A Look into the Random Forest Algorithm and its Applications

Daerian Dilkumar, Guan Yu Chen, Derek Frempong, Zong Li

April 4, 2018

Contents

1 Introduction

This project aims at the construction and implementation of the "Random Forest" algorithm, which generates predictions for problems in classification and regression. Relying on randomness, the algorithm selects both sample and feature spaces hundreds of times from the original dataset. The result is a prediction with minimal bias due to the lack of overfitting.

The term 'random forest' is derived from the collection of decision trees in which the algorithm uses. Each tree is trained over a bootstrapped [2.1] sample of the training set with a randomly selected set of variables to create its feature space. This means that each tree has high bias; however, the overall bias of the prediction drops significantly since the forest is intended for use with hundreds of trees. As a result, the algorithm is able to derive a very accurate prediction through aggregating all of the decisions returned by each tree.

2 Methodologies

Before we get into the algorithm for Random Forest, we need to go over some basic statistical methods.

2.1 Distribution Free Statistical Methods: Bootstrapping

Bootstrapping in statistics is any test or metric that relies on random sampling with replacement. This is commonly used to get the variance of a complicated distribution. But how is this implemented? The algorithm is as follows:

Algorithm 1 *Let $\hat{\Psi}_i$, $i = 1, 2, \ldots, n$ be the bootstrap samples and $\widehat{F(x)}$ be an Empirical distribution. Suppose we want to estimate the variance.*

1. *Draw a sample of size n with replacement from $\widehat{F(x)}$*
2. *Calculate $\hat{\Psi}$ from the sample in (1)*
3. *Repeat steps (1) and (2) for m times to get $\hat{\Psi}_1, \hat{\Psi}_2, \ldots, \hat{\Psi}_n$ such that $m << n^n$*
4. *Calculate*

$$\widehat{Var}_{\hat{F}}(\hat{\Psi}) = \frac{1}{m-1}\left[\sum_{i=1}^{m}(\hat{\Psi}_i - \bar{\hat{\Psi}})^2\right]$$

where $\bar{\hat{\Psi}} = \frac{1}{m}\sum_{i=1}^{m}\hat{\Psi}_i$

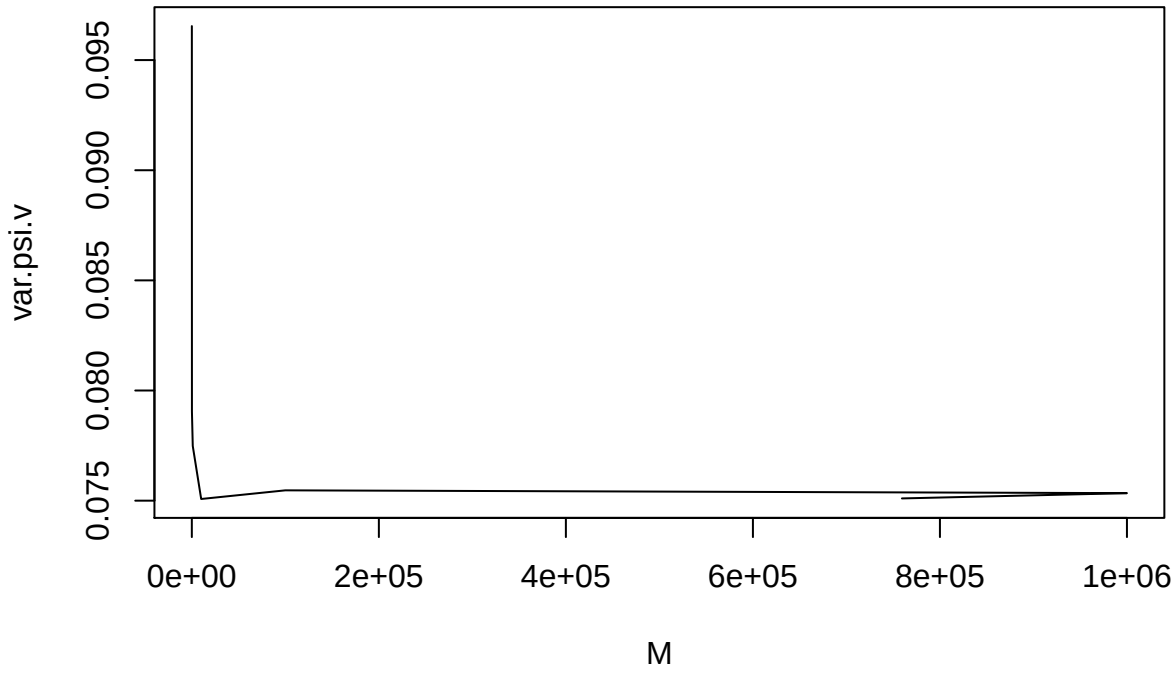
A sample implementation in R and an example is given below.

```
data = c(1.06, -1.28, 0.4, 1.36, -0.35,
         -1.42, 0.44, -0.58, -0.24, -1.34,
         0.00, -1.02, -1.35, 2.05, 1.06,
         0.98, 0.38, 2.13, -0.03, -1.29)
n = length(data)
M = c(10,10^2,10^3,10^4,10^5,10^6,15^5)
var.psi.v = rep(NA, length(M))
j=1
for (m in M) {
  samp.var = rep(NA, m)
  for (i in 1:m) {
    samp = sample(data, n, replace=TRUE)
    samp.var[i] = ((n-1)*var(samp))/n
```

```
  }
  psi_bar = sum(samp.var)/m
  var.psi = (sum((samp.var-psi_bar)^2))/(m-1)
  var.psi.v[j] = var.psi
  j = j+1
}
var.psi.v
```

```
## [1] 0.09654352 0.07904018 0.07748273 0.07507635 0.07546783 0.07534098
## [7] 0.07510138
```

```
plot(M,var.psi.v,type="l")
```



As we can see in the above plot and the numbers, the numbers become more and more stable as we increase the number of bootstrap samples. A very similar approach is used to create decision trees for Random Forest. The minor modification is that we sample the size of the dataset instead of choosing an arbitrary sample size.

2.2 Random Forest

The procedure is as follows:

1. Select a number of trees for the forest, and how big the feature space is.
2. For each tree, choose variables from the feature space, and get a bootstrap.
3. Train the model for each tree with these variables for the given labels.
4. Collect the trees into a forest to get them ready for predictions

5. Finally, given testing data, use the forest to create as many predictions are there are trees.
6. Aggregate the predictions for a final prediction for each observation in the testing data.

The libraries we used are as follows:

- tidyverse
- dplyr
- rpart
- rpart.plot

3 Development Process

During May of 2017, Daerian and Derek attended the DataFest competition where several of the winning teams implemented a variant of the Random Forest algorithm in their analysis. Inspired by the top participants at DataFest, we have decided to delve deeper into the heart of the Random Forest algorithm, ultimately implementing the algorithm in R and putting it through rigorous testing on various classification and regression datasets.

"Elements of Statistical Learning" (Hastie, n.d.) and "Introduction of Statistical learning with Applications in R" (Gareth, n.d.) greatly aided us in the development of the algorithm. We started with little prior knowledge of Random Forest, we tasked ourselves to learn about the statistical methods and algorithms before exploring the implementations of the Random Forest algorithm. We discovered that Random Forest uses an amalgamation of bootstrapped decision trees that sample (without replacement) the predictors to reduce bias and increase variance.

First, we have applied a function that generated a sample tree, using bootstrapping, with sampled predictors that $m \leq p$ where p are the number of predictors. [7.1.1] We then created a function that loops and creates B amount of trees, creating a "Random Forest". [7.1.2] We then put the B trees into a nested list, where the ith element is a list with three elements; One being the ith bootstrapped tree, second being the anti join of the data, and the final element being the sample used to generate the the bootstrapped tree. Anti-joins become useful when classifying as we implement "out of bag sampling" - when predicting each point, we need to aggregate the trees that did not contain the observation we are predicting as to stay accurate. We had to revise our algorithm numerous times due to impossibly high and near-zero accuracy results. We are currently experimenting with controls as a way to increase accuracy.

4 Our Story

Our tale is one of fights, frustration and victory.

We start our progress by attempting to research the topic of random forests, being as thorough as possible. Being the first time hearing of bootstrapping and bagging, it took us a few weeks to understand the material. We held multiple late-night research sessions and had many arguments.

Through plentiful office hour sessions, thorough examinations of papers in the library, meetings, and arguments razing the ears of the many students in the halls of BV, we have finally developed a game plan. We wrote our general ideas and shared them. Meanwhile, we had our team test each others' sample code. During this research process, we stumbled upon the concepts of cross entropy and confusion matrices, which quickly became an important concept to be later implemented. We strived to understand the intricacies involved in such notions, as they will be come essential to test the accuracy of the Random Forest algorithm.

Upon our understanding of the new material, we have finally obtained some confidence to begin the implementation of our Random Forest algorithm. We have decided to consult with our supervising professor,

Dr. Ken Butler, for assistance in attaining access to increased processing power. With luck on our side, we were granted access to a portion of our institution's server processing power, `cms-chorus`: a virtual machine server hosted by the University of Toronto, with 6 CPUs at 3.6 gHz and 64 gigabytes of RAM (better to have it and not need it rather than need it and not have it).

Abiding by our Professor's advice, we began experimenting with `Rpart` which calculated the splits for our trees. `Rpart` is a package in R which plays a crucial role in creating decision trees via method of splitting the feature space, in our algorithm. During our experimentation, we have discovered that `Rpart` is mostly a black-box library. Feature selection, bootstrapping, and model setting for each tree were left for us to explore due to the black-boxing. The algorithms which we have implemented for these components worked for the time-being; however, we quickly found in our attempts, that we were required to constantly switch the type of splitting done and how the variables were handled. In fact, we were so confused that we wound up spending two weeks working on it. In the end, we simply had to leave a single variable out, which improved our result. This is sub-optimal, however; we would like to proceed in updating the scripts to be able to take the type of splitting as a parameter from the user and update before packing the scripts for deployment on GitHub.

At this point, we created a GitHub account and set up everything required to have an object-oriented development environment with shared statistical resources. We created multiple branches for each team member, with the master branch containing an amalgamation of all our finalized work. We first began by creating a series of functions we thought would be useful. This included functions such as the library setter (which would automatically scout for and install all required libraries), a tree creator [7.1.1], and a script to clean up the data set (only one at the time) to be in a compatible format. As we were testing our algorithm, we ran into various problems with tree generation, so some time was spent diagnosing and repairing our algorithm to generate trees. After our repairs were completed, we ran and compared our brand new creations. Needless to say we were proud that we had achieved this step. Although very primitive, we had managed to generate trees! As logic goes, they were very poor at generating an optimal, nonetheless a feasible tree. With imperfections aside, we were pleased to have our algorithm running without error.

We began production on other functions after our small success. We were generating trees, but were missing a process to organize them into a forest. To begin this process, we created a function which took the number of bootstraps, the number of features for each tree, and the data the trees would be based on as input. We took these parameters and generated a number of bootstrap samples as specified by the input, and creating a tree using the previous function. As the algorithm progressed, some key problems had arised: How do we store these trees? How do we retrieve which observations were affected by the tree? How can we test its efficiency? As it turns out, R does not allow us to create datatypes with the same efficiency as other programming languages, so we had to improvise. Our solution was to create empty lists for everything and then append to them. Our function which generated the trees now consists of: the tree itself, the set of all observations in the original training data, and a list of variables which appeared the most frequently in the tree. These lists returned by the function would then be stored as nested lists, which represents the generated tree. Each part of the tree (such as the splits, its important variables, the methods it has, function calls, etc) is also stored in a similar fashion. [7.1.1][7.1.2] As you can imagine, this results in objects of substantial order. One of the challenges of creating this script was tracking the location of resources we needed in this large cluster of topographies. Fortunately, we have discovered a work-around to this chaos, and retrieved all the things we needed.

Now that we are generating working forests, we needed to compare them. We started the process of running large batches of forests and comparing their respective trees; needless to say, the most difficult part of this process was simply waiting. The trees took time to compute and as consequence, so to did the forests. As a result of this process, we achieved what we set out to find - the results of the comparisons. We have observed many interesting points from these results. One point worth mentioning was the number of splits in each tree. On average, we had less splits than other algorithms publicly available on CRAN. This is an issue which would later be resolved via the use of "controls."

From here we had to create comparisons in classification and regression datasets, and devise ways to evaluate the performance of the random forest on our test data. We created two functions: one for classification and one for regression. These functions will prepare models based on their respective methods of prediction.

Under classification, we have loss function which calculates false positives. For regression, we calculate an MSE and R^2, using their corresponding formulas. Specifically, the formulas that we use are:

$$R^2 = 1 - \frac{SSRes}{SSTot} = \frac{SSReg}{SSTot}$$

and

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

, where $\hat{Y}$ is a vector of n predictions and Y is a vector of observed values. This allows us to account for all possible conditional distributions of the multivariable regressions since we use only the final residual, which is leftover from the total minus the sum of all regressive sum of squares.

Computing the accuracy of the Random Forest algorithm was challenging. We came to the realization that the formula for MSE and R^2 we were using relied on regression. However, as we were working with multiple predictors, we had no choice but to re-do the function to use the error term instead of regular regression. This is due to the fact that with multiple predictors the MSE is unique to each of them. In order to get true error we have to use the error term which is the squared error of the fitted subtracted from the observed.

Before implementing the control mechanic, there was no way of controlling how `rpart` splits the feature space to construct the decision trees. Since there was no control mechanic, the only hyper-parameters available were number of trees used in the Random Forest algorithm, and number of variables used per tree. This was a major setback since these hyper-parameters were insufficient to produce feasible results. The control mechanic can control the increase of R^2 per split of the feature space, number of cross validations used for each decision tree, minimum number of splits of the feature space, and many other hyper-parameters used to construct a decision tree. Hyper-parameters are parameters that are unknown and should be estimated under different circumstances. Experimenting with the hyper-parameters in the control mechanic, the results improved significantly. Unfortunately, hyper-parameters related to constructing decision trees were near impossible to get 100% accurate, this caused the classification and regression results to be not optimal. This was somewhat solved by "simulation".

Not having optimal results because of inaccurate hyper-parameters is something we can deal with, but we still wanted to see the most optimal results. This process turned to be very frustrating as it could take hours to run 1 simulation. The simulation would proceed as follows

Algorithm 2 *1. Read, clean and format the data to be analyzed by the Random Forest algorithm*

2. *Set variables to be optimized to a initial value (0 works most of the time)*
3. *Loop through possible seeds for the Random Forest algorithm ($[0, 100]$ were used)*
4. *For each seed, loop through all feasible values of hyper-parameters*
5. *For each value of hyper-parameters, record the value according to 2 cases:*
   (a) ***Regression**: Let r_1 be the previous R^2 recorded and let r_2 be the new R^2. r_2 will only be recorded if $r_2 > r_1$.*
   (b) ***Classification**: Let c_1 be the previous accuracy recorded and let c_2 be the new accuracy. c_2 will only be recorded if $c_2 > c_1$.* NOTE*: Accuracy is defined as $1 - l$, where l is the loss.*

This algorithm was used in a few of the results shown below. Due to time constraint, not all of the results came from this simulation algorithm. If time allowed, this algorithm possibly would be more efficient and we would be able to get more optimal results for each case.

5 Examples And Results

NOTE: We will be working with our functions and algorithms, which are stored in the file `randomforest.R`, so we will be first setting this source before we proceed.

5.1 Regression

5.1.1 Wine Data

We will create an example using Wine Data. This data performs well using a regression model. The data is formatted so that the quality is the last column of the data set. We will properly read and process the data so that we can run it through our random forest algorithm. To start we start at $M = 1$ (1 variable is chosen per tree), and $B = 500$ (500 trees).

```
# Read red wine and white wine data from their respective files
redWineData = read_delim("winequality-red.csv", delim = ";")
whiteWineData = read_delim("winequality-white.csv", delim = ";")
```

Now, we that read the data into R . . . time to clean the data and set up the data to run the Random Forest!

```
# Remove NA values
redWineData = (redWineData[complete.cases(redWineData),])
whiteWineData = (whiteWineData[complete.cases(whiteWineData),])

# Prep Sets for merging library
redWineData = redWineData %>% mutate(Type = "Red")
whiteWineData = whiteWineData %>% mutate (Type = "White")

# Merge Data and get Final Dataset
wineData = rbind(redWineData,whiteWineData)

# Remove spaces from column names
names(wineData) = gsub(" ","_", names(wineData))

# Convert columns ot factor as needed and remove not needed columns
wineData$quality  = as.factor(wineData$quality)
wineData$Type = as.factor(wineData$Type)
wineData = wineData[,-ncol(wineData)]
```

Let's split the original dataset into training set and testing data.

```
# Split data into training data and testing data, 50% and 50% respectively
split = sample(2, nrow(wineData), prob=c(0.5,0.5), replace=TRUE)
training_set = wineData[split==1,]
testing_set = wineData[split==2,]
```

Lets see if this works.

```
coverage = as.data.frame(table(split))
coverage$Freq[1]/nrow(wineData)
```

```
## [1] 0.5040801
```

```
coverage$Freq[2]/nrow(wineData)
```

```
## [1] 0.4959199
```

The percentages are close enough to the split we want. Perfect! Let's get ready to perform regression.

```
# Acquire labels for the training set
training_labels = as.numeric(as.character(training_set$quality))

# Acquire labels for the testing set
testing_labels = as.numeric(as.character(testing_set$quality))
```

Note here that we need `as.character` since when converting factors with `as.numeric` it converts the underlying number and not the factor itself.

```
training_set = training_set[,-ncol(training_set)]
testing_set = testing_set[,-ncol(testing_set)]

# This is the number of trees we want to create
B = 500

# This is the number of variables we want to use
M = 1
f = PerformRegression(training_set,training_labels,testing_set,testing_labels,B,M)
```

```
## [1] "Results:"
## [1] "MSE = 0.671461738198936"
## [1] "R2 = 0.0990355159865147"
## [1] "Timings: "
##    user  system elapsed
##    8.78    0.09    8.87
```

The above results was achieved when we set the `seed` to 1. As we can see, the $R^2 = 0.0990355159865147$, let's see how high the R^2 can go. Therefore, we came up with a simulation that can give us the best result possible. Since we need a seed to keep the results consistent and reproducible, we looped through numbers 1 through 100 to set as possible seeds. First, we split the training set and testing set 50% and 50% respectively. this give us around the same number as when we set the training and testing set to 75% and 25% respectively. We decided to report the bigger number (obviously) which is $R^2 = 0.277918789503239$ (approx.) and $MSE = 0.529555537034112$ (approx.) when $B = 1500$ (number of trees), `seed`$= 64$ and $M = 5$ (number of variables used per tree).

How do we improve this further? We found a function in the `rpart` library that can control the `rpart` function, `rpart.control`. The code below contains a control added to the Random Forest algorithm in order to bring the R^2 up to around 0.2999. This code can be found in `experiment.R` (it takes at least 10 minutes to run depending on the parameter).

```
set.seed(64)
# Split data into training data and testing data, 75% and 25% respectively
split = sample(2, nrow(wineData), prob=c(0.75,0.25), replace=TRUE)
training_set = wineData[split==1,]
testing_set = wineData[split==2,]

# Acquire labels for the training set
training_labels = as.numeric(as.character(training_set$quality))

# Acquire labels for the testing set
testing_labels = as.numeric(as.character(testing_set$quality))

training_set = training_set[,-ncol(training_set)]
testing_set = testing_set[,-ncol(testing_set)]
M = 5
```

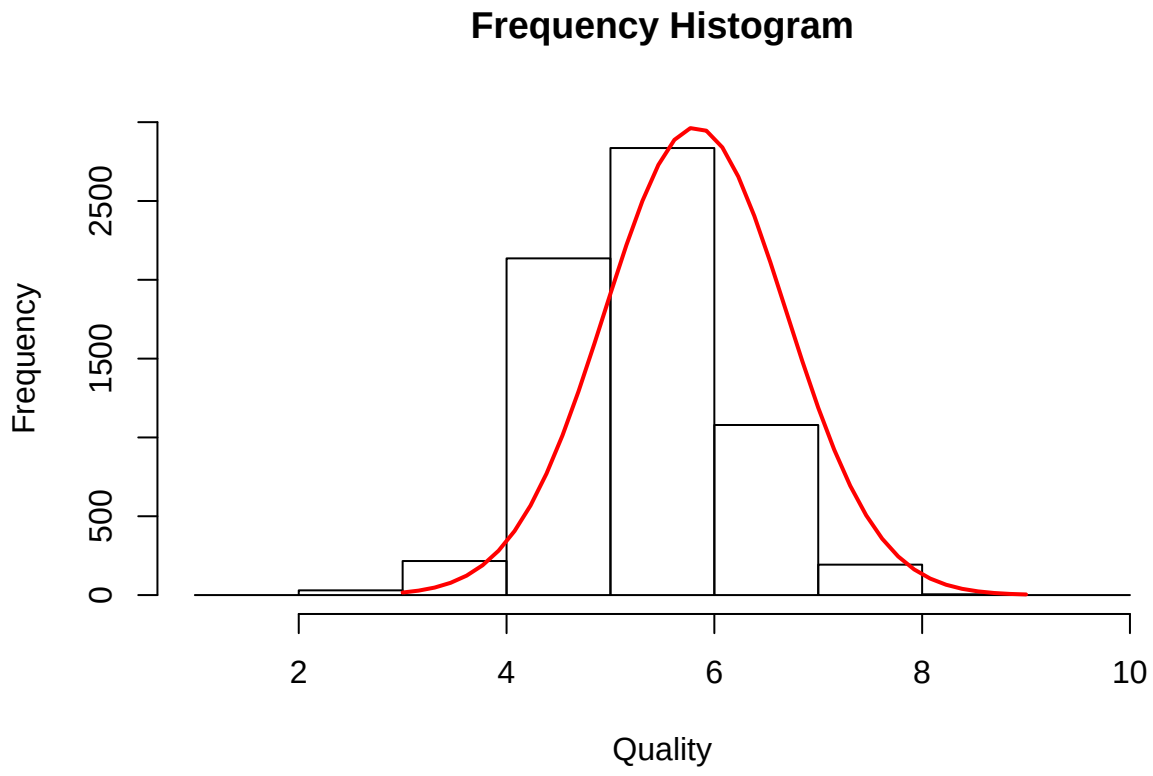
```
ctrl = rpart.control(cp = 0.005, minsplit = 500, xval = 10)
f = PerformRegression(training_set,training_labels,testing_set,testing_labels,B,M,ctrl)
```

```
## [1] "Results:"
## [1] "MSE = 0.513945996917796"
## [1] "R2 = 0.299203347654808"
## [1] "Timings: "
##    user  system elapsed
##   21.75    0.18   21.94
```

We can see an improvement in R^2 and MSE, but R^2 is still very low. One of the reasons that the R^2 is low could be the distribution of the data we want to classify. Let's take a closer look at the labels that the Random Forest algorithm was trying to classify, specifically the distribution.

```
#Draw histogram
data <- as.integer(as.character(wineData$quality))
h <- hist(data, breaks=1:10, ylim=c(0, 3000), freq=TRUE, main = "Frequency Histogram",
          xlab = "Quality")

#Fit curve with normal distribution
xfit<-seq(min(data),max(data),length=40)
yfit<-dnorm(xfit,mean=mean(data),sd=sd(data))
yfit <- yfit*diff(h$mids[1:2])*length(data)
lines(xfit, yfit, col="red", lwd=2)
```




```
# Occurences of labels in training set and testing set
table(training_labels)
```

```
## training_labels
##    3    4    5    6    7    8    9
##   24  163 1575 2084  783  147    5
```

```
table(testing_labels)
```

```
## testing_labels
##   3   4   5   6   7   8
##   6  53 561 752 296  46
```

The histogram above depicts the frequency of quality ratings of wine to be normally distributed, peaking at 6. We can also observe that quality 5 is also very popular amongst all estimated qualities using Random Forest. The less frequently estimated qualities, 1,2,3,4,7,8, and 9, are less popular due to the qualities appearing less in the training set and in the whole data file, in general. Taking a look at the mean and variance of the distribution, we get:

```
mean(data)
```

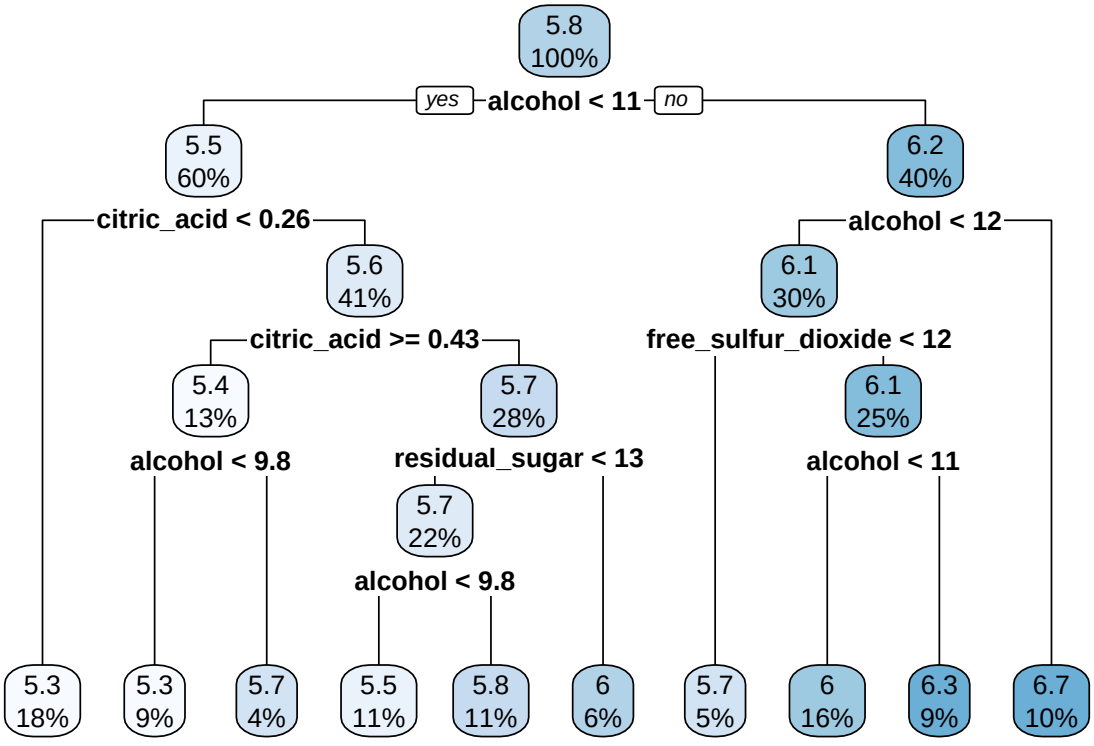
```
## [1] 5.81863
```

```
(sd(data))^2
```

```
## [1] 0.7626033
```

This tells us that most of our estimated data falls between 5.05 and 6.57, which supports our previous assumption that most wines have quality 5 or 6, hence the likelihood of estimating 5 or 6 is high. Let's take a look at a sample decision tree to confirm this hypothesis:

```
sample_dtree = BT_Tree(training_set, training_labels, 5, tree.print=TRUE)
```



We can see that the leaf nodes all have qualities/labels that are in the interval $[5, 7)$. Generating large

number of trees won't change this result. Even if there was a few decision trees that predicted other qualities, these predictions would get 'outnumbered' and would be insignificant to the final prediction.

Another reason is the decision trees that are used in the algorithm. We are using `rpart` to produce the trees. This is a built in R function that generates decision trees given a dataset. Since this function is black-box, we are not sure what exactly it does to separate the feature space to make the trees. If we had more time, we would love to learn more about how to split feature space and produce trees manually that helps us improve the accuracy.

The last possible reason we thought of that regression is not performing well is just the dataset alone and how we set up the data. Red wine and white wine might have different standards and the pattern or the model of the variables might be completely different. Therefore, we decided to run a little test. We took just the white wine data and ran the Random Forest regression on it. The result is surprising! Have a look!

```
whiteWineData.i = whiteWineData
whiteWineData.i = (whiteWineData.i[complete.cases(whiteWineData.i),])
names(whiteWineData.i) = gsub(" ","_", names(whiteWineData.i))
whiteWineData.i$quality  = as.factor(whiteWineData.i$quality)
split = sample(2, nrow(whiteWineData.i), prob=c(0.7,0.3), replace=TRUE)
training_set = whiteWineData.i[split==1,]
testing_set = whiteWineData.i[split==2,]
training_labels = as.numeric(as.character(training_set$quality))
testing_labels = as.numeric(as.character(testing_set$quality))
training_set = training_set[,-ncol(training_set)]
testing_set = testing_set[,-ncol(testing_set)]
B = 500
M = 5
ctrl = rpart.control(cp = 0.005, minsplit = 500, xval = 10)
f = PerformRegression(training_set,training_labels,testing_set,testing_labels,B,M,ctrl)
```

```
## [1] "Results:"
## [1] "MSE = 0.233403220628421"
## [1] "R2 = 0.693669082194203"
## [1] "Timings: "
##    user  system elapsed
##   15.02    0.03   15.05
```

Indeed, white wine and red wine are modeled completely different. Just using white wine data, our R^2 increased to 0.693669082194203. This is a significant increase from doing the regression with both data combined. Now, let's do the same with red wine data!

```
redWineData.i = redWineData
redWineData.i = (redWineData.i[complete.cases(redWineData.i),])
names(redWineData.i) = gsub(" ","_", names(redWineData.i))
redWineData.i$quality  = as.factor(redWineData.i$quality)
split = sample(2, nrow(redWineData.i), prob=c(0.8,0.2), replace=TRUE)
training_set = redWineData.i[split==1,]
testing_set = redWineData.i[split==2,]
training_labels = as.numeric(as.character(training_set$quality))
testing_labels = as.numeric(as.character(testing_set$quality))
training_set = training_set[,-ncol(training_set)]
testing_set = testing_set[,-ncol(testing_set)]
B = 500
M = 5
ctrl = rpart.control(cp = 0.005, minsplit = 500, xval = 10)
f = PerformRegression(training_set,training_labels,testing_set,testing_labels,B,M,ctrl)
```

```
## [1] "Results:"
## [1] "MSE = 0.232150301259436"
## [1] "R2 = 0.657545401652838"
## [1] "Timings: "
##    user  system elapsed
##    7.29    0.00    7.35
```

We can see that red wine data performs almost the same way as white wine data. The only difference is that we used 4:1 between the training set and testing set. Our R^2 is a bit lower than the R^2 of white wine data but it is still significantly better than the combined R^2. White wine data has more observation than red whine data which might result in the decrease in R^2.

5.2 Classification

5.2.1 Breast Cancer

Here's another example, but instead we will do classification on this dataset rather than regression. This data is is preprocessed already so we have to read in the data only.

```
BreastCancer = read.table("BreastCancer.csv",header=T, sep=",")
BreastCancer$Class = as.numeric(BreastCancer$Class)
BreastCancer$Class = (BreastCancer$Class) +1
BreastCancer$Class = as.factor(BreastCancer$Class)
BreastCancer = BreastCancer[-1]
```

Now that we have the data we can then perform Classification on it and see how it fares. Here, we will 80% of the data for the training set and 20% of the data for the testing set.

```
set.seed(41)
split = sample(2, nrow(BreastCancer), prob=c(0.8,0.2), replace=TRUE)
training_set2 = BreastCancer[split==1,]
testing_set2 = BreastCancer[split==2,]

# Training set labels
labelsBC = as.factor(unlist(training_set2[ncol(training_set2)]))

# Testing set labels
labelsBC2 = as.factor(unlist(testing_set2[ncol(testing_set2)]))

# Getting rid of the labels to prepare for classification using Random Forest
training_set2 = training_set2[,-ncol(training_set2)]
testing_set2 = testing_set2[,-ncol(testing_set2)]
B2 = 500
M2 = 3
ctrl = rpart.control(cp = 0.005, minsplit = 500, xval = 10)
f = PerformClassification(training_set2,labelsBC,testing_set2,labelsBC2,B2,M2,ctrl)
```

```
## [1] "Results:"
## [1] "Accuracy = 97.8723404255319%"
## [1] "Timings: "
##    user  system elapsed
##    4.92    0.00    4.92
```

Result was obtained by setting `seed`= 41, $B = 500$ and $M = 3$. The loss we got for classification on the recommended number of variables is about 0.9787, so approximately 97.87% accuracy. This is quite good. Based on what we have read about classification with Random Forest, if the data is simple the accuracy

will be quite high and due to the effect of having to do a combination of m of the p variables in the data, it reduces the possibility to overfit. So with that in mind, we can be quite certain that our accuracy we obtained makes sense.

Note that this data is a binary classification (only 2 classes to classify between). This is a good test data set to test the classification algorithm if it works. The dataset is not complicated and it is a simple 'Yes' or 'No' for each observation. Obtaining a good result for this dataset gave us confidence that it will work on datasets with more classes. The next dataset, Steel Data has 7 classes and more observations than Breast Cancer. Read on!

5.2.2 Steel Data

Now that we know previously that our implementation of Random Forests classified fairly well against data that has only two classes. Now we will see how our implementation of Random Forests fares against data that has 6 classes.

First things first, we got to read in the data! The parameter names and the data itself is in two separate files (we got a problem, Houston!); we need to first read in the data then set set the column names using the file containing the parameter names. Keep one dependent variables to classify, remove the other dependent variables.

```
data_file <- "Faults.NNA"
headers_file <- "Faults27x7_var.txt"
data <- read.table(data_file, sep = "" , header = F, na.strings ="",
                   stringsAsFactors= F)
data_headers <- scan(headers_file, what="", sep="\n")
colnames(data) <- data_headers

#Type of dependent variables (7 Types of Steel Plates Faults):
#1.Pastry
#2.Z_Scratch
#3.K_Scatch
#4.Stains
#5.Dirtiness
#6.Bumps
#7.Other_Faults
# change the values of the other categories to correspond the number of the type of steel
# fault

data = subset(data,select=-c(Other_Faults))
labels = subset(data, select=c("Pastry", "Z_Scratch", "K_Scatch", "Stains", "Dirtiness",
                               "Bumps"))

data <- (data[complete.cases(data),])
# now to make the categories into one columns
Fault_Type = names(labels)[max.col(labels == 1L)]
to_remove <- c("Pastry", "Z_Scratch", "K_Scatch", "Stains", "Dirtiness", "Bumps")
data <- data[ , !(names(data) %in% to_remove)]
data = cbind(data,Fault_Type)

data = within(data, Fault_Type <- factor(Fault_Type, labels = c(6,5,3,1,4,2)))
```

Delete any NA values in the dataset and make column names such that they contain no spaces (all spaces are replaced with underscores).

```
#Purge rows containing N/A entries
data <- (data[complete.cases(data),])

# Remove spaces from column names
names(data) = gsub(" ","_", names(data))
```

Before we run Random Forest, lets split the dataset. Here, we use 70% for training set and 30% for testing set.

```
# first permutate the data because the data needs to be random
split = sample(2, nrow(data), prob=c(0.7,0.3), replace=TRUE)
train_subset = data[split==1,]
test_subset = data[split==2,]
```

Splitting the labels and the data in order to run Random Forest.

```
# Training set labels
train_labels = as.factor(unlist(train_subset[ncol(train_subset)]))

# Testing set labels
test_labels = as.factor(unlist(test_subset[ncol(test_subset)]))
train_subset = train_subset[,-ncol(train_subset)]

# To perform classification
test_subset = test_subset[,-ncol(test_subset)]
num_vars = 5
num_trees = 800
ctrl = rpart.control(cp = 0.004, minsplit = 500, xval = 10)
f = PerformClassification(train_subset,train_labels,test_subset,test_labels,num_trees,
                          num_vars, ctrl)
```

```
## [1] "Results:"
## [1] "Accuracy = 50%"
## [1] "Timings: "
##    user  system elapsed
##   14.81    0.00   15.54
```

We obtained an accuracy of around 50%. This is significantly worse than the Breast Cancer data, which had only binary variables. There are more variables, more observations and more classes, so it should be more difficult to have as high of accuracy as the Breast Cancer Data. Like in the previous performs, we will try to add a control to try to increase the accuracy.

6 Wrap-Up

We enjoyed working on this project a lot. Researching about bootstrapping, bagging and Random Forest algorithm is very interesting. We came across hurdles while implementing the algorithm, issues concerning with runtime, complexity of the algorithm and alternatives for sections that we didn't have sufficient time to learn.

6.1 Future Work

Of course, there's room for improvement. Using Random Forest to do regression, our R^2 was not as high as we wanted. The MSE was satisfactory as well, as a consequence to the R^2 being low. For classification,

as the number of classes increased the accuracy dropped respectively. If time allowed, both of these can be solved by delving deeper into how to split the feature space manually and taking full control of how every tree in the Random Forest algorithm is constructed. Doing this will replace the use of `Rpart`, a black-box throughout the entire project that was very difficult to grasp how to control it. Replacing `Rpart` with our own algorithm also requires that all decision trees are built manually. Since R is not made for object-oriented programming, this will be easier to implement in a object-orientated language such as Python or C. C is preferred as R is built in C and C is low level language which makes it easier to optimize algorithms. This can be very big next step in improving this Random Forest algorithm.

7 Appendix

NOTE: The following libraries are used throughout the code: `tidyverse`, `dplyr`, `rpart`, and `rpart.plot`.

7.1 Random Forest Snippets

7.1.1 `BT_Tree`: Generate Decision Trees

```
BT_Tree = function(dat, labels, p, tree.print = "None", ctrl = "None") {
  # Set up Data to use
  dat = cbind(dat, labels) # add labels together with dataframe
  last_col = ncol(dat) # register the labels
  pred_name = paste(colnames(dat)[ncol(dat)],paste(" ~"))
  sample.dat = sample_n(dat, nrow(dat), replace=TRUE)
  sample.p.dat = sample(sample.dat[,-last_col], p, replace=FALSE)
  param = paste(pred_name, paste(names(sample.p.dat), collapse = " + "))
  sample.p.dat[,colnames(dat)[ncol(dat)]] = sample.dat[,last_col]

  # Set up Control sets and then perform the feature splits
  if (ctrl == "None"){
    ctrl = rpart.control(cp = 0.005, minsplit = 500, xval = 10)
  }

  trees = rpart(formula=param, data=sample.p.dat, control = ctrl)
  if (!(tree.print == "None")) {
    rpart.plot(trees)
  }
  ret = list()
  ret[[1]] = trees
  ret[[2]] = anti_join(dat, sample.dat, by=names(dat))
  ret[[3]] = trees[["variable.importance"]]
  return(ret)
}
```

7.1.2 `Get_Forest`: Generate the Random Forest

```
Get_Forest = function(dat, labels, B, p, print.tree = "None", ctrl = "None"){
  forest = list()
  for (i in 1:B) {
    forest[[i]] = BT_Tree(dat, labels, p, "None", ctrl)
  }
```

```
  return(forest)
}
```

7.1.3 `Classify`: Classification

```
Classify = function(forest, obs){
  # Will add all the predictions, then divide by numTrees to get average
  predictions = 0
  numTrees = length(forest)
  obs = as.data.frame(obs)
  pred_labels = list()
  i = 0
  # This for loop will add the predictions of every tree together, so they can be
  # aggregated
  for (i in 1:numTrees){
    predictions = predict(forest[[i]][[1]], obs)
    pred_labels[[i]] = colnames(predictions)[apply(predictions,1,which.max)]
  }
  pred_labels.matrix = do.call(cbind, pred_labels)
  ag_labels = apply(pred_labels.matrix, 1, function(x) as.numeric(as.character(names(
    which.max(table(x))))))
  return(ag_labels)
}
```

7.1.4 `Regress`: Regression

```
Regress = function(forest,obs){
  # Will add all the predictions, then divide by numTrees to get average
  predictions = 0
  numTrees = length(forest)
  obs = as.data.frame(obs)
  i = 0

  # This for loop will add the predictions of every tree together, so they can be
  # aggregated
  for (i in 1:numTrees){
    predictions = predictions + predict(forest[[i]][[1]], obs)
  }

  predictions = predictions/numTrees
  return (predictions)
}
```

8 References

Gareth, J. *An Introduction to Statistical Learning: with Applications in R*. Springer.

Hastie, T. *The elements of statistical learning [electronic resource] : data mining, inference, and prediction* (2nd ed.). Springer.

Random forest. (2018). *En.wikipedia.org*. Retrieved 22 January 2018, from https://en.wikipedia.org/wiki/Random_forest

Rpart Reference Document