Final Report

Problem Introduction:

Our project dives into the world of financial data and aims to predict retained earnings of different companies using key statistics found on the companies' income and cash flow statements as well as market-influenced metrics. We were motivated by the potential applications of a highly-precise model that predicts a company's retained earnings. By definition, retained earnings is a value recorded under shareholders' equity on the balance sheet, and refers to the percentage of a company's net earnings not paid out to investors as dividends, but retained by the company to be reinvested in its core business, or to pay debt. We are interested in predicting this value since it provides significant insights into a company's financial well-being, and can be a very useful reference for business analysts and portfolio managers to adjust their clients' investments. Throughout the next few sections, we will describe in details the process behind preliminary data processing, variable selection, and parameter tuning. Our objective throughout the project was to build a model that minimizes the APSE.

Data and Preprocessing:

The data set used in this project is collected from Quandl.com and published by Sharadar, an independent research and analytics company that specializes in extraction, standardization and organization of financial data from company filings. The **Core US Fundamentals Data** consists of over 100 essential and fundamental financial metrics for over 10000 companies, gathered over 16 years. These metrics are gathered from publically released **income statements**, **cash flow statements**, **balance sheets**, and **closing prices**. The value of interest as explained earlier is retained earnings, which is denoted as **retearn** in the data set.

One of the biggest challenges in this data set is the amount of missing data. Since 5000 of the companies have been delisted already, a lot of those companies have several columns with missing values. In order to retain the most amount of information possible, we need to do some pre-processing.

Below is a quick preview of what the data is like:

```
options(scipen=999)
all_data=read.csv('SHARADAR-SF1-2.csv')
dim(all_data)
```

```
## [1] 10000   111
```

```
head(all_data[,c(3:13)],10)
```

```
##    calendardate    datekey reportperiod lastupdated      accoci
## 1    2011-12-31 2011-10-31   2011-10-31  2018-03-06   116000000
## 2    2012-12-31 2012-10-31   2012-10-31  2018-03-06  -111000000
## 3    2013-12-31 2013-10-31   2013-10-31  2018-03-06    91000000
## 4    2014-12-31 2014-10-31   2014-10-31  2018-03-06  -334000000
## 5    2015-12-31 2015-10-31   2015-10-31  2018-03-06  -391000000
## 6    2016-12-31 2016-10-31   2016-10-31  2018-03-06  -503000000
## 7    2013-12-31 2013-12-31   2013-12-31  2018-02-26          NA
## 8    2014-12-31 2014-12-31   2014-12-31  2018-02-26 -1316000000
## 9    2015-12-31 2015-12-31   2015-12-31  2018-02-26 -1600000000
## 10   2016-12-31 2016-12-31   2016-12-31  2018-02-26 -3775000000
##         assets   assetsavg    assetsc    assetsnc assetturnover   bvps
## 1   9057000000  8625750000 5569000000  3488000000         0.767 12.415
## 2  10536000000  9701250000 4629000000  5907000000         0.707 14.891
## 3  10686000000 10551000000 4983000000  5703000000         0.369 15.922
```

```
## 4  10815000000 10714750000 5509000000  5306000000         0.378 15.919
## 5   7479000000  7415250000 3686000000  3793000000         0.545 12.627
## 6   7794000000  7617500000 3635000000  4159000000         0.552 13.015
## 7            NA          NA         NA          NA            NA     NA
## 8  18680000000          NA 2917000000 15763000000            NA     NA
## 9  16413000000 17546500000 2566000000 13847000000         0.638     NA
## 10 16741000000 16577000000 3181000000 13560000000         0.562 30.918
```

The csv file has about 10,000 observations, and about 111 columns, which means it has a maximum of 110 predictors, and 1 response column.

It can be seen that the data has quite a few blanks in there. To see how many of the rows are fully usable (including outliers), we can examine how much data is left, after removing all the NAs.

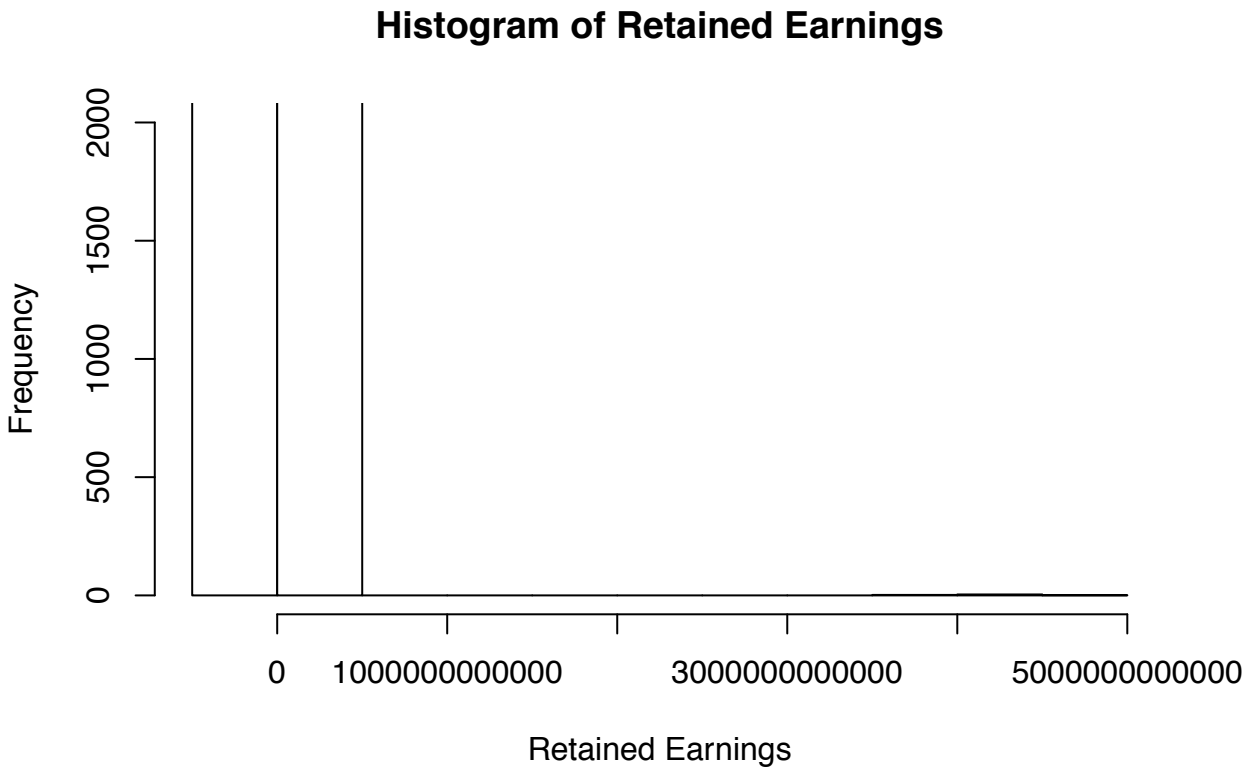
```
all_data=read.csv('SHARADAR-SF1-2.csv')
all_data=na.omit(all_data)
dim(all_data)
```

```
## [1] 5351  111
```

About half of the data gets removed, but the observations that currently exist now are unscaled, and may contain outliers

To figure out if it is worth scaling, we can look at a few histograms to see the overall spread

```
hist(all_data[,'retearn'],ylim=c(0,2000),xlab='Retained Earnings',ylab='Frequency',main='Histogram of Re
     col='white')
```




```
summary(all_data[,'retearn'])
```

```
##          Min.       1st Qu.        Median          Mean       3rd Qu.
##  -17470000000    -137300000        401000    6395000000     311400000
##          Max.
```

```
## 4656000000000
```

If we were to evaluate the goodness of fit of a model using APSE, the numbers that we would get will be extremely large to look at (around 26 digit), in order to address this, the data needs to be scaled down. There are many ways to do this such as:

Standard Scaler

The Standard Scaler is defined by the formula

$$\frac{x_i - mean(x)}{stddev(x)}$$

In Standard Scaler (Gaussian), the objective is to normalize the data by subtracting the mean, and then dividing by the standard deviation. Doing this brings a lot of the data closer together as in a cluster. However, the issue with this method is that outliers will have a strong influence on where the mean ends up being dictated, and the overall structure of the data gets changed to become more like gaussian. If the underlying data is not gaussian, this could end up causing issues. In the above histogram, most of the data is centred around 0 already, but there are a few very large outliers.

Min Max Scaler

The Min Max Scaler is defined by the formula

$$\frac{x_i - min(x)}{max(x) - min(x)}$$

In Min Max Scaler, the objective is to reduce all the values to go from 0 to 1, which can be achieved by dividing all the values in the column by the maximum value in the column. The issue with this method is that if there are any extremely large outliers, most of the data ends up being around 0 to 0.1, and the few outliers end up being around 1. In the above histogram, it can be seen that the data has a few extrmely large outliers.

Robust Scaler

The Robust Scaler is defined by the formula

$$\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}$$

It follow a similar approach to the min max scaler, except this scaler relies on the Interquartile Range. The advantage this technique have over the other methods is that this one is not sensitive to outliers, since it relies on the data's percentiles in order to do the transformations, which the histogram clearly showed the data does have some extremely large outliers. This technique will be able to scale the data down quite well, while at the same time preserving the natural underlying structure of the data

Scaling and Splitting

```
import pandas as pd
from sklearn.preprocessing import RobustScaler,robust_scale
import numpy as np
from sklearn.model_selection import train_test_split
all_data=pd.read_csv('SHARADAR-SF1-2.csv')
```

```
no_blanks=all_data.dropna()
#run: len(no_blanks['retearn']), will get = 5351, which means 5351 have no blanks
blanks=all_data.loc[pd.isnull(all_data).any(1),:]
#run: len(blanks['retearn']), will get = 4649, which means 4649 have a blank in their row

no_blanks=no_blanks.drop(no_blanks.columns[[0,1,2,3,4,5]],axis=1)

scaled=pd.DataFrame(robust_scale(no_blanks))
column_names=list(no_blanks)
scaled.columns=column_names
new_scaled=scaled.set_index('retearn').reset_index()

all_data=new_scaled

Y=all_data['retearn']
all_data=all_data.drop('retearn',1)

X_train, X_test, Y_train, Y_test = train_test_split(all_data,Y,test_size=0.1, random_state=123456)

pd.DataFrame.to_csv(X_train,'X_train.csv',index=False)
pd.DataFrame.to_csv(X_test,'X_test.csv',index=False)
Y_train=np.transpose((Y_train.values)[np.newaxis])
Y_test=np.transpose((Y_test.values)[np.newaxis])
np.savetxt('Y_train.csv',Y_train)
np.savetxt('Y_test.csv',Y_test)
```

The above code segment reads the csv file in Python, and then removes all the rows which contain blanks. It then removes the first 5 columns, since they are only timestamps and not relevant to our problem (it would be a relevant factor if we had serially correlated time-series data on a particular company, but in our data set, each row represents a different company). Finally, the function applies the Robust Scaler transformation to each of the columns and then splits the data into the training and the test set, using 123456 as the seed value for the random number generator. We decided to go with a splitting scheme of 90% training data and 10% testing data. The python script creates four file: X_train, Y_train, X_test, Y_test, which are used in order to look at the APSE of a leave 1 out cross validation. Then for the final assessment, a 5 fold cross validation was used, to do that, the data was split 4 more times as shown below.

```
import pandas as pd
from sklearn.preprocessing import RobustScaler,robust_scale
import numpy as np
from sklearn.model_selection import train_test_split
all_data=pd.read_csv('SHARADAR-SF1-2.csv')
no_blanks=all_data.dropna()
#run: len(no_blanks['retearn']), will get = 5351, which means 5351 have no blanks
blanks=all_data.loc[pd.isnull(all_data).any(1),:]
#run: len(blanks['retearn']), will get = 4649, which means 4649 have a blank in their row

no_blanks=no_blanks.drop(no_blanks.columns[[0,1,2,3,4,5]],axis=1)

scaled=pd.DataFrame(robust_scale(no_blanks))
column_names=list(no_blanks)
scaled.columns=column_names
new_scaled=scaled.set_index('retearn').reset_index()

all_data=new_scaled
```

```
Y=all_data['retearn']
all_data=all_data.drop('retearn',1)
#DUPES 2#

for i in (range(2,6)):
    X_traini, X_testi, Y_traini, Y_testi = train_test_split(all_data, Y, test_size=0.1, random_state=i)
    text_new_train = 'X_train' + str(i) + ".csv"
    text_new_test = 'X_test' + str(i) + ".csv"
    pd.DataFrame.to_csv(X_traini, text_new_train, index=False)
    pd.DataFrame.to_csv(X_testi, text_new_test, index=False)
    Y_traini = np.transpose((Y_traini.values)[np.newaxis])
    Y_testi = np.transpose((Y_testi.values)[np.newaxis])
    text_new_train = 'Y_train' + str(i) + ".csv"
    text_new_test = 'Y_test' + str(i) + ".csv"
    np.savetxt(text_new_train, Y_traini)
    np.savetxt(text_new_test, Y_testi)
```

The above code segment creates a for loop which creates 4 more randomely generated copies of the above files.

A quick preview of the train and test is below:

X__train/X__test

```
all_data=read.csv('X_train.csv')
head(all_data[,c(1:10)],10)
```

```
##              accoci       assets   assetsavg     assetsc    assetsnc
## 1      0.0005555556 -0.167008018 -0.16578359 -0.18835855 -0.13316412
## 2     -0.0056666667 -0.175240418 -0.17182569 -0.21753588 -0.13134623
## 3      0.2491111111 -0.187598955 -0.18333210 -0.24767911 -0.13590715
## 4      0.0005555556 -0.164381563 -0.15764459 -0.19569177 -0.12486147
## 5      0.0005555556 -0.007750646 -0.04222501  0.17059290 -0.05865145
## 6      0.0005555556 -0.190833518 -0.18745944 -0.25618120 -0.13677379
## 7      0.0005555556  0.290797831  0.32990545 -0.03774164  0.55139119
## 8      0.0005555556  0.510196773  0.58021154  0.06729673  0.86192032
## 9      0.0005555556 -0.184171140 -0.18117892 -0.25336451 -0.12714504
## 10 -1091.3327777800 37.435051703 37.77002140 34.78918624 44.02680251
##    assetturnover       bvps        capex    cashneq  cashnequsd
## 1     -0.4044032  0.2887832   0.13271749 -0.0444367 -0.04441221
## 2      1.2549247 -0.3693222   0.13232521 -0.1970320 -0.20507190
## 3      0.4553882 -0.5267539   0.13337551 -0.2404181 -0.25075080
## 4      4.8238702 -0.4646056   0.14033534 -0.2448944 -0.25546366
## 5      7.6662804 -0.1511692   0.00635242 -0.1455519 -0.15087112
## 6      0.8377752 -0.5929449   0.14384056 -0.2446156 -0.25517020
## 7      0.5909618  1.1823226  -1.15443214 -0.1954266 -0.20338161
## 8      1.1726535 -0.1078082  -0.74535906  0.2113865  0.22493075
## 9     -0.6604867 -0.3108205   0.14433439 -0.2397363 -0.25003297
## 10    -0.1425261  1.3399921 -49.42158810 36.5848198 38.52062579
```

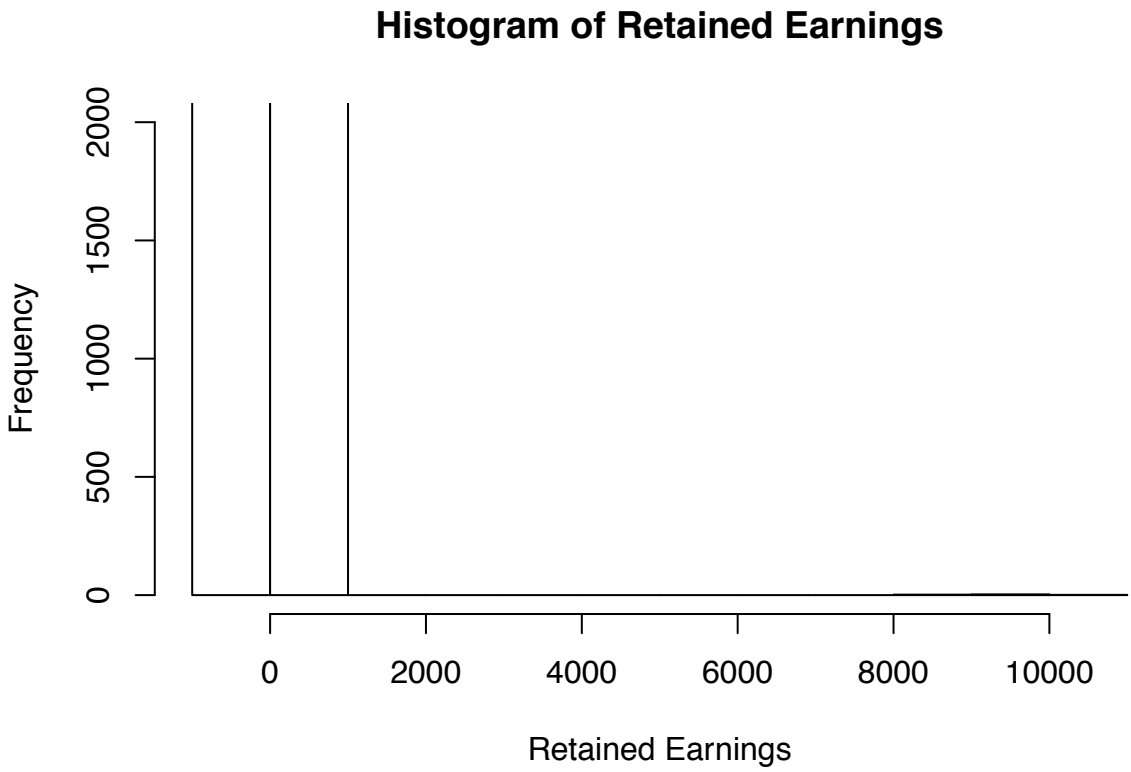
Y_train/Y_test

```
all_data=read.csv('Y_train.csv',header=F)
head(all_data[,1],10)
```

```
##  [1] -0.563392524 -0.355265577 -0.137120634  0.009097545  0.011038740
##  [6] -0.052410058  1.888638656  4.485896243 -0.140858503 67.613340985
```

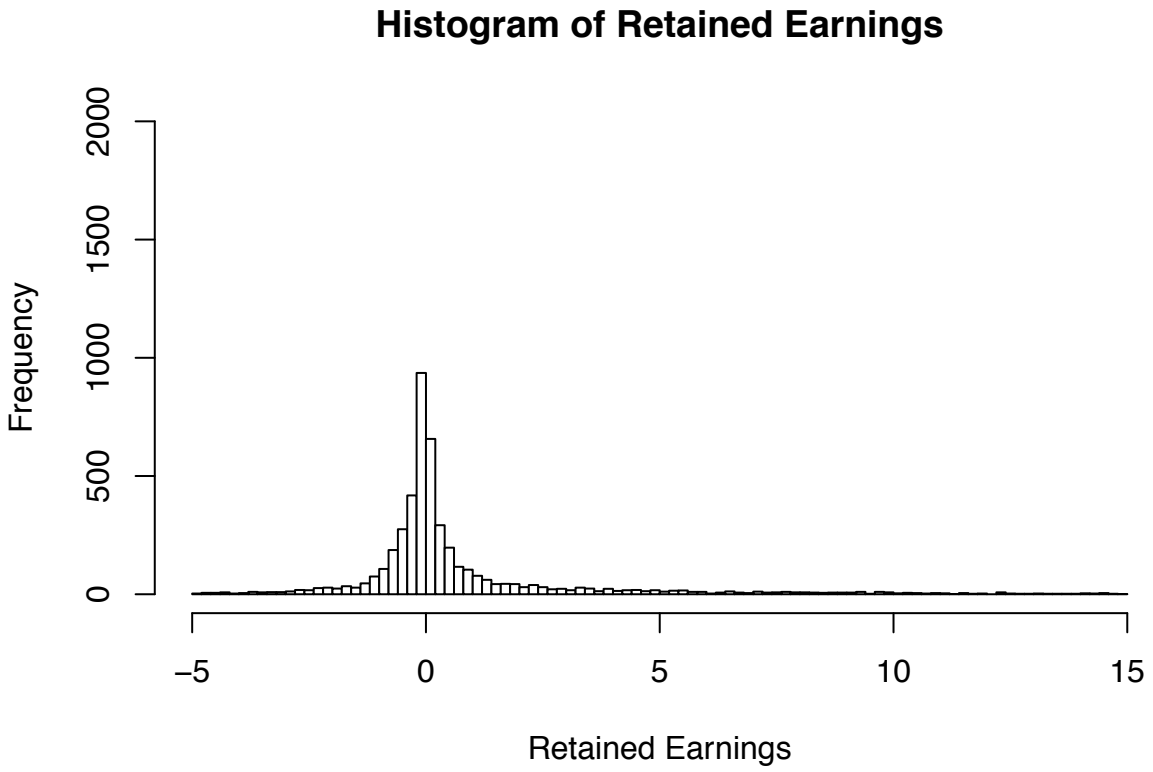
And the new scale of the dataset is:

```
hist(all_data[,1],ylim=c(0,2000),xlab='Retained Earnings',ylab='Frequency',main='Histogram of Retained
     col='white')
```



For a better preview, we can ignore the extreme values

```
all_data=subset(all_data, V1<=15 & V1>=-5)
hist(all_data[,1],breaks=100,ylim=c(0,2000),xlab='Retained Earnings',ylab='Frequency',main='Histogram o
     col='white')
```




```
all_data=read.csv('Y_train.csv',header=F)
summary(all_data[,1])
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
##   -38.950     -0.305      0.002     13.650      0.724 10380.000
```

Additionally, an important value to consider in this data set is **fxusd**, where a non-zero value indicates that foreign exchange rate was applicable in gathering the data. We believed that a company's country of origin would be a reasonable indicator value to have in determining its retained earnings, therefore we create a new categorical variable called **foreign** in the code segment below to indicate whether the company report is domestic or foreign. This variable would later be used in tree-based methods such as random forest and boosting. We also load any necessary libraries and pre-processed data in the code below.

```
library(randomForest)
```

```
## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.
```

```
library(mgcv)
```

```
## Warning: package 'mgcv' was built under R version 3.3.2
## Loading required package: nlme
## This is mgcv 1.8-22. For overview type 'help("mgcv-package")'.
```

```
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 3.3.2
## Loading required package: survival
## Loading required package: lattice
```

```
## Loading required package: splines
```

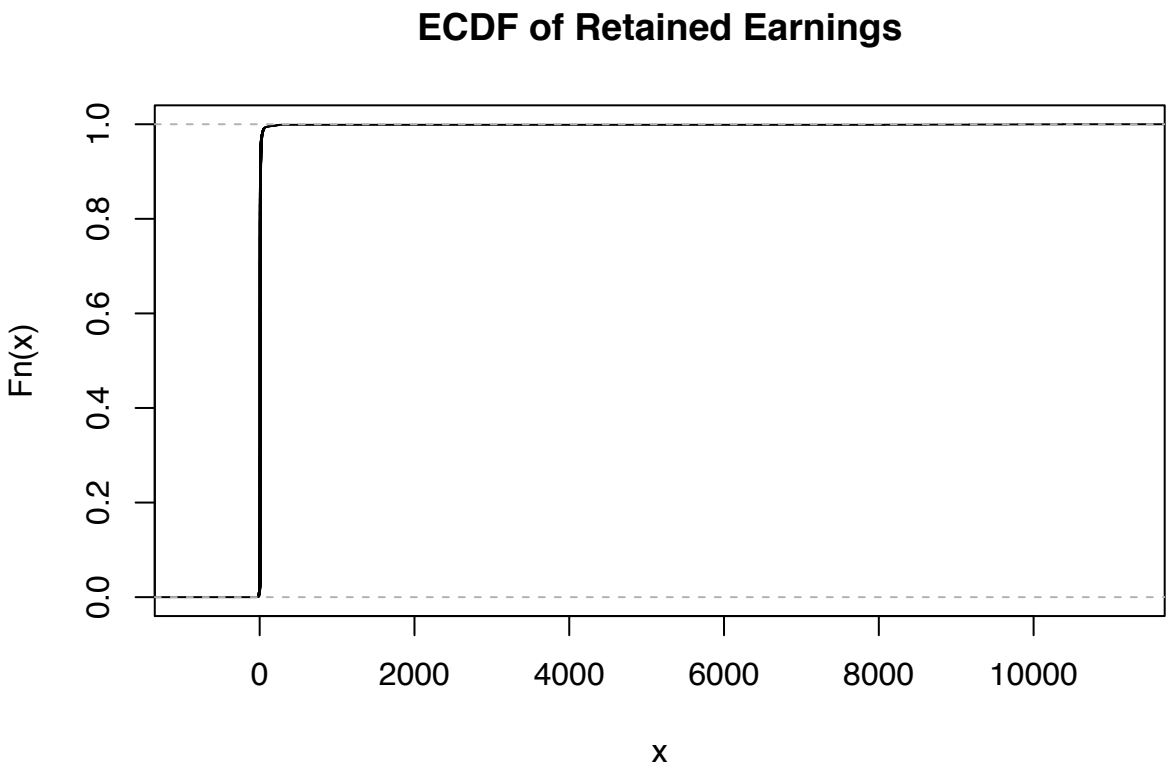
```
## Loading required package: parallel
```

```
## Loaded gbm 2.1.3
```

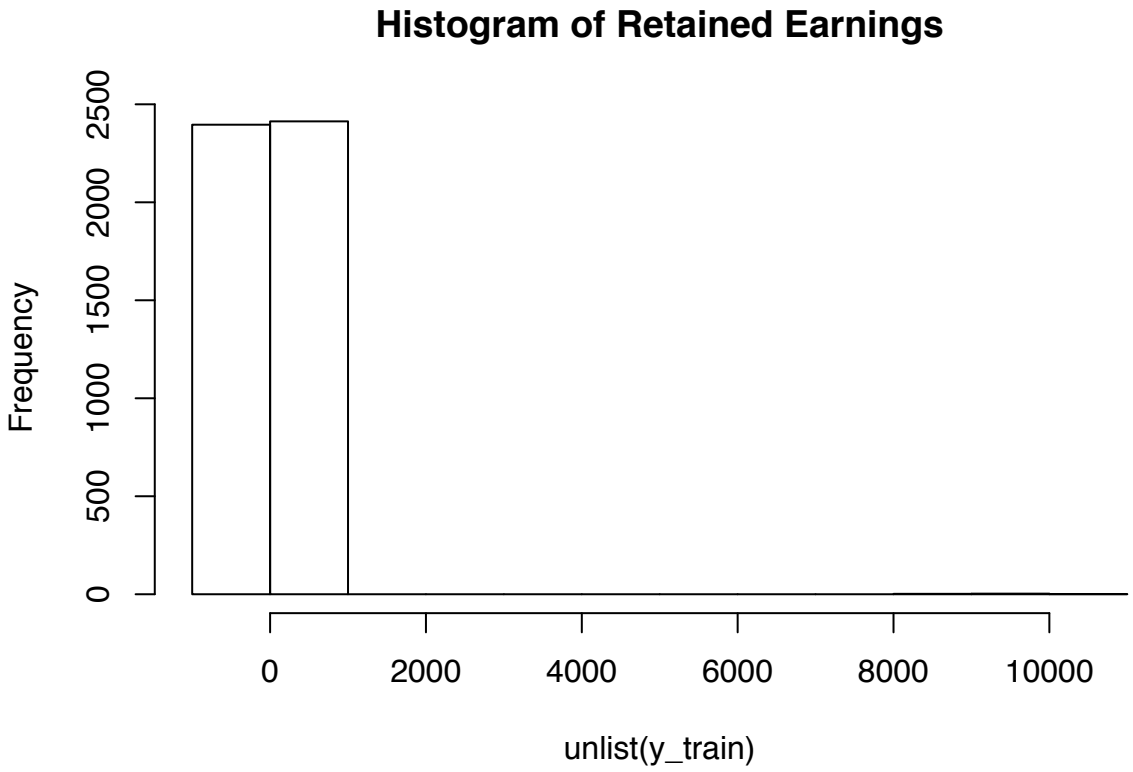
```
options(warn=-1)
# Reading data from csv cleaned files
x_test = read.csv ("X_test.csv",header = T)
y_test = read.csv ("Y_test.csv",header = F)
x_train = read.csv("X_train.csv",header = T)
y_train = read.csv ("Y_train.csv",header = F)

x_train$foreign <- factor(ifelse(x_train$fxusd!=0.000, "Foreign", "Domestic"))
x_test$foreign <- factor(ifelse(x_test$fxusd!=0.000, "Foreign", "Domestic"))
```

Next we need to empirically visualize the response variable, we do so by plotting the empirical CDF, the empirical PDF (histogram), the scatter plot, as well as querying the range of teh response variable:

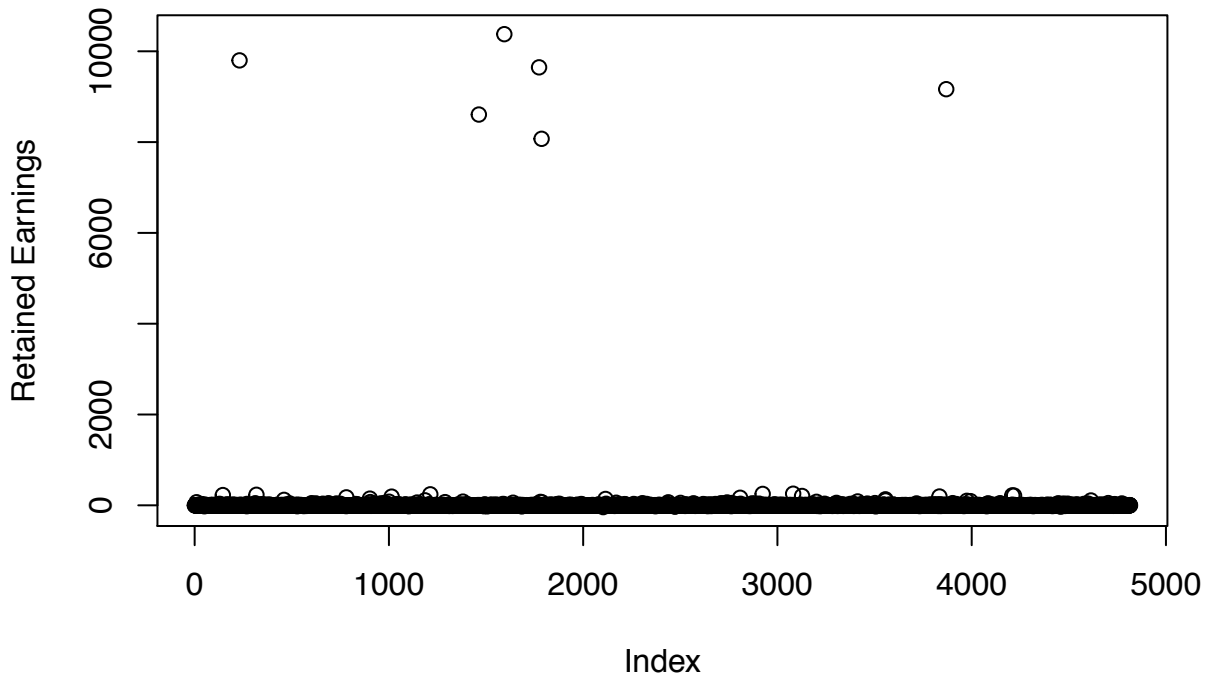
```
y_train_ecdf <- ecdf(unlist(y_train))
plot(y_train_ecdf, main = "ECDF of Retained Earnings")
```




```
hist(unlist(y_train), main = "Histogram of Retained Earnings")
```




```
plot(unlist(y_train), xlab = "Index", ylab = "Retained Earnings")
```



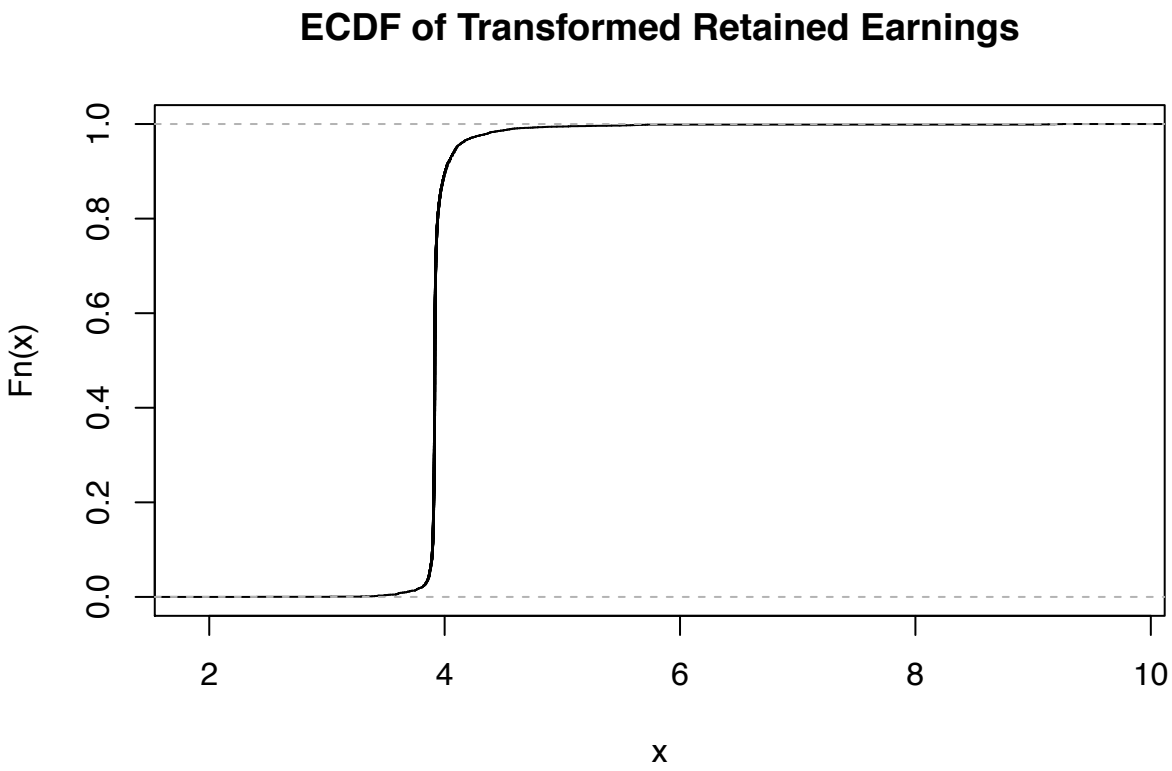

```
range(y_train)
```

```
## [1]   -38.94516 10377.12598
```

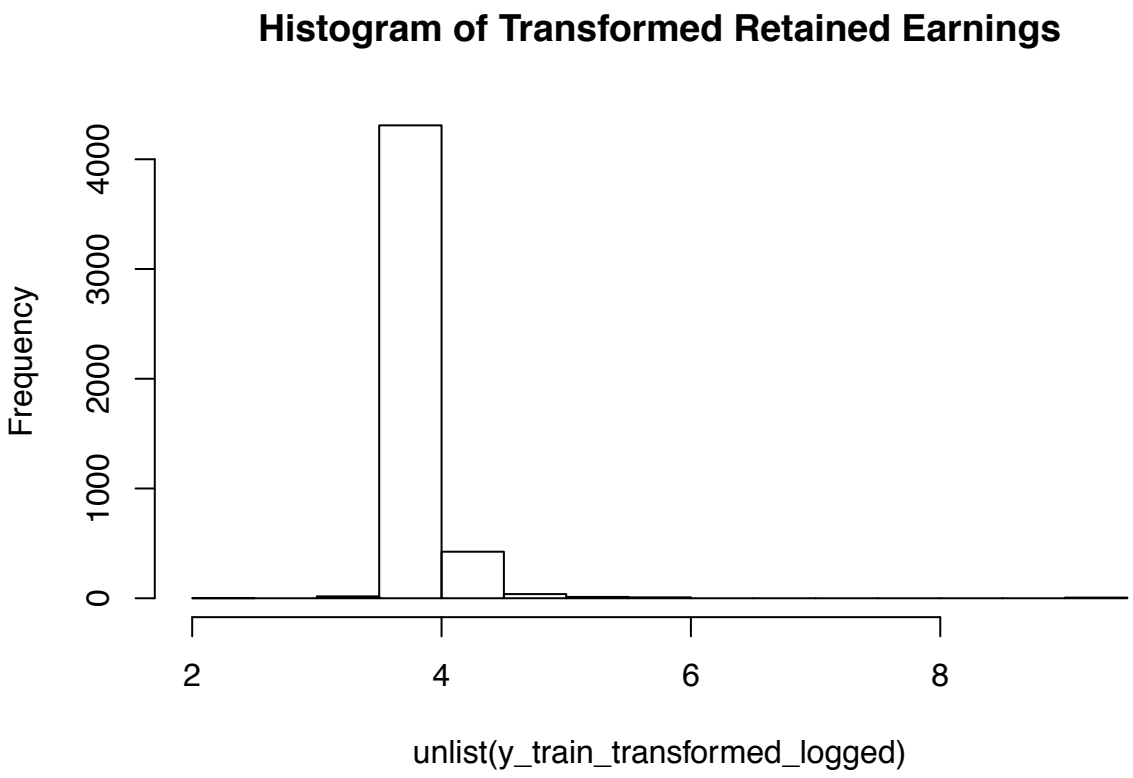
We can see from the plots and the range, that even after all the pre-processing, the response variable has outliers that could drastically affect the function fitting. In the later stages of model building, we would rectify that by first vertically translating the response variable by 50 units (so that all values are positive), then

taking the natural log. The value 50 was arbitrary and could be any integer $i \geq 39$, since our main objective was to get rid of negative values so we coudl take natural logs. We will peform the same transformation on the test response as well. This variable transformation gives us a much narrower range of values to work with and would result in more precise models.

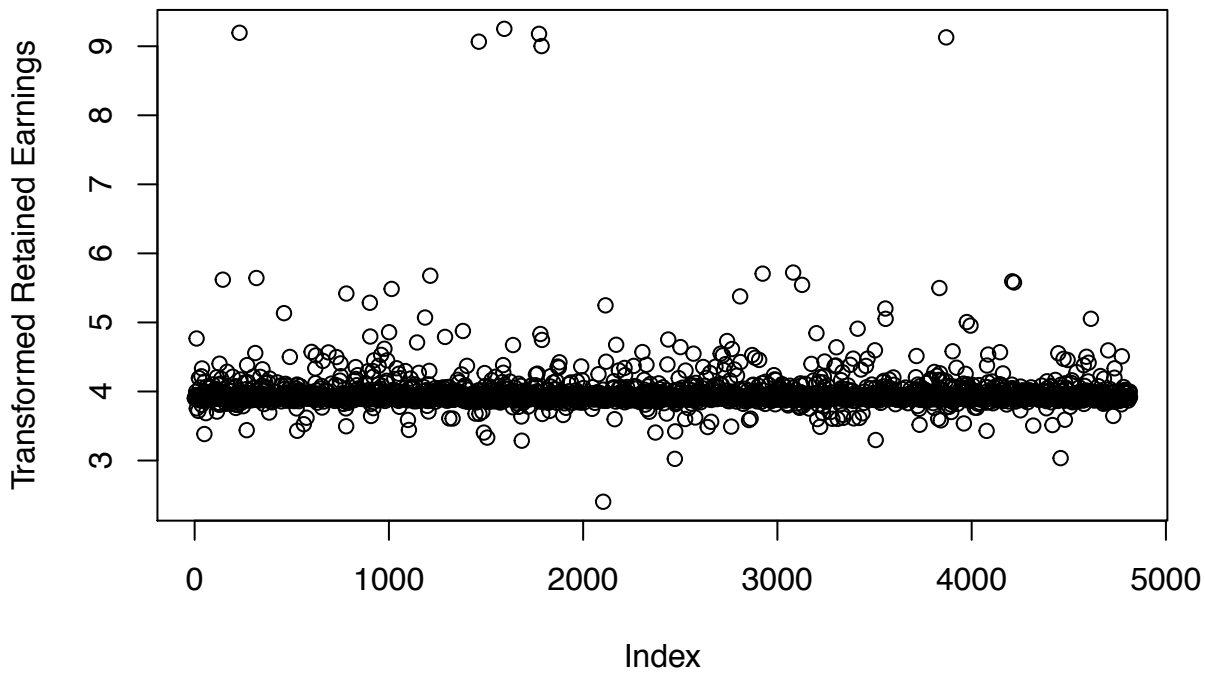
```
y_train_transformed_logged <- log(y_train + 50)
y_test_transformed_logged <- log(y_test + 50)
y_train_transformed_logged_ecdf <- ecdf(unlist(y_train_transformed_logged))
plot(y_train_transformed_logged_ecdf, main = "ECDF of Transformed Retained Earnings")
```




```
hist(unlist(y_train_transformed_logged), main = "Histogram of Transformed Retained Earnings")
```




```
plot(unlist(y_train_transformed_logged), xlab = "Index", ylab = "Transformed Retained Earnings")
```




```
range(y_train)
```

```
## [1]   -38.94516 10377.12598
```

Note that we are still keeping the outliers because while those data points are outliers, we should make our model work for any generalized situations instead of only for perfectly distributed data sets. Instead we decided to keep the outliers but transform the variable as a whole so that even outliers are within a

manageable range of values. We will use methods such as Tukey's bi-weight to down-weigh the outliers during variable selection and model tuning.

Variable Selection:

In class, we learened about the "curse of dimensionality", which states that as the number of dimensions increases, the volume of the sample space increases so fast that the space becomes sparse. This is clearly present in our project, as we have about 104 initial predictor values.

Approach to Initial Screening:

To tackling this issue, we initially screened the variables by determining the lowest APSE of each individual variable of each of the following methods: 1) Linear Model 2) Trees 3) Smoothing Splines

The code below shows initial screening of APSEs: (A preview of first 10 variables are shown below)

```
options(scipen = 999)
options (warn = -1)
library(tree)

# Reading data from csv cleaned files
x_test = read.csv ("X_test.csv",header = T)
y_test = read.csv ("Y_test.csv",header = F)
x_train = read.csv("X_train.csv",header = T)
y_train = read.csv ("Y_train.csv",header = F)

APSE_Formula=function(Y_True,Y_predicted){
  diff_vector=(Y_True-Y_predicted)^2
  mean_num=mean(diff_vector)
  median_num=median(diff_vector)
  mm_vector = c(mean_num,median_num)
  return (mm_vector)
}

# initialize variables
col_names = names(x_train)
APSE_matrix = matrix (col_names,nrow = 1,ncol = 104)
lm_mean = c()
lm_median = c()
t_mean = c()
t_median = c()
s_mean = c()
s_median = c()
over_mean = c()
gam_mean = c()
gam_median = c()
```

```
for (i in 1:104) {
  # Linear fit
  l_fit = lm(unlist(y_train) ~ eval (parse (text = col_names[i])),
             data = cbind(x_train,y_train))
  pred_l_y = predict (l_fit,x_test)
  APSE_l_fit = APSE_Formula (y_test,pred_l_y)
  lm_mean = c(lm_mean,APSE_l_fit[1])
```

```
  lm_median = c(lm_median,APSE_l_fit[2])

  # Tree fit
  tree_fit = tree(unlist(y_train) ~ eval (parse (text = col_names[i])),
                  data = cbind(x_train,y_train))
  pred_t_y = predict (tree_fit,x_test)
  APSE_t_fit = APSE_Formula (y_test,pred_t_y)
  t_mean = c(t_mean,APSE_t_fit[1])
  t_median = c(t_median,APSE_t_fit[2])

  # Smoothing Spline fit
  smooth_fit = smooth.spline(x=x_train[,col_names[i]],
                             y = unlist(y_train),tol=1e-4)
  pred_s_y = unlist(predict (smooth_fit,x_test[,col_names[i]])[2])
  APSE_s_fit = APSE_Formula (y_test,pred_s_y)
  s_mean = c(s_mean, APSE_s_fit[1])
  s_median = c(s_median,APSE_s_fit[2])

  avg_APSE = (APSE_l_fit[1] + APSE_l_fit[2] + APSE_t_fit[1] +
    APSE_t_fit[2] + APSE_s_fit[1] + APSE_s_fit[2])/6
  over_mean = c(over_mean,avg_APSE)
}

APSE_matrix = rbind (APSE_matrix,lm_mean)
APSE_matrix = rbind (APSE_matrix,lm_median)
APSE_matrix = rbind (APSE_matrix, t_mean)
APSE_matrix = rbind (APSE_matrix, t_median)
APSE_matrix = rbind (APSE_matrix, s_mean)
APSE_matrix = rbind (APSE_matrix, s_median)
APSE_matrix = rbind (APSE_matrix, over_mean)
APSE_matrix = rbind (APSE_matrix, gam_mean)
APSE_matrix = rbind (APSE_matrix, gam_median)

APSE_matrix[,1:10]
```

```
##           [,1]               [,2]               [,3]
##           "accoci"           "assets"           "assetsavg"
## lm_mean   "205839.591134999" "615.308497566787" "549.776056122691"
## lm_median "93.5972793919377" "14.2734341877955" "13.0889311306196"
## t_mean    "44095.189738612"  "643.790776842222" "643.790776842222"
## t_median  "4.47881055248813" "4.61781217738813" "4.61781217738813"
## s_mean    "90520.4091850435" "423.246227007431" "299.700125789169"
## s_median  "0.20342104942566" "0.119573051460559" "0.121125866339087"
## over_mean "56758.9115949414" "283.559386805514" "251.849137988071"
##           [,4]                [,5]                [,6]
##           "assetsc"           "assetsnc"          "assetturnover"
## lm_mean   "257.722267365775"  "1376.01229264964"  "180153.980436736"
## lm_median "13.0648057144264"  "14.8132056431279"  "195.897385536593"
## t_mean    "643.790776842222"  "643.790776842222"  "180188.404707304"
## t_median  "4.61781217738813"  "4.61781217738813"  "186.982578040779"
## s_mean    "47.3685064339034"  "886.658240297271"  "175077.837853807"
## s_median  "0.122941736290122" "0.136190115281583" "3.69735740109304"
## over_mean "161.114518378334"  "487.671419620821"  "89301.1333864708"
##           [,7]                [,8]                [,9]
```

```
##           "bvps"             "capex"             "cashneq"
## lm_mean   "180188.337223662" "5654.41364229527"  "32133.3722264812"
## lm_median "187.141206983243" "0.560202003742839" "5.08513879968367"
## t_mean    "15341.8872859698" "643.790776842222"  "880.8965196941"
## t_median  "4.25696460770583" "4.61781217738813"  "3.84083623779574"
## s_mean    "142959.850672537" "694.104430912453"  "2314.39598236821"
## s_median  "1.36427478831621" "0.133614311392656" "0.139164634942146"
## over_mean "56447.1396047581" "1166.27007975708"  "5889.62164470265"
##           [,10]
##           "cashnequsd"
## lm_mean   "178229.674870175"
## lm_median "4.63504116500015"
## t_mean    "208006.342447705"
## t_median  "31.8857969527218"
## s_mean    "191054.619902449"
## s_median  "0.13425772494785"
## over_mean "96221.2153860287"
```

The data clearly demonstrates that based on screening on 3 different models on each individual variable, smoothing splines (shown as s_mean and s_median) outperforms all other methods (Trees and Linear Model). We will be using smoothing splines to do variable selection where a minimum APSE will be our target goal.

Since the median APSE is already within 0.1 range, we will be attempting to find the minimum mean APSE by adding different predictors to our current model (By using Thin Plate Splines).

By simple observation and inspection, we see that assets has the lowest APSE in predicting retained earnings. We will be finding another variable which we can add to to further reduce this mean APSE.

The Thin Plate Spline code is demonstrated below:

```
      TPS_fit=Tps(x=x_train[,i],Y=y_train[,1])
      predicted=predict(TPS_fit,x_test[,i])
      new_APSE = APSE_Formula(predicted,y_test)
```

This simple function TPS from the 'fields' library takes about 2.5 minutes to run on 1 predictor. Hence we would need to figure out how to reduce the dimensionality of the data as well as how to run multithreaded processes in order to be more efficient.

With this concern in mind, we see that this is a computational challenging problem as there are 104 predictors and it wouldn't be computationally possible to iterate all predictors to find the best combination of predictors that give best APSE. We took a look at the correlation matrix, which tells us which variables are correlated with each other. The highly correlated ones can be ignored and skipped over when being added to the model. The R code and output is shown below:

```
corr_matrix_train = abs(cor((x_train)))
corr_df=as.data.frame(corr_matrix_train)
corr_df[1:10,1:10]
```

```
##                    accoci       assets    assetsavg      assetsc
## accoci        1.0000000000 0.3484459128 0.3443116819 0.3194302460
## assets        0.3484459128 1.0000000000 0.9998459880 0.9971405688
## assetsavg     0.3443116819 0.9998459880 1.0000000000 0.9968238123
## assetsc       0.3194302460 0.9971405688 0.9968238123 1.0000000000
## assetsnc      0.3625227236 0.9992520929 0.9991816974 0.9934726554
## assetturnover 0.0052644449 0.0156548846 0.0155177819 0.0122111981
## bvps          0.0006622118 0.0004014337 0.0004257735 0.0009433325
## capex         0.3967141482 0.9959393362 0.9958082085 0.9886634456
## cashneq       0.6142239645 0.8953059325 0.8913360861 0.8786459798
```

```
## cashnequsd    0.1453074095 0.1753389145 0.1713837643 0.1792811697
##                    assetsnc assetturnover         bvps        capex
## accoci        0.3625227236   0.005264445 0.0006622118 0.3967141482
## assets        0.9992520929   0.015654885 0.0004014337 0.9959393362
## assetsavg     0.9991816974   0.015517782 0.0004257735 0.9958082085
## assetsc       0.9934726554   0.012211198 0.0009433325 0.9886634456
## assetsnc      1.0000000000   0.017382397 0.0001232577 0.9974602995
## assetturnover 0.0173823969   1.000000000 0.0155668565 0.0136652079
## bvps          0.0001232577   0.015566857 1.0000000000 0.0002612049
## capex         0.9974602995   0.013665208 0.0002612049 1.0000000000
## cashneq       0.9018512065   0.013881373 0.0003680103 0.9057375199
## cashnequsd    0.1729339848   0.021766792 0.0052955333 0.1456993936
##                     cashneq  cashnequsd
## accoci        0.6142239645 0.145307410
## assets        0.8953059325 0.175338915
## assetsavg     0.8913360861 0.171383764
## assetsc       0.8786459798 0.179281170
## assetsnc      0.9018512065 0.172933985
## assetturnover 0.0138813733 0.021766792
## bvps          0.0003680103 0.005295533
## capex         0.9057375199 0.145699394
## cashneq       1.0000000000 0.249537093
## cashnequsd    0.2495370926 1.000000000
```

Any correlation coefficient with a number greater than 0.7 can be ignored due to high correlation. This brought our number of predictors down to about 60 predictors. This is still computationally hard, so we decided to access the school server and run the program on parallel using the doparallel library.

To implement concurrently running threads, we must create a cluster where we detect the number of cores available to us in the server.

Set up code for concurrency is shown below:

```
library(fields)
library(parallel)
library(doParallel)
library(foreach)
library(doSNOW)
library(doMPI)
c1 = parallel::makeCluster(detectCores(logical = TRUE))
registerDoParallel(c1)
```

The full code for executing is shown below to find best selection of variables for our Thin Plate Spline. The first itaeration is shown below where we selected Assets and we are finding the best 2nd variable to add to reduce APSE.

```
col_names = names(x_train)
c1 = parallel::makeCluster(detectCores(logical = TRUE))
registerDoParallel(c1)
system.time({
F1=foreach(i=1:104,.combine = "c",.packages = 'fields') %dopar% {
    print(i)
    if (i == 2) {
    }
    else {
      TPS_fit=Tps(x=x_train[,c(2,i)],Y=y_train[,1])
      predicted=predict(TPS_fit,x_test[,c(2,i)])
```

```
    new_APSE = APSE_Formula(predicted,y_test)
    file_name = paste(i, '.csv', sep = '')
    write_data = as.data.frame(c(col_names[i],new_APSE))
    write.csv(write_data, file_name)
```

After the function terminates, it creates 104 seperate .csv files which we took a look at the mean APSE's of each combination of variables and selected the variable which reduces our mean APSE. Next iteration, we simply added another variable to the TPS_fit as shown below:

```
  TPS_fit=Tps(x=x_train[,c(2,17,i)],Y=y_train[,1])
  predicted=predict(TPS_fit,x_test[,c(2,17,i)])
```

After repeating the process 4 times, we concluded that the best selection of variables are : (assets, debtnc, depamor, ncfdiv)

The mean APSE with the pre-transformed response variable started with 100 for Assets, and dropped to 20 when adding debtnc to our TPS. It further reduced to 12 when depamor was added. And finally 11.51 for ncfdiv. The APSE mean cannot be reduced further from 11.51. This means that, on average, we are off by $\sqrt{11.51} = 3.39$

Finalized Predictors:

From the variable selection procedure, we ended up with four final predictors: assets, debt non-current (with column name **debtnc**), depreciation, amortization & accretion (with column name **depamor**), and payment of dividends & other cash distributions (with column name **ncfdiv**). Additionally, our final set of predictors will also include the categorical variable **foreign** that indicates whether a company is foreign or domestic.

We will subset the data in two different ways. First we created a simple bootstrap re-sampling mechanism to create 100 bootstrapped training and testing sets with the number of observations just slightly smaller than the original sets. These bootstrapped samples will be used in the APSE calculations when tuning model complexity. Additionally, we created a set of training and testing sample sets that will be used for 5-fold cross validation for the random forest model.

```
## First need to re-load the data and re-create the categorical variable
x_test = read.csv ("X_test.csv",header = T)
y_test = read.csv ("Y_test.csv",header = F)
x_train = read.csv("X_train.csv",header = T)
y_train = read.csv ("Y_train.csv",header = F)

x_train$foreign <- factor(ifelse(x_train$fxusd!=0.000, "Foreign", "Domestic"))
x_test$foreign <- factor(ifelse(x_test$fxusd!=0.000, "Foreign", "Domestic"))

## Bootstrap Re-sample pairs of x_train, y_train ----------------------------------------
getSample <- function(n1, n2, x, y, seed_val) {
  set.seed(seed_val)
  sam <- sample(n1, n2)
  x_sample <- x[c(sam), ]
  y_sample <- y[c(sam), ]
  list(x=x_sample, y=y_sample)
}

x_train_final_predictors <- x_train[,c(2, 17, 20, 60, 105)]
x_test_final_predictors <- x_test[,c(2, 17, 20, 60, 105)]

## generate 100 bootstrap sampled training and testing sets
```

```
##   from x_train, y_train, x_test, and y_test
N_S <- 100

Ssamples <- lapply(1:N_S, FUN= function(i){
  getSample(4815, 4000, x_train_final_predictors, y_train_transformed_logged, i)
  })

Tsamples <- lapply(1:N_S, FUN= function(i){
  getSample(536, 500, x_test_final_predictors, y_test_transformed_logged, i)
  })

## 5-fold cross validation sets ------------------------------------------------------------
gen_K_fold <- function(k, pop){
  Ssamples <- vector(mode="list", length = k)
  Tsamples <- vector(mode="list", length = k)
  bin_size = nrow(pop)/k
  for (i in 1:k){
    test_int <- seq(from=(i-1)*bin_size+1, to=i*bin_size)
    rest_int1 <- seq(from=0, to=(i-1)*bin_size)
    rest_int2 <- seq(from=i*bin_size+1, to=nrow(pop))
    Tsamples[[i]] <- data.frame(pop[test_int,])
    Tsamples[[i]]
    Ssample <- data.frame(pop[rest_int1,])
    if (i != k){
      Ssample <- rbind(Ssample, data.frame(pop[rest_int2,]))
    }
    Ssamples[[i]] <- data.frame(Ssample)
  }
  output <- vector(mode="list", length = 2)
  output[[1]] = Ssamples
  output[[2]] = Tsamples
  output
}

pop <- cbind(rbind(x_train_final_predictors, x_test_final_predictors),
             rbind(y_train_transformed_logged, y_test_transformed_logged))
CVSets <- gen_K_fold(5, pop = pop)
```

Smoothing methods:

The smoothing method we will be using is **thin-plate splines**, as implemented in the **GAM** package in CRAN. We decided on this particular spline model based on results from the variable selection procedure as well as the mathematical properties of thin-plate splines, from here on referred to as TPS. From the documentation of GAM package on the CRAN website, we learned that TPS are low-rank isotropic smoothers that can take on any number of covariates. Since it is isotropic, rotation on the covariate coordinate system as a result of change of basis would not affect the result of smoothing. Additionally, these smoothers are low-rank which means that they will have far fewer coefficients than there are data to smooth. This rank reduction is achieved using a truncated eigen-decomposition. Hence this model provides an ideal balance between flexibility and ease of computation.

The hyper parameter we will tune for this model is k, where k is the basis dimension. The following helper functions are defined to help us visualize how the APSE is affected by complexity.

```
ave_y_mu_sq <- function(sample, predfun){
  mean(abs(sample$y - predfun(sample$x))^2)
}
getmuhat_tps <- function(sample, df) {
  fit <- gam(y~ s(x[,1],bs="tp",k = df) + s(x[,2], bs="tp",k=df) +
               s(x[,3], bs="tp",k = df) + s(x[,4], bs="tp",k=df), data = sample)
  muhat <- function(x){predict(fit, x=x)}
  muhat
}

apse <- function(Ssamples, Tsamples, hyper_parameter, which_function = 1){
  N_S <- length(Ssamples)
  mean(
    sapply(1:N_S,
           FUN=function(j){
             S_j <- Ssamples[[j]]
             if (which_function == 1){
               muhat <- getmuhat_tps(S_j, df=hyper_parameter)
             }else if (which_function == 2){
               muhat <- getmuhat_rf_mtry(S_j, mtry=hyper_parameter)
             } else if (which_function == 3){
               muhat <- getmuhat_boost(S_j, depth=hyper_parameter)
             } else if (which_function == 4){
               muhat <- getmuhat_tps_with_interaction(S_j, df=hyper_parameter)
             } else if (which_function == 5){
               muhat <- getmuhat_boost_lambda(S_j, lambda=hyper_parameter)
             }
             T_j <- Tsamples[[j]]

             if (which_function == 3 || which_function == 5){
               ave_y_mu_sq_for_boost(T_j, muhat)
             } else{
               ave_y_mu_sq(T_j, muhat)
             }
           }
    )
  )
}
```

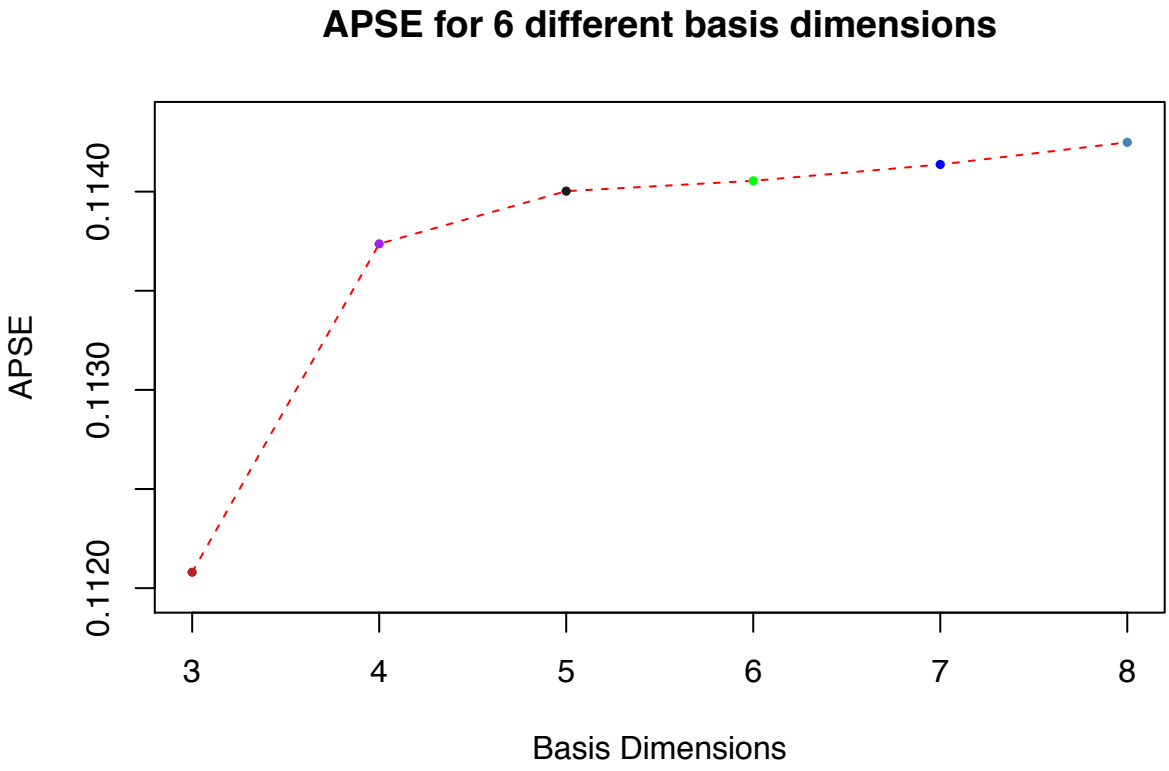
Note that the apse function is generalized to handle other models as well. We will define the other specific *getmuhat* functions in the next couple of sections.

Now let's calculate and plot the APSE of 6 different basis dimensions, averaged over the 100 boostrapped testing sets:

```
apses <- c()
complexity <- c(3, 4, 5, 6, 7, 8)
for (i in 3:8){
  model_apse <- apse(Ssamples, Tsamples, hyper_parameter = i, which_function = 1)
  apses <- c(apses, model_apse)
}
```

```
ylim = extendrange(apses)
plot(complexity, apses, xlab = "Basis Dimensions",
     ylab="APSE", ylim=ylim,
```

```
     main="APSE for 6 different basis dimensions",
     type="n") # "n" suppresses the plot, add values later
lines(complexity, apses, lwd=1, col="red", lty=2)
points(complexity,apses, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



From the graph, we could see that with a thin-plate spline, a 3-dimensional basis gives us the best APSE so far.

The above TPS model was fitted and tested under the assumption of no interaction between the predictors. We will model predictor interactions in the following code segment:

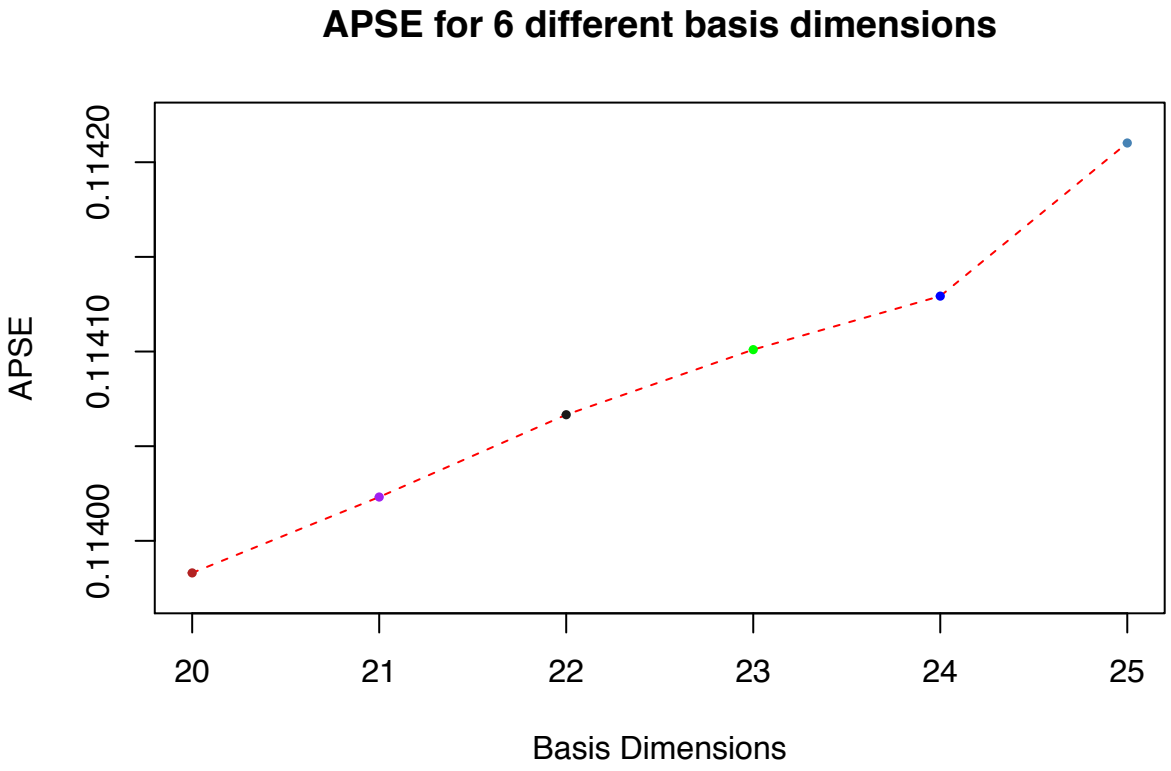
```
getmuhat_tps_with_interaction <- function(sample, df) {
  fit <- gam(y~ s(x[,1], x[,2], x[,3], x[,4], bs="tp",k = df), data = sample)
  muhat <- function(x){predict(fit, x=x)}
  muhat
}
```

Now let's plot the APSEs again:

```
apses <- c()
complexity <- c(20, 21, 22, 23, 24, 25)
for (i in 20:25){
  model_apse <- apse(Ssamples, Tsamples, hyper_parameter = i, which_function = 4)
  apses <- c(apses, model_apse)
}
```

```
ylim = extendrange(apses)
plot(complexity, apses, xlab = "Basis Dimensions",
     ylab="APSE", ylim=ylim,
     main="APSE for 6 different basis dimensions",
```

```
     type="n") # "n" suppresses the plot, add values later
lines(complexity, apses, lwd=1, col="red", lty=2)
points(complexity,apses, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



The value of k had to be increased up to minimum 20 when switching from an additive to an interactive model, otherwise R would generate warning messages and automatically re-set k to minimal required value.

We could see from the two APSE plots that an additive model where independence between the predictors is assumed, fited over a thin-place splien with $dim(basis) = 3$, gives us the current lowest APSE value at $apse \approx 0.112$, averaged over 100 bootstrapped samples.

Finally, we will report the estimate of the prediction error based on GCV score of the tuned thin-place spline, fitted over the entire data set.:

```
## since GCV scores are reported as a part of the model fit, we will use the full data set
x <- rbind(x_train_final_predictors, x_test_final_predictors)
y <- rbind(y_train_transformed_logged, y_test_transformed_logged)
fit <- gam(unlist(y)~s(x[,1],bs="tp",k=3) + s(x[,2], bs="tp",k=3) +
             s(x[,3],bs="tp",k=3) + s(x[,4], bs="tp",k=3))
GCVScore <- fit$gcv.ubre
GCVScore
```

```
##      GCV.Cp
## 0.009128537
```

The final prediction error for an additive thin-plate spline with a 3-dimensional basis is approximately 0.009 based on GCV score.

Radom Forests:

Use random forests models on your data, report the importance of variables and your prediction error. If you use CV to estimate prediction error, use 5-fold.

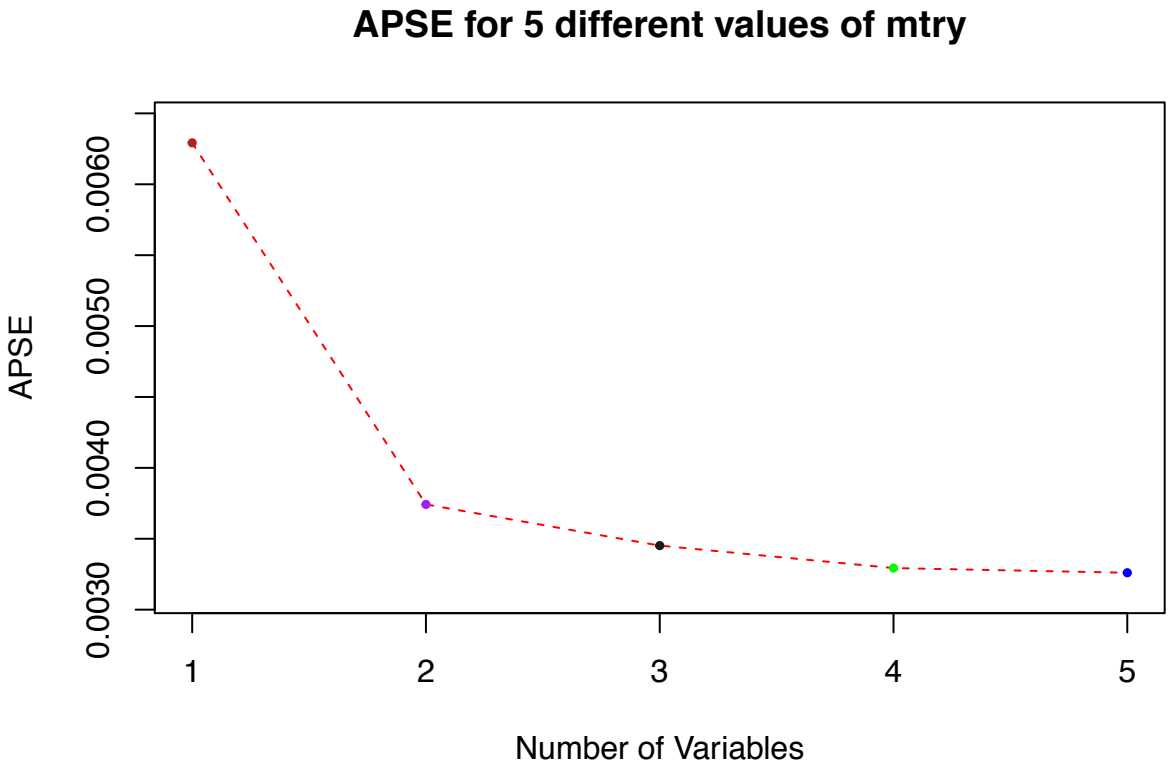
Now let's fit a random forest model on the dataset. The hyper parameter to be tuned is $mtry$, where $mtry$ is equal to the number of variables randomly checked at every internal node during the tree-building procedure. While it is true that both the number of trees and the number of variables affect the final prediction error, the estimated error converges for large enough number of trees. Hence in this report we will only be tuning the $mtry$ parameter and we will set $n.trees = 50$ for all different values of $mtry$.

A note on computational efficiency: throughout many trials prior to this final report draft, our group has experienced exponentially slow runtime as the number of training and testing samples increased. It is simply not a realistic task to average the APSE values over all 100 training and testing samples. Instead, we decided to lower the number down to 20. Since asymptotically random forest has a lower bound on estimated errors, even 20 samples should provide a good insight for parameter tuning.

```
## Random Forests -- we're interested in mtry
getmuhat_rf_mtry <- function(sample, mtry){
  fit <- randomForest(x = sample$x, y = sample$y, ntree = 50,
               mtry = mtry, importance = TRUE, proximity = TRUE)
  muhat <- function(x){predict(fit, newdata = x, ntree = 50)}
  muhat
}
```

```
apses <- c()
complexity <- c(1, 2, 3, 4, 5)
for (i in 1:5){
  model_apse <- apse(Ssamples[c(seq(1,20))], Tsamples[c(seq(1,20))],
                     hyper_parameter = i, which_function = 2)
  apses <- c(apses, model_apse)
}
```

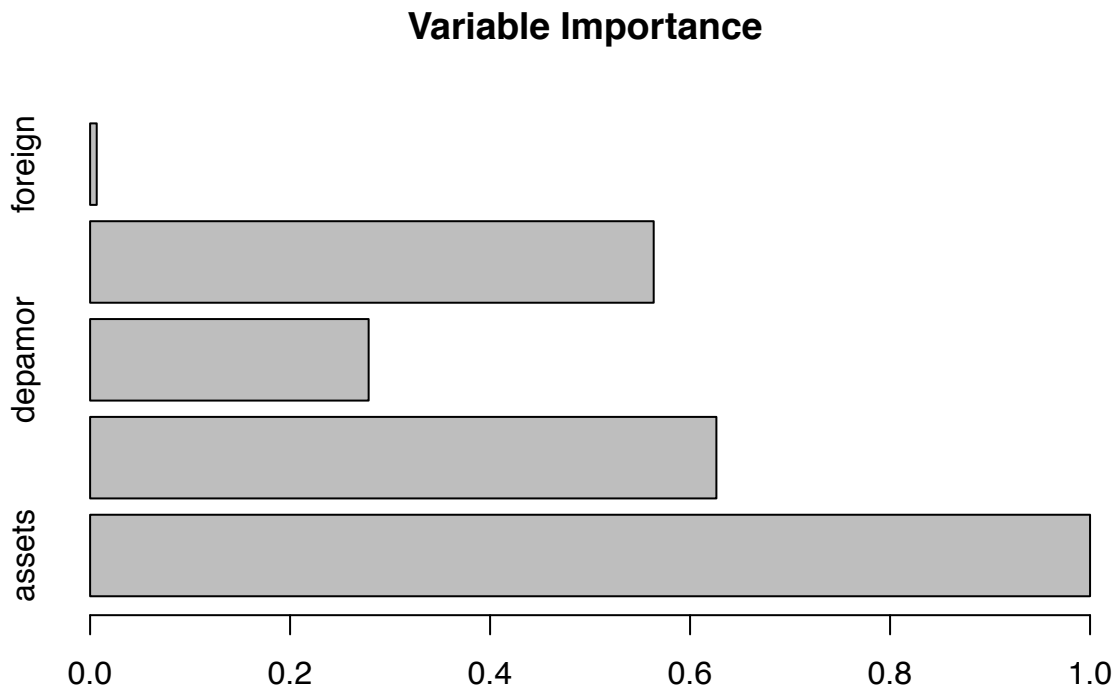
```
ylim = extendrange(apses)
plot(complexity, apses, xlab = "Number of Variables",
     ylab="APSE", ylim=ylim,
     main="APSE for 5 different values of mtry",
     type="n") # "n" suppresses the plot, add values later
lines(complexity, apses, lwd=1, col="red", lty=2)
points(complexity,apses, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



From the plots, we could see that a value of $mtry = 5$ at $ntree = 50$ gives us the lowest APSE so far. Set these parameters as our final model and get the following variable importance plot (normalized over the max value of all variable importance scores):

```
x <- rbind(x_train_final_predictors, x_test_final_predictors)
y <- rbind(y_train_transformed_logged, y_test_transformed_logged)
model <- randomForest(x = x[,1:5], y = y[,1], ntree = 50,
                      mtry = 5, importance = TRUE, proximity = TRUE)
```

```
barplot(model$importanceSD/max(model$importanceSD), main="Variable Importance", horiz=TRUE)
```

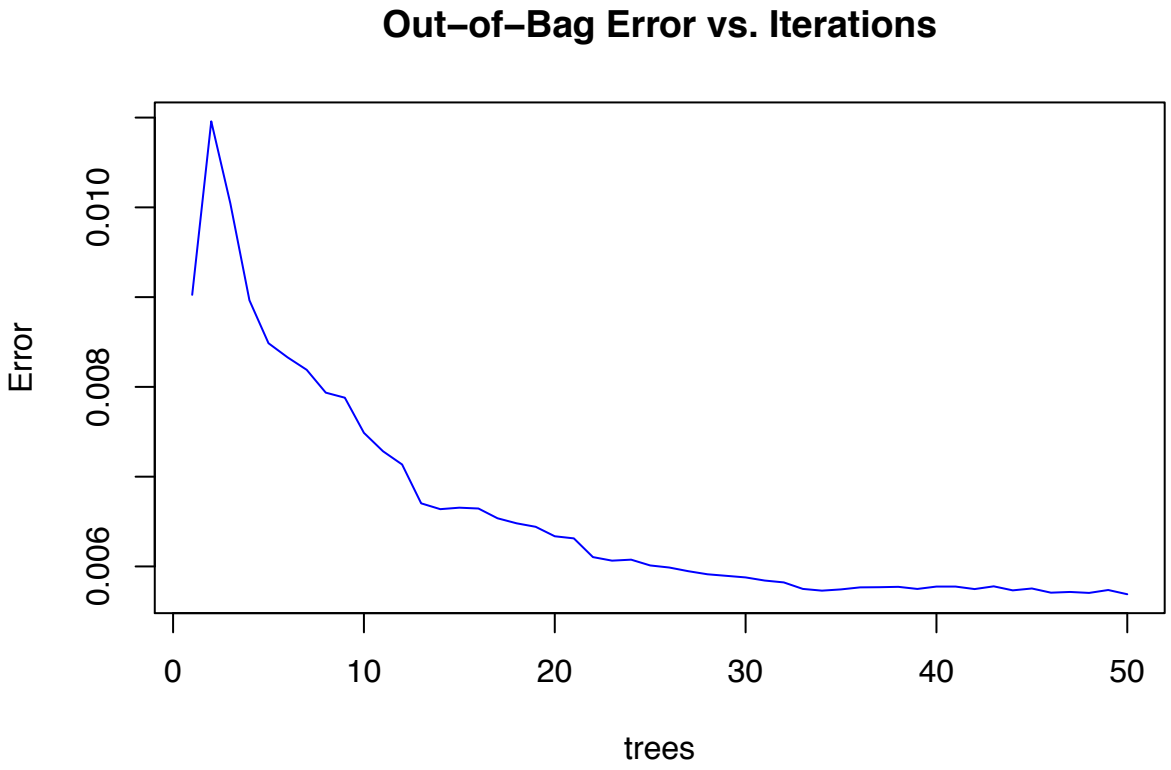



```
model$importanceSD/max(model$importanceSD)
```

```
##      assets      debtnc     depamor      ncfdiv     foreign
## 1.000000000 0.626317415 0.278562196 0.563606883 0.006603596
```

Out-of-bag error usually provides a rough estimate of the true error. While it is not a reliable estimate, we could visualize how the true error is reduced over increasing number of iterations by plotting the out-of-bag error:

```
plot(model, main = "Out-of-Bag Error vs. Iterations", col = 'blue')
```



To obtain an estimate of the prediction error, we could either use the out-of-bag error or the cross-validation error. Since it was discussed in class that the out-of-bag error tends to underestimate the true error, we shall use a 5-fold cross validation to estimate the total error.

```
cverr <- 0

for (i in 1:5){
  train_x <- CVSets[[1]][[i]][,1:5]
  train_y <- CVSets[[1]][[i]][,6]
  test_x <- CVSets[[2]][[i]][,1:5]
  test_y <- CVSets[[2]][[i]][,6]
  model <- randomForest(x = train_x, y = train_y, ntree = 50,
               mtry = 5, importance = TRUE, proximity = TRUE)
  predicted_y <- predict(model, newdata = test_x, ntree = 50)
  cverr <- cverr + APSE_Formula(test_y, predicted_y)[1]
}
cverr <- cverr / 5
```

From the 5-fold cross-validated result, we have an estimated APSE of:

```
cverr
```

```
## [1] 0.00585257
```

Boosting:

For boosted trees, we need to define more customized *getmuhat* functions:

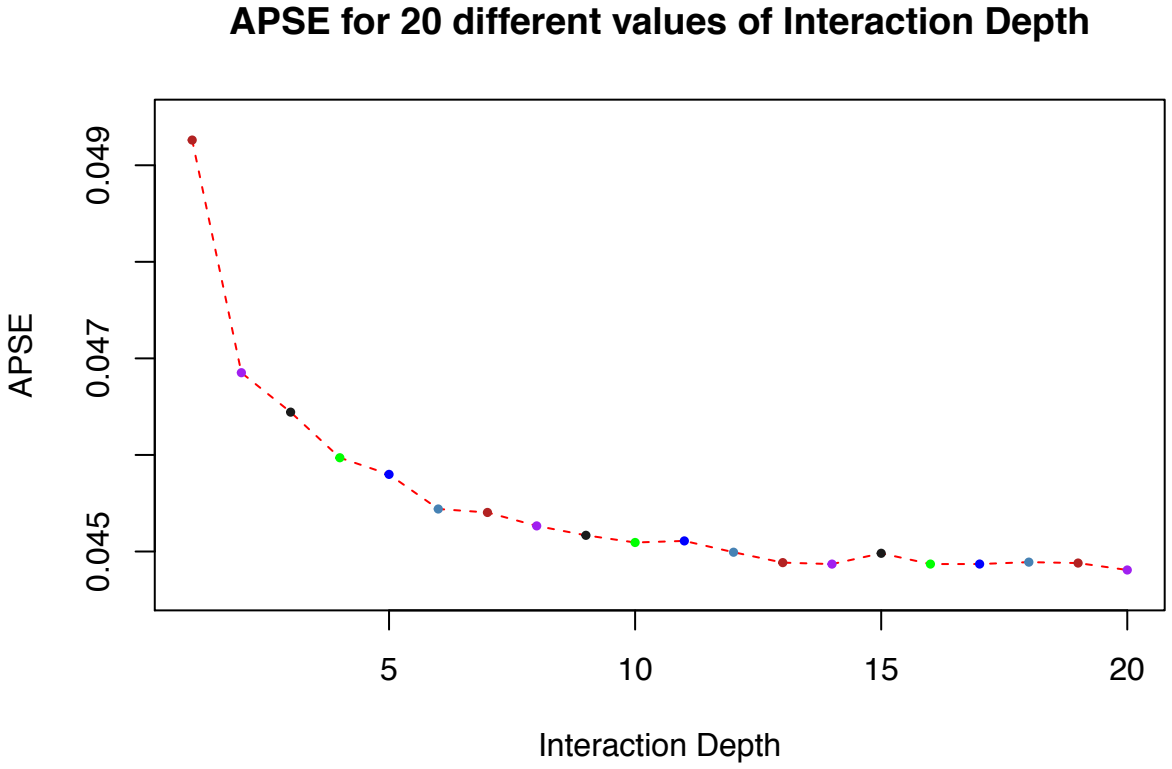
```
ave_y_mu_sq_for_boost <- function(sample, predfun){
  mean(abs(sample$y - predfun(sample))^2)
}
```

```
getmuhat_boost <- function(sample, depth){
  fit <- gbm(y ~ x[,1] + x[,2] + x[,3] + x[,4] + x[,5],
             data = sample, n.trees = 50, shrinkage = 0.01,
             interaction.depth = depth, distribution = "gaussian")
  print(fit)
  muhat <- function(input){
    predict(fit, input, n.trees = 50)
    }
  muhat
}
```

We are interested in two factors affecting the APSE: **interaction depth** of the boosted trees and the **shrinkage factor**. We will tune these two hyper parameters separately:

```
apses <- c()
complexity <- c(seq(1, 20))
for (i in 1:20){
  model_apse <- apse(Ssamples, Tsamples, hyper_parameter = i, which_function = 3)
  apses <- c(apses, model_apse)
}
```

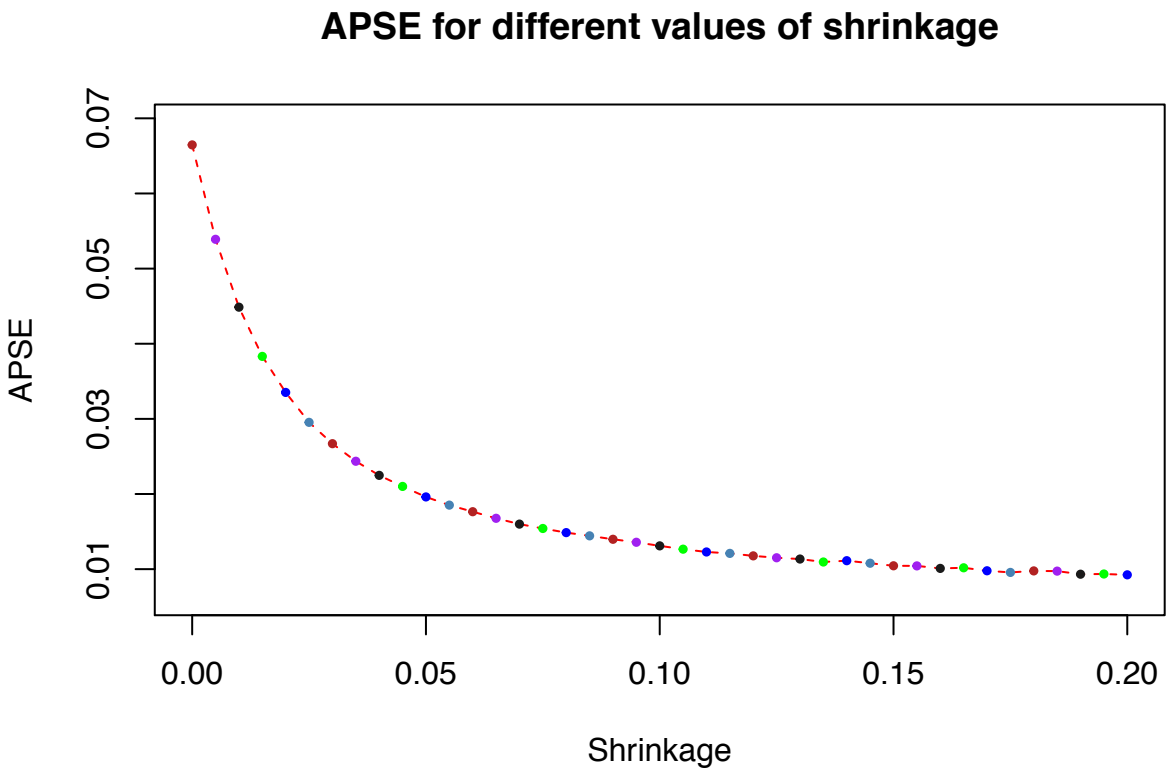
```
ylim = extendrange(apses)
plot(complexity, apses, xlab = "Interaction Depth",
     ylab="APSE", ylim=ylim,
     main="APSE for 20 different values of Interaction Depth",
     type="n") # "n" suppresses the plot, add values later
lines(complexity, apses, lwd=1, col="red", lty=2)
points(complexity,apses, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



From the plot, could see that the APSE decreases drastically then slows down and stabilizes after $depth \geq 15$. Hence we will move on to tune the *shrinkage* parameter and set *interactin.depth* to a fixed value of 20.

```
getmuhat_boost_lambda <- function(sample, lambda){
  fit <- gbm(y ~ x[,1] + x[,2] + x[,3] + x[,4] + x[,5],
             data = sample, n.trees = 50, shrinkage = lambda,
             interaction.depth = 20, distribution = "gaussian")
  print(fit)
  muhat <- function(input){
    predict(fit, input, n.trees = 50)
    }
  muhat
}

apses <- c()
complexity <- c(seq(0, 0.2, 0.005))
for (i in complexity){
  model_apse <- apse(Ssamples, Tsamples, hyper_parameter = i, which_function = 5)
  apses <- c(apses, model_apse)
}
```

```
ylim = extendrange(apses)
plot(complexity, apses, xlab = "Shrinkage",
     ylab="APSE", ylim=ylim,
     main="APSE for different values of shrinkage",
     type="n") # "n" suppresses the plot, add values later
lines(complexity, apses, lwd=1, col="red", lty=2)
points(complexity,apses, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



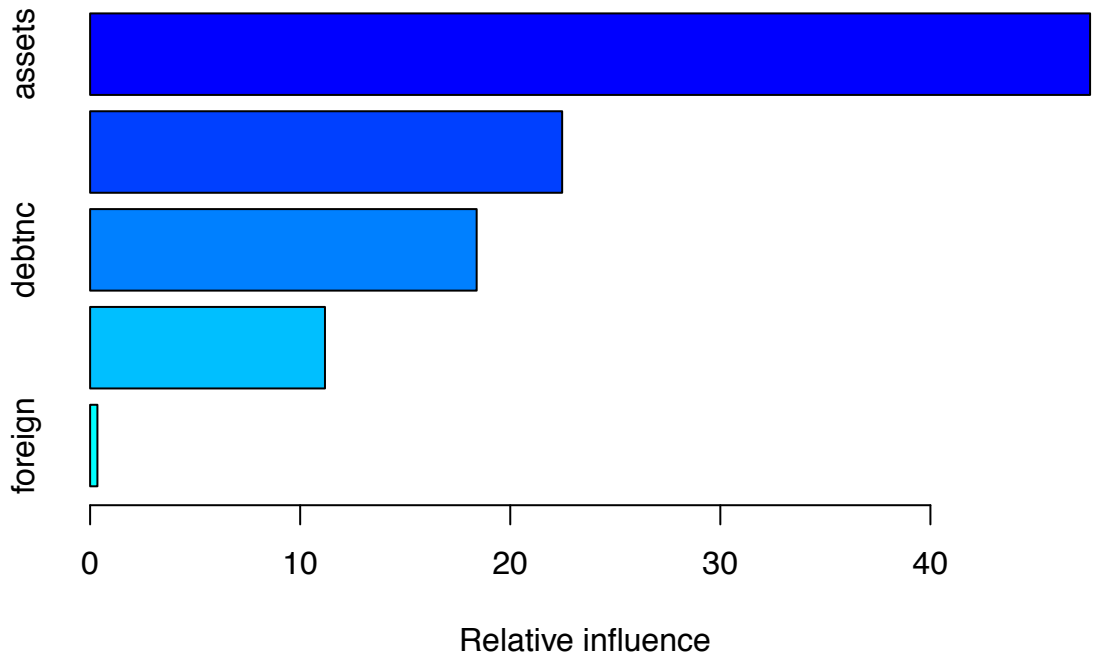
Our finalized hyper parameters for the boosted tree model is $interaction.depth = 20$ and $shrinkage = 0.20$,

with $n.trees = 50$.

We can obtain a plot of the variable importance normalized over the max importance score in the following code segment:

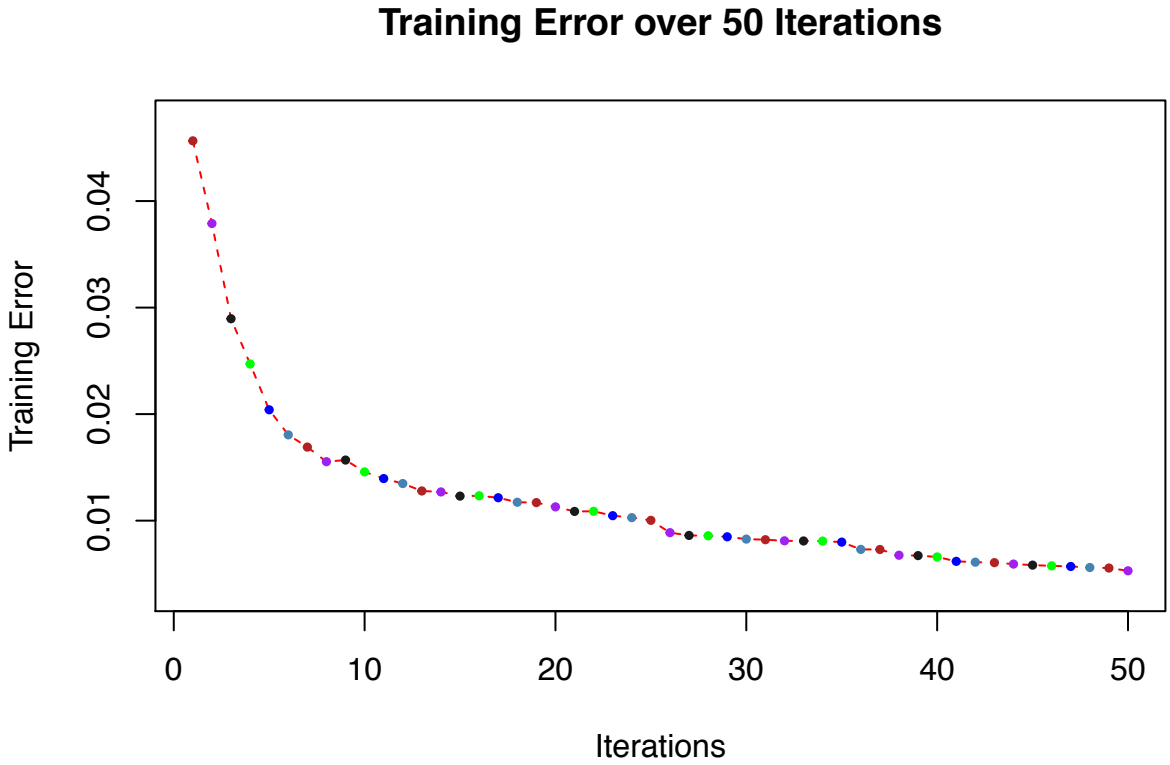
```
x <- rbind(x_train_final_predictors, x_test_final_predictors)
y <- rbind(y_train_transformed_logged, y_test_transformed_logged)
data <- cbind(x, y)
model <- gbm(V1~., n.trees = 50, data = data,
             shrinkage = 0.20, interaction.depth = 20, distribution = "gaussian")
```

```
summary(model)
```



We can also visualize how the training error decreases over each iteration:

```
ylim = extendrange(model$train.error)
plot(c(seq(1, model$n.trees)), model$train.error, xlab = "Iterations",
     ylab="Training Error", ylim=ylim,
     main="Training Error over 50 Iterations",
     type="n") # "n" suppresses the plot, add values later
lines(c(seq(1, model$n.trees)), model$train.error, lwd=1, col="red", lty=2)
points(c(seq(1, model$n.trees)),model$train.error, pch=19, cex=0.5,
       col=c("firebrick", "purple", "grey10", "green", "blue", "steelblue"))
```



Now let's perform 5-fold cross validation to obtain an estimate of the prediction error:

```
cverr <- 0
for (i in 1:5){
  training_data <- CVSets[[1]][[i]]
  testing_data <- CVSets[[2]][[i]]
  model <- gbm(V1~., n.trees = 50, data = training_data,
               shrinkage = 0.20, interaction.depth = 20, distribution = "gaussian")
  predicted_y <- predict(model, newdata = testing_data, n.trees = 50)
  cverr <- cverr + APSE_Formula(testing_data[,6], predicted_y)[1]
}
cverr <- cverr / 5
```

From the 5-fold cross-validated result, we have an estimated APSE of:

```
cverr
```

```
## [1] 0.01656328
```

Statistical Conclusions:

From the three models we constructed and fine-tuned, and based on GCV and 5-fold cross validation scores, we've obtained the following results:

- Prediction Error: In terms of prediction error, random forest gives the most precise model with the lowest estimated error, followed by thin-plate splines and then boosted trees. The estimated error given by the finalized random forest model is almost half the estimated error given by the other two methods.
- Ease of Use: In terms of ease of use, we personally found the R packages *randomForest* and *gbm* to be easier to use than *gam*. That being said, *gam* had far more detailed documentation on a wide variety of different multivariate spline methods.

- Computation Time: Gradient Boosted Trees is a top contender in terms of the computation time, followed by thin-plate splines, and then random forest. As mentioned in the Random Forest parameter tuning section, we were forced to reduce the number of bootstrapped training and testing samples down to 20 because the runtime did not scale well with increased sample sizes.

Conclusions in the context of the problem:

Based on our findings, we conclude that assets, non-current debt, depreciation, amortization & accretion, and payment of dividends & other cash distributions are the most important factors when modelling the retained earnings of a company. While a company's country of origin (domestic vs. foreign) improves the accuracy of the predictions, the benefits are only marginal. Hence the most efficient and practical way of modelling retained earnings is to use the above-mentioned four factors. As an extension of our results, we can further make the inference that a company's retained earnings is significantly affected by its cash managent and its asset-liability ratios. Therefore even in the absence of one or two of the four important predictors, a decent model could still be obtained as long as some form of data on assets and debt are available.

In terms of application, the models we derived can be used by a variety of groups. For business analysts, predicted retained earnings can be used to conduct market analysis. For sales and marketing executives, our models can be used to compile pitchbooks for new investment packages. Last but not least, hedge fund managers can use our model to make seasonal or annual adjustments to their clients' portfolios. While it took a lot of effort to pre-process the data and tune the parameters, our finalized models are all very concise and easy to run, regardless of the users' background knowledge in statistics or computer science.

Future work:

Aspects we wish to have done better:

1. More sophisticated variable selection methods: due to time constraints and the sheer number of potential predictors, we were unable to investigate all potential interactions between the predictors. If we had more time, we would try more complicated variable screening techniques.
2. Imputation: missing value imputation is a topic that has been researched in-depth by a lot of scholars, and we wish we had more time to try out different imputation packages in R such as $MICE$ so that we could retain more rows from the original data set.

Weakness of the current methods:

1. Runtime of random forests: while random forests give the lowest APSE out of all models we constructed and tuned, it does not scale well with increased number of observations and predictors. This inefficiency cripples the strength of the model by a lot, as in real life analysts would deal with much larger databases and many more predictors. A quick way to improve the runtime is to concurrently execute m threads where m is the number of iterations, since each iteration builds a tree and therefore can be executed independently. We would have tried concurrent threads on random forest if we had more time for this project.

Other statistical and Machine Learning methods:

1. Due to time constraints, we were only able to perfect three potential models. There are many more variations of spline methods available in the *gbm* package, but we were unable to read up on all of them in details other than thin-plate splines. Given more time, we would definitely have explored the *gbm* package further.

2. We also would be interested in using a neural net on this problem and see if it performs better or worse than the models we currently have.

Data Collection:

1. We were interested in obtaining more back-dated data on the companies and analyze how serially correlated data could hurt or help our current models, as well as how to construct a mixed model that contains categorical predictors, continuous predictors, continuous time-series predictors, and categorical time-series predictors. All of this requires scraping more data off the internet, which we did not have the time for.
2. Additionally, as we had seen with this project, raw data in real life is usually a mess to work with and often requires a lot of pre-processing, imputation, cleaning, and scaling. Given more time, we would have contunued to improve the quality of our data set as well.

Contribution:

- Abhi: Data cleaning and pre-processing
- Alvin: Variable screening and selection
- Rosie: Model selection and parameter tuning, report compilation

Appendix:

A Note on Reproductibility:

For faster PDF compilation time, we disabled some code block executions and directly used cached results, namely the python code blocks in the data processing section and the R code for concurrent threads in the variable selection section. To fully reproduce the results, please go through all code blocks in the markdown file and reset *eval* to $TRUE$ when applicable.