Stat 154 Final Project Report

Jack Moorer and Jaime Pimentel

Introduction

The Census Income data set provided by Ronny Kohavi and Berry Becker is a data set documenting several characteristics of individuals across the globe. Amongst these characteristics are age, working status, sex, hours worked per week, native country, and even relationship status. The objective of this report is to utilize these given characteristics and create several classification models in order help predict whether a certain person makes over $50,000 in yearly earnings.

In this report, we will focus on fitting three different types of classifiers on our data: classification trees, bagged trees, and random forest classifiers. For each classification method, we will use several statistical measures in order to hypertune and eventually arrive at an optimal model fit. In doing so, we will put these three models against each other to finally choose the model we believe fits and predicts our data the best.

In building a model to predict if people make over 50k a year or not, preditive performance is obviously important, but also crucial to this project is finding the importance of each feature. In this report we will find, based on our 3 methods, which features are more important in predicting whether a person makes 50k a year or not.

The first part of this report details some of what we did in our Exploritory Data Analysis. We then used the training data to fit a classification tree, bagged tree, and random forest. For each model we looked at the performance on the training data by looking at the training error rate and looking at the training ROC and AUC. For the classification tree, we tuned the hyperparameter based on the size of the classification tree, or the number of terminal nodes. We examined the variable importance of the classification tree by looking plotting the tree. For the bagged tree we tuned the parameter dealing with number of trees used for estimation. We examined the variable importance of the bagged tree using the mean decreased gini. Finally, we tuned the random forest based on both the number of trees used for estimation, and the number of features to use for each split. It was important that the number of features was not maximum, because then the random forest would be a bagged tree, so we limited the number of features the random forest could use using tuneRF.

Finally, we looked at the test performance of each model. For each model we looked at the test data error rate. We then found the confusion matrix for each model, and used it to find the sensitivity and specificity of each model. We then plotted the ROC curves for each classifier, and examined each AUC. In the end we will report which model performed the best on the test data.

A note on the file structure of this project. Data is stored in the data folder. This file, in the report folder, is the main report for this project. Inside the report folder is a sub-report folder, with Rmd files and pdfs of analysis performed in EDA, classification trees, bagged trees, and random forest. Each Rmd file also has an occompaning R script file in the R folder, with the except of EDA, whose R script code is in the cleandata.R script. Each R script is used to save the plots and output of each of the sections of this report. You can find plots in the images folder, and all other text results in the results folder.

EDA

Before we made any exploritory data analysis, we cleaned up the data. You can see how we cleaned the data in the cleandata.R script in the R folder.

First the data came in the form of a .data file for the train and a .test file for the test data. I noticed that you can read in files that are .data or .test using the read.table() function, however, I noticed it would have been

a lot nicer if the data was in csv file format. In order to change it, I just renamed adult.data to adult.csv and adult.test to test.csv. We also specificially set stringAsFactors to false so that we can define later define the levels of each column so that the test and training data had the same levels for categorical variables. The data did not have names, so I named each column based on the name provided in the attribute information section from where we got the data, and named the response variable Over50k.

The Over50k variable originally had two levels, ">50k" and "<=50k". I decided I wanted them to be categorized as "Yes" and "No", so I changed the labels so that ">50k" was "yes" and "<=50k" was no. I also added a column over_50k_numeric, that just acted as a numeric indicator variable for Over50k with Yes as 1 and No as 0, however, it turned out it was not needed.

Next we noticed there were missing values set to a quetion mark "?". I replaced these with NA values. We then decided to omit data that had NA values. The reason we felt this was ok was that there were still plenty of data points left after ommitted data.

Next we looked at the distrubution and correlations of the data. One particular variable that stood out to me when looking at the distrubtion of the data was native_country. At first, we kept native_country as it was, however, we ran into several issues. For one, there were more than 32 countries represented, so in order to run a classification tree we had to keep native_country as a character vector. I looked at the distribution of native_country in the training set, and found that by far the frequency of people from the United States trumped the rest of the countries. To deal with native_country, we decided it then made sense, based on this representation, to remove the native_country column, and replace it with a column called US_citized, that indicated whether the person's native country was the United States or not.

Another part of the EDA we noticed was that the frequency of people having over 50k was about half of that of those that were less than equal to 50k. We noticed that could be an issue for predictive power, as a model that overpredicted "No" could have high accuracy but low true performance.

Next, we looked at the features correlation using the library polycor. We decided that education and education_num were far too correlated, so we decided to remove the variable education_num.

We did not change the scale of any variables. One of the main reasons is that many of our variables were categorical, and we decided not to convert categorical variables in dumby variables. We also decided not to deal with outliers, between with over 30,000 training observations the impact of outliers would be relatively small.

Finally, I wrote the cleaned training data as clean_train.csv and wrote the cleaned test data as clean_test.csv in the data folder.

```
#read in the clean data
train <- read.csv("/Users/jackmoorer/Stat154/Projects/Project/data/clean_train.csv", header = TRUE)
test <- read.csv("/Users/jackmoorer/Stat154/Projects/Project/data/clean_test.csv", header = TRUE)
```

Building Models

Clasification Tree

Build the Tree

There seems to be two main ways to build a classification tree in r. The first is to use the package rpart, which was used in the examples in APM. The second way, which is the method they used in ISL and we used in lab, was using the tree package. After looking at booth methods, we decided to build our classification tree using the method form the tree package.

```
#load packages
library(ISLR)
```

```
library(tree)
library(rpart)
```

The tree method allows for categorical response variables and categorical predictor variables, so we don't we use the numeric over 50k indicator variable or transform our predictors into dumby variables.

```
#remove numeric reponse column
train_tree <- train[,-ncol(train)]
```

I am going to start out by building a classification tree using the default tree function, just to get an idea of what is happening.

```
#default classification tree
classification_tree <- tree(Over50k ~., data = train_tree)
summary(classification_tree)
```

```
##
## Classification tree:
## tree(formula = Over50k ~ ., data = train_tree)
## Variables actually used in tree construction:
## [1] "relationship" "capital_gain" "education"    "occupation"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7118 = 21460 / 30150
## Misclassification error rate: 0.1589 = 4794 / 30162
```

I can now move on to tuning optimal parameters. From what I can see there are to main ways to tune hyperparameters using the tree package. . Both ways use cv.tree from the tree package, with the only difference being the prune function to perform cross validation. I am going to start by using prune.misclass, which looks at the parameters by looking at the misclassification rate of the tree with those parameters. The parameter we can tune is one of the cost-complexity parameter, k, or the number of terminal nodes in the tree, size. I am going to focus on the number of terminal nodes, size, as the parameter to tune, but I will still show the results of the cost-complexity parameter as well.

```
#tune parameters
set.seed(100)
classification_tree_cv <- cv.tree(classification_tree, FUN = prune.misclass)
```

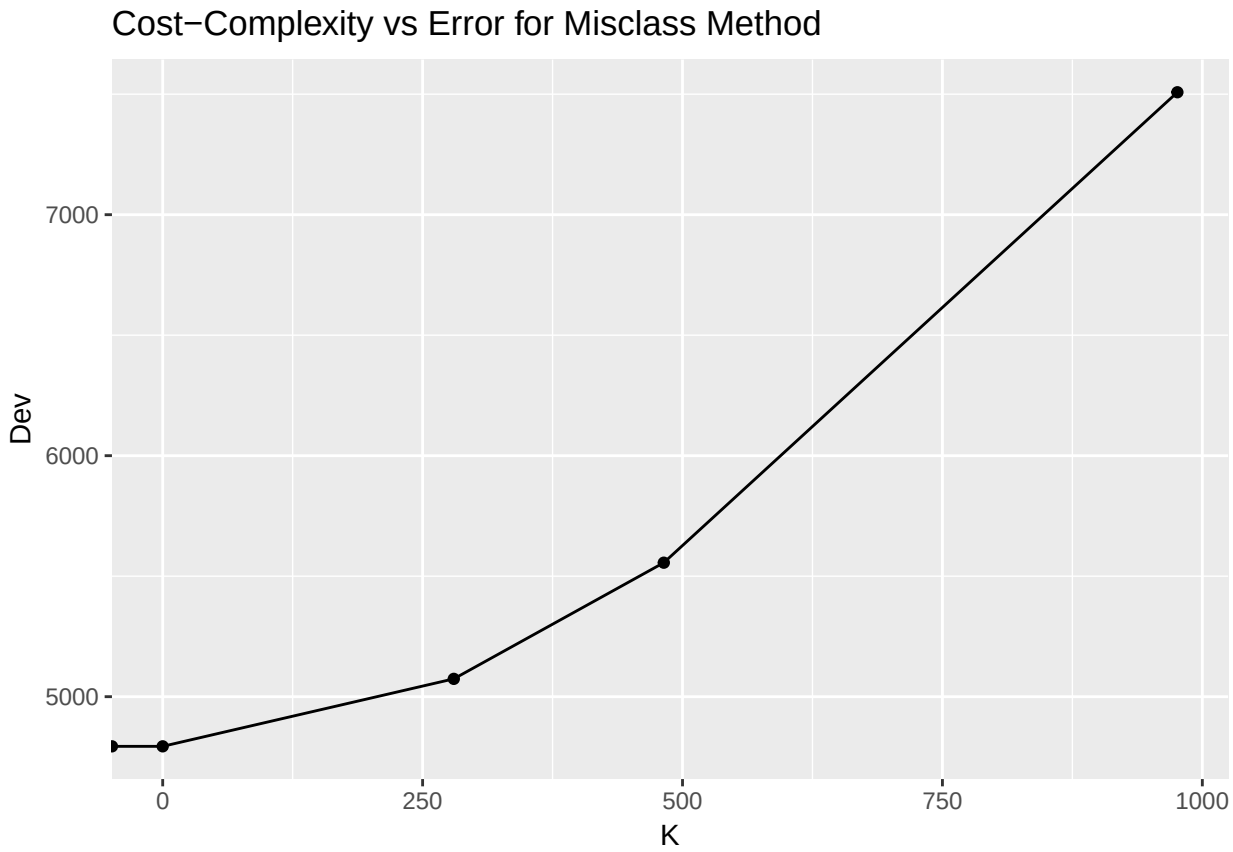
In cv.tree, the performance of the parameter is represented with dev, which looks at the misclassification rate of the tree with the given parameters. We can plot the results of the different terminal node sizes or cost-complexity parameters with dev.

```
#look at data frame of cv results
library(ggplot2)
Size <- classification_tree_cv$size
K <- classification_tree_cv$k
Dev <- classification_tree_cv$dev
Misclass <- data.frame(Size, K, Dev)
Misclass
```

```
##   Size    K  Dev
## 1    8 -Inf 4794
## 2    5    0 4794
## 3    4  280 5074
## 4    3  482 5556
## 5    1  976 7508
```

```
#plot cv results of number of terminal nodes
ggplot(data = Misclass, aes(x = Size, y = Dev)) + geom_point() + geom_line() + ggtitle("Size of Tree vs
```


```
#plot cv results of number cost-complexity
ggplot(data = Misclass, aes(x = K, y = Dev)) + geom_point() + geom_line() + ggtitle("Cost-Complexity vs
```



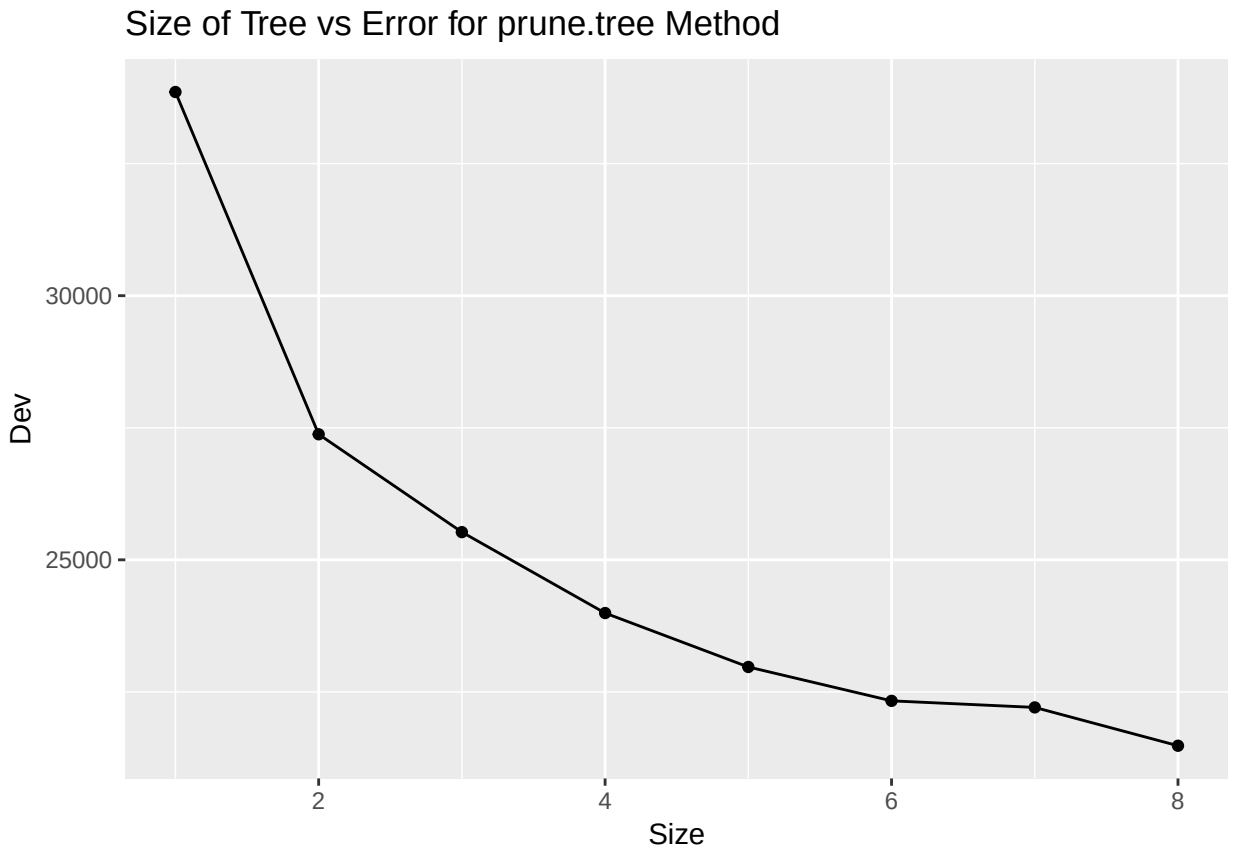
We could use k = 0 as a cost complexicity hyper-parameter. However, if we look at number of trees using 5 or 8 both give us the same minimum. We can also look at a different type of cross validation approach using the default prune.tree function from cv.tree(). In theory I could just pick the smaller value of 5, since an increased number of terminal nodes may overfit the data, however, I am going to compare the results of this tuning process to one using prune.tree has the function in cv.tree.

```
#cv using prune.tree
set.seed(200)
classification_tree_cv_default <- cv.tree(classification_tree, FUN = prune.tree)
```

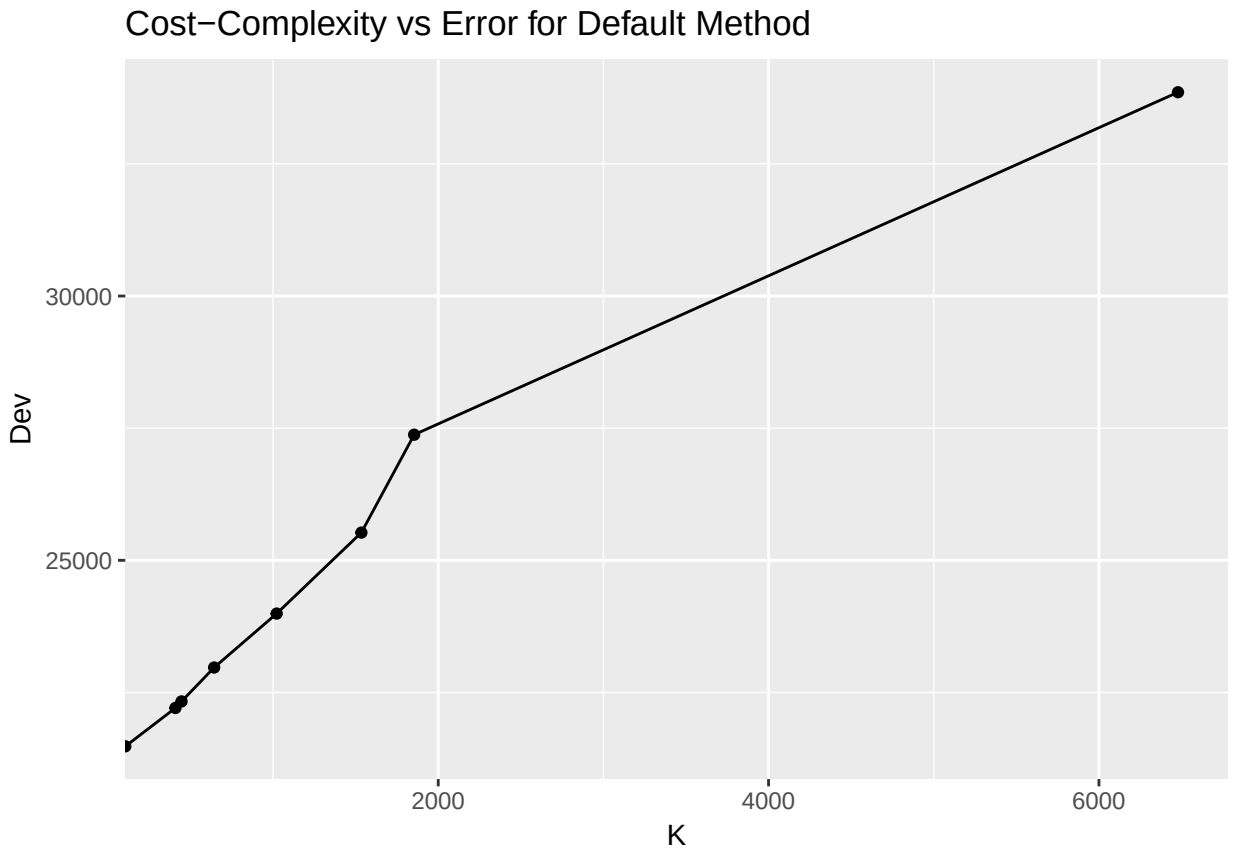
```
#creat data frame of results
Size <- classification_tree_cv_default$size
K <- classification_tree_cv_default$k
Dev <- classification_tree_cv_default$dev
default <- data.frame(Size, K, Dev)
default
```

```
##   Size         K      Dev
## 1    8      -Inf 21480.40
## 2    7  408.9358 22206.32
## 3    6  445.8046 22330.03
## 4    5  643.1663 22971.42
## 5    4 1021.8448 23992.21
## 6    3 1534.7488 25524.19
## 7    2 1854.1982 27376.34
## 8    1 6478.6784 33855.37
```

```
#plot cv results for size
ggplot(data = default, aes(x = Size, y = Dev)) + geom_point() + geom_line() + ggtitle("Size of Tree vs
```




```
#plot cv results for cross complexity
ggplot(data = default, aes(x = K, y = Dev)) + geom_point() + geom_line() + ggtitle("Cost-Complexity vs
```



From both of these results it seems like minimum cost-complexity and maximum size is ideal for our tuned parameters. I am going to use size = 8 as the optical parameter.

```
#get best size parameter from cross validation
names <- classification_tree_cv_default$size
values <- classification_tree_cv_default$dev
names(values) <- names
size <- as.numeric(names(which.min(values)))
```

I can now build an optimal classification tree using the best size parameter using prune.misclass.

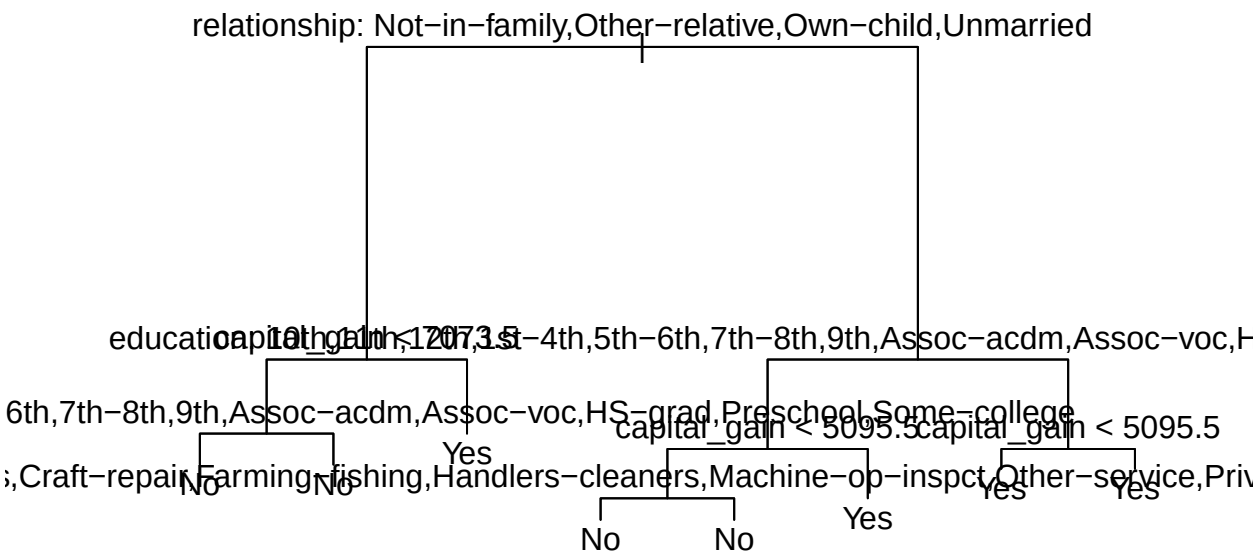
```
#build classification tree
set.seed(4)
prune_classification_tree <- prune.misclass(classification_tree, best = size)
```

```
#let's look at the tree
prune_classification_tree
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 30162 33850.00 No ( 0.751078 0.248922 )
##    2) relationship: Not-in-family,Other-relative,Own-child,Unmarried 16293  8237.00 No ( 0.930338 0.
##      4) capital_gain < 7073.5 15993  6614.00 No ( 0.947164 0.052836 )
##        8) education: 10th,11th,12th,1st-4th,5th-6th,7th-8th,9th,Assoc-acdm,Assoc-voc,HS-grad,Prescho
##        9) education: Bachelors,Doctorate,Masters,Prof-school 3257  2786.00 No ( 0.847099 0.152901 ) *
##      5) capital_gain > 7073.5 300    87.69 Yes ( 0.033333 0.966667 ) *
##    3) relationship: Husband,Wife 13869 19140.00 No ( 0.540486 0.459514 )
##      6) education: 10th,11th,12th,1st-4th,5th-6th,7th-8th,9th,Assoc-acdm,Assoc-voc,HS-grad,Preschool
```

```
##       12) capital_gain < 5095.5 9219 11370.00 No ( 0.692917 0.307083 )
##         24) occupation: Armed-Forces,Craft-repair,Farming-fishing,Handlers-cleaners,Machine-op-inspct
##         25) occupation: Adm-clerical,Exec-managerial,Prof-specialty,Protective-serv,Sales,Tech-suppo
##       13) capital_gain > 5095.5 500    90.15 Yes ( 0.018000 0.982000 ) *
##      7) education: Bachelors,Doctorate,Masters,Prof-school 4150  4798.00 Yes ( 0.264819 0.735181 )
##       14) capital_gain < 5095.5 3512  4362.00 Yes ( 0.312358 0.687642 ) *
##       15) capital_gain > 5095.5 638    27.05 Yes ( 0.003135 0.996865 ) *
```

```
#lets look at the structure of the tree
plot(prune_classification_tree)
text(prune_classification_tree, pretty = 0)
```



Based on the tree structure, the most important variable is relationship status, followed by capital gain, and then education, then finally occupation. These are the only variables used to make the tree. However, there are some further insights to be had. The most important categories that split the data in relationship are not-in-family, other-relative, own-child, and unmarried. Next is capital gain and education level, with a capital gain boundary of 7055.5 as the next best split of the data. The important features for education are the levels corresponding to non-college graduates. The tree is implying that the third most important feature for over 50k classification is whether of not you got at least a college degree, and the results imply if you did not get at least a college degree you will not make over 50k. Finally, occupation is the last varibale used in the tree.

I could not find how to report importance statists of variables from classification trees using the tree library, only the rpart library.

Training Results

Now I can look at the training accuracy rate. I also want to look at the training confusion matrix

```
#compute confusion matrix
real_train <- train_tree$Over50k
train_preds <- predict(prune_classification_tree, train_tree, type = "class")
table(train_preds, real_train)
```

```
##            real_train
## train_preds    No   Yes
##         No  21536  3676
##         Yes  1118  3832
```

```r
#get error rate of classification tree
err_rate <- mean(train_preds != real_train)
err_rate
```

```
## [1] 0.1589417
```

```r
#get accuracy
training_accuracy_class_tree = 1 - err_rate
```

The training accuracy rate for the classification tree we built is ‘r training_accuracy_class_tree’.

Now I can plot the ROC curve and find the AUC of the training data.

```r
#load package
library(ROCR)
```

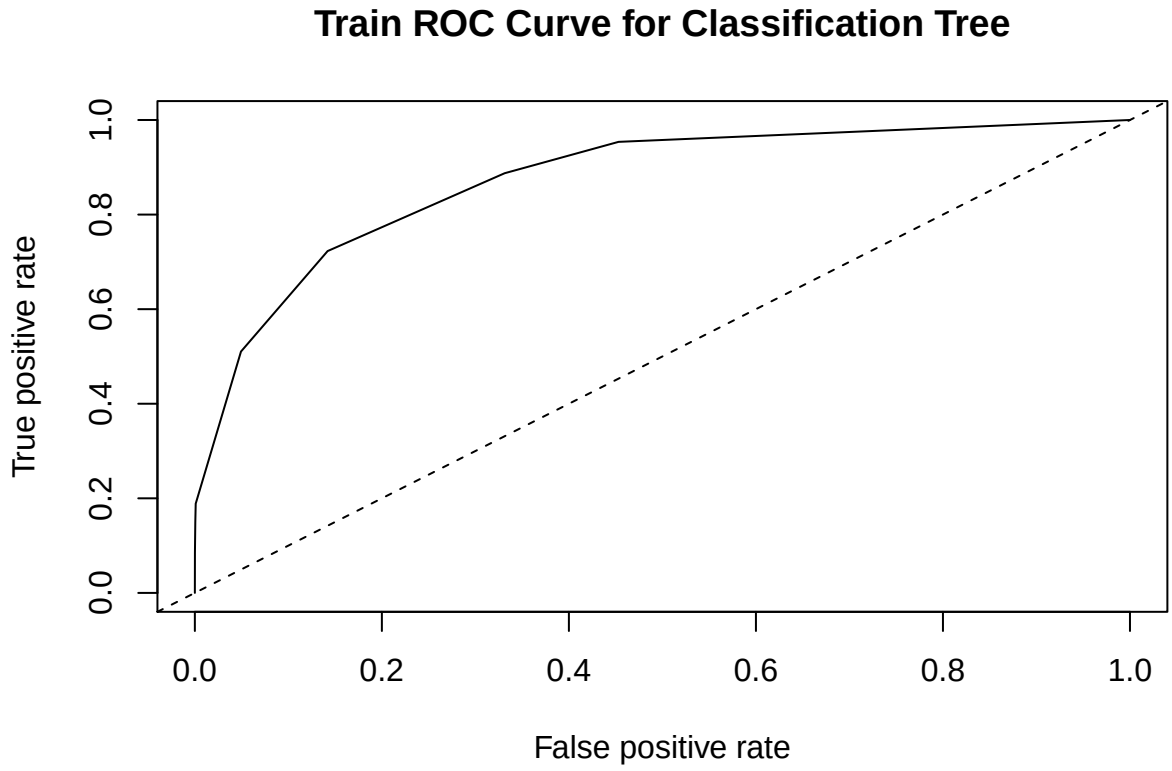
```
## Loading required package: gplots
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

```r
#prepare roc plot
train_probs <- predict(prune_classification_tree, train_tree)
train_prediction <- prediction(train_probs[,2], real_train)
train_performance <- performance(train_prediction, measure = "tpr", x.measure = "fpr")
```

```r
#plot roc
plot(train_performance, main = "Train ROC Curve for Classification Tree")
abline(a=0, b=1, lty=2)
```



```r
#get train auc
class_tree_train_auc <- performance(train_prediction, measure="auc")@y.values[[1]]
class_tree_train_auc
```

```
## [1] 0.8729808
```

The training auc for the classification tree is ‘r class_tree_train_auc’.

Bagged Tree

Build Model

```r
#load packages
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
```

```r
library(caret)
```

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

```r
#prepare data
train_bag = train[, -ncol(train)]
test_bag = test[, -ncol(test)]
train_bag_response = train_bag$Over50k
test_bag_response = test_bag$Over50k
```

There are several ways on how to fit a bagged tree model in R. We decided to use the randomForest package and utilize the randomForest function. Although this function is primarily used in modeling random forests, we took advantage of the fact that bagged trees are actually a special case of random forests where all predictors variables are used as candidates at each split.

In order to help determine which model is the best, we began hypertuning the number of trees via five-fold cross validation. We decided to use a total of 5 ntree values which are given by (100, 300, 500, 700, 900).

```r
#cross validate
ntree = seq(100, 1000, by = 200)
CV_mat = matrix(ncol = 5, nrow = 5)
rownames(CV_mat) = ntree
colnames(CV_mat) = paste0("Fold", 1:5)
set.seed(200)
folds = createFolds(train_bag_response, k = 5)
for (j in 1:length(ntree)) {
  err_rate = numeric(5)
  for (i in 1:length(folds)) {
    fold = folds[[i]]
    train_set = train_bag[-fold, ]
    test_set = train_bag[fold, ]
```

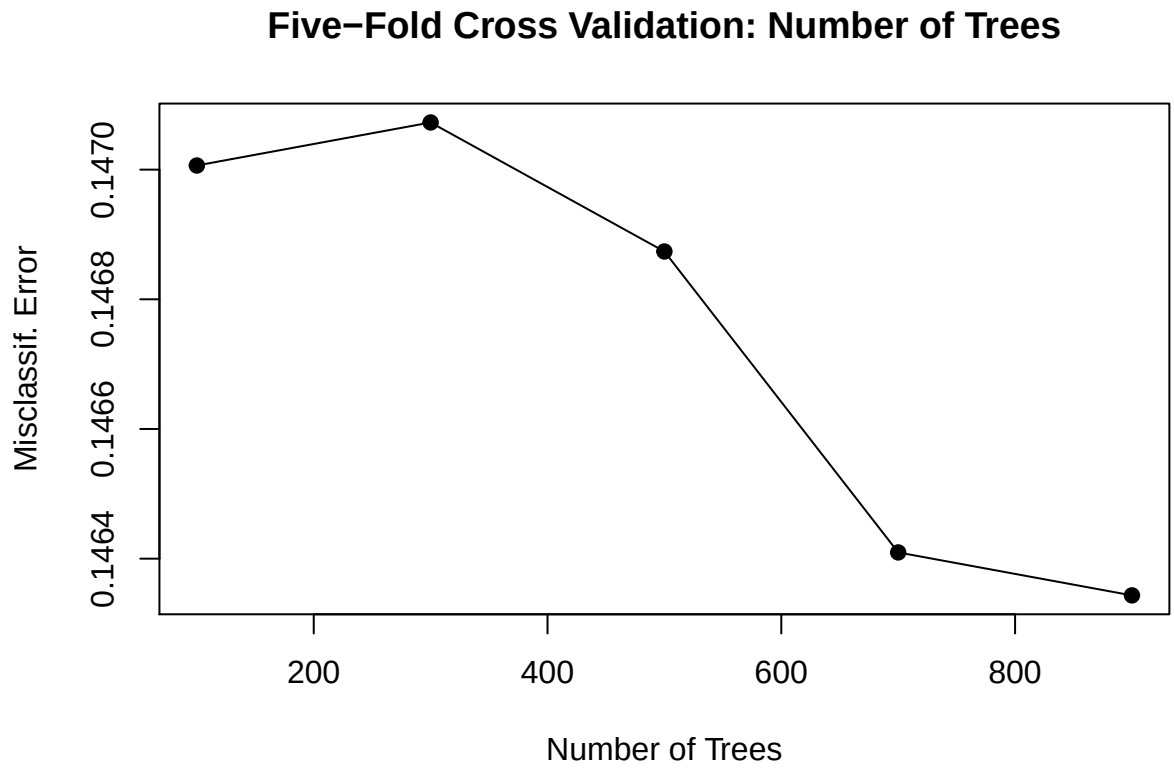
```
    bag_fit = randomForest(Over50k ~., data = train_set, ntree = ntree[j],
                           mtry = ncol(train_bag) - 1, importance = TRUE)
    bag_pred = predict(bag_fit, test_set)
    err_rate[i] = 1 - mean(test_set$Over50k == bag_pred)
  }
  CV_mat[j,] = err_rate
}
CV_mat = cbind(CV_mat, rowMeans(CV_mat))
colnames(CV_mat) = c(paste0("Fold", 1:5), "Average")
```

```
CV_mat
```

```
##         Fold1     Fold2     Fold3     Fold4     Fold5   Average
## 100 0.1457228 0.1515252 0.1501989 0.1418863 0.1456987 0.1470064
## 300 0.1483753 0.1492042 0.1515252 0.1418863 0.1443726 0.1470727
## 500 0.1452255 0.1480438 0.1535146 0.1430466 0.1445384 0.1468738
## 700 0.1448939 0.1498674 0.1505305 0.1423836 0.1443726 0.1464096
## 900 0.1460544 0.1497016 0.1516910 0.1410575 0.1432123 0.1463434
```

Here are the results of the cross-validation, where the last column shows the average missclassification rate for each values of ntree. As we can see, some fits yield a lower rates than others.

```
plot(as.numeric(rownames(CV_mat)), CV_mat[,6], type = "p", pch = 19,
     xlab = "Number of Trees", ylab = "Misclassif. Error",
     main = "Five-Fold Cross Validation: Number of Trees")
lines(as.numeric(rownames(CV_mat)), CV_mat[,6])
```



We decided to plot the number of trees against average CV-missclassification error. According to our plot, ntree = 900 performed the best. We see as the number of trees grows, our misclassification rate decreases. We expected this to be the case since higher levels of complexity.

```r
sort(CV_mat[,6])[1:2]
```

```
##       900       700
## 0.1463434 0.1464096
```

In order to ensure ourselves that we are not overfitting, we will choose the two values of ntree that yielded the best results from cross validation. The values that correspond to these are ntree = 700, 900.

Traiing results

```r
AUC_vec = numeric(2)
AUC_vec1 = numeric(2)
train_misclass_vec = numeric(2)
test_misclass_vec = numeric(2)
test_sensitivity = numeric(2)
test_specificity = numeric(2)
```

```r
set.seed(100)
bag_fit = randomForest(Over50k ~., data = train_bag, ntree = 700,
                       mtry = ncol(train_bag) - 1, importance = TRUE)

#Training Set Classification Error
bag_pred = predict(bag_fit, newdata = train_bag)
train_misclass_vec[1] = mean(bag_pred != train_bag_response)
train_probabilities = predict(bag_fit, newdata = train_bag, type = "prob")
prediction_train = prediction(train_probabilities[,2], train_bag_response)
performance_train = performance(prediction_train, measure = "tpr", x.measure = "fpr")
AUC_vec1[1] = performance(prediction_train, measure="auc")@y.values[[1]]


#Test Set Classification Error
bag_pred = predict(bag_fit, newdata = test_bag)
test_misclass_vec[1] = mean(bag_pred != test_bag_response)


confusion_mat = table(bag_pred, test_bag_response)
test_sensitivity[1] = confusion_mat[2, 2]/(confusion_mat[2, 2] + confusion_mat[1, 2])
test_specificity[1] = confusion_mat[1, 1]/(confusion_mat[1, 1] + confusion_mat[2, 1])


test_probabilities = predict(bag_fit, newdata = test_bag, type = "prob")
prediction_test = prediction(test_probabilities[,2], test_bag_response)
performance_test = performance(prediction_test, measure = "tpr", x.measure = "fpr")
AUC_vec[1] = performance(prediction_test, measure="auc")@y.values[[1]]
```

```r
set.seed(100)
bag_fit = randomForest(Over50k ~., data = train_bag, ntree = 900,
                       mtry = ncol(train_bag) - 1, importance = TRUE)
summary(bag_fit)
```

```
##                 Length Class  Mode
## call                 6 -none- call
## type                 1 -none- character
## predicted        30162 factor numeric
```

```
## err.rate         2700  -none- numeric
## confusion           6  -none- numeric
## votes           60324  matrix numeric
## oob.times       30162  -none- numeric
## classes             2  -none- character
## importance         52  -none- numeric
## importanceSD       39  -none- numeric
## localImportance     0  -none- NULL
## proximity           0  -none- NULL
## ntree               1  -none- numeric
## mtry                1  -none- numeric
## forest             14  -none- list
## y               30162  factor numeric
## test                0  -none- NULL
## inbag               0  -none- NULL
## terms               3  terms  call
```

```
#Training Set Classification Error
bag_pred = predict(bag_fit, newdata = train_bag)
train_misclass_vec[2] = mean(bag_pred != train_bag_response)
train_probabilities = predict(bag_fit, newdata = train_bag, type = "prob")
prediction_train = prediction(train_probabilities[,2], train_bag_response)
performance_train = performance(prediction_train, measure = "tpr", x.measure = "fpr")
AUC_vec1[2] = performance(prediction_train, measure="auc")@y.values[[1]]


#Test Set Classification Error
bag_pred = predict(bag_fit, newdata = test_bag)
test_misclass_vec[2] = mean(bag_pred != test_bag_response)

#Confusion Matrix
confusion_mat = table(bag_pred, test_bag_response)
test_sensitivity[2] = confusion_mat[2, 2]/(confusion_mat[2, 2] + confusion_mat[1, 2])
test_specificity[2] = confusion_mat[1, 1]/(confusion_mat[1, 1] + confusion_mat[2, 1])

#Find AUC values
test_probabilities = predict(bag_fit, newdata = test_bag, type = "prob")
prediction_test = prediction(test_probabilities[,2], test_bag_response)
performance_test = performance(prediction_test, measure = "tpr", x.measure = "fpr")
AUC_vec[2] = performance(prediction_test, measure="auc")@y.values[[1]]
```

Bagged Tree Selection

For both fits, we created a vector that stored the values of training accuracy rate, test accuracy rate, test sensitivity, test specificity, test AUC, and train AUC. We then created a matrix that allowed us to facilitate comparisons amongst the three bagged trees.

```
comparison_mat = cbind(train_misclass_vec, test_misclass_vec, test_sensitivity, test_specificity, AUC_v
rownames(comparison_mat) = c(700, 900)
colnames(comparison_mat) = c("train misclass.", "test misclass.", "test sensitivity",
                             "test specificity", "test AUC", "train AUC")


comparison_mat
```

```
##     train misclass. test misclass. test sensitivity test specificity
## 700     3.31543e-05      0.1512616        0.6218919        0.9226232
## 900     3.31543e-05      0.1507968        0.6229730        0.9228873
##      test AUC train AUC
## 700 0.9022351         1
## 900 0.9025868         1
```

Training Results

```
comparison_mat[,c(1,6)]
```

```
##     train misclass. train AUC
## 700     3.31543e-05         1
## 900     3.31543e-05         1
```

Final Fit

```
set.seed(100)
bag_fit = randomForest(Over50k ~., data = train_bag, ntree = 900,
                       mtry = ncol(train_bag) - 1, importance = TRUE)
summary(bag_fit)
```

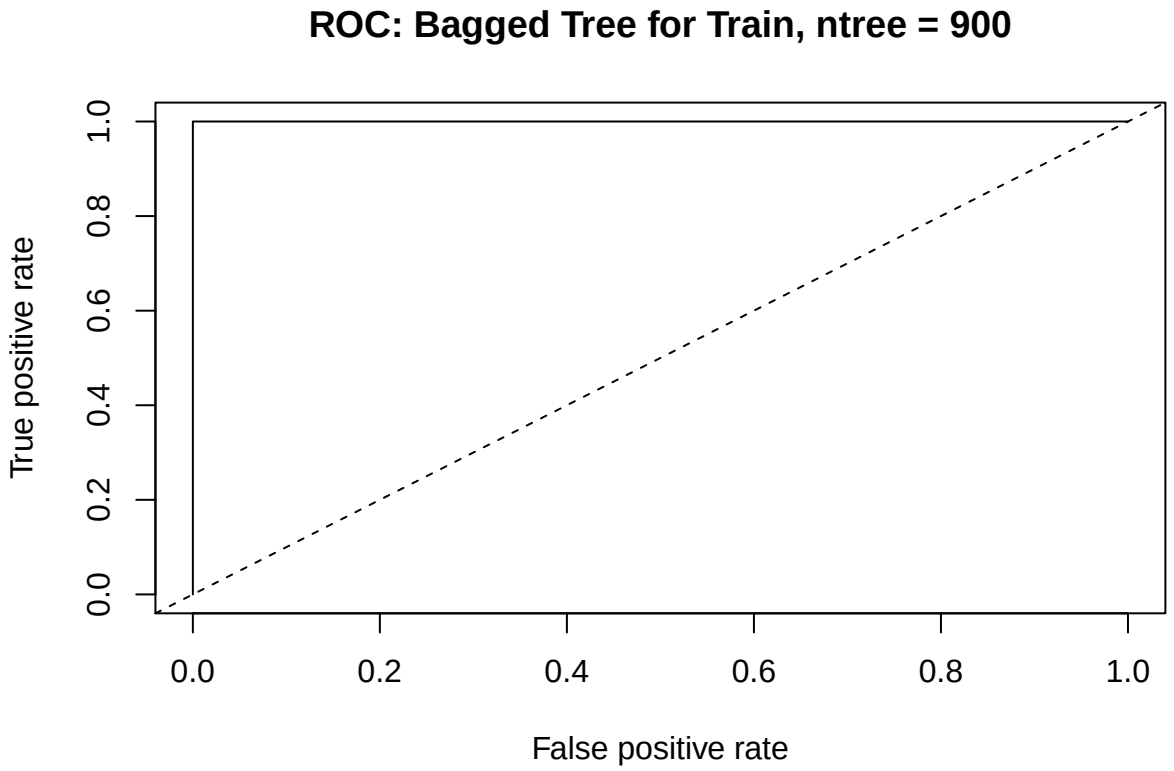
```
##                 Length Class  Mode
## call                 6 -none- call
## type                 1 -none- character
## predicted        30162 factor numeric
## err.rate          2700 -none- numeric
## confusion            6 -none- numeric
## votes            60324 matrix numeric
## oob.times        30162 -none- numeric
## classes              2 -none- character
## importance          52 -none- numeric
## importanceSD        39 -none- numeric
## localImportance      0 -none- NULL
## proximity            0 -none- NULL
## ntree                1 -none- numeric
## mtry                 1 -none- numeric
## forest              14 -none- list
## y                30162 factor numeric
## test                 0 -none- NULL
## inbag                0 -none- NULL
## terms                3 terms  call
```

```
bag_pred = predict(bag_fit, newdata = test_bag)
```

According to our results, running a bagged tree with ntree = 900 provided the most reasonable results and predictions. From the two most accurate fits via 5-fold cross validation, the aforementioned model yielded that best train misclassification rate, test misclassification rate, test sensitivity, test specificity, test AUC, and train AUC. We considered choosing ntree = 700 in order to decrease our chances of overfitting; however, this fit did not perform nearly as well as the other.

```
#plot roc
train_probabilities = predict(bag_fit, newdata = train_bag, type = "prob")
```

```
prediction_train = prediction(train_probabilities[,2], train_bag_response)
performance_train = performance(prediction_train, measure = "tpr", x.measure = "fpr")
plot(performance_train, main = "ROC: Bagged Tree for Train, ntree = 900")
abline(a = 0, b = 1, lty = 2)
```

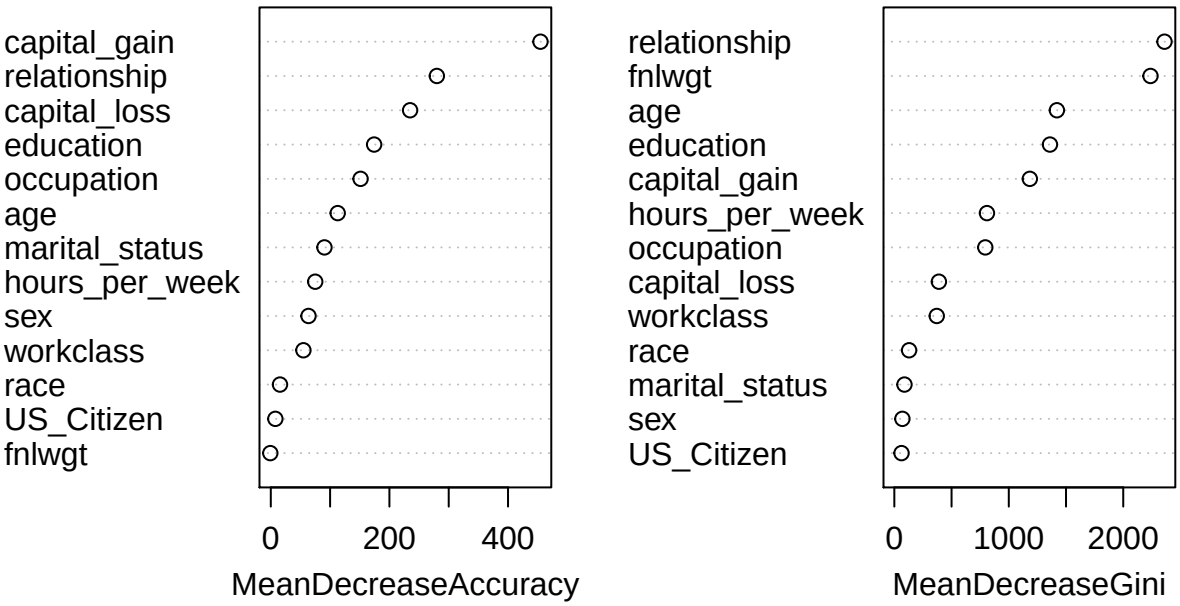


The variable importance of the bagged trees are found below:

```
imp_data = as.data.frame(importance(bag_fit))
imp_data[order(imp_data$MeanDecreaseGini, decreasing = TRUE)[1:6],]
```

```
##                         No         Yes MeanDecreaseAccuracy
## relationship   149.731629 108.3809724          280.0129225
## fnlwgt          -0.818157  -0.2805546           -0.8028489
## age             12.259746 157.2065249          112.8529110
## education      100.615105 149.3574302          174.2686156
## capital_gain   346.713111 382.9122489          454.7111638
## hours_per_week  22.943279  84.8629305           74.9176787
##                MeanDecreaseGini
## relationship         2360.3956
## fnlwgt               2238.2162
## age                  1420.3148
## education            1359.6727
## capital_gain         1183.9115
## hours_per_week        809.2784
```

```
varImpPlot(bag_fit)
```



According to the MeanDecreaseGini measure, the relationship variable was the most important. We were interested to see that fnlwgt was second, since we did not anticipate this variable gaining such a high value for MeanDecreaseGini. Some other important variables were age, education, capital_gain, and hours_per_week.

Random Forest

Build Model

```
#load packages
library(randomForest)
library(caret)
```

```
#prepare data
train_forest <- train[, -ncol(train)]
```

We are going to build a random forest using the randomForest package in r. The important distinction between random forest and bagged trees is that random forest takes a random subset of the features to build when splitting each node, unlike bagging, that uses all features. This means on of the most important parameters to tune for our random forest is the number of features to use, or mtyr in r. The randomForest package has a method called rfcv that performs cross validation in order to tune the value of mtyr. There are two issues with the rfcv function, for one, it allows for the maximum number of features as a possible value for mtyr, but we specifically don't want to use all of the features to make our ranfom forest distinct from our bagged tree. The second issue with rfcv is that it is hard to tune the number of trees used during the random forest estimation. I attempted to loop though the number of trees, then run rfcv on each number of trees, but this was taking a very long time. You can see the details and results of me trying rfcv in the R script file and results and plot files, or look at the separate random-forest.Rmd file in the sub-reports folder.

Instead of rfcv I am going to use the function tuneRF to tune the number of trees and number of features to

use for the random forest.

```
#seperate predictors and response
train_predictors <- train_forest[, -ncol(train_forest)]
train_response <- train_forest$Over50k
```

```
#set seed
set.seed(200)
#number of trees to look at
ntree <- c(50, 100, 500, 1000, 1500)
#how I will store the results
cv_list <- as.list(rep(0, 5))
names(cv_list) <- ntree
i <- 1
#loop through number of trees and run tuneRF to look at combination of mtyr and number of trees
for (tree in ntree) {
  tune_param <- tuneRF(train_predictors, train_response, ntreeTry = tree, trace = FALSE, plot = FALSE)
  cv_list[[i]] <- tune_param
  i <- i + 1
}
```

```
## 0.003301108 0.05
## -0.06366423 0.05
## -0.011597 0.05
## -0.05146171 0.05
## -0.0004873294 0.05
## -0.05165692 0.05
## 0.008464329 0.05
## -0.04353083 0.05
## 0.002679006 0.05
## -0.04627375 0.05
```

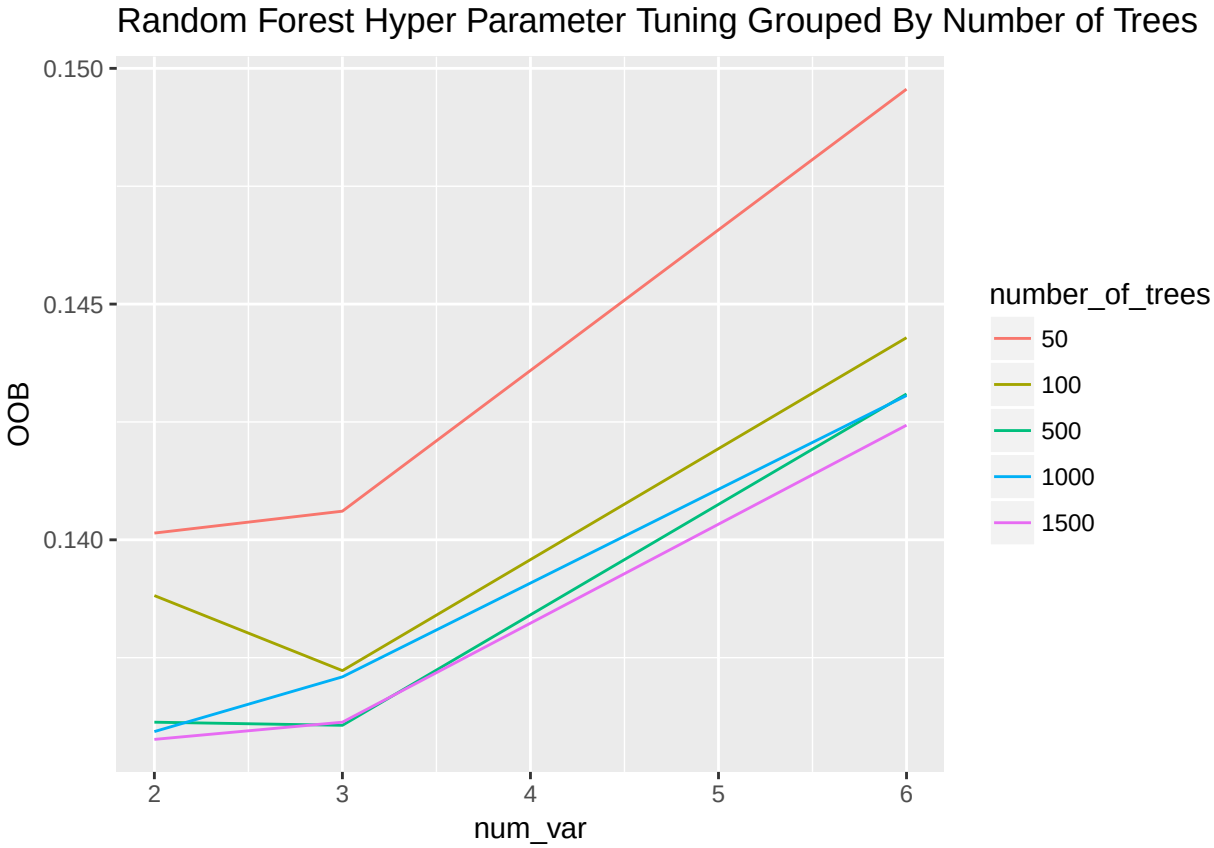
```
#prepare cv results for plotting
library(stringr)
matrix <- matrix(rep(0, 3*5*3), nrow = 15, ncol = 3)
index <- 0
for (i in 1:5) {
  cur_cv <- cv_list[i]
  errs <- cv_list[[i]][,2]
  for (j in 1:length(errs)){
    cur_err <- cv_list[[i]][,2][[j]]
    val <- str_sub(names(cv_list[[i]][,2][j]), 1, 1)
    val <- as.numeric(val)
    matrix[index + j, ] <- c(ntree[i], val, cur_err)
  }
  index <- index + j
}
cv_df <- data.frame(matrix)
names(cv_df) <- c("ntree", "num_var", "OOB")
cv_df
```

```
##    ntree num_var       OOB
## 1     50       2 0.1401432
## 2     50       3 0.1406074
## 3     50       6 0.1495590
## 4    100       2 0.1388171
```

```
## 5    100        3 0.1372256
## 6    100        6 0.1442875
## 7    500        2 0.1361316
## 8    500        3 0.1360652
## 9    500        6 0.1430940
## 10  1000        2 0.1359326
## 11  1000        3 0.1370930
## 12  1000        6 0.1430608
## 13  1500        2 0.1357669
## 14  1500        3 0.1361316
## 15  1500        6 0.1424309
```

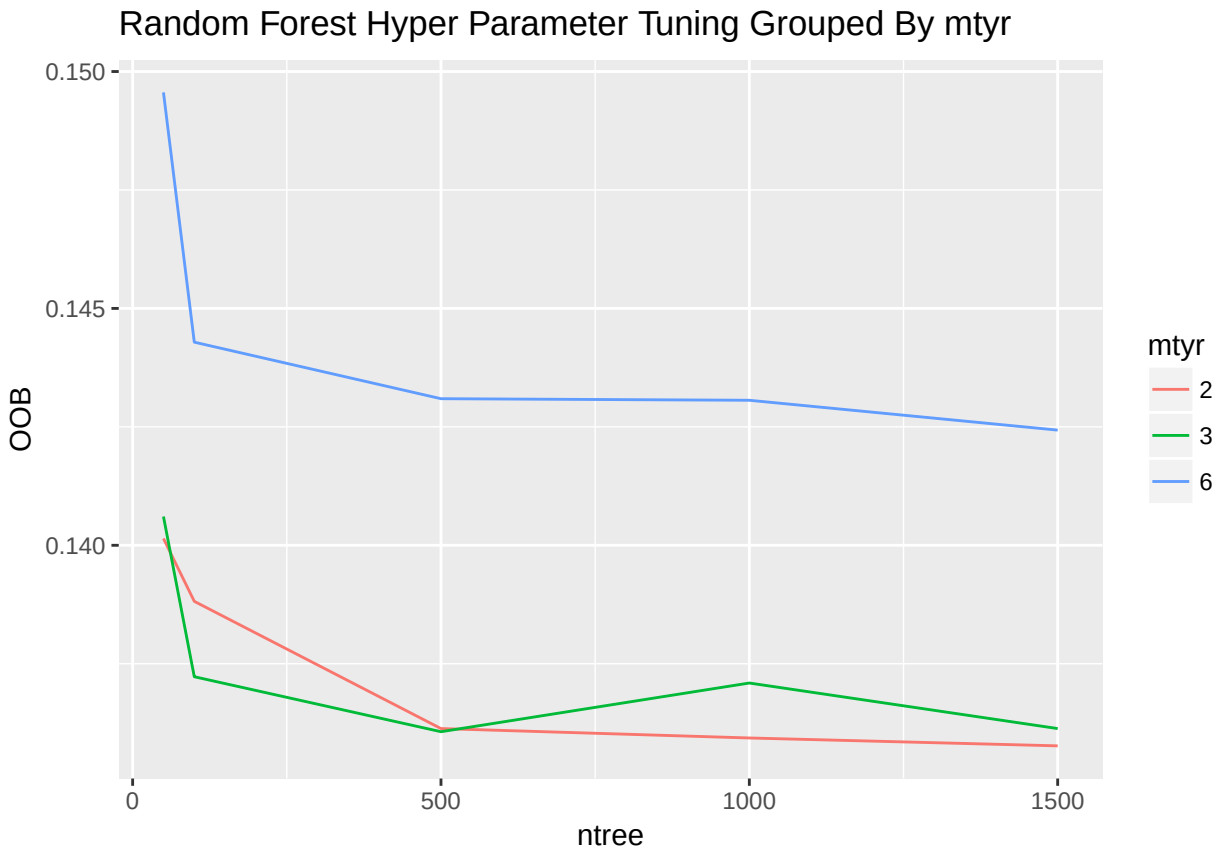
Here is the paramater tuning results grouped by number of trees.

```
cv_df$number_of_trees <- factor(cv_df$ntree)
ggplot(data = cv_df, aes(x = num_var, y = OOB, color = number_of_trees)) + geom_line() + ggtitle("Rando
```



Here is the paramater tuning results grouped by number of variables.

```
cv_df$mtyr <- factor(cv_df$num_var)
ggplot(data = cv_df, aes(x = ntree, y = OOB, color = mtyr)) + geom_line() + ggtitle("Random Forest Hype
```




```
oobs <- cv_df$OOB
row_num <- as.numeric(which.min(oobs))
num_tree <- cv_df[row_num, 1]
num_var <- cv_df[row_num, 2]
```

The best parameter combination based on our hyperparameter tuning restuls are number of trees is 'r num_tree' and number of features is 'r num_var'. These are based on the lowest OOB error rate during the cross validation process.

Now I can build the random forest using these parameters.

```
set.seed(100)
random_forest <- randomForest(Over50k ~., data = train_forest, ntree = num_tree, mtry = num_var, importa
```

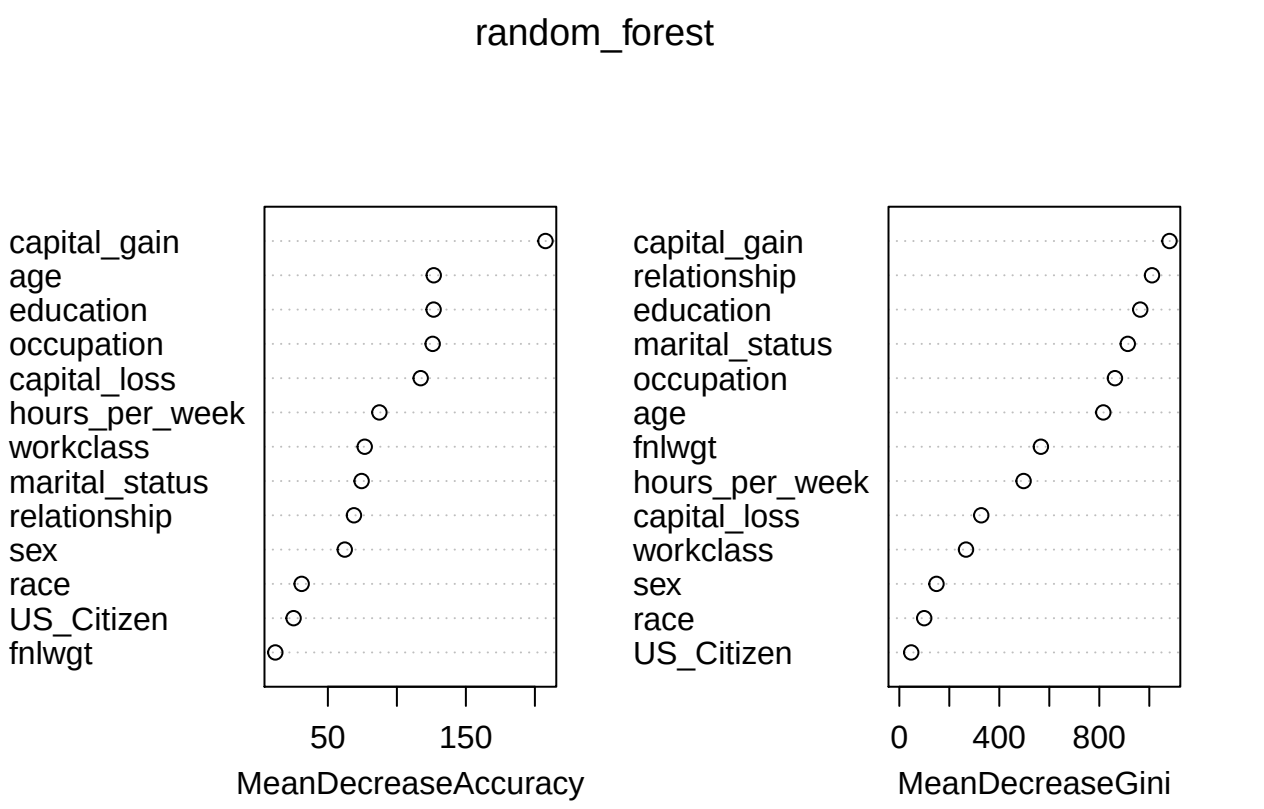
Now I can look at the variable importance using the importance function. Variable importance will be based on the greatest mean decreases gini.

```
#look at variable importance of random forest
var_imp <- data.frame(importance(random_forest))
var_imp
```

```
##                      No         Yes MeanDecreaseAccuracy
## age            2.916181 117.7812947            126.71208
## workclass     67.458005  27.2146935             76.69084
## fnlwgt        13.726050   0.5755618             11.91015
## education     82.102928  96.6243220            126.61319
## marital_status 75.476360  44.6633592             74.44127
## occupation    70.493598  96.1170909            125.93797
## relationship  45.553450  66.4228965             68.89658
```

```
## race            19.108144  20.2750338            31.00323
## sex             44.129040  10.9439606            62.22469
## capital_gain   172.812802 216.5955580           207.66669
## capital_loss    88.886756 115.6337429           117.23932
## hours_per_week  18.625739  84.8279608            87.33325
## US_Citizen      27.569738   1.2870478            25.15558
##                MeanDecreaseGini
## age                   815.99695
## workclass             266.76014
## fnlwgt                566.25831
## education             963.76462
## marital_status        913.81883
## occupation            862.54477
## relationship         1010.80850
## race                   99.57915
## sex                   148.65378
## capital_gain         1080.78006
## capital_loss          327.79417
## hours_per_week        497.17865
## US_Citizen             47.64475
```

```
#lets plot the variable importance based on mean decrease gini and mean decrease accuracy
varImpPlot(random_forest)
```



I am going to base varaible importance on mean decrease gini. Based on the plots above the most important features are capital_gain, relationship, education, matrical_status, occupation, and age, in that order. The variable importance statistics are:

```
var_imp_best_6 <- var_imp[order(var_imp$MeanDecreaseGini, decreasing = TRUE)[1:6],]
var_imp_best_6
```

```
##                         No       Yes MeanDecreaseAccuracy MeanDecreaseGini
## capital_gain   172.812802 216.59556            207.66669        1080.7801
## relationship    45.553450  66.42290             68.89658        1010.8085
## education       82.102928  96.62432            126.61319         963.7646
## marital_status  75.476360  44.66336             74.44127         913.8188
## occupation      70.493598  96.11709            125.93797         862.5448
## age              2.916181 117.78129            126.71208         815.9970
```

The variable importance statistics are:

- capitcal_gain: 1080.7801
- relationship: 1010.8085
- education: 963.7646
- marital_status: 913.8188
- occupation: 862.5448
- age: 815.9970

Training Results

```
#prediction the values of the training set
rf_train_pred <- predict(random_forest, train_forest[, -ncol(train_forest)])
```

```
real_train <- train_forest$Over50k
rf_train_err_rate <- mean(rf_train_pred != real_train)
rf_train_err_rate
```

```
## [1] 0.08938399
```

```
rf_train_acc <- 1 - rf_train_err_rate
rf_train_acc
```

```
## [1] 0.910616
```

The training data accuracy for random forest is ‘r rf_train_acc’.

Here is also the Random Forest training confusion matrix.

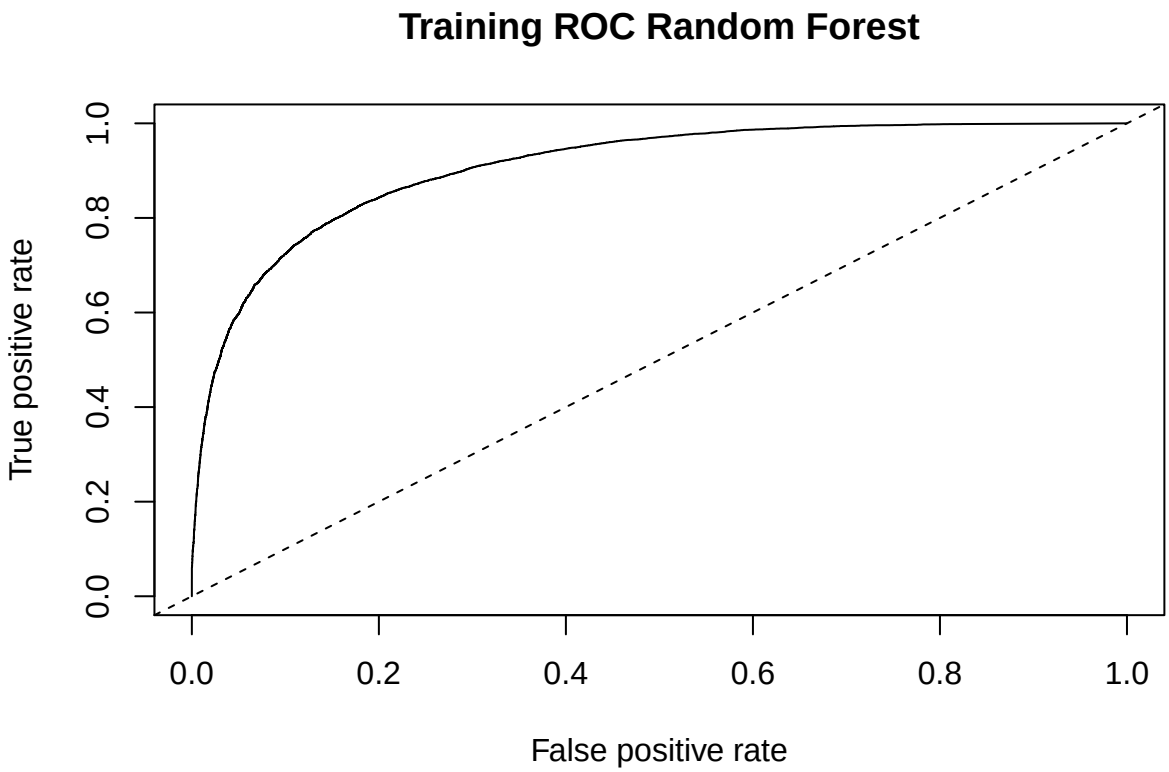
```
print(random_forest$confusion)
```

```
##        No  Yes class.error
## No  21308 1346  0.05941556
## Yes  2764 4744  0.36814065
```

Now we can look at the training ROC and AUC of the Random Forest.

```
#prepare trianing roc
rf_train_probs <- as.vector(random_forest$votes[,2])
rf_prediction <- prediction(rf_train_probs, real_train)
rf_performance <- performance(rf_prediction, measure = "tpr", x.measure = "fpr")
```

```
#plot training roc
plot(rf_performance, main="Training ROC Random Forest")
abline(a=0, b=1, lty = 2)
```




```
train_rf_auc <- performance(rf_prediction, measure="auc")@y.values[[1]]
train_rf_auc
```

```
## [1] 0.9071512
```

The Training AUC of the Random Forest is ‘r train_rf_auc’.

Testing Models

We are going to look at the performance of all of the methods, and then report the best at the end.

Classification Tree

This section will be based around looking at the test performance of the classificaiton tree we build.

Error Rate

```
#prepare test data
test_tree <- test[, -ncol(test)]
test_tree_preds <- test_tree[, -ncol(test_tree)]
real_test <- test_tree$Over50k
```

```
#get error rate
ct_test_preds <- predict(prune_classification_tree, test_tree_preds, type = "class")
ct_test_err_rate <- mean(ct_test_preds != real_test)
ct_test_err_rate
```

```
## [1] 0.1610226
```

```
#get test accuracy
ct_test_acc <- 1 - ct_test_err_rate
ct_test_acc
```

```
## [1] 0.8389774
```

The test accuracy of our classification tree is ‘r ct_test_acc’.

Confusion Matrix

```
#get classification tree confusion matrix
class_tree_confusionMatrix <- table(ct_test_preds, real_test)
class_tree_confusionMatrix
```

```
##              real_test
## ct_test_preds    No   Yes
##           No  10772  1837
##           Yes   588  1863
```

Specificity and Sensitivity

```
#calculate sensitivity
ct_sensitivity <- class_tree_confusionMatrix[2, 2]/(class_tree_confusionMatrix[2, 2] + class_tree_confus
ct_sensitivity
```

```
## [1] 0.5035135
```

```
#calculate specificity
ct_specificity <- class_tree_confusionMatrix[1, 1]/(class_tree_confusionMatrix[1, 1] + class_tree_confus
ct_specificity
```

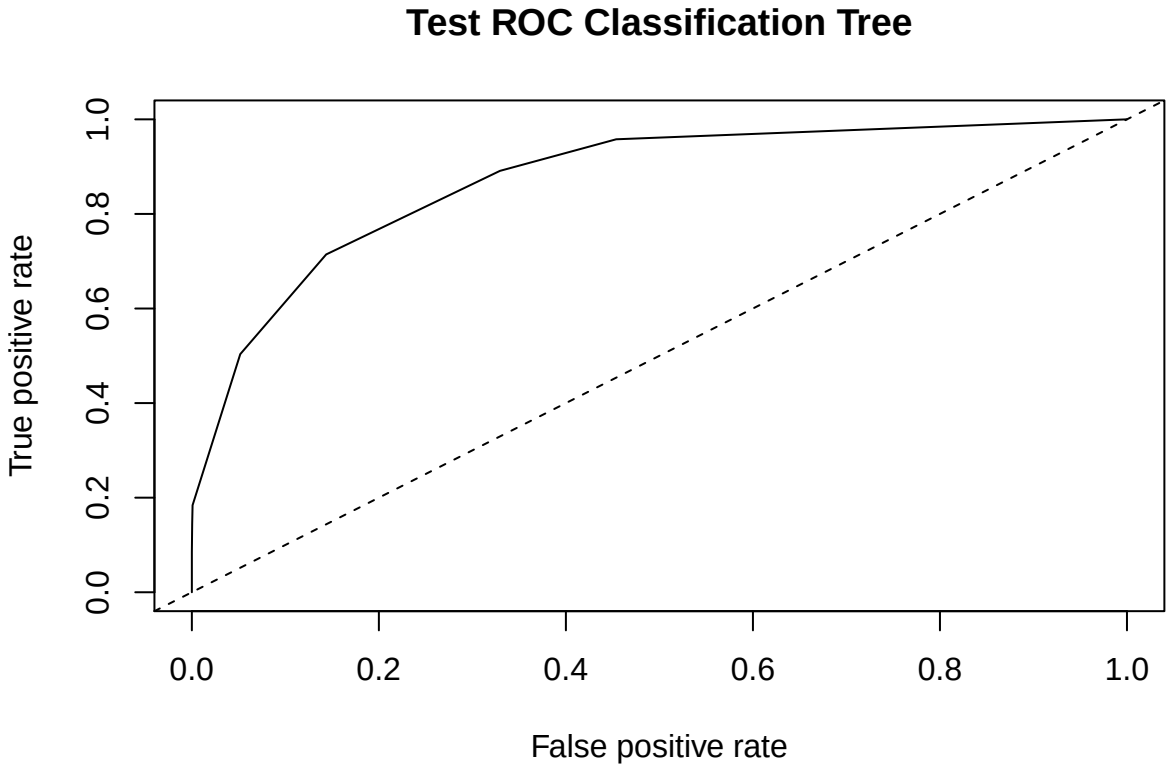
```
## [1] 0.9482394
```

The classification tree sensitivity is ‘r ct_sensitivity’ and the classification tree specificity is ‘r ct_specificity’. We have a low sensitivity and high specificity.

ROC and AUC

```
#prepare roc curve
ct_test_probs <- predict(prune_classification_tree, test_tree_preds)
ct_test_prediction <- prediction(ct_test_probs[,2], real_test)
ct_test_performance <- performance(ct_test_prediction,  measure = "tpr", x.measure = "fpr")
```

```
#plot roc curve
plot(ct_test_performance, main="Test ROC Classification Tree")
abline(a=0, b=1, lty=2)
```




```
#report test auc
ct_test_auc <- performance(ct_test_prediction, measure="auc")@y.values[[1]]
ct_test_auc
```

```
## [1] 0.8723657
```

The test auc for the classification tree is 'r ct_test_auc'.

Bagged Tree

Note that this section will look a little different than the previous sections. Jack did the work on the classification tree and random forest, while Jaime did the work on the bagged tree, and the structure of our code was different. The variables that make the these results are actually found in the training section of the bagged train.

Error Rate

Test Results

```
#test results for the tune different parameters we picked
comparison_mat[,2:5]
```

```
##     test misclass. test sensitivity test specificity  test AUC
## 700      0.1512616        0.6218919        0.9226232 0.9022351
## 900      0.1507968        0.6229730        0.9228873 0.9025868
```

```
#test results for the fit we picked
comparison_mat[2,]
```

```
##  train misclass.   test misclass. test sensitivity test specificity
```

```
##     0.0000331543     0.1507968127     0.6229729730     0.9228873239
##         test AUC        train AUC
##     0.9025867553     0.9999999971
```

Confusion Matrix

```
#Confusion Matrix
(confusion_mat = table(bag_pred, test_bag_response))
```

```
##         test_bag_response
## bag_pred    No   Yes
##      No  10482  1394
##      Yes   878  2306
```

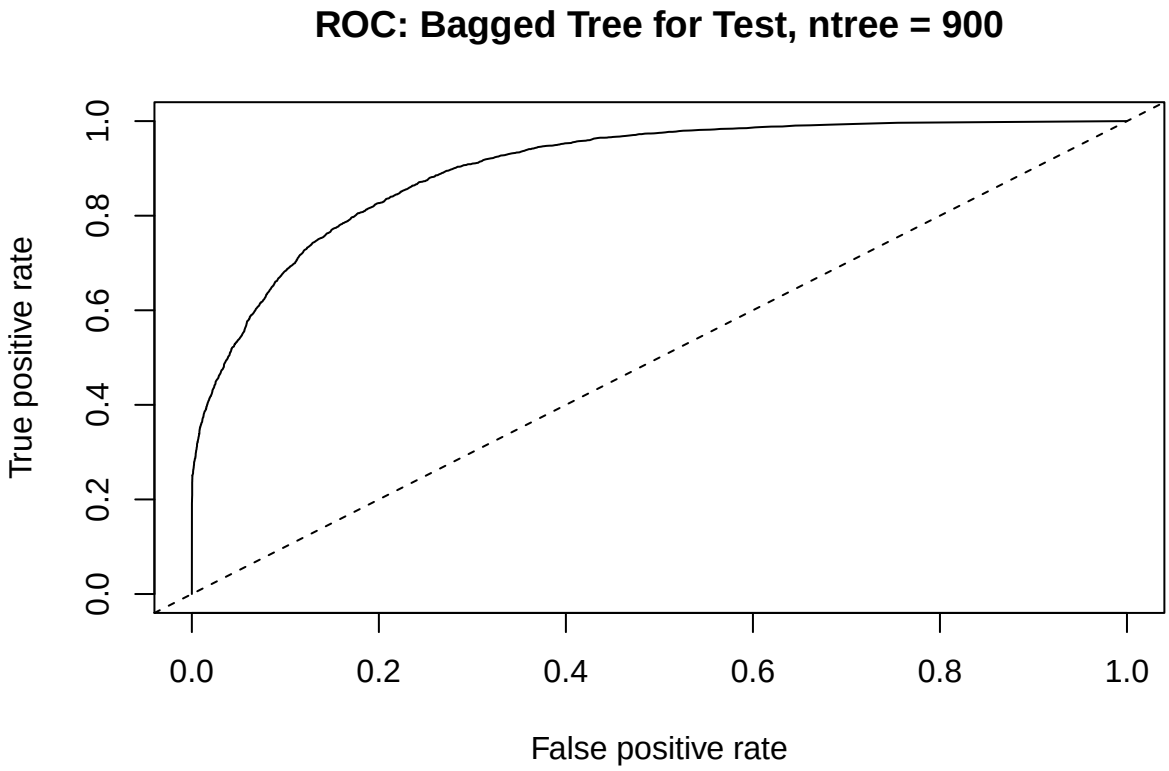
Specificity and Sensitivity

Look above for specificity and sensitivity

ROC and AUC

AUC is reported in an above section

```
test_probabilities = predict(bag_fit, newdata = test_bag, type = "prob")
prediction_test = prediction(test_probabilities[,2], test_bag_response)
performance_test = performance(prediction_test, measure = "tpr", x.measure = "fpr")
plot(performance_test, main = "ROC: Bagged Tree for Test, ntree = 900")
abline(a = 0, b = 1, lty = 2)
```



Random Forest

```
#prepare data
test_forest <- test[, -ncol(test)]
test_forest_preds <- test_forest[, -ncol(test_forest)]
real_test <- test_forest$Over50k
```

Error Rate

```
#get test error rate
rf_test_preds <- predict(random_forest, test_forest_preds)
rf_test_err_rate <- mean(rf_test_preds != real_test)
rf_test_err_rate
```

```
## [1] 0.1394422
```

```
#get test acc
rf_test_acc <- 1 - rf_test_err_rate
rf_test_acc
```

```
## [1] 0.8605578
```

The random forest test accuracy is ‘r rf_test_acc’.

Confusion Matrix

```
#get test confusion matrix
rf_test_confusionMatrix <- table(rf_test_preds, real_test)
rf_test_confusionMatrix
```

```
##              real_test
## rf_test_preds    No   Yes
##           No  10638  1378
##           Yes   722  2322
```

Specificity and Sensitivity

```
# get test sensitivity
rf_test_sensitivity <- rf_test_confusionMatrix[2, 2]/(rf_test_confusionMatrix[2, 2] + rf_test_confusionM
rf_test_sensitivity
```

```
## [1] 0.6275676
```

```
#get test specificity
rf_test_specificity <- rf_test_confusionMatrix[1, 1]/(rf_test_confusionMatrix[1, 1] + rf_test_confusionM
rf_test_specificity
```

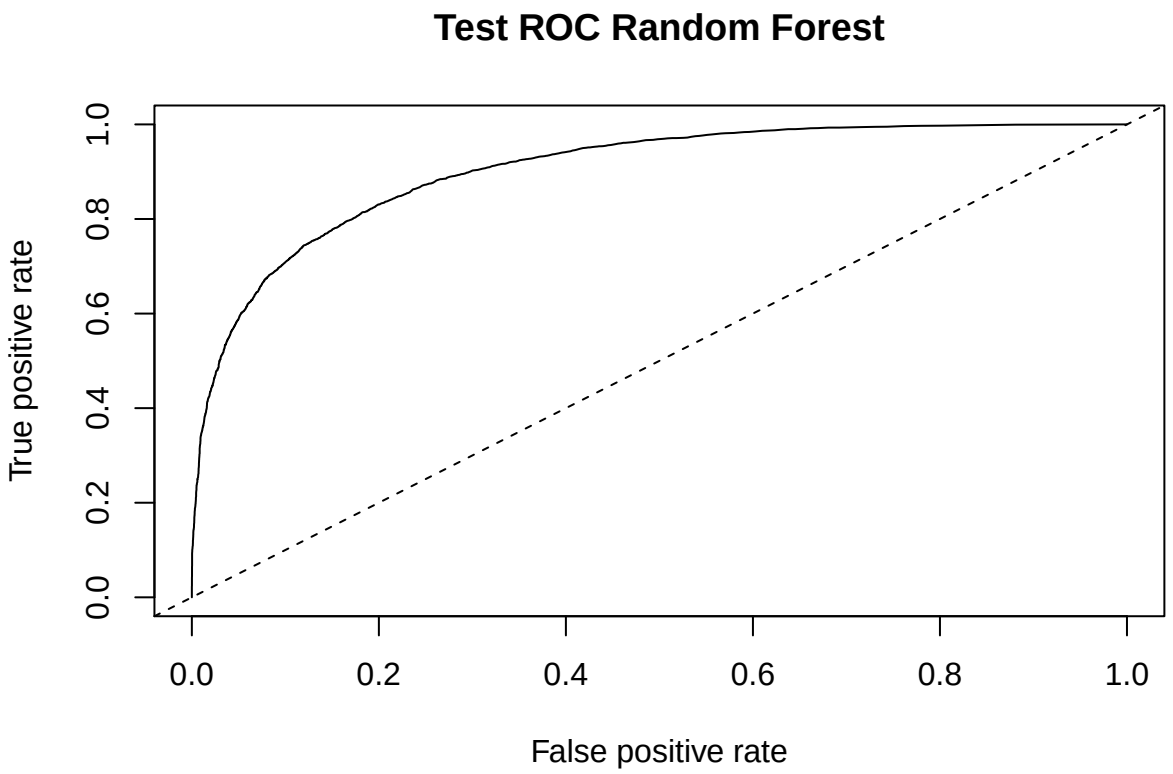
```
## [1] 0.9364437
```

The random forest sensitivity is ‘r rf_test_sensitivity’ and specificity is ‘r rf_test_specificity’. We have a pretty high specificity, but a pretty low sensitivity.

ROC and AUC

```
#prepare roc cruve
rf_test_probs <- predict(random_forest, test_forest_preds, type = "prob")
rf_test_prediction <- prediction(rf_test_probs[,2], real_test)
rf_test_performance <- performance(rf_test_prediction,  measure = "tpr", x.measure = "fpr")
```

```
#plot random forest roc
plot(rf_test_performance, main="Test ROC Random Forest")
abline(a=0, b=1, lty=2)
```




```
#get test auc
rf_test_auc <- performance(rf_test_prediction, measure="auc")@y.values[[1]]
rf_test_auc
```

```
## [1] 0.9031879
```

The random forest test AUC was ‘r rf_test_auc’.

Best Model

Our Random Forest performed the best out of the 3 models. The random forest had the lower test error rate, greatest test AUC, greatest sensitivity, and greatest specificity of all 3 models, making it far an away the best model based on the performance of the test data.

The most important variables for our random forest with variable importance statistic were:

- capitcal_gain: 1080.7801
- relationship: 1010.8085

- education: 963.7646
- marital_status: 913.8188
- occupation: 862.5448
- age: 815.9970

Conclusion

In this report we used the Census Income data set to find the best model to predict whether a certain person makes over $50,000 in yearly earnings, and found the most important features in classifying this prediction. Using the training data set from the adult.data file, we trained a classification tree, bagged tree, and random forest, and then tested each model on the test data set from the adult.test file. For each model we tuned certain hyperparamters using cross validation. For the classification tree we tuned the size, or number of terminal nodes for the tree. Using this parameter, we fit a pruned classification tree on the training data, and examined the variable importance via the plot structure of the tree. For the bagged tree we tuned the number of trees used as estimators during the bootstrap estimation process, finding the best number of trees was 900. In the bagged tree we did worry about overfitting, so we also looked at the second best number of trees, 700. Finally, for the random forest we tuned both the number of trees used as estimators and the maximum number of features used for each split, making sure that the maximum features we less than the full number of features, as that would have made it a bagged tree. We found that the best combination of parameters was 1500 trees and 2 features.

We then tested our models on both the training on test data, reporting the results of each. Based on the training data, bagged trees seemed to be the best model, however, this may have been due to overfitting. On the test data, however, the random forest model performed the best based on every metric we used to test performance. This implies the random forest reduced the variance captured in the bagged tree, without a huge increase in bias. In the end we determined the random forest was the best predictive model.