Classifying Income Brackets Using Machine Learning Algorithms

Modern Statistical Prediction and Machine Learning — Audrey Webb, Vahan Aslanyan
Professor: Gaston Sanchez-Trujillo — GSI: Johnny Hong

Problem Statement

We are given a raw data set called Census Income Data Set from UC Irvine Machine Learning Repository. Our goal is to predict whether a person makes over 50K a year.

Introduction

We are going to follow these steps to accomplish our goal:

- **Early data analysis**—this step is going to include our first impressions from the data, some preprocessing and exploration including: binning, handling missing values, changing scales, observing outliers, visualization, and dummification.
- **Model fitting** — Once we have a "clean" dataset we can fit models. This stage consists of fitting multiple models— classification trees, bagged trees, and random forests
- **Selecting a model** —We select the best model from our fitted models based on AUC(area under curve) parameter from each method's Receiver Operating Characteristic (ROC) curve.

Data

The data set for this project is the Census Income Data Set donated by Ronny Kohavi and Barry Becker to the UCI Machine Learning Repository. The data set describes 15 variables on a sample of individuals from the US Census database. Our multivariate data set has 14 predictors(both categorical and integers), 48842 observations with missing values. The data was divided into training and testing sets by the original contributors. Our corresponding variables are in Table 1.

Early Data Analysis

Missing data was encoded as ?. Our first step was turning this to NA, using the gsub function. We moved on to bin our training and testing sets as changing the structure of our data was a necessary step. For relevant categorical predictors, we grouped by level. We grouped

Table 1: Data Variables

Variable	Type	Comments
Income	categorical variable	Our response variable, should remain untouched
Age	continuous numeric	Scale
Workclass	categorical variable	Either dummifying or binning (too many levels)
Final Weight:	continuous numeric	Good for estimation, bad for prediction
Education	categorical variable	Bin and then potentially dummify
Education-num	continuous numeric	Provides insights into education
Marital-status	categorical variable	Binning and then dummifying
Occupation	categorical variable	We cannot bin because of our bias
Relationship	categorical variable	Hard to interpret, will keep intact
Race	categorical variable	Pretty straightforward
Sex	categorical variable	Again straightforward
Capital Gain	continuous numeric	Scale
Capital Loss	continuous numeric	Scale
Hours-per-week	continuous numeric	Scale
Native Country	categorical variable	Bin into 2 major groups and dummify

education from very specific year levels into college education and above versus high school education and below (college and non-college). We did the same thing for marital status (grouped them into married, and not married(single)), and for the countries (we grouped them by US and non-US). We removed final weight, as it did not contain relevant information for our purposes, but it can be generally used for population estimation inquiries. We decided to leave occupation as a categorical variable, because grouping them into categories would come down to a judgment problem on authors part and introduce external bias into our problem.For numerical predictors, we used a correlogram to observe correlation. From this, we did not notice strong correlations.

Our next step is handling missing data. Typically, we want our data to be missing at random, but we cannot expect this because some underlying procedure of data gathering usually contributes to missing values. So, as a result, our missing data might be correlated. We had several options for handling missing values; options such as dropping all observations with missing data, imputing them with mode of levels, use k-Nearest Neighbors, use Multivariate Imputation using Chained Equation (MICE), or using the missForest algorithm. The first two options will skew our data, as assumptions of distributions must be made. MICE can run both parametric and nonparametric versions (see comparison of both methods here [1]). K-NN, and missForest are non-parametric methods so they will not make assumptions on underlying distributions. K-NN needs tuning of k(number of neighbors) and has a general complexity of O(k*n*d) where d is our dimension of data and n is the number of observations. MissForest takes a different approach by recasting the missing data problem as a prediction problem. MissForest has been shown to outperform well known methods,

Algorithm 1 Impute missing values with RF.

Require: $\mathbf{X}$ an $n \times p$ matrix, stopping criterion γ

1. Make initial guess for missing values;
2. $\mathbf{k} \leftarrow$ vector of sorted indices of columns in $\mathbf{X}$ w.r.t. increasing amount of missing values;
3. **while** not γ **do**
4. $\quad \mathbf{X}_{\text{old}}^{\text{imp}} \leftarrow$ store previously imputed matrix;
5. $\quad$ **for** s in $\mathbf{k}$ **do**
6. $\quad\quad$ Fit a random forest: $\mathbf{y}_{\text{obs}}^{(s)} \sim \mathbf{x}_{\text{obs}}^{(s)}$;
7. $\quad\quad$ Predict $\mathbf{y}_{\text{mis}}^{(s)}$ using $\mathbf{x}_{\text{mis}}^{(s)}$;
8. $\quad\quad$ $\mathbf{X}_{\text{new}}^{\text{imp}} \leftarrow$ update imputed matrix, using predicted $\mathbf{y}_{\text{mis}}^{(s)}$;
9. $\quad$ **end for**
10. $\quad$ update γ.
11. **end while**
12. **return** the imputed matrix $\mathbf{X}^{\text{imp}}$

Figure 1: MissForest Algorithm

such as k-nearest neighbors and parametric MICE. Thus, we decided to use the missForest algorithm. In addition, it decreases computation time.[2]

MissForest is used to impute missing values particularly in the case of mixed-type data, and can be used to impute continuous and/or categorical data including complex interactions and nonlinear relations. Additionally, it yields an out-of-bag (OOB) imputation error estimate. Figure 1 contains the complete algorithm for the method.
After imputing values we changed the scales for numeric variables. We do this for several purposes. First, some algorithms converge faster when performed on standardized data. Second, it is easier to interpret our findings using means and standard deviations than using independent different-scaled values.

Once we put everything on the same scale we visualized our distributions. An example for numeric variables is shown in Figure 2. We can see for some variables, even after scaling, there is a lot of variability. That makes us question the importance of some of the variables,

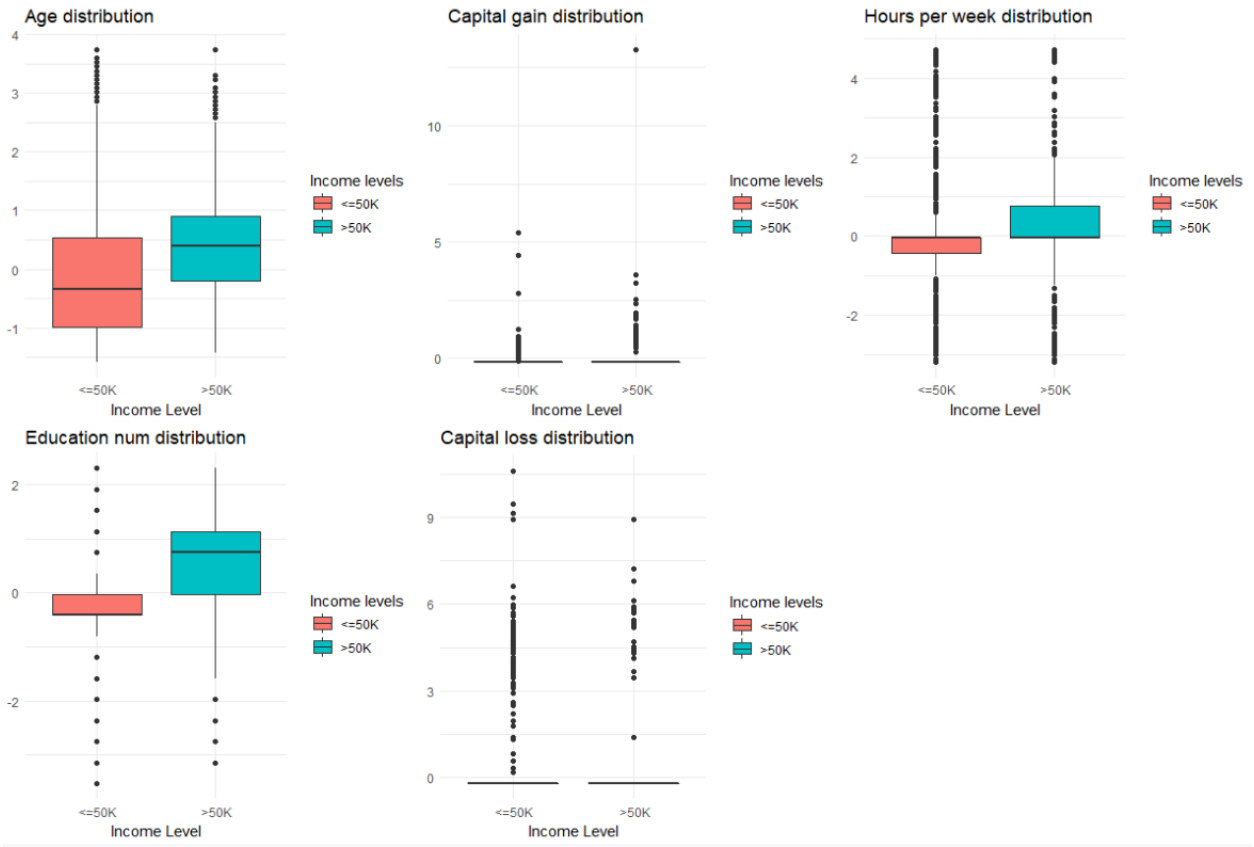


Figure 2: Boxplots of numeric variables based on income

especially of capital gain and loss. However, it might be logical to conjecture that a person with a capital gain might have better chances of making more than $50000, so we leave it there. Graphs on categorical variables reinforce our common knowledge (men make more money than women, white people are better off etc) Complete outputs of the distribution can be found in "Images" Folder.

The last step in early data analysis is dummification. We dummified our data because some built-in R functions are unable to deal with multiple categorical variables. Based on our binning we have 26 predictors after dummifying. Even though random forests should be able to handle categorical values natively, we look for a different implementation so we don't have to encode all of the features and use up all our memory. Some other algorithms (notably **tree()**) can only work with 32 factors.

As a final step we decided to visualize the entire dataset using k-Means clustering. We use this to generally visualize our dataset. We partitioned our data into two partitions(clusters) and the result is shown in Figure 3. We can see there is a clear partition in our data, so we should expect our models to have relatively high accuracy. We also visualize the proportion

of people making more than $50K vs people making less. The axes here represent discriminant coordinates. We can also see that we can partition our data relatively easy, meaning a support vector machine might perform well for our project too.

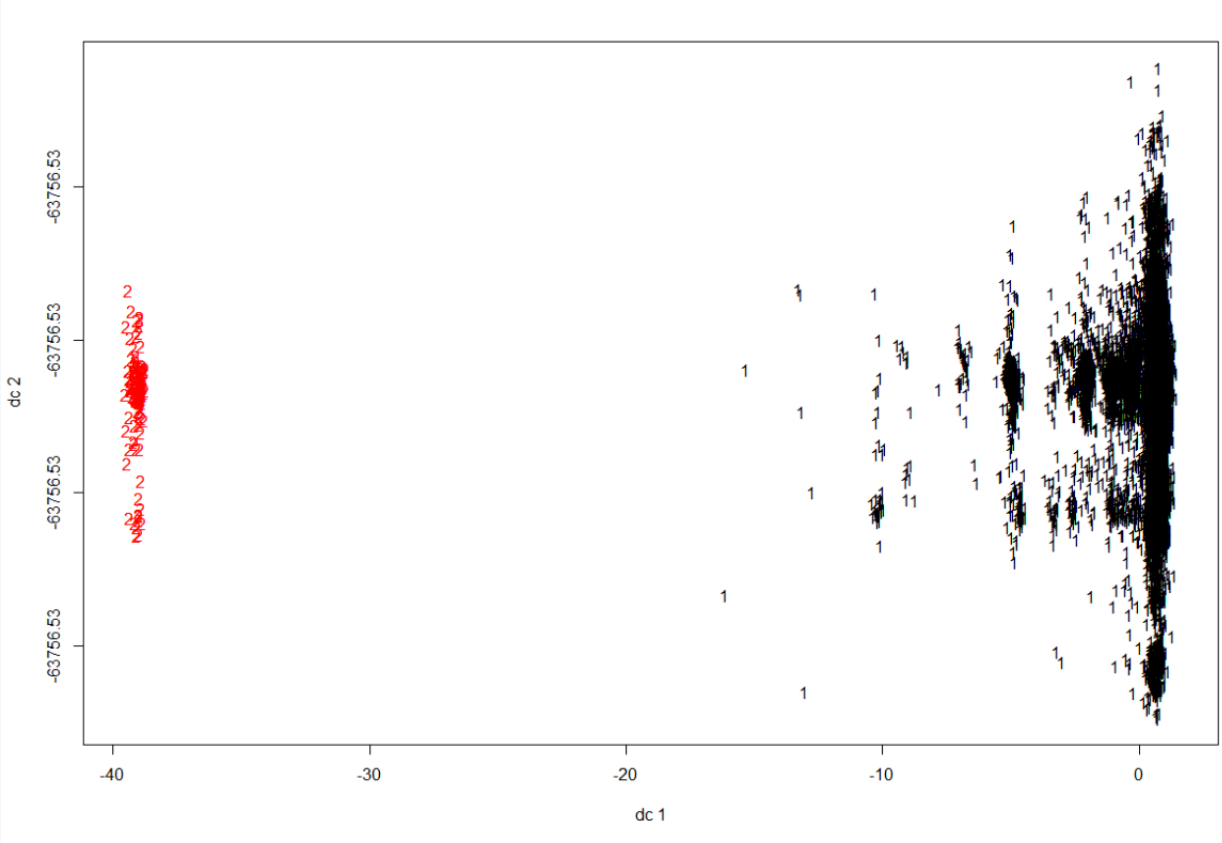


Figure 3: 2 Cluster partition of our data

Model Fitting

Classification trees

Classification tree recursively partitions the space of the independent variables one variable at the time. This method is easier to interpret and deals with different types of data. We use the following code to fit our tree to the data.

```
trctrl<-trainControl(method = "repeatedcv",number=10,repeats = 8,
                     search = "grid", savePred=TRUE,
                     verboseIter = TRUE, classProbs = T)
set.seed(154)
tree_fit <- train(x=training_dummy[,c(1:26)], y=training_dummy[,27],
```

```
method = "rpart",
parms = list(split = "information"),
trControl=trctrl,
tuneLength = 10
)
```

We are using the caret package for all of our model fitting purposes. We are setting three parameters of trainControl() method. The method parameter holds the details about re-sampling method. We used repeated cross validation. The number parameter holds the number of re-sampling iterations. The repeats parameter contains the complete sets of folds to compute for our repeated cross-validation. We are using setting number equal 10 and repeats equal 8. This trainControl() methods returns a list, and we will pass this on to our train() method. We put our savePred to be true so that the model saves our predictions, and the class probabilities to be true so that we will be able to construct ROC plots and extract AUC values later. If we use the formula interface (Y~X), the caret train() function will convert any factor variables in the formula to dummy variables. To preserve their status as factors, so these results match those from the randomForest() model, we will use the non-formula interface.

For training Decision Tree classifier, train() method should be passed with method parameter as rpart. The package rpart is specifically available for decision tree implementation. One difference between the results of train and rpart is that the latter uses the "one SE" method for pruning while train prunes to the depth with the best performance. Caret links its train function with others to make our work simple. We use the split information. Information gain is based on the concept of entropy from information theory. Information gain is used to decide which feature to split on at each step in building the tree. For each node of the tree, the information value "represents the expected amount of information that would be needed to specify whether a new instance should be classified yes or no, given that the example reached that node."[3] Finally, we keep our length to be 10, and then fit the tree. This takes approximately 8-10 minutes. Our final tree is in Figure 4. The nodes represent the variables and splitting values. We can see that the root of our tree is marital status. For married people capital gain the only decisive variable for their income bracket. For non-married people, occupation plays the second important role (our first "branch"). Then capital gain again comes into play, followed by scaled years spend on education and a successive branching by occupations again and capital gains/loss.

We next interpret the tree output. First it specifies that CART algorithm was used. Then it gives us some information about dataset and cross validation and outputs re-sampling results across tuning parameters. cp stands for complexity parameter. The complexity parameter specifies how the cost of a tree is penalized by the number of terminal nodes. Usually a small cp results in large trees and potential over-fitting. The next parameter is accuracy, the proportion of correct guesses in the test sets for each cross-validation. Kappa parameter reported by this summary reports if the agreement is greater than by chance.

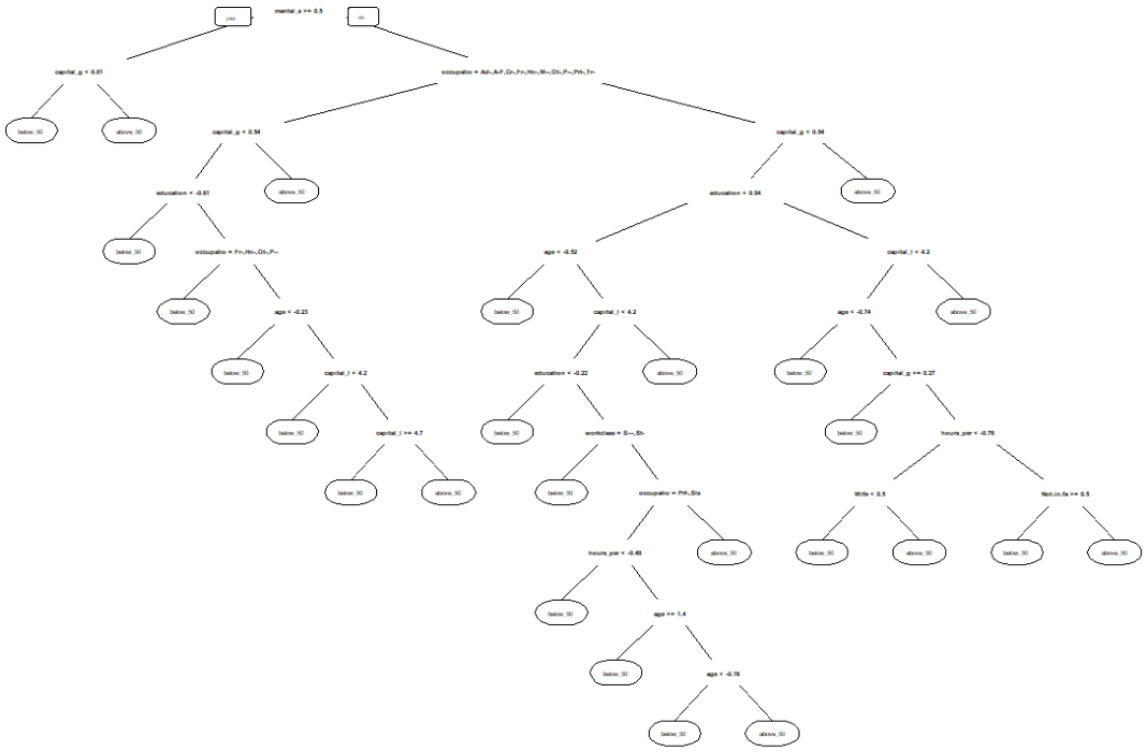


Figure 4: Crossvalidated classification tree

Our research focuses on complexity and accuracy, so we ignore the Kappa output. We also plot complexity parameter vs Accuracy and see a general trend when higher accuracies correspond to lower complexity parameter. These can be found in "Images" folder as well. The ROC plot and AUC value will be reported when we are select the optimal model for our data. In addition, we observe each predictors level of importance after analysis and see that the top seven predictors are: age, education number, capital gain, capital loss, hours per week, workclass.Federal.gov, and workclass.Local.gov.

Bagged Trees

Bagging is a powerful method to reduce overfitting of more complex models. Instead of fitting the model on one sample of the population, several models are fitted on different samples (with replacement) of the population. Then, these models are aggregated by a voting system. We first need to note that bagging is necessary because classification trees suffer from high variance and averaging a set of observations reduces variance. We can bootstrap by taking repeated samples from the (single) training data set. Because bagging is an extension of tree fitting we wont be adding more to the theoretical discussion. Instead we will focus on our loop that is written to save time and trades a little bit of accuracy for it.

To not confuse with actual RandomForest algorithm that is also using similar principles, we write our own script, following closely steps by EnhanceDataScience[4]

```
train_index<-sample.int(nrow(training_dummy),size=
                round(nrow(training_dummy)*0.8),replace = F)
n_model<-100
bagged_models<-list()
for (i in 1:n_model)
{
 new_sample=sample(train_index,size=length(train_index),replace=T)
 bagged_models=c(bagged_models,list(rpart(income~.,
 training_dummy[new_sample,],control=rpart.control(minsplit=6))))
}
```

We define train _ index to use later for bootstrapping purposes. We use 80—20 split. We decide to aggregate together 100 models (to save computation time). For each of this 100 models, we sample again, and fit a classification tree using rpart(for consistency purposes). We use minsplit=6 again to save time. Once we are done with fitting, we continue to predict our results:

```
bagged_result=NULL
i=0
for (from_bag_model in bagged_models)
{
 if (is.null(bagged_result))
 bagged_result=predict(from_bag_model,testing_dummy)
 else
 bagged_result=(i*bagged_result+predict(from_bag_model,testing_dummy))/(i+1)
 i=i+1
}
```

This loop basically predicts values for our test set from our 100 fitted models and creates probabilities of belonging to classes. We perform a simple if-else loop to get actual factors. The exact steps can be found in our R script. Additionally, we cant see the important variables because our script doesnt produce a single best fitted tree. This approach was chosen for time efficiency, so we traded potential accuracy and some features in order to save time. However,we noticed that the loop contains no improvement over our regular tree fit. So we resorted back to our initial treatment of bagged tree as a random forest, that, unlike random forests, uses all the available variables. So we returned to our package caret and wrote the following:

```
cvCtrl <- trainControl(verboseIter = TRUE,
          summaryFunction = twoClassSummary,classProbs = TRUE)
newGrid<- expand.grid(mtry = 26)
set.seed(154)
```

```
bagged_tree_rf <- train(y=training_dummy[,27],x=training_dummy[,c(1:26)],
                        ntrees=100, trControl = cvCtrl, method = "rf",
                        tuneGrid = newGrid,importance=TRUE)
```

cvCtrl is our training control. We left our method blank so that caret will choose bootstrapping with 25 re-sampling iterations. VerboseIter just helps us to keep track of the process. We introduced twoClassSummary and classProbs to force caret to use ROC for fitting. We created a new grid that contains sole 26 as mtry, forcing our forest to run like a bagged tree. Once the process was done we extracted importance of variables, the top seven being: age, capital gain, occupation, education number, single marital status, hours per week, and married marital status. This will guide us in next step and followed the exact same steps we did for classification tree. In addition, the ROC plot and AUC value will be reported when we select the optimal model for our data.

Random Forest

Random Forest provides an improvement over bagged trees by way of a small tweak that decorrelates the trees. Similar to bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. Increasing the number of trees in the ensemble won't have any effect on the bias of our model, but will reduce the variance of our model. We can achieve a higher variance reduction by reducing the correlation between trees in the ensemble. This is the reason why we randomly select 'm' predictors at each split because it will introduce some randomness in to the ensemble and reduce the correlation between trees. Hence 'm' is the major attribute to be tuned in a random forest ensemble. In general best 'm' is obtained by cross validation. Some of the factors affecting 'm' are :

- A small value of m will reduce the variance of the ensemble but will also increase the bias of an individual tree in the ensemble.
- the value of m also depends on the ratio of noisy variables to important variables in our data set. If we have a lot of noisy variables then small 'm' will decrease the probability of choosing an important variable at a split thus affecting our model.

In our case we sample 9,11,13 from 27 total predictors. This is done in order to manage complexity. So, we kept number of trees fixed and the depth of tree on default, and we tried to change the mtry to 9,11, and 13. Initial iterations of the model show that the higher the number of predictors the better; this is what our experience shows, but may not be generally true. The general rule and default for mtry is the closest integer to the square root number of total predictors for large data sets (our data set is not considered to be large). On average fitting and tuning a random forest takes 8 hours, even with parallel computing. This is due to complexity O(M*n*m*log(n)), where M: number of trees, m: number of predictors, and n: number of observations.

Some observations while fitting the forest. As we are using laptops, we do not generally have a lot of computing power. Regular R script **randomForest()** gives an error about memory allocation straight away (the laptop we were running the script on has 8 GB of RAM). We then resorted to using the caret package code to circumvent this. Another way to cut time is using parallel computing in R. We used the packaged **foreach()** and enabled all 4 cores of our processor and still managed to cut time only by an hour. The code for this is very similar to that of bagged tree and classification tree so we am not going to display it here. We just changed the bootstrapping method back to repeat cross validation and reduced the number of folds to 6 and the number of repeats to 3 in order to save time. Please refer to .Rmd file for more insights. In addition, the top seven variables of importance from our analysis is capital gain, capital loss, occupation, education number, age, hours per week, and work class. The ROC plot and AUC value will be reported when we select the optimal model for our data.

Model Selection

The results of our classification methods—classification tree, bagging, and random forest — are shown using a confusion matrix output, ROC curve, and AUC values. We used the **confusionMatrix()** function from the caret package instead of calculating it on our own because of the additional information that it supplies in its output. From the confusion matrix we see our sensitivity and specificity values, with the class "over 50K a year" used as the positive event. Sensitivity, also referenced as the true positive rate, represents the measures the proportion of positives that are correctly identified as such. Specificity, also referenced as 1 minus the false positive rate, measures the proportion of negatives that are correctly identified as such. They are both statistical metrics that can be easily extracted from our confusion matrices, and measuring the performance of our classification methods. We summarize the information from our confusion matrices in Table 2

Table 2: Model Performance

Model	Accuracy	Sensitivity	Specificity	AUC
Classification Tree	0.8573	0.5606	0.9490	0.8722551
Bagged Tree	0.8435	0.6235	0.9115	0.8854973
Random Forest	0.8598	0.6381	0.9283	0.8998767

Another method for measuring performance is the the Receiver Operating Characteristic (ROC) curve and its corresponding Area Under the Curve (AUC) value. This is created by plotting the sensitivity against one minus the specificity. The accuracy of our classification methods used depends on how well the method classifies, and this accuracy is measured by

the area under the ROC curve, aka the AUC. The output from our models can be found in Figure 5-6. We chose AUC values to compare amongst our models in order to choose the best one, instead of looking at the accuracy rate from the confusion matrix. We can clearly see that our bagged tree has the smallest accuracy, yet it improves sensitivity of tree by 6%. Navigating these values without having a specific target for the fit is hard, so we look at more holistic characteristics—the AUC. The AUC values for tree, bagging, and random forest are 0.8722551, 0.8854973, and 0.8998767 respectively. Thus, by these performance metrics the optimal classification method for our data is Random Forest with $mtry = 9$.

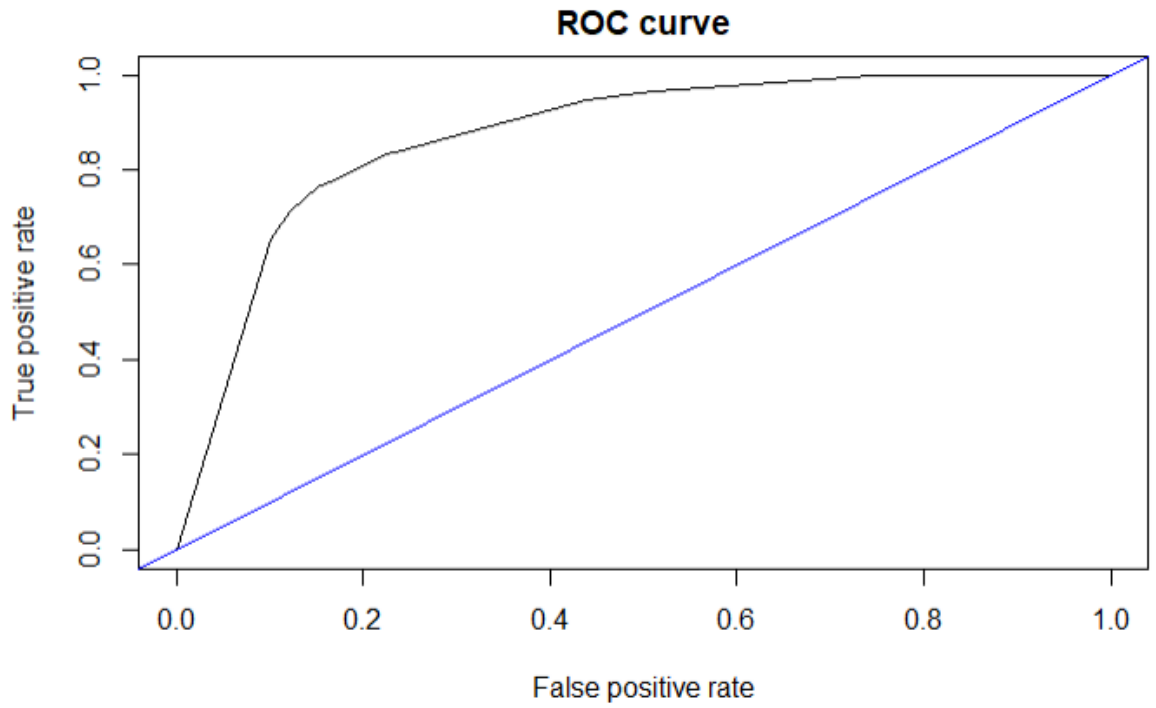
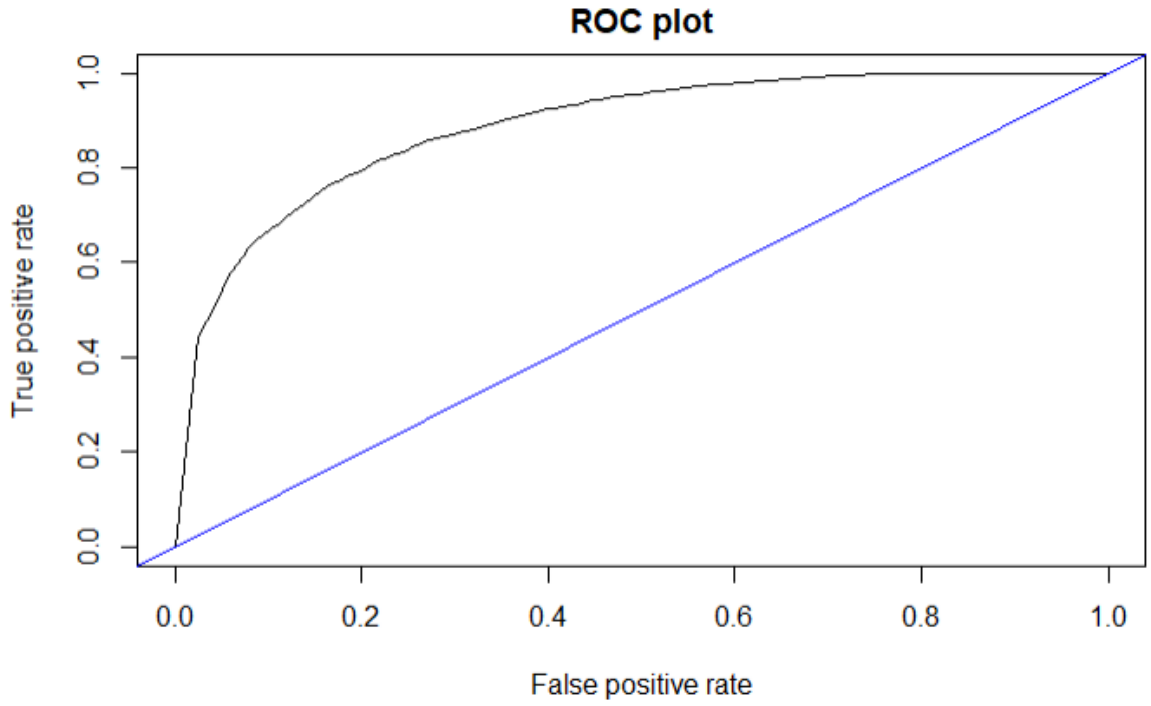
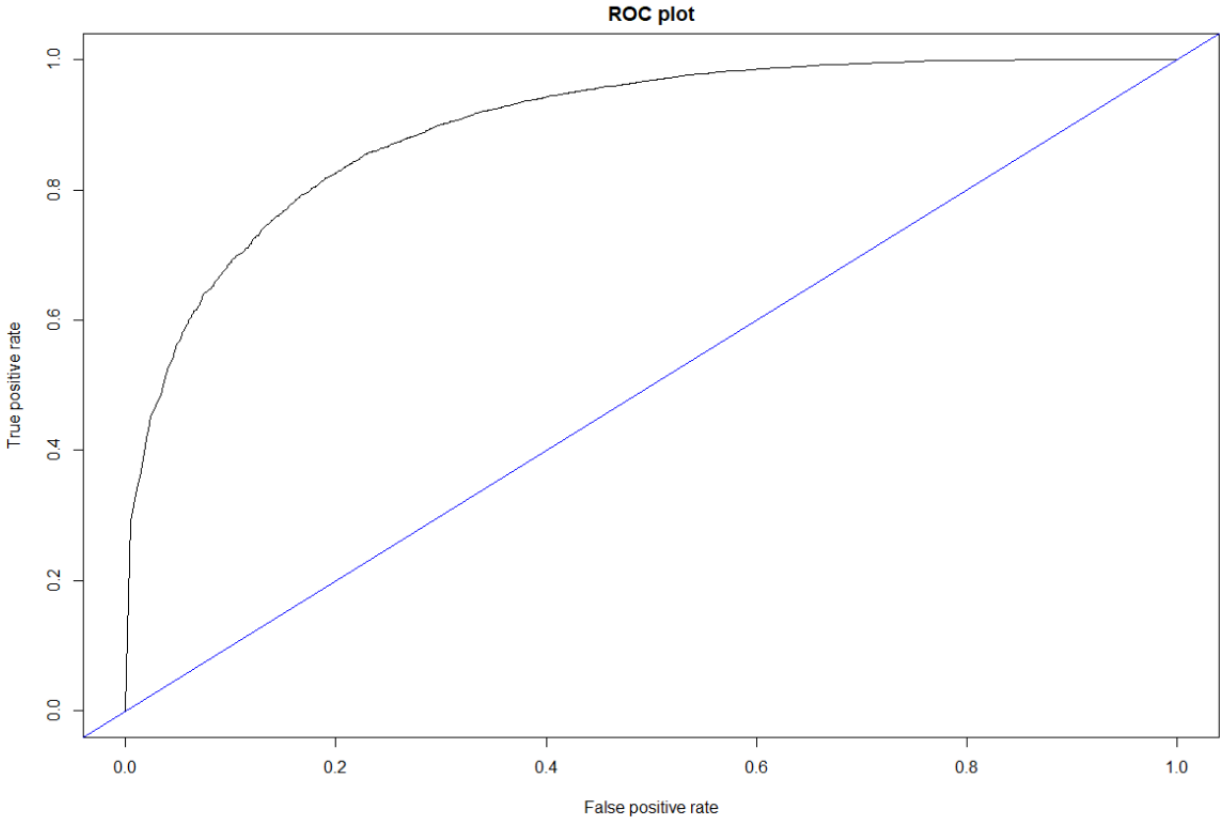


Figure 5: Classification Tree ROC



(a) Bagged Trees ROC



(b) Random Forest ROC

Figure 6: Bagged Trees and Random Forest ROC.

Notes

Our RStudio would run a couple of hours and crash every time we tried to knit the .Rmd file. We looked up online and could not find a solution (usually because of syntax issues RMarkdown gives you the error right away). Thus, we uploaded our .Rmd file and our workspace. Once the reader downloads the two files together, the project should run smoothly.

References

[1] https://cran.r-project.org/web/packages/CALIBERrfimpute/vignettes/simstudy_survival.pdf

[2] https://arxiv.org/pdf/1701.05305.pdf

[3] Witten, Ian; Frank, Eibe; Hall, Mark (2011). Data Mining. Burlington, MA: Morgan Kaufmann. pp. 102 - 103.

[4] https://www.r-bloggers.com/machine-learning-explained-bagging/