

We now can import the ACS file and examine its structure. The file contains a total of eight features, one of which is the target (`coverage`).

```r
health <- read.csv("data/lecture7.csv")
str(health)
```

```
## 'data.frame':    22354 obs. of  7 variables:
##  $ coverage: Factor w/ 2 levels "Coverage","No Coverage": 2 2 2 2 2 2 2 2 2 2 ...
##  $ age     : int  33 22 28 25 42 51 36 26 55 23 ...
##  $ wage    : int  0 0 0 0 50000 0 0 0 37000 9400 ...
##  $ cit     : Factor w/ 2 levels "Citizen","Non-citizen": 1 1 1 1 1 1 1 1 1 1 ...
##  $ mar     : Factor w/ 5 levels "Divorced","Married",..: 2 2 3 2 2 3 3 3 1 3 ...
##  $ educ    : Factor w/ 4 levels "Graduate Degree",..: 2 2 2 3 2 2 2 3 3 2 ...
##  $ race    : Factor w/ 8 levels "Amer. Ind.","Asian",..: 8 8 8 8 8 3 8 3 8 3 ...
```

K-Nearest Neighbors (Discrete Case)

As covered in Lecture 6, KNNs are a weak learning algorithm that treats input features as coordinate sets. Given a class label y associated with input features x, a given record i in a dataset can be related to all other records using Euclidean distances in terms of x:

$$\text{distance} = \sqrt{\sum (x_{ij} - x_{0j})^2}$$

where j is an index of features in x and i is an index of records (observations). For each i, a neighborhood of taking the k records with the shortest distance to that point i. From that neighborhood, the value of y can be approximated. Given a discrete target variables, y_i is determined using a procedure called *majority voting* where the most prevalent value in the neighborhood around i is assigned.

Recall that in the case of KNNs, all variables should be in the same scale such that each input feature has equal weight. A review of the data indicates that the health data is not in the appropriate form to be used.

Data preparation: Mixed variable formats

The continuous variable can be discretized by first binning records at equal intervals. For simplicity, we'll bin the age and wage varaibles in the following manner:

- `age`: 10 year intervals.
- `wage`: $20,000 intervals, topcoded at $200,000.

Upon binning, each variable needs to be set as a factor.

```r
#Age
  age <- round(health$age / 10) * 10
  age <- factor(age)

#Wage
  wage <- round(health$wage / 20000) * 20000
  wage[wage > 200000] <- 200000
  wage <- factor(wage)
```

For all discrete features including the newly added `age` and `wage` variables, we can convert them into dummy matrices (e.g. all except one level in a discrete feature is converted into a binary variable). The former can be easily achieved by using the `model.matrix()` method, which returns a binary matrix for all levels:

```r
  model.matrix(~health$variable - 1)
```

As is proper in preparation of dummy variables, if there are k levels in a given discrete variable, we should only keep $k - 1$ dummy variables For example, citizenship is a two level variable, thus we only need to keep

one of two dummies. It's common to leave out the level with the most records, but any consistently appearing level will do.

```
#Keep only column 2
  cit <- as.data.frame(model.matrix(~ health$cit - 1)[, 2])

#Keep columns 1 to 4, leave out column 5
  mar <- as.data.frame(model.matrix(~ health$mar - 1)[,1:4])

#Keep columns 1 to 4, leave out column 5
  educ <- as.data.frame(model.matrix(~ health$mar - 1)[,1:4])

#Keep columns 1 to 7, leave out column 8
  race <- as.data.frame(model.matrix(~ health$race - 1)[,1:7])

#Keep columns 2 to 11, leave out column 1
  wage <- as.data.frame(model.matrix(~ wage - 1)[,2:11])

#Keep columns 2 to 8, leave out column 1
  age <- as.data.frame(model.matrix(~ age - 1)[,2:8])
```

Now the data can be combined. Notice that the new dataset `knn.data` has 34 features. Compared with the `health` dataset's 7 features. Note that perform these transformations are necessary given mixed variable types; however, a datasets containing continuous variables only does not require any manipulation other than scaling.

```
#Combine all the newly transformed data
  knn.data <- as.data.frame(cbind(coverage = as.character(health$coverage),
                                  wage, age, cit, educ, mar, race))

#Dimensions
  dim(health)
```

```
## [1] 22354     7
```

```
  dim(knn.data)
```

```
## [1] 22354    34
```

Sample partition

As is proper, the next step is to partition the data. For simplicity, we'll create a vector that will split the data into two halves, denoting the training set as `TRUE` and the test set as `FALSE`. We then split the data into four objects:

- Two objects contain the input features for each train and test sets.
- Two objects contain the labels for each train and test sets.

Splitting into four objects is common practice. `ytrain` and `xtrain` are used to train the model. Then, `xtest` is used to produce predictions. Then, `ytest` is used to calculate accuracy of the predictions.

```
#Train-Test
  rand <- runif(nrow(knn.data))
  rand <- rand > 0.5

#Create x-matrix. Use "-1" in the column argument to keep everything except column 1
  xtrain <- knn.data[rand == T, -1]
  xtest <- knn.data[rand == F, -1]
```

```r
#Create y-matrix
  ytrain <- knn.data[rand == T, 1]
  ytest <- knn.data[rand == F, 1]
```

Model

Now the data is ready, the process is fairly simple. The supervised learning library "class" needs to be called to be able to take advantage of the `knn()` method.

```r
#Call "class" library
  library(class)
```

The `knn()` syntax is simple, only requiring a few parameters:

- training features: `xtrain`
- test features: `xtest`
- training labels: `cl`
- number of neighbors: `k`
- return probability?: `prob`

In this trial run, we set $k = 10$, meaning the 10 nearest neighbors in euclidean distance are used to predict a given observation's value. The method returns an object containing the predicted labels of the test set. If `prob = TRUE`, the object also contains the probabilities, which can be accessed by using the `attr()` method.

```r
#Run model
  pred <- knn(xtrain, xtest, cl = ytrain, k = 10, prob = TRUE)

#Check model object
  str(pred)
```

```
##  Factor w/ 2 levels "Coverage","No Coverage": 2 2 2 1 2 2 2 2 2 1 ...
##  - attr(*, "prob")= num [1:11309] 0.698 0.689 0.851 0.718 0.806 ...
```

```r
#Extract probabilities
  test.prob <- attr(pred, "prob")
```

Using the extracted probabilities, we now can calculate the accuracy using the True Positive Rate (TPR) using a probability cutoff of 0.5. Typically, one would expect a 2×2 matrix given a binary label; However, our confusion matrix is a 1×2 matrix with a TPR of 49.9%, indicating that the model does not perform well at the $p = 0.5$ cutoff does not yield promising results. The selected cutoff is fairly arbitrary and one cutoff does not necessarily provide an indication of the model's robustness [or lack thereof].

```r
#TPR
  pred.class <- test.prob
  pred.class[test.prob >= 0.5] <- "Coverage"
  pred.class[test.prob < 0.5] <- "No Coverage"

#Confusion matrix
  table(ytest, pred.class)
```

```
##              pred.class
## ytest         Coverage
##   Coverage        5669
##   No Coverage     5640
```

To more robustly test the model, we will rely on the Receiving-Operating Characteristic (ROC) Curve and the Area Under the Curve (AUC). The ROC calculates the TPR and FPR at many thresholds, that produces

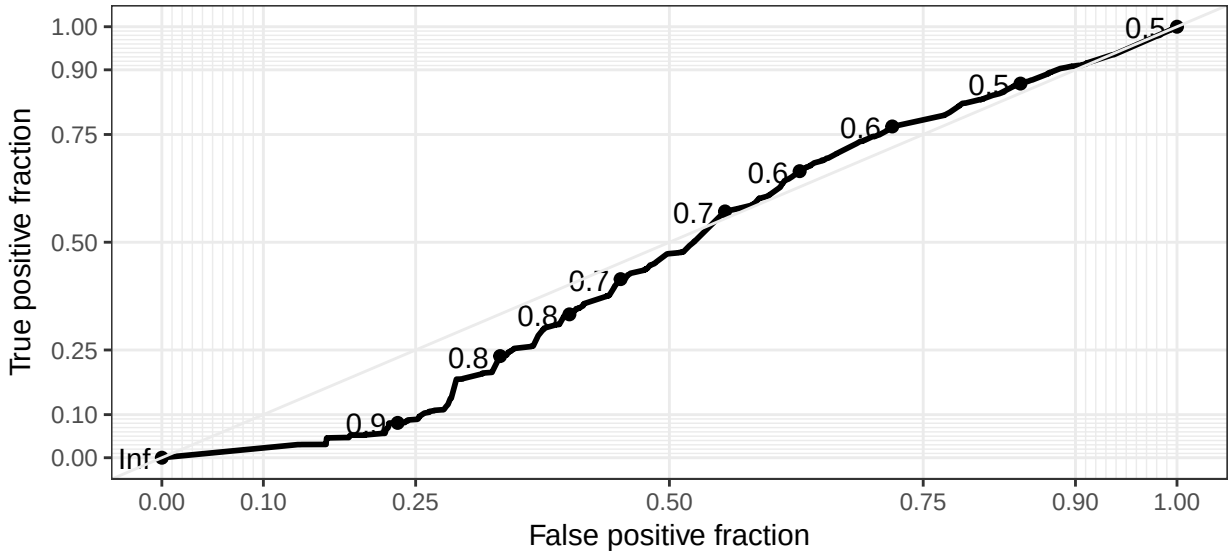
a curve that indicates the general robustness of a model. The AUC is literally the area under that curve, which is a measure between 0.5 and 1 where the former indicates no predictive power and 1.0 indicates a perfect model.

To assess accuracy, we will install and load the `devtools` and `plotROC` libraries. The former allows for loading libraries available via Github and the latter is an extension of `ggplot2` that calculates accuracy and graphs ROC plots. In order to calculate the ROC, we will create a new data frame `input` that is comprised of the labels for the test set `ytest` and the predicted probabilities `test.prob`.

```
#Load libraries
  #install.packages("devtools")
  #devtools::install_github("sachsmc/plotROC")
  library(ggplot2)
  library(plotROC)

#Set up test data frame
  input <- data.frame(ytest = ytest, prob = test.prob)
```

We then will first create a ggplot object named `base` that will contain the labels (`d =`) and probabilities (`m =`), then create the ROC plot using `geom_roc()` and `style_roc()`. A ROC curve for a well-performing model should sit well-above the the 45 degree diagonal line, which is the reference for an AUC of 0.5 (the minimum expected for a positive predictor). However, as the curve is below the 45 degree line, we may have a seriously deficient model.

```
#Base object
  roc <- ggplot(input, aes(d = ytest, m = prob)) + geom_roc() + style_roc()

#Show result
  roc
```



To calculate the AUC, we can use the `calc_auc()` method, from which we find that $AUC = 0.453$, which is below the minimum acceptable values of AUC. This begs the question: Is the data bad or can we find a better classification algorithm? To answer that, we'll give decision tree learning a try.

```
  calc_auc(roc)$AUC
```

```
## [1] 0.4639484
```

Decision Trees

In everyday policy setting and operations, decision trees are a common tool used for communicating complex processes, whether for how an actor moves through intricate and convoluted bureaucracy or how a sub-population can be described based on a set of criteria. While the garden variety decision tree can be laid out qualitatively, supervised learning allows decision trees to be created in an empirical fashion that not only have the power to aesthetically communicate patterns, but also predict how a non-linear system behaves.

The structure of a decision tree can be likened to branches of a tree: moving from the base of the tree upwards, the tree trunk splits into two or more large branches, which then in turn split into even smaller branches, eventually reaching even small twigs with leaves. Given a labeled set of data that contains input features, the branches of a decision tree is grown by subsetting a population into smaller, more homogeneous units. In other words, moving from the root of the tree to the terminating branches, each subsequent set of branches should contain records that are more similar, more homogeneous or purer.

```
1. Let Sample = S, Target = Y, Input Features = X
2. Screen records for cases that meet termination criteria.
      If each base case that is met, partition sample to isolate homogeneous cases.
3. For each X:
      Calculate the attribute test comparing all X's and Y
4. Compare and identify Xi that yields the greatest separability
5. Split S using input feature that maximizes separability
6. Iterate process on steps 3 through 5 until termination criteria is met
```

As was demonstrated at the beginning of this chapter, decision trees use a form of recursive partitioning to learn patterns, doing so using central concepts of *information theory*. There are a number of decision tree algorithms that were invented largely in the 1980s and 1990s, including the ID3 algorithm, C4.5 algorithm, and Classification And Regression Trees for Machine Learning (CART). All these algorithms follow the same framework that includes the following elements: (1) nodes and edges, (2) attribute tests, and (3) termination criteria.

(1) Nodes + Edges

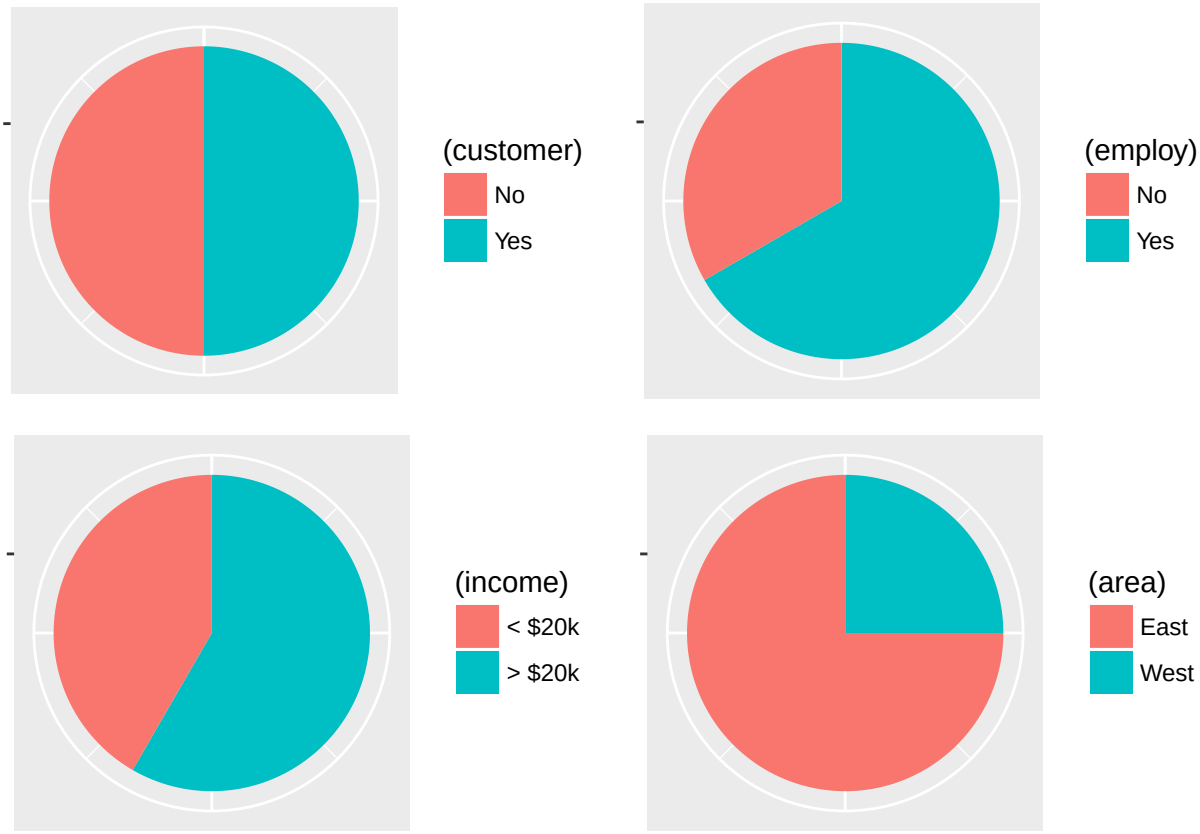
Recalling the healthcare insurance decision tree, the tree can be characterized by nodes and edges.

- Nodes (circles) contain records.
- Edges (lines) show dependency between nodes and is the product of a split decision. Nodes are split based on an attribute test – a technique to identify the optimal criterion to subset records into more homogeneous groups of the target variable.
- The node at the top of the tree is known as the *root*and represents the full population.
- Each time a node is split, the result is two nodes – each of which is referred to as a child node. A node without any child nodes is known as a leaf.

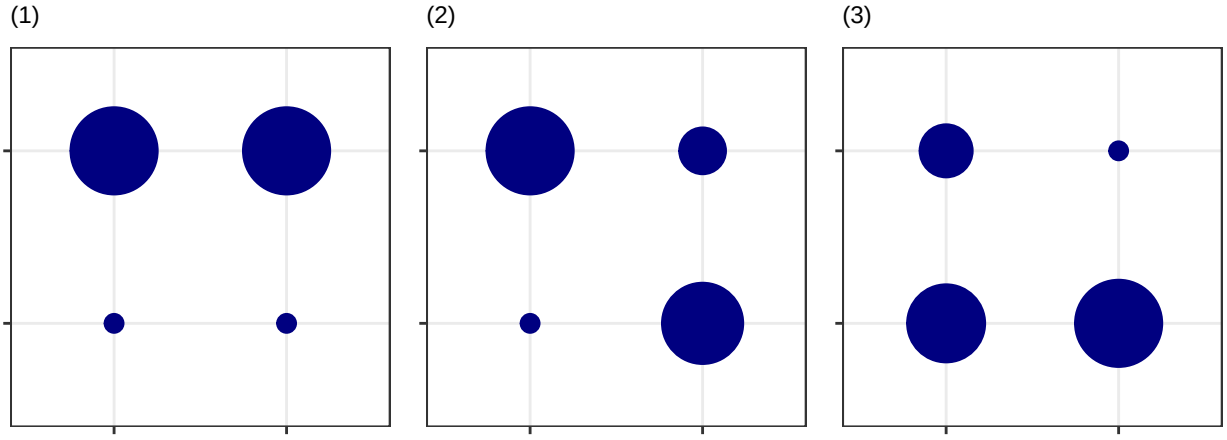
The goal is to grow a tree from the root node into as many smaller, more homogeneous child nodes with respect to the target variable.

(2) Attribute tests

To understand attribute tests means to have a thorough understanding of separability. Let's suppose we have a list of residents of a town. The list contains both users and non-users of a given healthcare service. For each person, the inventory captures whether a given person is employed, has income over $20k, and lives on the west side or east side of town. Each of the features are plotted in the pie chart below. 50% of town residents use the health service, but which of the features is best at separating users from non-users?



To answer that question, we can rely on a visual cross-tabulation where the size of the circles is scaled proportional to the number of records. The objective is to identify the matrix where the circles are the largest along any diagonal – this would indicate that given usership, a feature is able to serve as a criterion that separates users from non-users. Of the three graphs below, graph #2 is able to separate a relatively large proportion of users from non-users. For a relatively low-dimensional dataset (fewer attributes), a visual analysis is accomplishable. However, on scale, undertaking this process manually may be onerous and prone to error.



Enter attribute tests.

Decision trees are grown by splitting a data set into many smaller samples. Attribute tests are the mode of finding the split criterion, following an empirical process to systematically test all input features to find the feature with the greatest separability. The process starts from the root node where the algorithm examines each input feature to find the one that maximizes separability at that node:

```
Let Sample = S, Target = Y, Input Features = X
    For each X:
        Calculate the attribute test statistic comparing X and Y
        Store statistic
    Compare and identify Xi that yields the greatest separability
    Split S using input feature that maximizes separability
    Iterate process on child node
```

Upon finding the optimal feature for a given node, the decision tree algorithm splits the node into two child nodes based on the optimal feature, then moves onto the next node (often times a child node) and runs the same process to find the next split. There are a number of attribute tests, of which we will cover two: *Information Gain* and *Gini Impurity*.

Information gain is a form of *Entropy*, which is a measure of purity of information. Based on these distinct states of activity, entropy is defined as:

$$\text{Entropy} = \sum -p_i log_2(p_i)$$

where i is an index of states, p is the proportion of observations that are in state i, and $log_2(p_i)$ is the Base 2 logarithm of the proportion for state i. Information Gain (IG) is variant of entropy, which is the entropy of the root node *less* the average entropies of the child nodes.

$$\text{IG} = \text{Entropy}_{\text{root}} - \text{Avg Child Entropy}$$

How does this work in practice? Starting from the root node, we need to calculate the root entropy, where the classes are based on the classes of the target `usership`.

$$\begin{aligned}\text{Entropy}_{\text{usership}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= (-\tfrac{6}{12} log_2(\tfrac{6}{12})) + (-\tfrac{6}{12} log_2(\tfrac{6}{12})) \\ &= 1.0\end{aligned}$$

Then, the attribute test is applied to the root node by calculating the weighted entropy for each proposed child node. Using the `income` feature, the calculation is as follows:

- Split the root node into two child nodes using the `income` class. This yields the following subsamples as shown in the table below:

	$<$ \$20k	$>$ \$20k
No	0	6
Yes	5	1
Total	5	7

- For each child node (the columns in the table), calculate entropy:

$$\begin{aligned}\text{Entropy}_{\text{income} < 20\text{k}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= -\tfrac{5}{5} log_2(\tfrac{5}{5}) = 0\end{aligned}$$

$$\begin{aligned}\text{Entropy}_{\text{income} < 20\text{k}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= -\tfrac{6}{7} log_2(\tfrac{6}{7}) + -\tfrac{1}{7} log_2(\tfrac{1}{7}) = 0.5916728\end{aligned}$$

- Calculate the weighted average entropy of children:

$$\text{Entropy}_{\text{income split}} = \tfrac{5}{12}(0) + \tfrac{7}{12}(0.5916728) = 0.3451425$$

- Then calculate the information gain:

$$\begin{aligned}\text{IG}_{\text{income}} &= \text{Entropy}_{\text{root}} - \text{Entropy}_{\text{income split}} \\ &= 1 - 0.3451425 = 0.6548575\end{aligned}$$

- We then can perform the same calculation on all other features (e.g. employment, part of town) and compare results. The goal is to *maximize* the IG statistic at each decision point. In this case, we see that income is the best attribute to use for splitting. This split is easily interpretable: "The majority of users of health services can be predicted to earn less than $20,000."

Measure	IG
Employment	0.00
Income	0.6548575
Area of Town	0.027119

Gini Impurity is closely related to the entropy with a slight modification:

$$\text{Gini Impurity} = \sum p_i(1 - p_i) = 1 - \sum p_i^2$$

Using Gini Impurity as an attribute test is also similar to Information Gain:

$$\text{Gini Gain} = \text{Gini}_{\text{root}} - \text{Weighted Gini}_{\text{child}}$$

(3) Stopping Criteria + Tree Pruning

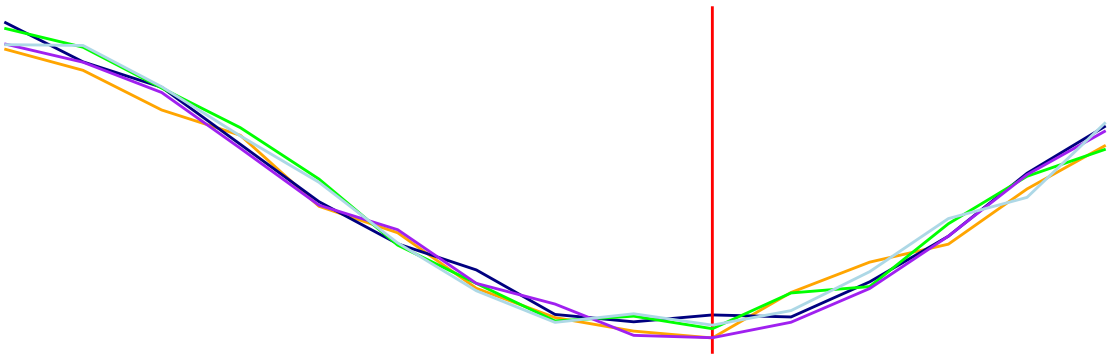
Both Gini Gain and Information Gain attribute tests can be recursively applied until there are no longer input features available to split the data. This is also known as a "fully grown tree" or an "unpruned tree". While the terminal leafs may yield a high degree of accuracy in training, trees may grow to epic and complex proportions that have leaf sizes are often times too small to provide accurate and generalizable results. While fully grown trees are considered to have low bias, their out-of-sample performance may be high in variance. There [theoretically] exists some optimal balancing point where trees are complex enough to capture statistical patterns, but are not too complex to yield misleading results.

Fortunately, the methodologists who invented decision tree learning have designed two approaches to balance accuracy and generalizability: stopping criteria and pruning.

Recall that a leaf is defined as a node with no child nodes. Otherwise stated, a leaf is a terminal node in which no additional attribute testing is conducted – it's placed out of commission. Stopping criteria are employed to determine if a node should be labeled a leaf during the growing process, thereby stopping tree growth at a given node. These criteria are specified before growing the tree and take on a number of different forms including:

- A node has fewer records than a pre-specific threshold;
- The purity or information gain falls below a pre-specified level or is equal to zero;
- The tree is grown to n-number of levels (e.g. Number of levels of child nodes relative to the root exceeds a certain threshold).

While stopping criteria are useful, the results in some studies indicate their performance may be sub-optimal. The alternative approach involves growing a tree to its fullest, then comparing the prediction performance given tree complexity (e.g. number of nodes in the tree) using cross-validation. In the example graph below, model accuracy degrades beyond a certain number of nodes. Thus, optimal number of nodes is defined as when cross-validation samples (e.g. train/test, k-folds) reaches a minimum across samples. Upon finding the optimal number of nodes, the tree is *pruned* to only that number of nodes.



Issues

Like any technique, decision trees have strengths and weaknesses:

Strengths	Weakness
- Rules (e.g. all the criteria that form the path from root to leaf) can be directly interpreted. - Method is well-suited to capture interactions and non-linearities in data. - Technique can accept both continuous and continuous variables without prior transformation. - Feature selection is conducted automatically	- Data sets with large number of features will have overly complex trees that, if left unpruned, may be too voluminous to interpret. - Trees tend to overfitted at the terminal leafs when samples are too small.

Decision Trees in Practice

To put decision trees into practice, we will use a simple train/test design to partition the `health` data set into two equal parts (a 50-50 partition). Then, test a few different versions of the model before testing accuracy using an AUC statistic.

```
#Create index of randomized booleans of the same length as the health data set
  set.seed(100)
  rand <- runif(nrow(health))
  rand <- rand > 0.5

#Create train test sets
  train <- health[rand == T, ]
  test <- health[rand == F, ]
```

With the data ready, we can call the `rpart` library, which is the most popular classification trees library.

```
#Load rpart library
  library(rpart)
```

The main method we're concerned with is the `rpart()` method, which requires a few arguments:

```
object <- rpart(<target> ~ <features>, method = "class", data = df, control)
```

- *object*. Outputs is a class object containing all the learned rules and insights.
- *rpart()*. The method.
- *target*. A discrete feature.
- *features*. Input features of any kind
- *method*. Indicates the type of tree: "class" for classifier, "anova" for regression tree.

- *data*. Specifies the source data frame.
- *control*. Is a catch all for stopping and pruning conditions. The two that will come in handy in this exercise are:
  - *cp*. Parameter that indicates the complexity of the tree. $cp = 1$ is a tree without branches, whereas $cp = 0$ is the full unpruned tree. If *cp* is not specified, `rpart()` defaults to a value of 0.01.
  - *minbucket*. Minimum number of observations in any terminal leaf.
  - *minsplit*. Minimum number of observation in a node to qualify for an attribute test.

As a first pass, we'll run `rpart()` with the default assumptions. Note that in `rpart()` automatically conducts k-folds cross-validation for each level of tree growth. If one were to use `summary()` or `str()` to check the structure of the output object named `fit`, the inner workings would likely be found to be quite exhaustive and rather complex. Fortunately, the `printcp()` method can be used to obtain a summary of the overall model accuracy for tree at different stages of growth. Key features of the `printcp()` output include:

- A listing of the variables actually used in construction (note that `cit` and `race` were not used)
- In the table, `CP` indicates the tree complexity, `nsplit` is the number of splits, `rel error` is the prediction error in the training data, `xerror` is the cross-validation error, and `xstd` is the standard error.

To choose the best tree, the *rule of thumb* is to first find the tree with the lowest cross-validation `xerror`, then find the tree that has the lowest number of splits that is still within one standard deviation `xstd` of the best tree. This rule ensures that overfitting is minimized within reason. In this first model, the best tree has $nsplit = 4$ and $xerror = 0.54245$. By applying the rule, the upper bound of acceptable error is $xerror_{\text{limit}} = 0.54245 + 0.0084772 = 0.5509272$. As it turns out, the tree with $nsplit = 3$ is within one standard deviation and is thus the best model.

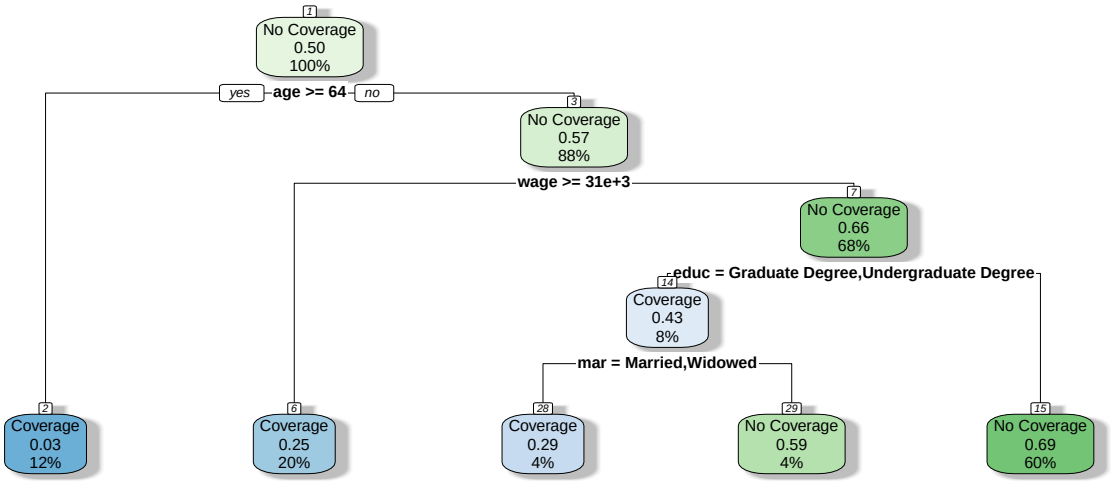
```
#Fit decision tree under default assumptions
  fit <- rpart(coverage ~ age + wage + cit + mar + educ + race,
               method = "class", data = train)

#Tools to review outpu
  printcp(fit)
```

```
##
## Classification tree:
## rpart(formula = coverage ~ age + wage + cit + mar + educ + race,
##     data = train, method = "class")
##
## Variables actually used in tree construction:
## [1] age  educ mar  wage
##
## Root node error: 5512/11083 = 0.49734
##
## n= 11083
##
##         CP nsplit rel error  xerror      xstd
## 1 0.231495      0   1.00000 1.01306 0.0095494
## 2 0.199565      1   0.76851 0.76851 0.0092809
## 3 0.022315      2   0.56894 0.56894 0.0086030
## 4 0.013425      3   0.54663 0.54953 0.0085117
## 5 0.010000      4   0.53320 0.54245 0.0084772
```

The model's learned rules contained in `fit` can be plotted with `plot()`, but it takes a bit of work to get the plot into a presentable format. The substitute is using the `rpart.plot` library, which auto-formats the tree and color codes nodes based on the concentration of the target variable.

```
#Plot
  library(rpart.plot)
```

```
rpart.plot(fit, shadow.col="gray", nn=TRUE)
```



While this answer is valid, it should be noted that the CP lower threshold is 0.01, which is the default value. For robustness, we should run the model once more, this time specifying $cp = 0.0$ to obtain the full, unpruned tree (see below). Applying the error minimization rule once more, the minimum $xerror = 0.47859$, which corresponds to $nsplit = 38$. The maximum $xerror$ within one standard deviation is $xerror = 0.47859 + 0.0081339 = 0.4867239$, which corresponds to $nsplit = 32$ with $xerror = 0.48657$ and $cp = 0.0010281$

```
#cp = 0
  fit.0 <- rpart(coverage ~ age + wage + cit + mar + educ + race,
                 method = "class", data = train, cp = 0)
  printcp(fit.0)
```

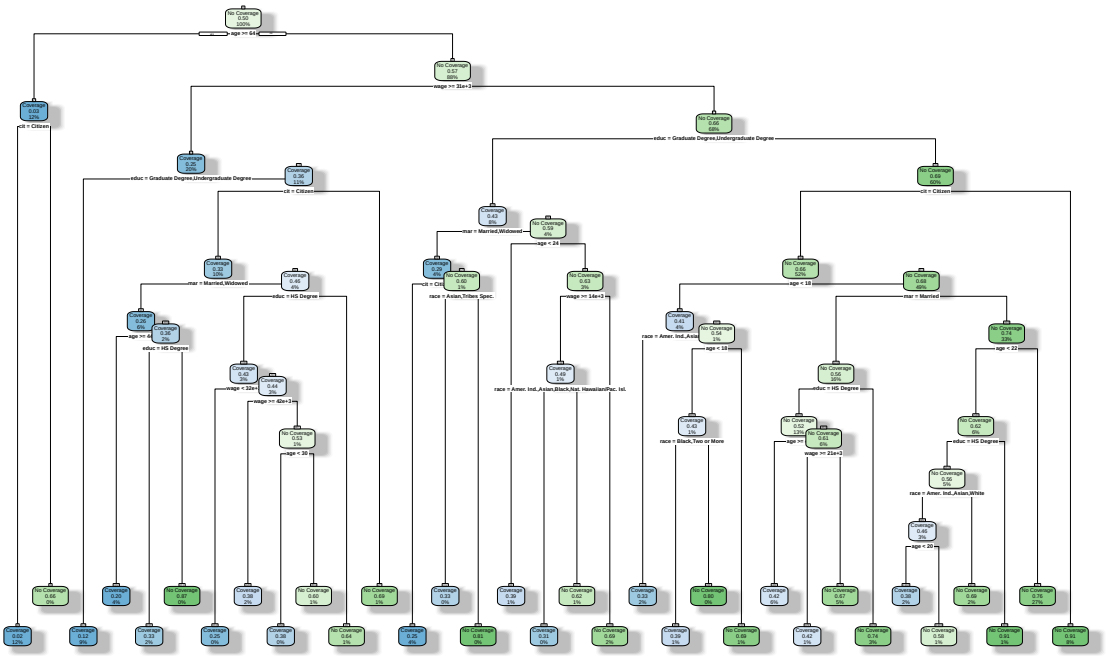
```
##
## Classification tree:
## rpart(formula = coverage ~ age + wage + cit + mar + educ + race,
##     data = train, method = "class", cp = 0)
##
## Variables actually used in tree construction:
## [1] age  cit  educ mar  race wage
##
## Root node error: 5512/11083 = 0.49734
##
## n= 11083
##
##            CP nsplit rel error  xerror      xstd
## 1  2.3149e-01      0   1.00000 1.01179 0.0095495
## 2  1.9956e-01      1   0.76851 0.76851 0.0092809
## 3  2.2315e-02      2   0.56894 0.56894 0.0086030
## 4  1.3425e-02      3   0.54663 0.55679 0.0085464
## 5  6.3498e-03      4   0.53320 0.53302 0.0084301
## 6  4.3541e-03      9   0.50109 0.50907 0.0083051
## 7  3.1749e-03     10   0.49673 0.50671 0.0082923
## 8  2.5399e-03     12   0.49038 0.50417 0.0082785
## 9  2.3585e-03     13   0.48784 0.50472 0.0082814
## 10 2.1771e-03     14   0.48549 0.50345 0.0082745
## 11 1.8142e-03     16   0.48113 0.49710 0.0082395
## 12 1.6328e-03     17   0.47932 0.49147 0.0082079
## 13 1.4514e-03     18   0.47769 0.48712 0.0081832
```

```
## 14 1.3909e-03     20   0.47478 0.48694 0.0081821
## 15 1.3607e-03     24   0.46662 0.48549 0.0081738
## 16 1.0885e-03     26   0.46390 0.48385 0.0081644
## 17 1.0281e-03     32   0.45737 0.48186 0.0081529
## 18 9.0711e-04     35   0.45428 0.47896 0.0081360
## 19 7.2569e-04     36   0.45337 0.47714 0.0081254
## 20 6.6522e-04     38   0.45192 0.47859 0.0081339
## 21 6.3498e-04     41   0.44993 0.47896 0.0081360
## 22 5.4427e-04     43   0.44866 0.47986 0.0081413
## 23 4.9891e-04     64   0.43650 0.47841 0.0081328
## 24 4.5356e-04     68   0.43451 0.48041 0.0081445
## 25 3.6284e-04     72   0.43269 0.48240 0.0081561
## 26 2.7213e-04     99   0.42253 0.48639 0.0081790
## 27 2.4190e-04    119   0.41636 0.49256 0.0082141
## 28 2.2678e-04    134   0.41165 0.49329 0.0082181
## 29 1.8142e-04    139   0.41038 0.49329 0.0082181
## 30 1.3607e-04    172   0.40403 0.50708 0.0082943
## 31 1.2095e-04    182   0.40239 0.50998 0.0083099
## 32 9.0711e-05    199   0.40022 0.51343 0.0083284
## 33 7.2569e-05    247   0.39532 0.51905 0.0083581
## 34 6.0474e-05    258   0.39351 0.52540 0.0083912
## 35 4.5356e-05    267   0.39296 0.52540 0.0083912
## 36 3.6284e-05    274   0.39260 0.52540 0.0083912
## 37 3.0237e-05    284   0.39224 0.52758 0.0084024
## 38 2.5917e-05    302   0.39169 0.52776 0.0084033
## 39 0.0000e+00    309   0.39151 0.52776 0.0084033
```

At this point, we'll re-run the decision tree once more with the updated *cp* value, assign the decision tree object to `fit.opt`, and plot the resulting decision tree. Notice how the rendered tree is significantly more complex relative to the default and interpretation may be more challenging with a plethora of criteria.

```
fit.opt <- rpart(coverage ~ age + wage + cit + mar + educ + race,
             method = "class", data = train, cp = 0.0010281)
rpart.plot(fit.opt, shadow.col="gray", nn=TRUE)
```

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



In lieu of a thorough review of the learned rules, we may rely on a measure of variable importance, that is defined as follows:

$$\text{Variable Importance}_k = \sum \text{Goodness of Fit}_{\text{split, k}} + (\text{Goodness of Fit}_{\text{split,k}} \times \text{Adj. Agreement}_{\text{split}})$$

Where *Variable Importance* for variable k is the sum of *Goodness of Fit* (e.g. Gini Gain or Information Gain) at a given split involving variable k. In otherwords, a variable's importance is the sum of all the contributions variable k makes towards predicting the target. Below, we can see that the measure can be extracted from the `fit.opt` object. As it turns out, `age` is the most important factor.

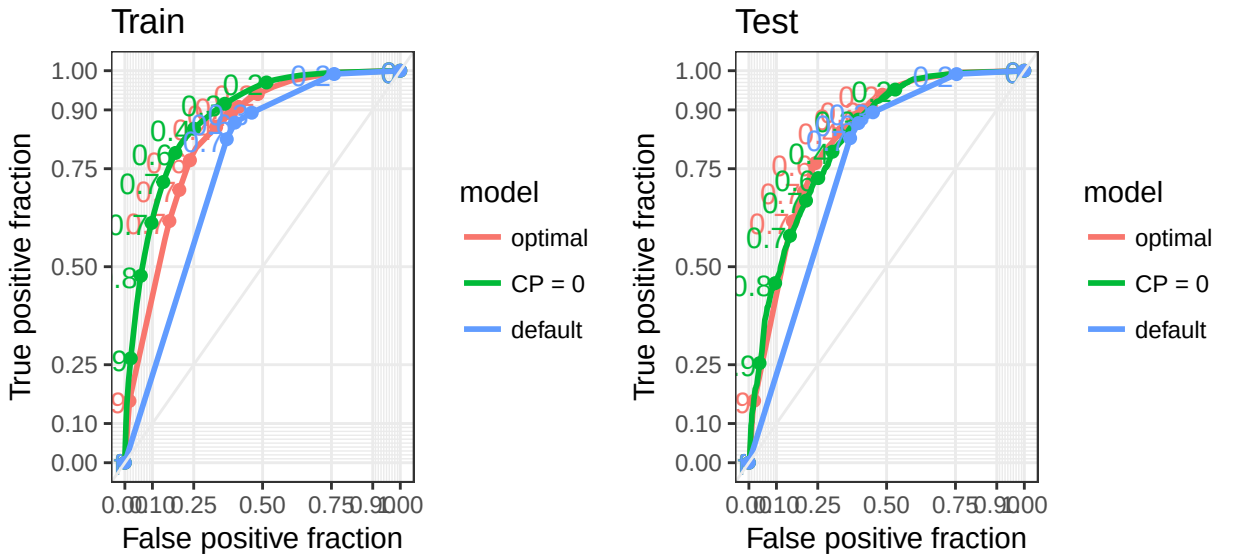
```
#Extract variable importance list from fit object
  fit.opt$variable.importance
```

```
##       age      wage      educ       mar       cit      race
## 831.43469 628.80645 282.40482 205.42914 147.29141  62.20588
```

Using the `plotROC` package once again, we calculate the AUC score for each model to assess predictive performance on both the training and test set. One particularly striking difference is the switch in position of the *optimal* and $cp = 0$ curves: $cp = 0$ is higher in the training set, but are at the approximate safe height in test. This indicates that $cp = 0$ notably overfits, likely to the extra low bias of unpruned leafs.

```
#plotROC
  library(plotROC)
  library(gridExtra)

#Predict values for train set
  pred.opt.train <- predict(fit.opt, train, type='prob')[,2]
  pred.0.train <- predict(fit.0, train, type='prob')[,2]
  pred.default.train <- predict(fit, train, type='prob')[,2]

#Predict values for test set
  pred.opt.test <- predict(fit.opt, test, type='prob')[,2]
  pred.0.test <- predict(fit.0, test, type='prob')[,2]
  pred.default.test <- predict(fit, test, type='prob')[,2]
```

```
#Set up ROC inputs
  input.test <- rbind(data.frame(model = "optimal", d = test$coverage, m = pred.opt.test),
                      data.frame(model = "CP = 0", d = test$coverage,  m = pred.0.test),
                      data.frame(model = "default", d = test$coverage,  m = pred.default.test))
  input.train <- rbind(data.frame(model = "optimal", d = train$coverage,  m = pred.opt.train),
                      data.frame(model = "CP = 0", d = train$coverage,  m = pred.0.train),
                      data.frame(model = "default", d =  train$coverage,  m = pred.default.train))


#Graph all three ROCs
  roc.test <- ggplot(input.test, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  + ggtitle("Test")
  roc.train <- ggplot(input.train, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  +ggtitle("Train")

#Plot
  grid.arrange(roc.train, roc.test, ncol = 2)
```



Lastly, we can extract the AUC statistics using `calc_auc()`. As multiple AUCs were calculated, we will need to extract the labels for the AUCs from the `input` file in order to produce a a 'prettified' table using `xtable`. The resulting table below presents the results of the three models that were trained. For all models, we should expect that the training AUC will be greater than the test AUC. This is generally true, but occassionally the test AUC may be greater and is largely a matter of how the data was sampled.

Starting from the top of the table:

- *Full grown.* The unpruned tree is the most complex model, which means the model has a higher chance of overfitting. This is characterized by an artificially inflated training AUC and a large drop in test AUC. As seen, the AUC drops from 0.88 to 0.826 in the test sample. The unreliable results of an unpruned tree are likely due to the algorithm's sensitivity to irregular noise at leafs.
- *Optimal.* The optimal tree achieves a consistent $AUC = 0.83$ with minimal loss of accuracy as an appropriate level of complexity was precisely tuned.
- *Default.* An underfit model will have consistently low performance in both training and testing. As we can see, these patterns are played out in the table below containing AUCs for each the default decision tree, the optimal model complexity and the fully grown tree.

As the result of tuning towards an optimal model, we can see that the decision tree yields a marked

improvement over the kNN model's $AUC = 0.44$. For a social science problem, this is considered to be a decent result.

```
#Calculate AUC
  calc_auc(roc.test)$AUC
```

[1] 0.8300719 0.8261042 0.7551470

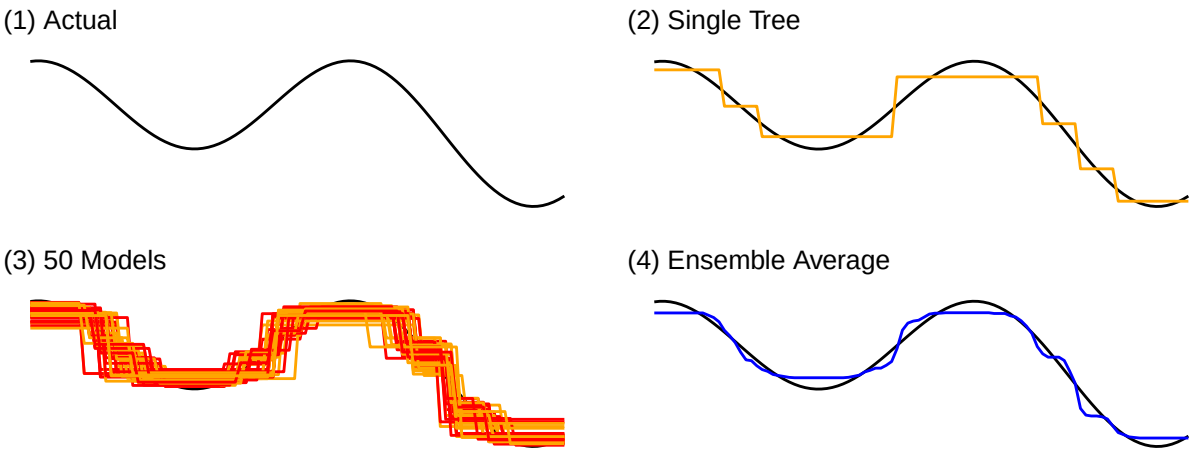
```
#Assemble a well-formatted table
  library(xtable)
  tab <- data.frame(model = unique(input.test$model),
                    train = calc_auc(roc.train)$AUC,
                    test = calc_auc(roc.test)$AUC)
  print(xtable(tab, digits = 5), comment=FALSE)
```

	model	train	test
1	optimal	0.83372	0.83007
2	CP = 0	0.88008	0.82610
3	default	0.75252	0.75515

Random Forests

In much of modern data references, we see more uncertainty being characterized. When a hurricane approaches the US Eastern Seaboard, forecasters often map the "cone of uncertainty" that provides the possible range of motion of a storm based on the results of many forecasted simulations. In presidential elections, often times the most polling results are ones that ensemble or average the results of many other similarly conducted polls. The reliance on predictions from a group of models with the same aim may very well improve prediction quality. In statistical learning, average the results of multiple models is known as *ensemble learning* or *ensembling* for short.

Ensemble methods combine the results of many models to obtain more stable results. Each model's accuracy can be characterized, among other things, by the appropriateness and stability of the prediction method and the randomness of the underlying sample. For example, the curve below in graph #1 can be approximated using a decision tree algorithm. The result of a single tree only loosely fits the curve in a jagged fashion (graph #2). In order to produce a more natural prediction that hugs the curve more closely, we may rely on boostrapping. Recall from elementary statistical that bootstrapping is defined as any statistical process that involves sampling records with replacement. Iterative sampling with replacement is like treating a sample as the population and by repeatedly sampling from the same sample, we can expose and characterize the qualities of an estimator. Many times, the results of an estimator are due to the "luck of the draw", meaning that some influential observations may be in one sample, but not in another. Bootstrapping helps to de-emphasizing the influence of errors by building a probability distribution of an estimator as well as averaging out the luck. We can bootstrap the decision tree by (1) sampling the data with replacement up to the full size of the sample, then (2) run the decision tree. The result of repeating the process 50 times is graph #3 that shows the "range of motion" of the decision tree process. By averaging the 50 bootstrap runs, we can produce a more organic, more accuracy result.



Random forests are an extension of decision trees using a modified form of bootstrapping and ensemble methods to mitigate overfitting and bias issues associated with single trees. The Random Forests algorithm creates a manifold of classification trees using a nuanced bootstrapping method known as *bootstrap aggregation* or *bagging*: Not only are individual records bootstrapped, but input features are bootstrapped such that if K variables are in the training set, then k variables are randomly selected to be inputted into the model such that $k < K$. Each bootstrap sample is trained using tree learning, allowing the trees to grow to a fully grown, unpruned tree. The resulting predictions of hundreds of trees are ensembled. The algorithmic process is described below.

Pseudo-code

```
Let S = training sample, K = number of input features
  1. Randomly sample S cases with replacement from the original data.
  2. Given K features, select k features at random where k < K.
  3. With a sample of s and k features, grow the tree to its fullest complexity.
  4. Predict the outcome for all records.
  5. Out-Of-Bag (OOB). Set aside the predictions for records not in the s cases.
Repeat steps 1 through 5 for a large number of times saving the result after each tree.
Vote and average the results of the tree to obtain predictions.
Calculate OOB error using the stored OOB predictions.
```

A key feature of this technique is the *Out-Of-Bag* (OOB) prediction. In each bootstrap sample, approximately one-third of observations are naturally left un-selected for training the decision tree. These unselected records are used as the basis of calculating the test error.

Not described in the pseudo-code, but nonetheless a key metric is *variable importance*, which, like for a decision tree, measures the contribution of a feature to the homogeneity of a classifier. Unlike decision trees, variable importance for a Random Forest is calculated as the mean decrease in the Gini coefficient of a split relative to the Gini coefficient of the root node. Gini coefficients measures homogeneity on a scale of 0 to 1, where 0 is perfect homogeneity and 1 is perfect heterogeneity. The Gini changes are summed for each variable and normalized.

Tuning

Tuning Random Forest parameters have an impact on different aspects of trees. The principal tuning parameters include: Number of features and number of trees.

- *Number of input features.* As k number of parameters need to be selected in each sampling round, the value of k needs to be determined through trial and error. For regression-based Random Forests, the rule of thumb is to selection $k = \frac{K}{3}$. For classification problems, start with $k = K^{\frac{1}{2}}$. Values of $2 \times k$ and $frack2$ should also be tested. The k that minimizes the error on the OOB predictions is the keeper.

- *Number of trees* influences the stability the Variable Importance metric that is commonly used to infer variable influence in decision tree learning. More trees help to stabilize the Variable Importance estimate. To determine the number of trees, keep adding trees to a sample until the OOB error for a randomly select set of trees is approximately equal to that of the ensemble.

Random Forests in Practice

Like decision trees, much of Random Forests rely on easy to use methods made available through the `randomForest` library. The basic method requires the following parameters:

- *object*. Outputs is a class object containing all the learned rules and insights.
- *randomForest()*. The method.
- *target*. A discrete feature.
- *features*. Input features of any kind
- *method*. Indicates the type of tree: "class" for classifier, "anova" for regression tree.
- *data*. Specifies the source data frame.
- *mtry*. Number of variables to be randomly sampled per iteration. Default is $\sqrt{k}$ for classification.
- *ntree*. Number of trees. Default is 500.

```
randomForest(<formula>, <data>, <control>)
```

Using the same formula as the `rpart()` function, we can train a naive Random Forest and check the OOB error. Approximately 80% and 77% of people without and with coverage, respectively, can be accurately classified. The algorithm also tried 2 variables per bagging iteration to start.

```
#Load randomForest library
  library(randomForest)

#Run Random Forest
  fit.rf <- randomForest(coverage ~ age + wage + cit + mar + educ + race, data = train)

#Check OOB error
  fit.rf
```

```
##
## Call:
##  randomForest(formula = coverage ~ age + wage + cit + mar + educ +      race, data = train)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 23.68%
## Confusion matrix:
##             Coverage No Coverage class.error
## Coverage        3999        1513   0.2744920
## No Coverage     1111        4460   0.1994256
```

Using the `importance()` method, we can see the Mean Decrease Gini, which calculates the mean of Gini coefficients. `age` has the largest value of 1129, indicating that age is the best predictor of coverage; However, the values themselves do not have any meaning outside of a comparison with other Gini measures.

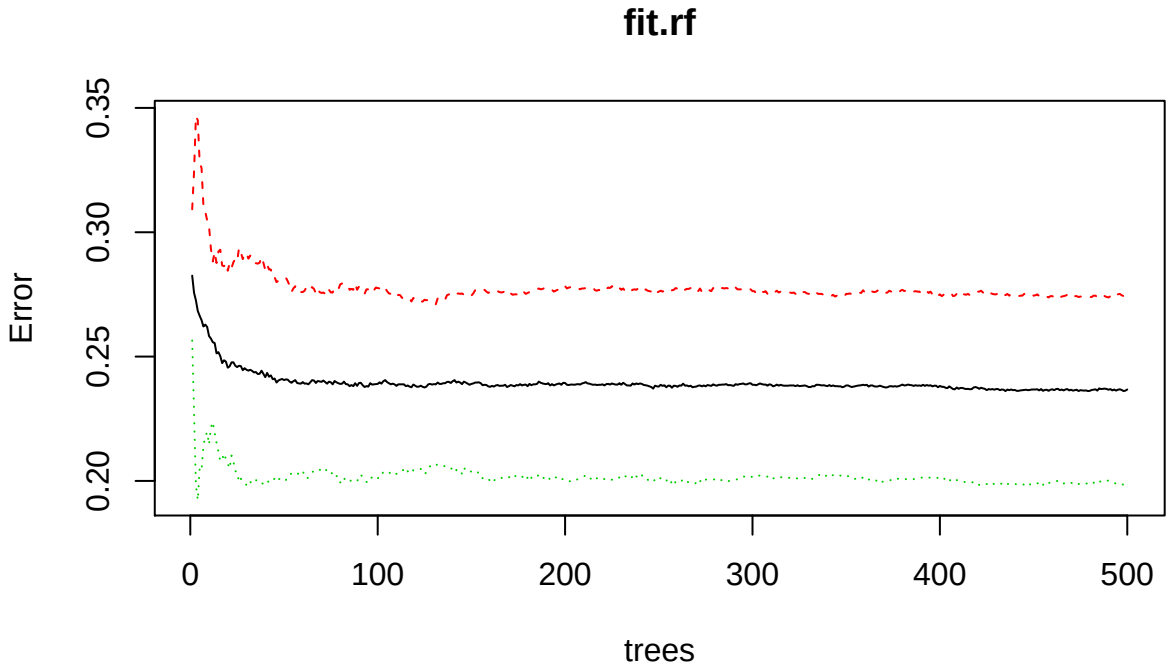
```
#Variable importance
  print(importance(fit.rf, type = 2))
```

```
##      MeanDecreaseGini
## age         1128.1027
## wage         730.1619
```

```
## cit          172.1726
## mar          337.1476
## educ         362.7965
## race         179.3288
```

By default, the `randomForests` library sets the number of trees to equal 500. By plotting the fit object, we can see how OOB error and the confidence interval converges asymptotically as more trees are added to the ensemble. Otherwise stated, more trees will help up to a certain point and the default is likely more than enough.

```
  plot(fit.rf)
```



As we know that $n = 500$ trees is more than enough, we will now need to tune the tree for the number of variables. To tune the algorithm, we will use the `runeRF()` method. The method searches for the optimal number of variables per split by incrementally adding variables. While it's a useful function, it is relatively verbose. In addition to the target and input features, a number of other parameters need to be specified:

- *ntreeTry*. Number of trees.
- *mtryStart*. Number of variables to start.
- *stepFactor*. Factor increase in number of variables tested per iteration.
- *improve*. Minimum relative improvement in OOB error for search to continue.
- *trace*. Print search progress.
- *plot*. Plot the search results.

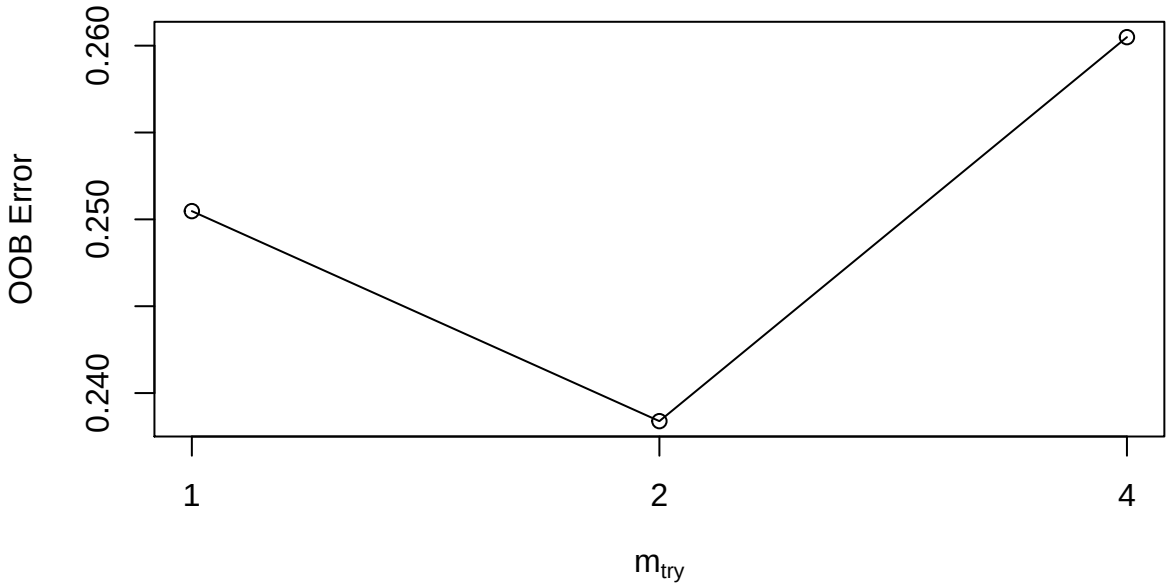
```
  tuneRF(<x-vals>, <y-vals>, <ntreeTry>,  <mtryStart>, <stepFactor>, <improve>, <trace>, <plot>)
```

Below, we conduct a search from `mtryStart = 1` with a `stepFactor = 2`. The search result indicates that *2* variables per split are optimal.

```
#Search for most optimal number of input features
  fit.tune <- tuneRF(train[,-1], train$coverage, ntreeTry = 500,
                     mtryStart = 1, stepFactor = 2,
                     improve = 0.001, trace = TRUE, plot = TRUE)
```

```
## mtry = 1  OOB error = 25.05%
## Searching left ...
```

```
## Searching right ...
## mtry = 2     OOB error = 23.84%
## 0.04827089 0.001
## mtry = 4     OOB error = 26.05%
## -0.09273278 0.001
```




```
  fit.tune
```

```
##       mtry  OOBError
## 1.OOB    1 0.2504737
## 2.OOB    2 0.2383831
## 4.OOB    4 0.2604890
```

```
  tune.param <- fit.tune[fit.tune[, 2] == min(fit.tune[, 2]), 1]
```

Using the optimal result, we can plug back into the `randomForest()` method and re-run. However, as the default model already has the same parameters as the optimal model, we can proceed to calculating the model accuracy. Comparing the training and test models for the Random Forest algorith, we see a large drop in the AUC between train and test, indicating quite a bit of overfitting.

```
#plotROC
  library(plotROC)

#Predict values for train set
  pred.rf.train <- predict(fit.rf, train, type='prob')[,2]

#Predict values for test set
  pred.rf.test <- predict(fit.rf, test, type='prob')[,2]


#Set up ROC inputs
  input.rf <- rbind(data.frame(model = "train", d = train$coverage, m = pred.rf.train),
                    data.frame(model = "test", d = test$coverage,  m = pred.rf.test))

#Graph all three ROCs
  roc.rf <- ggplot(input.rf, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  +ggtitle("Train")
```

```r
#AUC
  calc_auc(roc.rf)
```

```
##   PANEL group       AUC
## 1     1     1 0.9069204
## 2     1     2 0.8359834
```

To be continued in Lecture 8