Analysis and Prediction of Boeing's Stock Price Trend of Next Trading Day Using Various Machine Learning Algorithms

Thomas Z. Su

PHID: 1616244

Due: Dec. 14, 2020

Introduction

In the midterm report, the machine learning algorithms of K-nearest Neighbors and Principal Component Analysis were used to analyze the historical data set of Boeing's stock in the past ten years and predict its future stock price trend. In this report, it's continued to study with the same data set as what had been selected to perform in the midterm report, but by applying different algorithms, such as Decision Tree, Random Forest, Support Vector Machine.

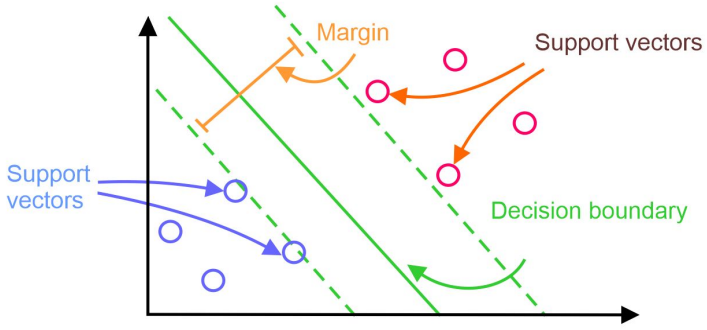
The original file of the data set was provided from the following link *quandl.com/data/EOD/BA-The-Boeing-Company-BA-Stock-Prices-Dividends-and-Splits* and downloaded in a CSV format.

A brief reminder of **Principal Component Analysis (PCA)** we have done in the midterm report: it is an unsupervised machine learning where it has to look at the data set and then divide it based on its algorithms without having any training. It is the linear model associated to obtain new uncorrelated variables, and have maximum variance, and applied to reduce redundancy information and data dimensionality.

Decision Tree is a tree-shaped algorithm used to determine a course of action. Each branch of the tree represents a possible decision, occurrence, or reaction. This model is supervised machine learning that learns from the past input data and makes future predictions as output. So, it is used for both classification and regression. The classification is dealt with identifying to which set an object belongs. For example, an apple is red while a banana is yellow. Regression problems have continuous or numerical valued output variables, such as predicting the earning or loss of a company.

Random Forest is a supervised machine learning algorithm where lots of data are used to train the model that operates by constructing multiple decision trees during the training phase. The decision of the majority of the trees is chosen by the random forest as the final decision, so the random forest outperforms the decision tree. The random forest has three major advantages. First of all, it reduces the risk of overfitting by using multiple trees. Training time is much less than one of the K-nearest neighbors. Secondly, It runs efficiently on large databases for large data and produces highly accurate predictions. Lastly, it can maintain accuracy when a large proportion of data is missing.

Support Vector Machine (SVM) is a supervised machine learning model that looks at and separates data using hyperplanes like a decision boundary between various classes with the maximum margin. Generally, SVM can be used to generate multiple separating hyperplanes so that the data set is divided into segments. Each of these segments only contains one class/kind of data. It is mainly used for classification that classifies a data set into two different segments. For classifying non-linear data, the data can not be separated with a simple straight line, so the kernel trick is applied. The kernel trick is to transform data into another dimension so that the hyperplane can be easily drawn between the different classes. The following diagram shows how the SVM algorithm works:



Question 0: Description of Data Set

The original data set has 10 usable feature columns in total excluding the features *time t*, *dividend*, *split*. They are named: “open_t” as price when the market opens, “high_t” as the highest price at the day t, “low_t” as lowest price at day t, “close_t” as price when the market closes at day t, “volume_t” as the number of shares that changed hands at day t, “adj_open_t” as adjusted open price, “adj_high_t” as the adjusted high price, “adj_low_t” as adjusted low price, “adj_close_t” as the adjusted close price, “adj_volume_t” as adjusted volume. Note that the last five features are adjusted variables that factor in things like dividends, stock splits, new stock offerings, etc. Based on the original features, three extra features are added. They are, “return_t” as a daily return in which the close price of day t minus the close price of day t-1, “gap_trading_t” as the earning/loss while gap trading in which the open price of day t minus the close price of day t-1, “adj_return_t” as the adjusted daily return. At last, extract only 2705 row cases of trading days from January 1 of 2010 to September 30 of 2020 to analyze.

As the rule of thumb, it is recommended to use a plus and minus 0.6% threshold of daily return to label bullish or bearish stock for its company having a large market capitalization. In the research, Boeing company’s market gap has been maintained at least ten billion US dollars since more than ten years ago. Therefore, Boeing is considered as having a large market cap, and the rule of thumb is applied to the true set, which is the feature column one.

Based on the rule, the data set can be divided into three classes of stock trends the next trading day on its true set. CLASS 1, CLASS 2, and CLASS 3 respectively represent bullish/uptrend, bearish/downtrend, and neither of them. Now, the data set has 13 features and 1 added feature of the true set. The number of cases of each class is, respectively, 918, 809, and 978.

```
> n1 = nrow(CL1); print(n1)
[1] 918
> n2 = nrow(CL2); print(n2)
[1] 809
> n3 = nrow(CL3); print(n3)
[1] 978
```

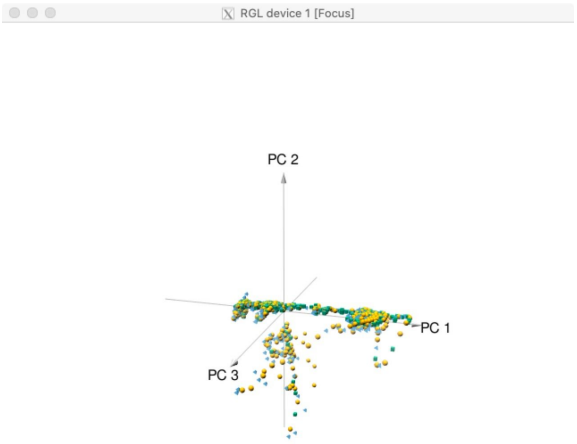
Since the Principle Component Analysis is still performed in this report, it's needed to increase the memory of the classifier to the original data set. The transformed data set that is expanded from 13 features to 78 features by adding features of the earlier five days will be employed instead.

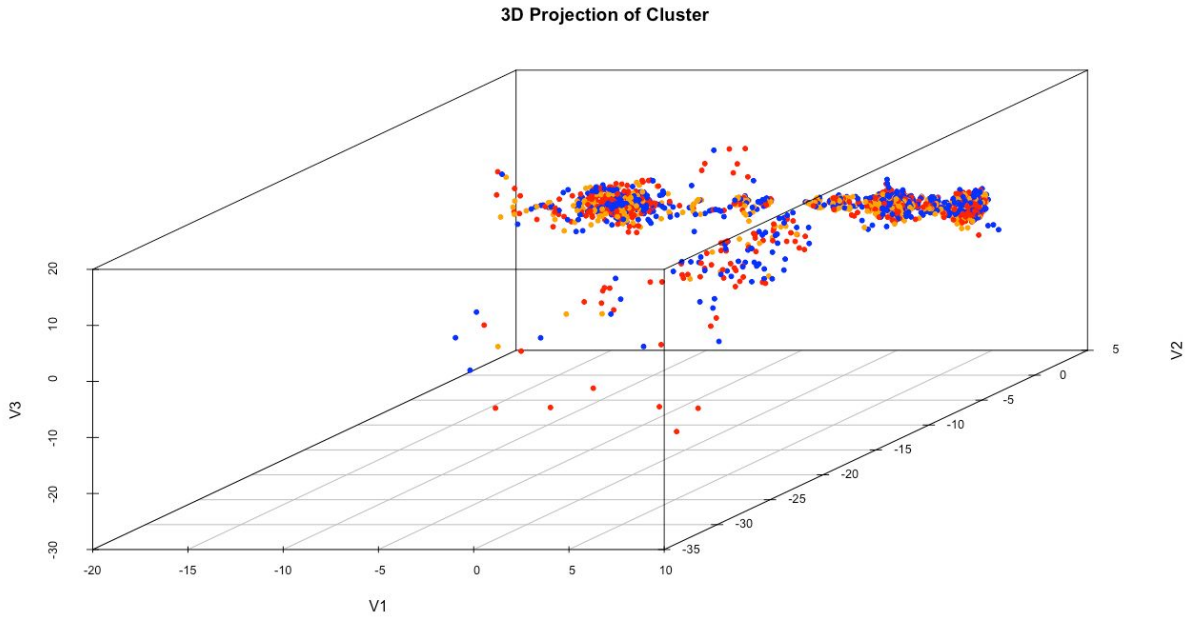
Part 1: Principal Component Analysis (PCA)

In this part, PCA is performed on standardized features SDATA. Using R, create the rectangular covariance matrix Σ by a set of codes "***sigma = cov(SDATA)"*,** where X is the matrix of the revised data set excluding the feature of the true set. Convert the variance-covariance matrix computed to a correlation matrix named R by a set of codes "***R = cov2cor(sigma)***". Find a matrix of eigenvectors and eigenvalues of the features correlation matrix by sets of codes "***Q = eigen(R)$vectors"*** and ***"D = eigen(R)$values"***. The ordinal number of the column of the eigenvectors' matrix is the ordinal number of the component. Through a vector of eigenvalues in decreasing order, it can be concluded that 95.211% of the variance will be accounted for with the first nine principal components because the sum of the first nine components' eigenvalues divided by the sum of all eigenvalues is equal to 0.95211. There is an alternative way to find out which eigenvectors explain approximately 95% of the total variance, using R, run the codes "***summary(princomp(X,cor=TRUE,scores=TRUE))***" and directly see the summary of the principal component analysis with standard deviation, the proportion of variance, and cumulative proportion of variance for ordinal column components.

```
> Q[1:5, 1:9] # eigenvector of first five rows and first nine columns
              [,1]       [,2]       [,3]       [,4]       [,5]       [,6]       [,7]       [,8]       [,9]
[1,] -1.4425e-01  0.0038002 -0.023607 -0.011305 0.0038980 -0.0050869 -0.0043908  0.00186638  0.0039603
[2,] -1.4426e-01  0.0017198 -0.022618 -0.013493 0.0027398 -0.0068416 -0.0049457 -0.00036405  0.0033905
[3,] -1.4416e-01  0.0065137 -0.026243 -0.013741 0.0027589 -0.0082914 -0.0055627 -0.00097063 -0.0020040
[4,] -1.4418e-01  0.0044155 -0.024931 -0.016116 0.0009672 -0.0111704 -0.0051743 -0.00337552 -0.0012230
[5,] -4.8009e-05 -0.2844891  0.021838 -0.020560 0.0039345 -0.0077275  0.0034996  0.00068512  0.1672531
```

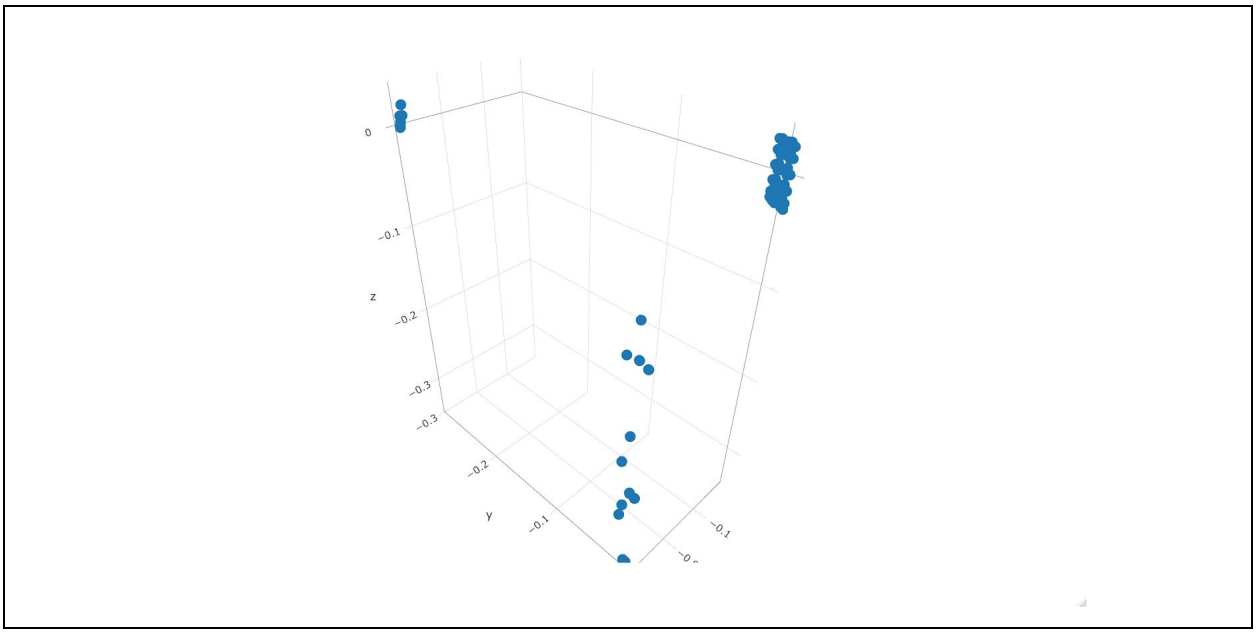
```
> D[1:3] # first three eigenvalues
[1] 47.9001 10.7393  3.1322
```





Three classes are not separated clearly.

```
> library(ggplot2)
> library(GGally)
> ggpairs(data = data.frame(Q[,1:3]), columnLabels = c("V1", "V2", "V3"))
```



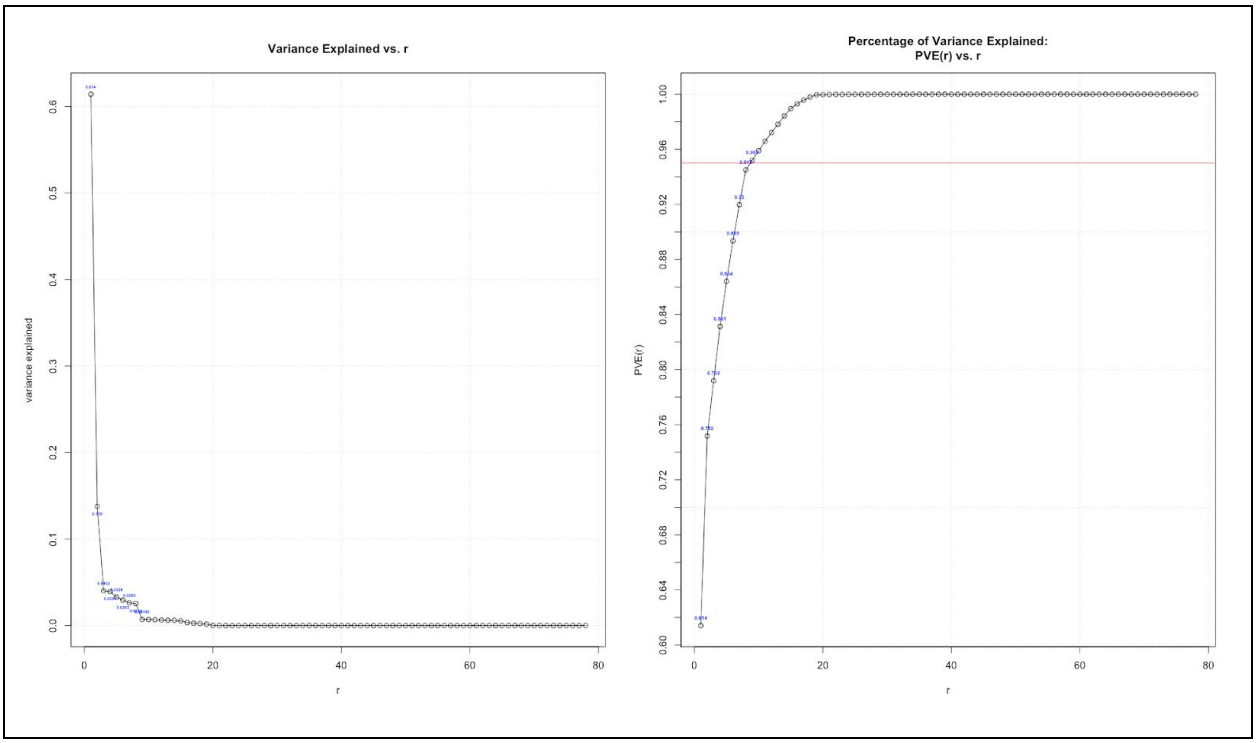


Figure 1. Left: A scatter plot depicting the Variance Explained by each of the 78 principal components in our font DATA set. Right: Percentage of Variance Explained by every 78 principal components in the font DATA set. The plot of the PVE(r) vs r. X-axes from 0 to 78 features, Y-axes from 0.0 to 1.00 (percentage)

As we see in Figure 1 left, the graph of variance explained vs. r is a steep decreasing curve followed by a bend and then a straight line. The components that are important are the ones in the steep curve before the graph begins to set into a horizontal trend. By eyeballing Figure 1 left, we can conclude that a fair amount of variance is explained by the first twenty principal components. After all, it does appear to represent a horizontal line getting close to zero. We also can see that the first principal component (PC1) explains 61.4% of the variance in the data, the second principal component (PC2) explains 13.8% of the variance of the data, and the third principal component (PC3) explains 4.02% of the variance and so forth.

Now that we know what our variance explained represents, we take a look at our Percentage of Variance Explained we calculated using the cumsum(D)/sum(D) functions in R, where D is defined as our eigenvalues. We are basically calculating the cumulative distribution of our variance explained. In Figure 1 Right, we can see for our Percentage of Variance Explained (PVE(r)), overall 75.2% is being captured by r = 2 components and so on. The red horizontal line represents nearly 95% (PVE(r) = 95%) of the variance, we identified to be r = 9 dimensions (or about 11.54%). Therefore, we concluded that 95% of our data falls within 1 to 9 dimensions and computed the PCA: new features also known as principal components.

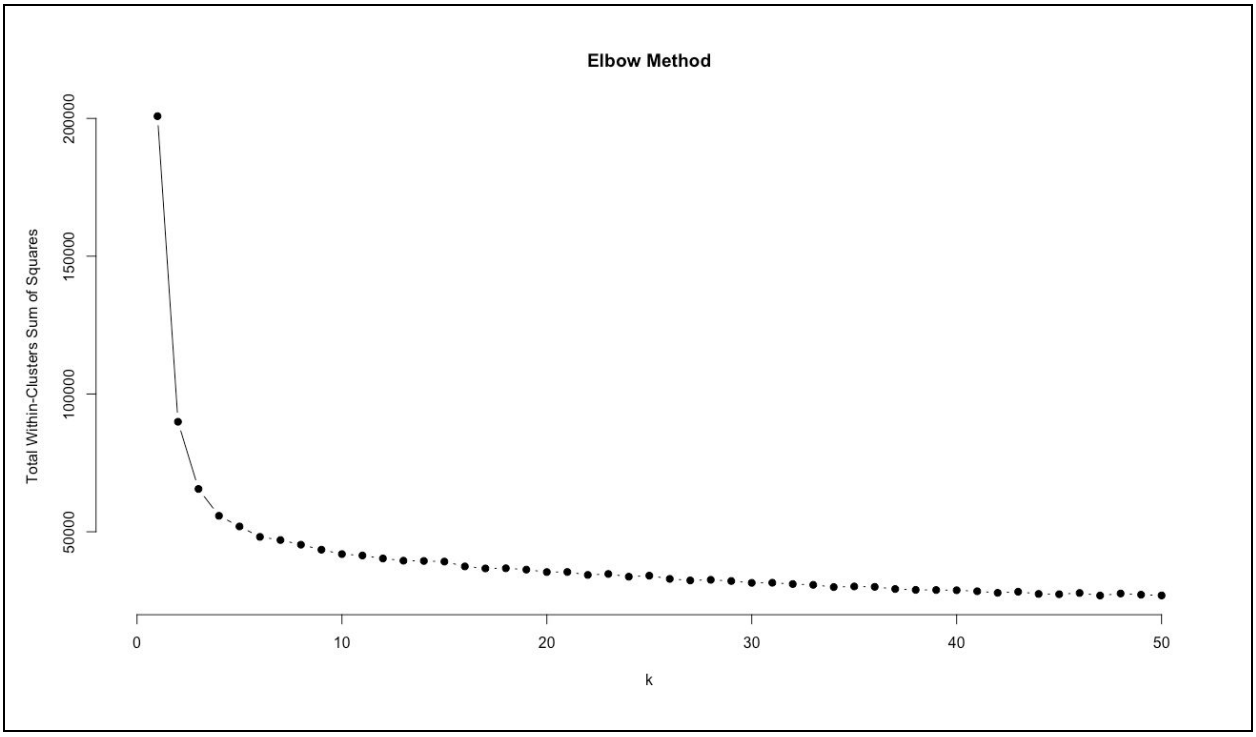
Y1(n), Y2(n),.., Yr(n) for each case in n and r = 9.

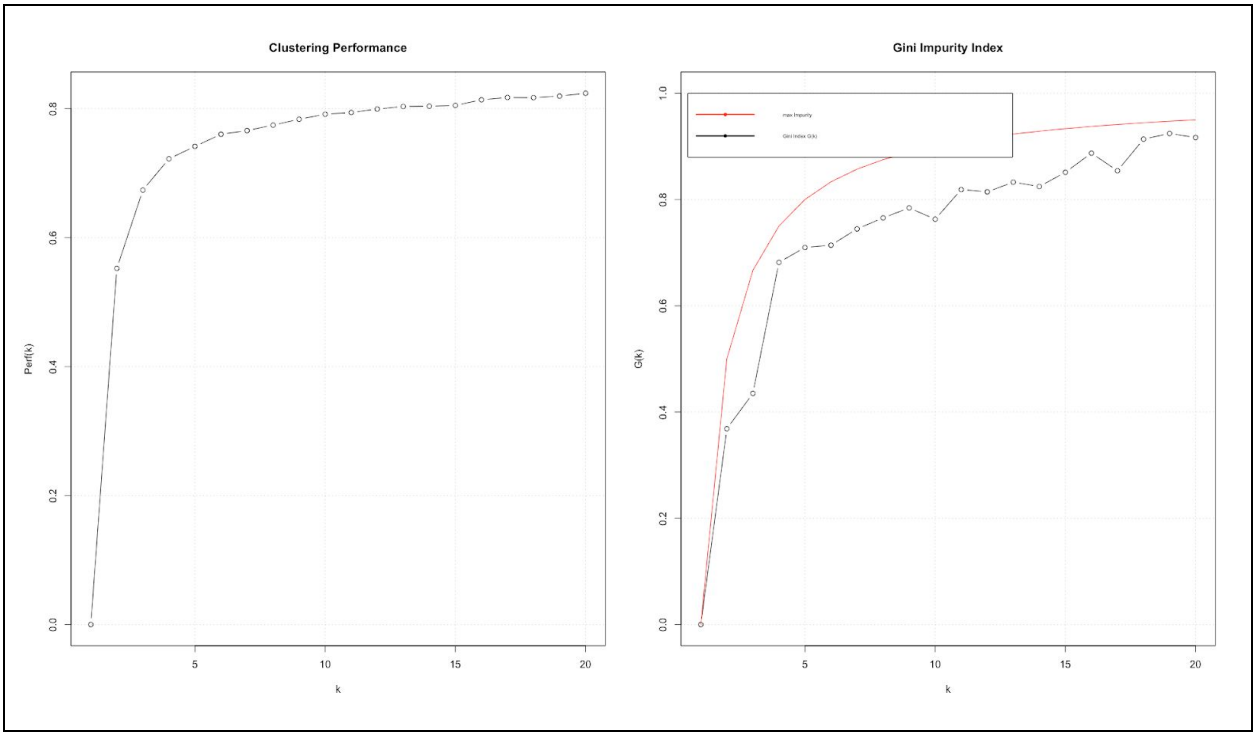
Part 2: *Kbest* Selection with K-means Clustering

There are two major indicators that can be traced to the optimal value of cluster k. They are the Clustering Performance and the Gini Impurity Index.

The **Clustering Performance** is evaluated by equation $1 - SWS(k)/SWS(1)$, where SWS(k) function is defined as the total within-clusters sum of squares by *kmean(k)$tot.withinss* in R. A higher clustering performance is expected.

The **Gini Impurity Index** is calculated as the sum of squares of probabilities for each class. The maximum Gini Index is followed by equation $1 - 1/r$, where r is the number of classes when records are equally distributed among all classes. The minimum Gini index is zero that exists when all records belong to one class implying the most interesting information. The lower the Gini impurity index, the higher the homogeneity of the nodes.





Look at the two graphs above, the left one represents the cluster k vs. clustering performance. The percent of accuracy is higher than 80% when cluster k is greater than and equal to 3. The optimum k would be selected from 3 to the maximum k. The right one represents the cluster k vs. Gini impurity index. Because the lower the Gini index, the better the classification. It is observed that the Gini index jumped up to approximately 70% from approximately 45% at k = 4, which is unexpected. The optimum k should be picked among [1, 2, 3]. In conclusion, the kbest is 3.

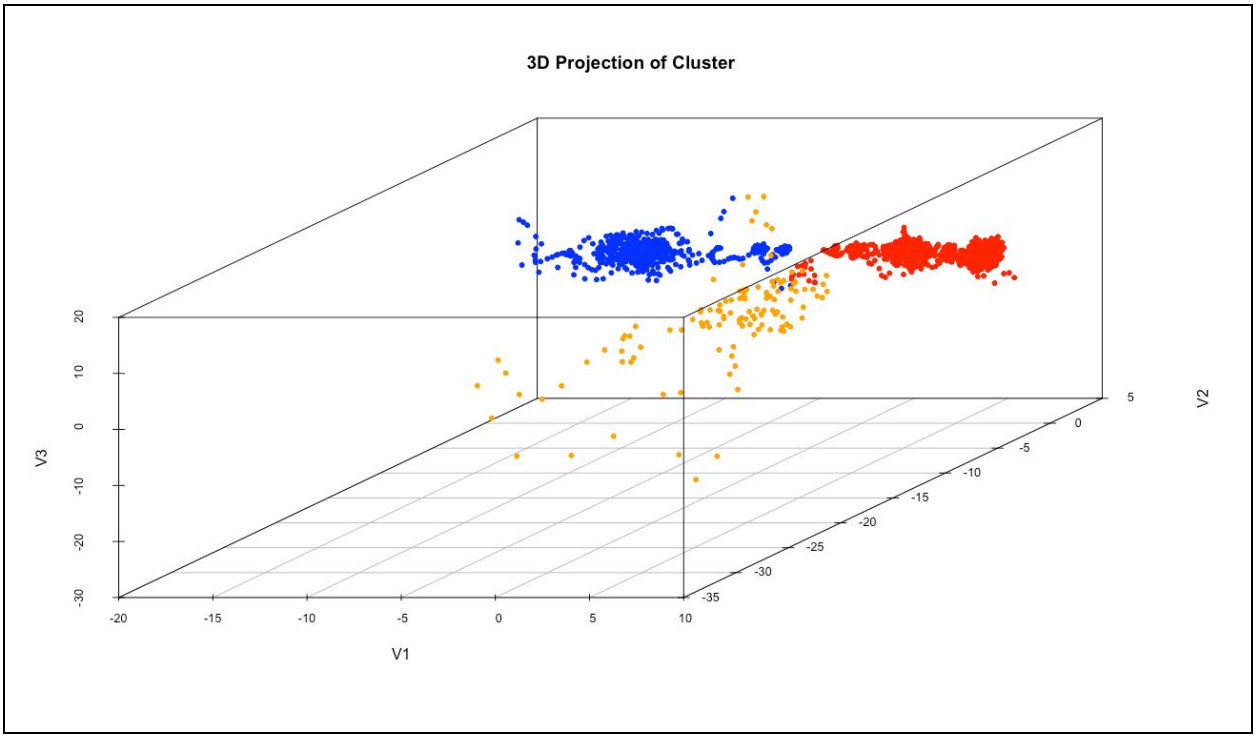
Part 3: Summary of K-means Clustering for kbest

As kbest = 3 that is confirmed. The center A_j , size S_j, dispersion DIS_j, gini index G_j and frequencies of each class in CL_j are outputted below.

```
> options(digits = 5)
> for (j in 1:kbest) {
+     cat("when j =", paste0(j, ", we have:\n"))
+     cat("center A_j:\n")
+     print(kmean(j)$center)
+     cat("Size S_j:\n")
+     print(kmean(j)$size) # table(kmean(j)$cluster)
+     cat("Dispersion DIS_j:", SWS(j), "\n")
+     cat("Gini G_j:", G(j), "\n")
+     cat("Frequency fm_j:\n")
+     FREQ_j = c()
+     for (k in 1:j) {
+         FREQ_j = append(FREQ_j, fm(k))
+     }
+     FREQ_j = matrix(FREQ_j, nrow = j)
+     print(FREQ_j) # frequencies of each class in CLj
+     cat("\n")
+ }
when j = 1, we have:
center A_j:
             X1           X2           X3           X4           X5           X6           X7           X8           X9
1 1.4828e-15 -1.5055e-15 3.0536e-17 1.1841e-17 2.2943e-17 -1.7731e-17 6.1565e-19 -9.912e-18 -8.1882e-18
Size S_j:
[1] 2705
Dispersion DIS_j: 200812
Gini G_j: 0
Frequency fm_j:
       [,1]   [,2]    [,3]
[1,] 0.27514 0.3309 0.39396

when j = 2, we have:
center A_j:
          X1        X2          X3           X4           X5           X6           X7           X8           X9
1    3.6286 -0.19405  0.0016634   0.0091304  0.00086591 -0.0045101 -0.00091558  0.00017908  0.00031079
2 -11.2658  0.60246 -0.0051644 -0.0283473 -0.00268839  0.0140025  0.00284260 -0.00055600 -0.00096491
Size S_j:
[1] 2046  659
Dispersion DIS_j: 89916
Gini G_j: 0.36854
Frequency fm_j:
       [,1]    [,2]    [,3]
[1,] 0.27514 0.39396 0.34848
[2,] 0.33090 0.33182 0.31970

when j = 3, we have:
center A_j:
             X1        X2           X3         X4           X5           X6           X7          X8           X9
1    3.869666   0.65807 -0.0048804   0.023321 -1.8006e-05  0.0010265 -0.00085227  0.0018126  0.00140027
2 -11.254401   0.60145 -0.0132865 -0.032070 -4.8986e-03  0.0153064  0.00557297  0.0063763 -0.00061841
3    0.016001 -13.17296  0.1439246 -0.187196  2.5934e-02 -0.0958102 -0.01621162 -0.0610056 -0.01808709
Size S_j:
[1] 1919  660  126
Dispersion DIS_j: 65534
Gini G_j: 0.43501
Frequency fm_j:
          [,1]      [,2]     [,3]
[1,] 0.27514 0.33182 0.492063
[2,] 0.33090 0.34848 0.420635
[3,] 0.39396 0.31970 0.087302
```

The 3D projection of cluster CLj on eigenvectors V1, V2, V3 is displayed below.

Each color represents a type of categories/classes.



Part 4: Data Cloning

Using an unbalanced data that will lead to the skewed or biased prediction in the Random Forest algorithm we will perform next session. To combat the imbalanced classes, we clone to increase the number of cases in the two smaller classes until the case numbers of the training set and the test set among three classes respectively match. As demonstrated in the table below, CL1 (Bearish), CL2 (Bullish), and CL3 (Neither) have different case sizes.

```
> table(true_set)
true_set
Bearish Bullish Neither
    809     918     978
```

So, here is how it works. For the training set of one of both smaller classes, we randomly select 80% of the cases as its training set at first. Then, we clone each case of its training set n times, where “n” is the ceiling value of the training set’s size ratio of the bigger class to this smaller class. Now, the modified training set of this smaller class has a larger size than the intact training set of the bigger class does. To reach a balanced state between both classes’ training sets, we randomly select “m” quantities of only cloned cases in the modified training set of this smaller class to remove out, where “m” is the size difference between the modified training set of this smaller class and the intact training set of the bigger class.

For the test set of this smaller class, because we selected the training set in the original data set just now, the rest of the 20% of the cases are automatically classified as its test set. Then the next procedures are all the same as how we did in the training set just now.

As for another small class, repeat the completely same cloning approach of the first smaller class. As demonstrated in the table below, at the end of time, the case numbers of the training set and the test set between three classes are respectively equal, each case is utilized once at least that is assured.

```
> # Verification!!!
> # setdiff(x, y) is the material that is in x, that is not in y
> setdiff(x1, pd_CL1train) # verification for all the original 80% of cases are not missed, so we expect the output be integer(0)
integer(0)
> setdiff(x2, pd_CL2train)
integer(0)
> setdiff(y1, pd_CL1test) # verification for all the original 20% of cases are not missed, so we expect the output be integer(0)
integer(0)
> setdiff(y2, pd_CL2test)
integer(0)
> intersect(pd_CL1train, pd_CL1test) # verification for test set and training set do not intersect
integer(0)
> intersect(pd_CL2train, pd_CL2test)
integer(0)
```

```
> trainset_target = as.factor(rbind(trainCL1, trainCL2, trainCL3)[,1])
> table(trainset_target)
trainset_target
Bearish Bullish Neither
    782     782     782
```

```
> testset_target = as.factor(rbind(testCL1, testCL2, testCL3)[,1])
> table(testset_target)
testset_target
Bearish Bullish Neither
    196     196     196
```

Part 5: Random Forest (RF)

To get started in performing random forest, we first make sure to set our cloned *trainset_target* into factors using as.factor(). Next using the function randomForest(), from the package: library(randomForest) in R. Defined as ***rf*** we input the factored cloned *trainset_target* as our model as a function of all the other remaining variables and set the data = cloned *trainset*.

Fixing the number of variables tried at each split (mtry) to sqrt(p), where p is the number of features, we repeat applying random forest to classify our data set into three classes by exploring ntrees = [100, 200, 300, 400]. Reminder that we are using the p = 9 features we found in our PCA result.

To compute the accuracy of the train set, Using the *predict()* function we input ***rf*** associated to the randomForest and train = TRAINSET from the original data set. Using the *confusionMatrix()* function, we input our prediction associated with the randomForest and true train = TRAINSET_TARGET from the original data det. The *trainperf* is followed by the equation as the sum of the diagonal coefficients of its confusion matrix divided by the total values of its confusion matrix.

To compute the accuracy of the test set, Using the *predict()* function we input ***rf*** associated to the randomForest and test = TESTSET from the original data set. Using the *confusionMatrix()* function, we input our prediction associated with the randomForest and true test = TESTSET_TARGET from the original data det. The *testperf* is followed by the equation as the sum of the diagonal coefficients of its confusion matrix divided by the total values of its confusion matrix.

> # for an accuracy of the training set

ntree = 100:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	545	46	41
Bullish	38	642	54
Neither	61	57	691

Accuracy
0.863448

ntree = 200:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	550	42	40
Bullish	42	642	50
Neither	52	61	696

Accuracy
0.868046

ntree = 300:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	550	43	36
Bullish	36	641	53
Neither	58	61	697

Accuracy
0.868046

ntree = 400:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	551	42	37
Bullish	38	642	51
Neither	55	61	698

Accuracy
0.869425

> # for an accuracy of the test set

ntree = 100:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	150	8	9
Bullish	11	157	7
Neither	4	8	176

Accuracy
0.911321

ntree = 200:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	148	8	6
Bullish	10	156	9
Neither	7	9	177

Accuracy
0.907547

ntree = 300:

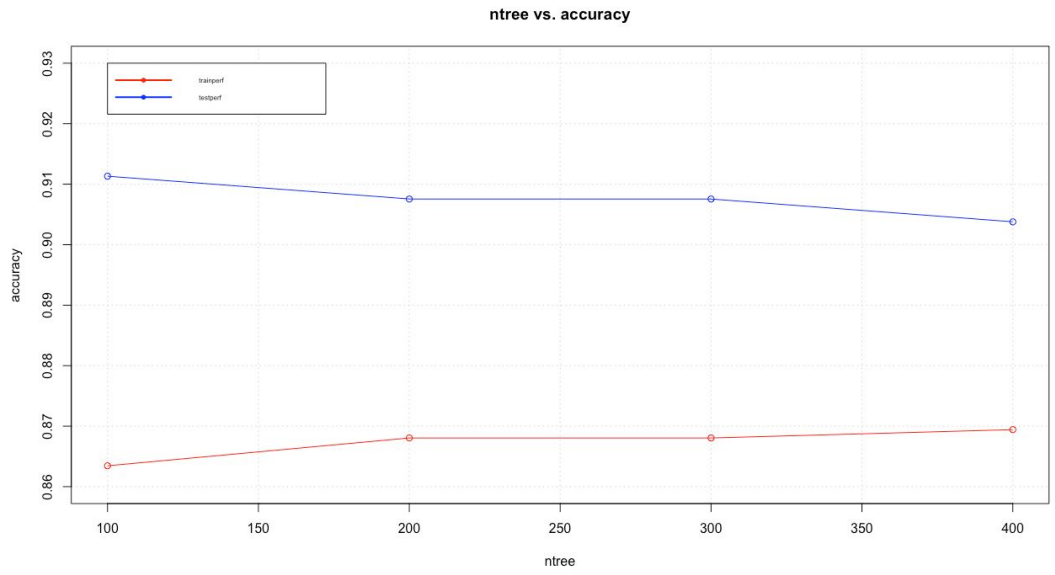
Prediction	Reference: Bearish	Bullish	Neither
Bearish	147	7	8
Bullish	12	159	9
Neither	6	7	175

Accuracy
0.907547

ntree = 400:

Prediction	Reference: Bearish	Bullish	Neither
Bearish	146	9	8
Bullish	14	156	7
Neither	5	8	177

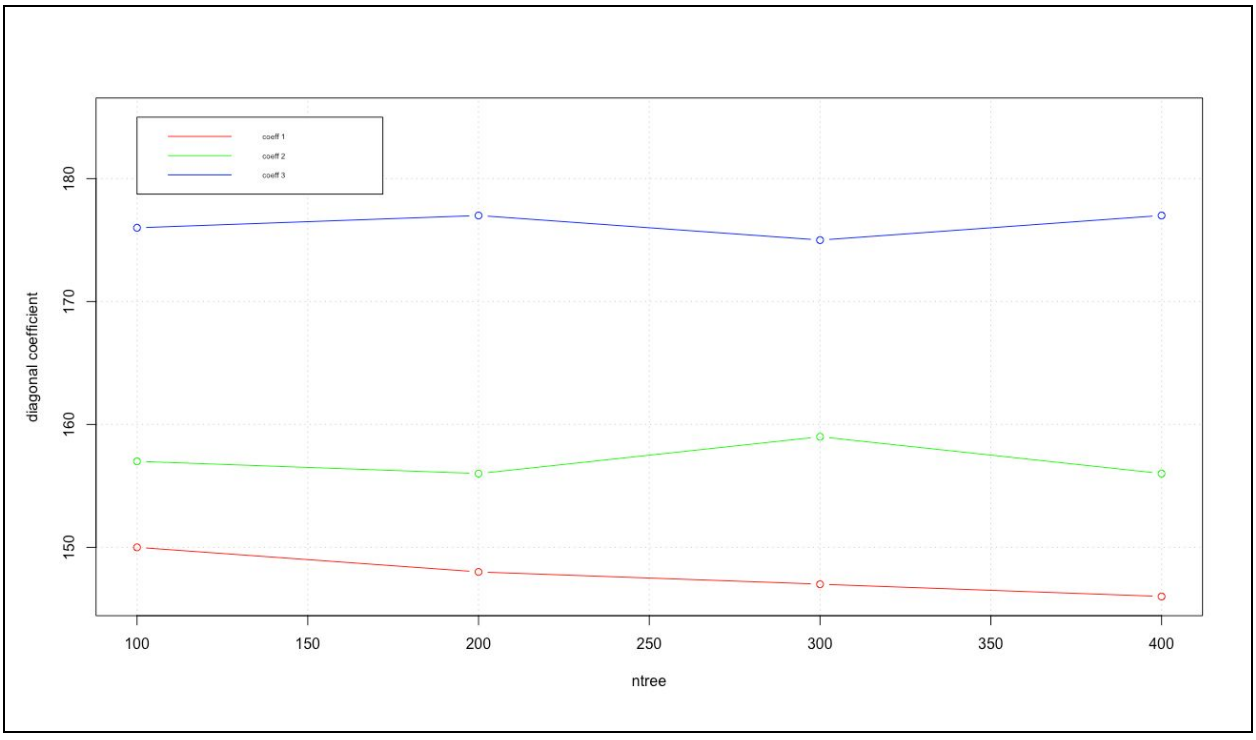
Accuracy
0.903774



Part 6: Selecting Best Value for *ntrees* in RF Model

The following curve below represents the diagonal coefficients vs, the number of decision trees (ntrees) of each class. Compare it to 6 confusion matrices above, we observe that the **Red** line is the first diagonal coefficients of the 4 matrices (1,1), {150, 148, 147, 146}, the **Green** line is the second diagonal coefficients of the 4 matrices (2,2), {157, 156, 159, 156}. The **Blue** line is the third diagonal coefficient of the 4 matrices (3,3), {176, 177, 175, 177}. Once again, these coefficients represent the times a font is correct while the model also predicted it to be that specific font.

Also, let's look back to the test set's accuracy vs ntrees in part 5. This time we are looking at our accuracy of which total classified correct divided by total cases in the test set. As we can see, it appears the highest test set accuracy is at when ntrees = 100 with an accuracy of 91.13%. Therefore, we select the best ntree value, BNT = 100.



Part 7: Feature Importance of Best RF Model

The **Feature Importance** refers to techniques that assign a score to input features based on how useful they are at predicting a target variable. It helps with a better understanding of the solved problem and leads to model improvements by employing feature selection that is the process where features are manually or automatically selected which contribute most to the target(true set) variable. In R, use the *importance()* function to view the importance of each variable.

The best RF model's importance IM_1, …, IM_p of standard. features Y_1, …, Y_p are computed, and features in decreasing order are reordered. The decreasing values of reordered importances are displayed below.

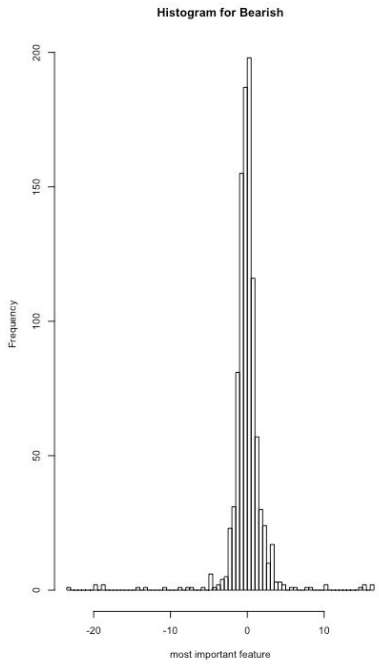
```
> sort(IM, decreasing = TRUE) # display decreasing values of recorded importances
[1] 183.703 177.858 177.749 177.118 173.403 172.609 168.101 166.991 165.797
```

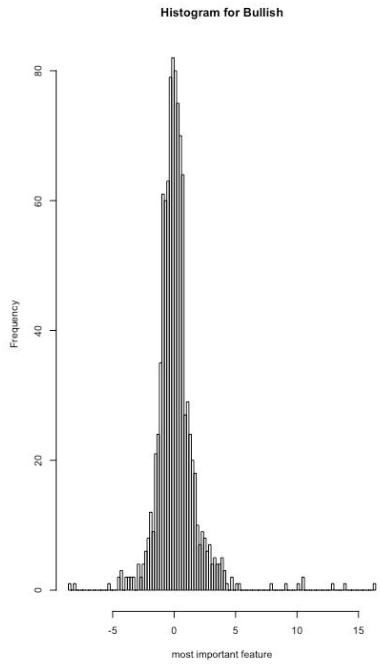
Part 8: KS Test for Most/Least Important Feature Z

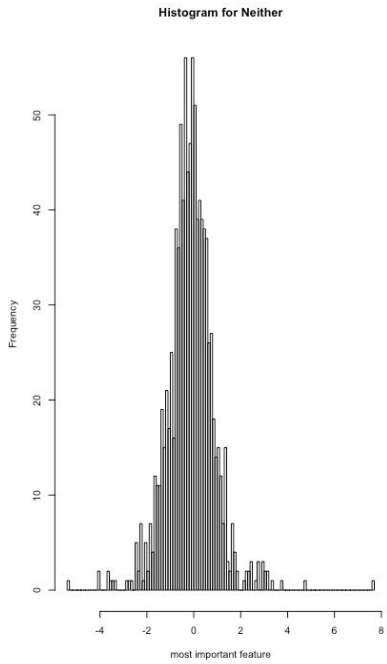
The Kolmogorov Smirnov Goodness-of-Fit test (KS test) is used to figure out if two samples share the same population with a specific distribution. Apply the hypothesis test and set up the null hypothesis that both samples follow a specific distribution and the alternate hypothesis be that both samples do not follow a specific distribution. Using R, the test statistic is calculated by a set of codes *ks.test(x, y)$statistic*. Also, test statistics, D, refer to the distance between the empirical distribution function of two samples in the graph of the KS test. The shorter the distance is, the more possible two samples come from the same population that is. The critical values for the KS test are determined by the simulation for the Anderson-Darling Test.

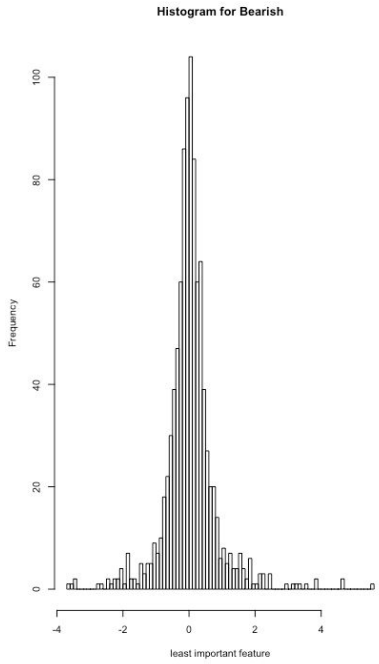
The first option of the decision rule is that reject H_0 if F-statistics $\geq$ critical value, and fail to reject H_0 if F-statistics $<$ critical value. The second option of it is that reject H_0 if p-value $\leq$ alpha, and fail to reject H_0 if p-value $>$ alpha. Alpha = 0.05 that is assumed.

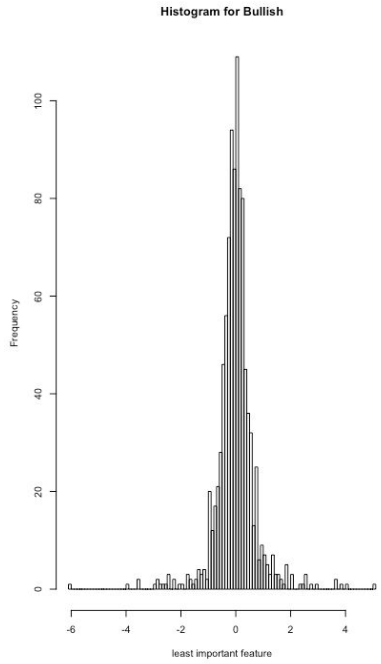
Based on the calculation, for the most important feature Z, the p-values of CL1 vs. CL2, CL1 vs. CL3, and CL2 vs. CL3 are, respectively, 0.386645, 0.00544649, and 0.000196495, so only one pair of CL1 & CL2 follow the same specified distribution. For the least important feature, the p-values of CL1 vs. CL2, CL1 vs. CL3, and CL2 vs. CL3 are 0.300945, 0.00860227, and 0.0721516, so two pairs of CL1 & CL3, and CL2 & CL3 follow the same specified distribution.

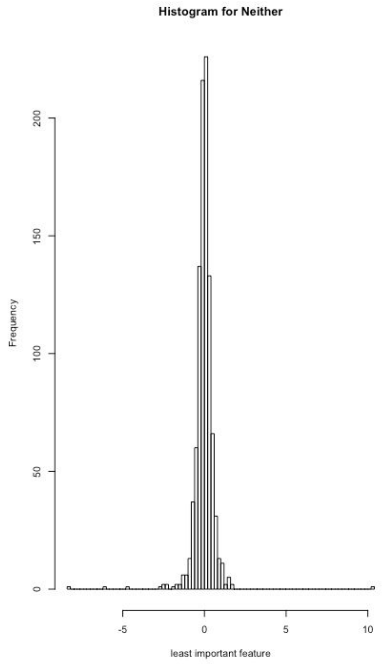

Histogram for Bearish
Frequency
0
50
100
150
200
-20
-10
0
10
most important feature


Histogram for Bullish
Frequency
0
20
40
60
80
-5
0
5
10
15
most important feature


Histogram for Neither
Frequency
0
10
20
30
40
50
-4
-2
0
2
4
6
8
most important feature


Histogram for Bearish
Frequency
0
20
40
60
80
100
-4
-2
0
2
4
least important feature


Histogram for Bullish
Frequency
0
20
40
60
80
100
-6
-4
-2
0
2
4
least important feature


Histogram for Neither
Frequency
0
50
100
150
200
-5
0
5
10
least important feature

> for (i in c(**p_1, p_2, p_3**)) {
+ cat("p_value is", paste0(i, ".\n"))
+ if (i <= alpha) {
+ cat("The null hypothesis is rejected. Two samples do not follow the same specified distribution.\n")
+ } else {
+ cat("The null hypothesis is not rejected. Two samples follow the same specified distribution.\n")
+ }
+ cat("\n")
+ }
p_value is 0.386645.
The null hypothesis is **not rejected**. Two samples follow the same specified distribution.
 	
p_value is 0.00544649.
The null hypothesis is **rejected**. Two samples do not follow the same specified distribution.
 	
p_value is 0.000196495.
The null hypothesis is **rejected.** Two samples do not follow the same specified distribution.
	> for (i in c(**p_4, p_5, p_6**)) {
	+ cat("p_value is", paste0(i, ".\n"))
	+ if (i <= alpha) {
	+ cat("The null hypothesis is rejected. Two samples do not follow the same specified distribution.\n")
	+ } else {
	+ cat("The null hypothesis is not rejected. Two samples follow the same specified distribution.\n")
	+ }
	+ cat("\n")
	+ }
	p_value is 0.300945.
	The null hypothesis is **not rejected**. Two samples follow the same specified distribution.
	
	p_value is 0.00860227.
	The null hypothesis is **rejected**. Two samples do not follow the same specified distribution.
	
	p_value is 0.0721516.
	The null hypothesis is **not rejected**. Two samples follow the same specified distribution.

Part 9 & 10

Let CLj = cluster with minimum gini Gj computed in part 3, Train a new RF classifier dedicated ONLY to cases belonging to CLj. Choose the training/test set in CLj and set ntry = sqrt(f) = 3, ntrees = BNT = 100. The balance of classes should be processed. Then, compute the confusion matrices of test set for each cluster CL1, CL2, CL_k*. When k = 3, the accuracy is getting highest.

```
> for (k in c(2:4)){
…
…
…
…
…
…
+     trainconf_CLUj = table(pred_CLUj_2,
TRAIN_CLUj[,1])
+     cat("-----------------------------------\n")
+     cat("when k =", paste0(k, ":\n\nconfusion matrix for test
set:"))
+     print(testconf_CLUj)
+     cat("\naccuracy of test set:\n",
sum(diag(testconf_CLUj))/sum(testconf_CLUj),"\n\n")
+     cat("confusion matrix for training set:")
+     print(trainconf_CLUj)
+     cat("\naccuracy of training set:\n",
sum(diag(trainconf_CLUj))/sum(trainconf_CLUj),"\n")
+     cat("-----------------------------------\n")
+ }
```

```
-----------------------------------
when k = 2:

confusion matrix for test set:
pred_CLUj_1    1    2
          1  312  312
          2   97   97

accuracy of test set:
 0.5

confusion matrix for training set:
pred_CLUj_2    1    2
          1 1258 1263
          2  379  374

accuracy of training set:
 0.498473
-----------------------------------
```

```
-----------------------------------
when k = 3:

confusion matrix for test set:
pred_CLUj_1    1    2    3
          1  383    0    0
          2    1   25    0
          3    0    0  132

accuracy of test set:
 0.998152

confusion matrix for training set:
pred_CLUj_2    1    2    3
          1 1533    0    0
          2    2  101    0
          3    0    0  528

accuracy of training set:
 0.999076
```

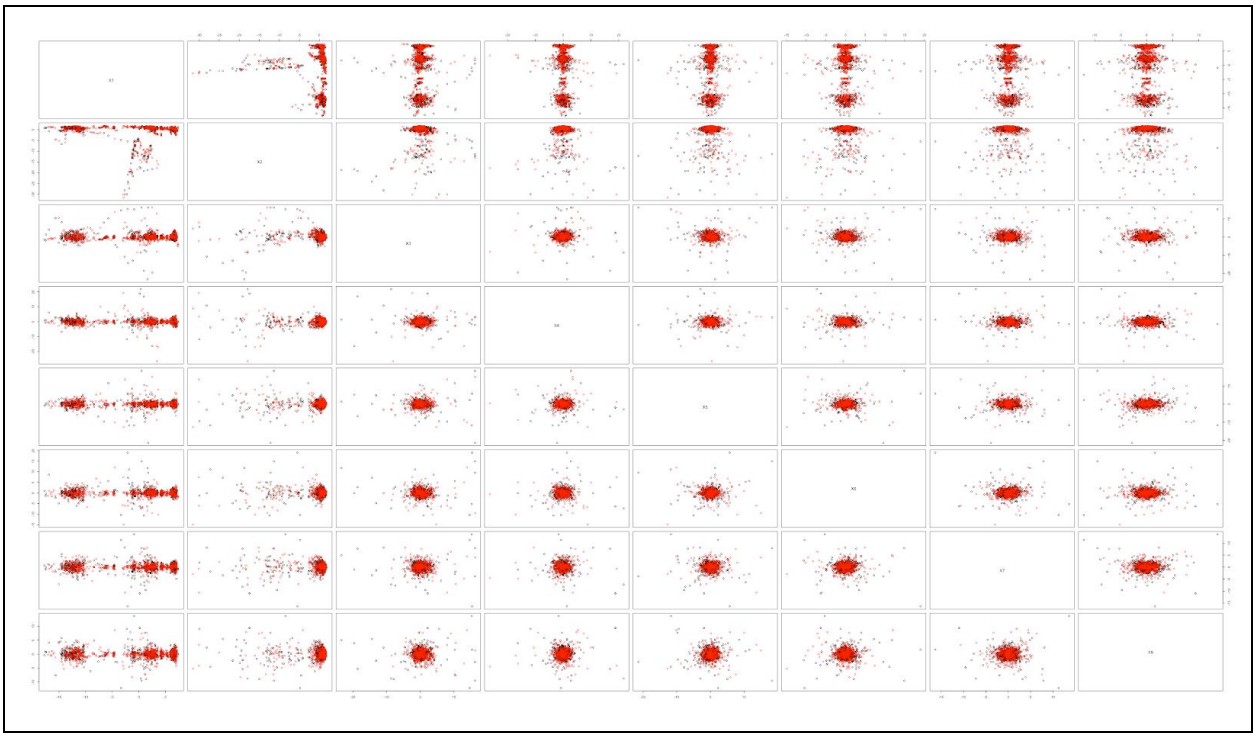
```
-----------------------------------
when k = 4:

confusion matrix for test set:
pred_CLUj_1    1    2    3    4
          1  148  150    0    0
          2   58   56    0    0
          3    0    0  143  147
          4    0    0   43   39

accuracy of test set:
 0.492347

confusion matrix for training set:
pred_CLUj_2    1    2    3    4
          1  616  612    0    0
          2  206  210    0    0
          3    0    0  569  573
          4    0    0  177  173
```

----------	accuracy of training set: 0.5 ----------

Part 11: Support Vector Machine (SVM)

Before implementing SVM in R, select two classes of three classes to create a new training set and a new test set to train a linear SVM to classify two classes selected.

It is supposed that It won't perform very well, because the data set has a lot of noise, which means the true set is overlapping as the scatterplots' matrix displayed below. It will result in a bad classification accuracy of the test set.



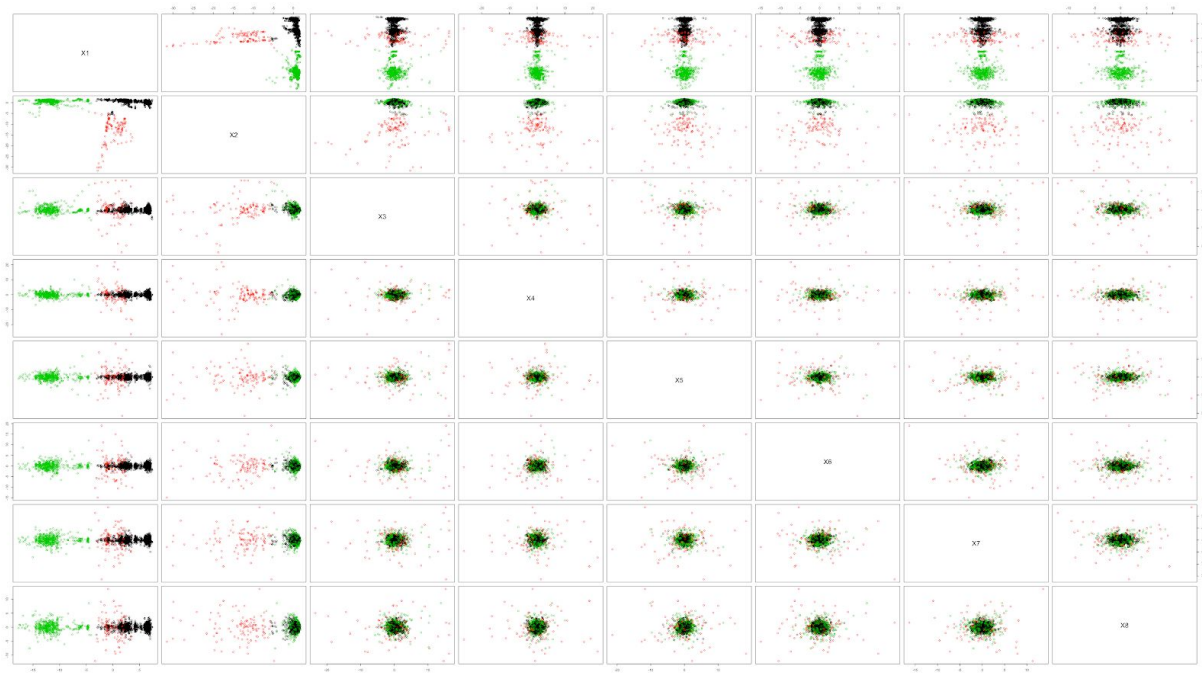
Just as we thought, the SVM is very sensitive to noise that a relatively small number of mislabeled examples can dramatically drop down the performance. Based on the calculation, we have the accuracy of the test set is 50% that is pretty low.

```
> print(conf)
          true_test
pred       Bearish Bullish
  Bearish       62      46
  Bullish      119     134
```

```
> sum(diag(conf))/sum(conf)
[1] 0.542936
```

If we want to improve the performance, we have to make the classes/categories distinct or may reduce the case size of the data set. Our data set has less than 3000 cases that are not considered as large. Let's try to replace the true set with the k-means clustering with the kbest. Now, the performance hit the max as 100%, because the true set wasn't classified properly.

```
> print(conf)
      true_test
pred  1   2   3
   1 24   0   0
   2  0  92   0
   3  0   0 245
> sum(diag(conf))/sum(conf)
[1] 1
```



Conclusion

Compared to the SVM, the random forest performs much better with this data set. Because the Random Forest is composed of decision trees and the average vote of the forest is selected as the predicted output. Random Forest and Decision trees are better for categorical data and they deal with collinearity better than SVM.

Unlike the Random Forest derive hyper-rectangles in input space to solve the problem, the SVM is flexible to use kernel tricks to solve nonlinear problems. But, The SVM is very sensitive to noise and outliers that some noisy or outlier training samples can significantly alter the hyperplane and lower the performance hence. But, it works well with a clear margin of separation and is effective in high dimensional spaces (more features). In conclusion, no one is the best or the worst, so, we need to base on the actual situation to choose a suitable machine learning model.

Time Used for Each Part

```
> print(t1) # part 1
   user  system elapsed
  4.334   0.358   4.868
>
> print(t2) # part 2
   user  system elapsed
156.117   3.319 159.618
>
> print(t3) # part 3
   user  system elapsed
  2.849   0.335   3.194
>
> print(t4) # part 4
   user  system elapsed
  0.020   0.005   0.022
>
> print(t5) # part 5
   user  system elapsed
 17.549   0.764  18.339
>
> print(t6) # part 6
   user  system elapsed
 26.115   1.138  27.326
>
> print(t7) # part 7
   user  system elapsed
  0.437   0.032   0.478
>
> print(t8) # part 8
   user  system elapsed
  0.201   0.022   0.231
>
> print(t9) # part 9 & 10
   user  system elapsed
  1.392   0.101   1.497
>
> print(t11) # part 11
   user  system elapsed
  3.130   0.046   3.207
```

Reference

- towardsdatascience.com/comparative-study-on-classic-machine-learning-algorithms-24f9ff6ab222
- r-bloggers.com/2011/10/a-new-dimension-to-principal-components-analysis/amp/
- sthda.com/english/wiki/scatterplot3d-3d-graphics-r-software-and-data-visualization
- plotly.com/r/3d-surface-plots/
- datacamp.com/community/tutorials/support-vector-machines-r
- r-bloggers.com/2018/01/how-to-implement-random-forests-in-r/
- itl.nist.gov/div898/handbook/eda/section3/eda35g.htm

R Codes

```
rm(list = ls())
cat("\f")

options(digits = 6)

setwd("~/library/Mobile Documents/com~apple~CloudDocs/Study Files (Graduate)/MATH
6350/MATH6350_Midterm/Dataset")
BA = data.frame(read.csv("BA_02.csv", na.strings = ""))
c.discard = match(c("t"), names(BA))
BA = na.omit(BA[, -c.discard])
str(BA)
start_col = match("Open_t", names(BA))
end_col = match("Adj_Return_t.5", names(BA))

#-----------------
#Q1
t1_start = proc.time()

# PCA
X = BA[, start_col:end_col]
SDATA = scale(X)
#pca = princomp(X,cor=TRUE,scores=TRUE)
#summary(pca)
## we can conclude that 95.211% of the variance will be accounted for with first nine principle components
#plot(pca, type = "line")

sigma = cov(SDATA) # variance-covariance matrix "sigma"
R=cov2cor(sigma)
D = eigen(R)$values
# or D = (summary(pca)$sdev)^2
Q = eigen(R)$vectors
p = order(abs(cumsum(D)/sum(D) - 0.95), decreasing = FALSE)[1]
V = Q[, 1:p]

layout(matrix(c(1:2), nrow = 1, ncol = 2))
# The variance explained by each principal component is obtained by squaring and then
# plot of the variance explained vs r
plot(1:length(D), D[1:length(D)]/sum(D), panel.first = grid(), type = "o", col = "black",
    xlab = "r", ylab = "variance explained", main = "Variance Explained vs. r")
text(x = (D[1:length(D)]/sum(D))[1:p], labels = signif((D[1:length(D)]/sum(D))[1:p], 3),
    cex = 0.4, pos = rep(c(3, 1), times = ceiling(p))[1:p], font = 2, col = "blue")

# compute the proportion of variance explained by each principal component and then
# plot of the PVE(r) vs r
plot(1:length(D), cumsum(D)/sum(D), panel.first = grid(), type = "o", col = "black",
    xlab = "r", ylab = "PVE(r)", main = "Percentage of Variance Explained:\nPVE(r) vs. r", yaxt = "n")
```

```
text(x = (cumsum(D)/sum(D))[1:p], labels = signif((cumsum(D)/sum(D))[1:p], 3),
    cex = 0.5, pos = 3, font = 2, col = "blue")
axis(2, at = seq(signif(D[1]/sum(D),1)-0.1, 1, 0.02))
abline(a = 0.95, b = 0, col = "red")

#---
library(ggplot2)
library(GGally)
ggpairs(data = data.frame(Q[,1:3]), columnLabels = c("V1", "V2", "V3"))

Q1 = Q[,1] # first PCA eigenvector:V1
Q2 = Q[,2] # second PCA eigenvector:V2
Q3 = Q[,3] # third PCA eigenvector:V3

#library(rgl)
#plot3d(Q[,1:3])

library(plotly)
plot_ly(x = Q1, y = Q2, z = Q3, type = "scatter3d", mode = "markers")
plot_ly(z = ~Q[,1:3]) %>% add_surface()
#---

true_set = BA[,1]
Y = data.frame(true_set, SDATA%*%V)
CL1 = Y[Y[,match("true_set", names(Y))] == "Bearish",]
CL2 = Y[Y[,match("true_set", names(Y))] == "Bullish",]
CL3 = Y[Y[,match("true_set", names(Y))] == "Neither",]

n1 = nrow(CL1)
n2 = nrow(CL2)
n3 = nrow(CL3)
N = sum(n1, n2, n3)

N == nrow(BA) # verification

SX = Y[,-1]

library(pca3d)
pca2d(prcomp(BA[,-1], scale. = TRUE), group = BA[,1])
pca3d(prcomp(BA[,-1], scale. = TRUE), group = BA[,1])

t1_stop = proc.time()
t1 = t1_stop - t1_start
print(t1)

#------
#Q2
t2_start = proc.time()
```

```
kmean = function(k) {
  set.seed(123)
  kmeans(x = SX, centers = k, nstart = 50, iter.max = 50)
}
cluster = function(k) {
  kmean(k)$cluster
}
SWS = function(k) { # Sum/Total Within-Clusters Sum of Squares
  kmean(k)$tot.withinss
}
# Clustering Performance Perf(k): % reduction of total dispersion by k-clustering
Perf = function(k) {
  1 - SWS(k)/SWS(1)
}
# Elbow Method for choosing the best k
k = c(1:50)
layout(matrix(c(1), 1, 1))
plot(k, sapply(k, SWS),
     type = "b", pch = 19, col = "black", frame = FALSE,
     xlim = c(0, max(k)),
     xlab = "k", ylab = "Total Within-Clusters Sum of Squares",
     main = "Elbow Method")

layout(matrix(c(1:2), nrow = 1, ncol = 2))
k1 = c(1:20)
plot(k1, sapply(k1, Perf), col = "black",
     type = "b", xlab = "k", ylab = "Perf(k)",
     main = "Clustering Performance",
     panel.first=grid())

s = function(k, c) {
  length(cluster(k)[cluster(k)==c])
}
f = function(k, c) { #class frequencies
  s(k, c)/length(cluster(k))
}

G = function(k) {
  1 - sum((mapply(f, c(k), c(1:k)))^2)
}
plot(k1, sapply(k1, G), col = "black",
     type = "b", ylim = c(0, 1), xlab = "k", ylab = "G(k)",
     main = "Gini Impurity Index", panel.first=grid())
lines(k1, sapply(k1, function(k) {1 - 1/k}), type = "l", col = "red")
legend(0.5, 1, c("max Impurity", "Gini Index G(k)"), lwd = 2,
       col = c("red", "black"), pch = 1, cex = 0.5)

kbest = 3
```

```
t2_stop = proc.time()
t2 = t2_stop - t2_start
print(t2)
```

#Q3

```
t3_start = proc.time()

fm = function(j) {
  A = function(m, j) {
    # class 1: Bearish; class 2: Bullish; class3: Neither
    length(intersect(which(as.integer(as.factor(true_set))==m),
                     which(cluster(3)==j)))
  }
  fm = rbind(A(m = 1, j)/s(k = kbest, c = j),
             A(m = 2, j)/s(k = kbest, c = j),
             A(m = 3, j)/s(k = kbest, c = j))
  return(fm)
}

for (j in 1:kbest) {
  cat("when j =", paste0(j, ", we have:\n"))
  cat("center A_j:\n")
  print(kmean(j)$center)
  cat("Size S_j:\n")
  print(kmean(j)$size) # table(kmean(j)$cluster)
  cat("Dispersion DIS_j:", SWS(j), "\n")
  cat("Gini G_j:", G(j), "\n")
  cat("Frequency fm_j:\n")
  FREQ_j = c()
  for (k in 1:j) {
    FREQ_j = append(FREQ_j, fm(k))
  }
  FREQ_j = matrix(FREQ_j, nrow = j)
  print(FREQ_j) # frequencies of each class in CLj
  cat("\n")
}

library(scatterplot3d)
par(mfrow=c(1,1))
#scatterplot3d(Y[,2:4],
#              color = c("red", "blue", "orange")[as.numeric(Y[,1])],
#              pch=20, xlab='V1', ylab='V2', zlab='V3',
#              main='3D Projection of Cluster')

scatterplot3d(Y[,2:4],
              color = c("red", "blue", "orange")[kmean(kbest)$cluster],
```

```
          pch=20, xlab='V1', ylab='V2', zlab='V3',
          main='3D Projection of Cluster')

t3_stop = proc.time()
t3 = t3_stop - t3_start
print(t3)
#----
```

#Q4

```
t4_start = proc.time()

#---
table(true_set)

set.seed(123)
x1 = sample(x = nrow(CL1),
          size = as.integer(0.8*nrow(CL1)),
          replace = FALSE)
x2 = sample(x = nrow(CL2),
          size = as.integer(0.8*nrow(CL2)),
          replace = FALSE)
time1 = ceiling(as.integer(0.8*nrow(CL3))/length(x1))
time2 = ceiling(as.integer(0.8*nrow(CL3))/length(x2))
i1 = sample(x = setdiff(c(1:(length(x1)*time1)), seq(1, length(x1)*time1, time1)),
          size = length(x1)*time1 - as.integer(0.8*nrow(CL3)),
          replace = FALSE)
i2 = sample(x = setdiff(c(1:(length(x2)*time2)), seq(1, length(x2)*time2, time2)),
          size = length(x2)*time2 - as.integer(0.8*nrow(CL3)),
          replace = FALSE)

pd_CL1train = rep(x1, each = time1)[-i1]
pd_CL2train = rep(x2, each = time2)[-i2]
setdiff(x1, pd_CL1train) # verification for all the original 80% of cases are not missed, so we expect the output be
integer(0)
setdiff(x2, pd_CL2train)
# setdiff(x, y) is the material that is in x, that is not in y

y1 = setdiff(c(1:nrow(CL1)), x1)
y2 = setdiff(c(1:nrow(CL2)), x2)
TIME1 = ceiling((nrow(CL3) - as.integer(0.8*nrow(CL3)))/length(y1))
TIME2 = ceiling((nrow(CL3) - as.integer(0.8*nrow(CL3)))/length(y2))
j1 = sample(x = setdiff(c(1:(length(y1)*TIME1)), seq(1, (length(y1)*TIME1), TIME1)),
          size = length(y1)*TIME1 - (nrow(CL3) - as.integer(0.8*nrow(CL3))),
          replace = FALSE)
j2 = sample(x = setdiff(c(1:(length(y2)*TIME2)), seq(1, (length(y2)*TIME2), TIME2)),
          size = length(y2)*TIME2 - (nrow(CL3) - as.integer(0.8*nrow(CL3))),
          replace = FALSE)
pd_CL1test = rep(y1, each = TIME1)[-j1]
```

```
pd_CL2test = rep(y2, each = TIME2)[-j2]

setdiff(y1, pd_CL1test) # verification for all the original 20% of cases are not missed, so we expect the output be integer(0)
setdiff(y2, pd_CL2test)
intersect(pd_CL1train, pd_CL1test) # verification for test set and training set do not intersect
intersect(pd_CL2train, pd_CL2test)

pd_CL3train = sample(x = nrow(CL3), size = round(0.8*nrow(CL3)), replace = FALSE)

trainCL1 = CL1[pd_CL1train,]
testCL1 = CL1[pd_CL1test,]

trainCL2 = CL2[pd_CL2train,]
testCL2 = CL2[pd_CL2test,]

trainCL3 = CL3[pd_CL3train,]
testCL3 = CL3[-pd_CL3train,]

trainset = rbind(trainCL1, trainCL2, trainCL3)[,-1] # newTRAJ
testset = rbind(testCL1, testCL2, testCL3)[,-1] # newTESJ

trainset_target = as.factor(rbind(trainCL1, trainCL2, trainCL3)[,1])
testset_target = as.factor(rbind(testCL1, testCL2, testCL3)[,1])
#---
#---
set.seed(123)
pd1 = sample(x = 2, size = nrow(CL1), replace = TRUE, prob = c(0.8, 0.2)) # partition data of CL1
pd2 = sample(x = 2, size = nrow(CL2), replace = TRUE, prob = c(0.8, 0.2)) # partition data of CL2
pd3 = sample(x = 2, size = nrow(CL3), replace = TRUE, prob = c(0.8, 0.2)) # partition data of CL3
traincl1 = CL1[pd1==1, ]
testcl1 = CL1[pd1==2, ]
traincl2 = CL2[pd2==1, ]
testcl2 = CL2[pd2==2, ]
traincl3 = CL3[pd3==1, ]
testcl3 = CL3[pd3==2, ]

TRAINSET_TARGET = rbind(traincl1, traincl2, traincl3)[,1] # true train set
TESTSET_TARGET = rbind(testcl1, testcl2, testcl3)[,1] # true test set

TRAINSET = rbind(traincl1, traincl2, traincl3)[,-1]
TESTSET = rbind(testcl1, testcl2, testcl3)[,-1]


t4_stop = proc.time()
t4 = t4_stop - t4_start
print(t4)

#---
```

#Q5

```
t5_start = proc.time()

library(randomForest)
rf = function(ntrees) {
  set.seed(123)
  rf = randomForest(trainset_target~., data = trainset, ntree = ntrees, mtry = sqrt(p))
  return(rf)
}
library(caret)
trainconf = function(ntrees) {
  pred1 = predict(rf(ntrees), TRAINSET)
  trainconf = confusionMatrix(pred1, TRAINSET_TARGET)$table
  return(trainconf)
}
trainperf = function(ntrees) {
  pred1 = predict(rf(ntrees), TRAINSET)
  trainperf = confusionMatrix(pred1, TRAINSET_TARGET)$overall[1]
  return(trainperf)
}
testconf = function(ntrees) {
  pred2 = predict(rf(ntrees), TESTSET)
  testconf = confusionMatrix(pred2, TESTSET_TARGET)$table
  return(testconf)
}
testperf = function(ntrees) { # accuracy
  pred2 = predict(rf(ntrees), TESTSET)
  testperf = confusionMatrix(pred2, TESTSET_TARGET)$overall[1]
  return(testperf)
}
n = c(100, 200, 300, 400)
for (i in n) {
  cat("ntree =", paste0(i, ":\n"))
  print(trainconf(i))
  print(trainperf(i))
  cat("\n")
}
for (i in n) {
  cat("ntree =", paste0(i, ":\n"))
  print(testconf(i))
  print(testperf(i))
  cat("\n")
}

plot(n, mapply(trainperf, n),
     ylim  = c(0.86, 0.93),
     panel.first=grid(),
     ylab = "accuracy",
```

```
    xlab = "ntree",
    main = "ntree vs. accuracy",
    col = "red",
    type = "o")
lines(n, mapply(testperf, n),
     col = "blue",
     type = "o")
legend(100, 0.93, c("trainperf", "testperf"), lwd = 2,
      col = c("red", "blue"), pch = 1, cex = 0.5)

t5_stop = proc.time()
t5 = t5_stop - t5_start
print(t5)
```

#Q6

```
t6_start = proc.time()

conf = function(x) {
  n = n[x]
  conf = testconf(n)
  return(conf)
}
coeff = function(k) {
  coeff = NULL
  for (i in c(1:length(n))) {
    coeff[i] = diag(matrix(mapply(conf, i), 3))[k]
  }
  return(coeff)
}
library(plyr) # for round_any() function
layout(matrix(c(1), 1, 1))
plot(n, coeff(1), type = "b", col = "red",
    ylim = c(min(mapply(coeff, 1:3)), 5 + round_any(max(mapply(coeff, 1:3)), accuracy = 5, f = ceiling)),
    panel.first=grid(), xlab = "ntree", ylab = "diagonal coefficient")
lines(n, coeff(2), type = "b", col = "green")
lines(n, coeff(3), type = "b", col = "blue")
legend(n[1], 185, legend = c("coeff 1","coeff 2","coeff 3"), col=c("red","green","blue"), lty=1:1, cex=0.5)

BNT = 100 # best value for ntrees

t6_stop = proc.time()
t6 = t6_stop - t6_start
print(t6)
```

#Q7

```
t7_start = proc.time()
```

```
bestRF = rf(ntrees = BNT)
IM = bestRF$importance
sort(IM, decreasing = TRUE) # display decreasing values of reordered importances

t7_stop = proc.time()
t7 = t7_stop - t7_start
print(t7)
```

#Q8

```
t8_start = proc.time()

target = rbind(trainCL1, trainCL2, trainCL3, testCL1, testCL2, testCL3)[,1]

Z1 = order(IM, decreasing = TRUE)[1] # the most important feature
layout(matrix(c(1:3), 1, 3))
c_1 = rbind(trainset, testset)[target=="Bearish",Z1]
c_2 = rbind(trainset, testset)[target=="Bullish",Z1]
c_3 = rbind(trainset, testset)[target=="Neither",Z1]
hist(x = c_1, breaks = 100,
    main = "Histogram for Bearish", xlab = "most important feature")
hist(x = c_2, breaks = 100,
    main = "Histogram for Bullish", xlab = "most important feature")
hist(x = c_3, breaks = 100,
    main = "Histogram for Neither", xlab = "most important feature")

Z2 = order(IM, decreasing = FALSE)[1] # the least important feature
layout(matrix(c(1:3), 1, 3))
c_4 = rbind(trainset, testset)[target=="Bearish",Z2]
c_5 = rbind(trainset, testset)[target=="Bullish",Z2]
c_6 = rbind(trainset, testset)[target=="Neither",Z2]
hist(x = c_4, breaks = 100,
    main = "Histogram for Bearish", xlab = "least important feature")
hist(x = c_5, breaks = 100,
    main = "Histogram for Bullish", xlab = "least important feature")
hist(x = c_6, breaks = 100,
    main = "Histogram for Neither", xlab = "least important feature")

# reference of Kolmogorov-Smirnov Goodness-of-Fit Test (KS test):
# https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm
# used to decide if a sample comes from a population with a specific distribution
# H0:     The data follow a specified distribution
# Ha:     The data do not follow a specified distribution
p_value = function(x1, x2) {
  ks = ks.test(x1, x2, alternative = "two.sided")$p.value
  return(ks)
}
options(warn=-1) # turn the warning off
```

```
# Decision Rule:
# Reject H0 if test_statistic >= critical_value, and fail to reject H0 if test_statistic < critical_value
# Reject H0 if p-value <= alpha, and fail to reject H0 if p-value > alpha
alpha = 0.05

p_1 = p_value(c_1, c_2)
p_2 = p_value(c_1, c_3)
p_3 = p_value(c_2, c_3)
for (i in c(p_1, p_2, p_3)) {
  cat("p_value is", paste0(i, ".\n"))
  if (i <= alpha) {
    cat("The null hypothesis is rejected. Two samples do not follow the same specified distribution.\n")
  } else {
    cat("The null hypothesis is not rejected. Two samples follow the same specified distribution.\n")
  }
  cat("\n")
}

p_4 = p_value(c_4, c_5)
p_5 = p_value(c_4, c_6)
p_6 = p_value(c_5, c_6)
for (i in c(p_4, p_5, p_6)) {
  cat("p_value is", paste0(i, ".\n"))
  if (i <= alpha) {
    cat("The null hypothesis is rejected. Two samples do not follow the same specified distribution.\n")
  } else {
    cat("The null hypothesis is not rejected. Two samples follow the same specified distribution.\n")
  }
  cat("\n")
}

options(warn=0) # turn the warning back on

t8_stop = proc.time()
t8 = t8_stop - t8_start
print(t8)
```

#Q9 & Q10

```
t9_start = proc.time()

# parameter CL is the data of a class/category extracted from the original data set
pdTRAIN = function(CL, size) { # partition data
  # size: balanced size for each class
  set.seed(123)
  if (SIZE > nrow(CL)) {
    x = sample(x = nrow(CL),
```

```
            size = as.integer(0.8*nrow(CL)),
            replace = FALSE)
    time = ceiling(as.integer(0.8*SIZE)/length(x))
    i = sample(x = setdiff(c(1:(length(x)*time)), seq(1, length(x)*time, time)),
            size = length(x)*time - as.integer(0.8*SIZE),
            replace = FALSE)
    pd_CLtrain = rep(x, each = time)[-i]
    return(pd_CLtrain)
  } else if (SIZE <= nrow(CL)) {
    pd_CLtrain = sample(x = nrow(CL), size = round(0.8*SIZE), replace = FALSE)
    return(pd_CLtrain)
  }
}
pdTEST = function(CL, size) {
  set.seed(123)
  if (SIZE > nrow(CL)) {
    x = sample(x = nrow(CL),
            size = as.integer(0.8*nrow(CL)),
            replace = FALSE)
    y = setdiff(c(1:nrow(CL)), x)
    TIME = ceiling((SIZE - as.integer(0.8*SIZE))/length(y))
    j = sample(x = setdiff(c(1:(length(y)*TIME)), seq(1, (length(y)*TIME), TIME)),
            size = length(y)*TIME - (SIZE - as.integer(0.8*SIZE)),
            replace = FALSE)
    pd_CLtest = rep(y, each = TIME)[-j]
    return(pd_CLtest)
  } else if (SIZE <= nrow(CL)) {
    pd_CLtrain = pdTRAIN(CL)
    pd_CLtest = sample(x = c(1:nrow(CL))[-pd_CLtrain], size = round(0.2*SIZE), replace = FALSE)
    return(pd_CLtest)
  }
}
balanced_train = function(CL, size) {
  pd_CLtrain = pdTRAIN(CL, size)
  trainCL = CL[pd_CLtrain,]
  return(trainCL)
}
balanced_test = function(CL, size) {
  pd_CLtest = pdTEST(CL, size)
  testCL = CL[pd_CLtest,]
  return(testCL)
}
pd_original_train = function(CL) {
  set.seed(123)
  pd = sample(x = nrow(CL), size = round(0.8*nrow(CL)), replace = FALSE)
  return(pd)
}
original_train = function(CL) {
  original_train = CL[pd_original_train(CL),]
```

```
  return(original_train)
}
original_test = function(CL) {
  original_test = CL[-pd_original_train(CL),]
  return(original_test)
}

SIZE = as.integer(length(true_set)/length(table(true_set))) # balanced size for each class


# kbest = 3

CLUj = function(k) {
  kmeans(x = SX,
         center = k, # kbest = 3
         nstart = 50,
         iter.max = 50)$cluster
}

for (k in c(2:4)){
  set.seed(2020)
  SY = data.frame(SX)
  if (k == 4) {
    CL1_CLUj = SY[as.vector(CLUj(4)) == 1,]
    CL1_CLUj = data.frame(true_set = rep(1, nrow(CL1_CLUj)), CL1_CLUj)
    CL2_CLUj = SY[as.vector(CLUj(4)) == 2,]
    CL2_CLUj = data.frame(true_set = rep(2, nrow(CL2_CLUj)), CL2_CLUj)
    CL3_CLUj = SY[as.vector(CLUj(4)) == 3,]
    CL3_CLUj = data.frame(true_set = rep(3, nrow(CL3_CLUj)), CL3_CLUj)
    CL4_CLUj = SY[as.vector(CLUj(4)) == 4,]
    CL4_CLUj = data.frame(true_set = rep(4, nrow(CL4_CLUj)), CL4_CLUj)
    train_CLUj = rbind(balanced_train(CL1_CLUj, SIZE),
                       balanced_train(CL2_CLUj, SIZE),
                       balanced_train(CL3_CLUj, SIZE),
                       balanced_train(CL4_CLUj, SIZE))
    test_CLUj = rbind(balanced_test(CL1_CLUj, SIZE),
                      balanced_test(CL2_CLUj, SIZE),
                      balanced_test(CL3_CLUj, SIZE),
                      balanced_test(CL4_CLUj, SIZE))
    TRAIN_CLUj = rbind(original_train(CL1_CLUj),
                       original_train(CL2_CLUj),
                       original_train(CL3_CLUj),
                       original_train(CL4_CLUj))
    TEST_CLUj = rbind(original_test(CL1_CLUj),
                      original_test(CL2_CLUj),
                      original_test(CL3_CLUj),
                      original_test(CL4_CLUj))
    } else if (k == 2) {
      CL1_CLUj = SY[as.vector(CLUj(2)) == 1,]
```

```
    CL1_CLUj = data.frame(true_set = rep(1, nrow(CL1_CLUj)), CL1_CLUj)
    CL2_CLUj = SY[as.vector(CLUj(2)) == 2,]
    CL2_CLUj = data.frame(true_set = rep(2, nrow(CL2_CLUj)), CL2_CLUj)
    train_CLUj = rbind(balanced_train(CL1_CLUj, SIZE),
                       balanced_train(CL2_CLUj, SIZE))
    test_CLUj = rbind(balanced_test(CL1_CLUj, SIZE),
                      balanced_test(CL2_CLUj, SIZE))
    TRAIN_CLUj = rbind(original_train(CL1_CLUj),
                       original_train(CL2_CLUj))
    TEST_CLUj = rbind(original_test(CL1_CLUj),
                      original_test(CL2_CLUj))
    } else if (k == 3) {
      CL1_CLUj = SY[as.vector(CLUj(3)) == 1,]
      CL1_CLUj = data.frame(true_set = rep(1, nrow(CL1_CLUj)), CL1_CLUj)
      CL2_CLUj = SY[as.vector(CLUj(3)) == 2,]
      CL2_CLUj = data.frame(true_set = rep(2, nrow(CL2_CLUj)), CL2_CLUj)
      CL3_CLUj = SY[as.vector(CLUj(3)) == 3,]
      CL3_CLUj = data.frame(true_set = rep(3, nrow(CL3_CLUj)), CL3_CLUj)
      train_CLUj = rbind(balanced_train(CL1_CLUj, SIZE),
                         balanced_train(CL2_CLUj, SIZE),
                         balanced_train(CL3_CLUj, SIZE))
      test_CLUj = rbind(balanced_test(CL1_CLUj, SIZE),
                        balanced_test(CL2_CLUj, SIZE),
                        balanced_test(CL3_CLUj, SIZE))
      TRAIN_CLUj = rbind(original_train(CL1_CLUj),
                         original_train(CL2_CLUj),
                         original_train(CL3_CLUj))
      TEST_CLUj = rbind(original_test(CL1_CLUj),
                        original_test(CL2_CLUj),
                        original_test(CL3_CLUj))
    }

  rf_CLUj = randomForest(as.factor(as.character(train_CLUj[,1]))~.,
                         data = train_CLUj[,-1],
                         ntree = BNT,
                         mtry = sqrt(p))
  pred_CLUj_1 = predict(rf_CLUj, TEST_CLUj[,-1])
  testconf_CLUj = table(pred_CLUj_1, TEST_CLUj[,1])
  pred_CLUj_2 = predict(rf_CLUj, TRAIN_CLUj[,-1])
  trainconf_CLUj = table(pred_CLUj_2, TRAIN_CLUj[,1])
  cat("-----------------------------------\n")
  cat("when k =", paste0(k, ":\n\nconfusion matrix for test set:"))
  print(testconf_CLUj)
  cat("\naccuracy of test set:\n", sum(diag(testconf_CLUj))/sum(testconf_CLUj),"\n\n")
  cat("confusion matrix for training set:")
  print(trainconf_CLUj)
  cat("\naccuracy of training set:\n", sum(diag(trainconf_CLUj))/sum(trainconf_CLUj),"\n")
  cat("-----------------------------------\n")
}
```

```
t9_stop = proc.time()
t9 = t9_stop - t9_start
print(t9)
```

#Q11

```
t11_start = proc.time()

library(e1071)

newTRAIN = rbind(balanced_train(CL1, SIZE), balanced_train(CL2, SIZE))
newTEST = rbind(balanced_test(CL1, SIZE), balanced_test(CL2, SIZE))


svm_model = svm(as.factor(as.character(newTRAIN[,1]))~.,
                data = newTRAIN[,-1],
                kernel = "linear",
                scale = TRUE,
                cost = 1)

# when cost is small, margins will be wide, more tolerence for misclassification
summary(svm_model)

pred = predict(svm_model, newTEST[,-1])
conf = table(pred, true_test = as.factor(as.character(newTEST[,1])))
print(conf)
sum(diag(conf))/sum(conf)



sx = (rbind(newTRAIN, newTEST))[1:9]
plot(sx[,-1], col = as.factor(sx[,1]))

t11_stop = proc.time()
t11 = t11_stop - t11_start
print(t11)



t11_start = proc.time()

library(e1071)
true_cluster = as.vector(CLUj(3))
newTRAIN_CL1 = data.frame(true_cluster = true_cluster[as.integer(row.names(balanced_train(CL1, SIZE)))],
                          balanced_train(CL1, SIZE))[,-2]
```

```
newTRAIN_CL2 = data.frame(true_cluster = true_cluster[as.integer(row.names(balanced_train(CL2, SIZE)))],
                          balanced_train(CL2, SIZE))[,-2]

newTEST_CL1 = data.frame(true_cluster = true_cluster[as.integer(row.names(balanced_test(CL1, SIZE)))],
                         balanced_test(CL1, SIZE))[,-2]
newTEST_CL2 = data.frame(true_cluster = true_cluster[as.integer(row.names(balanced_test(CL2, SIZE)))],
                         balanced_test(CL2, SIZE))[,-2]

newTRAIN = rbind(newTRAIN_CL1, newTRAIN_CL2)
newTEST = rbind(newTEST_CL1, newTEST_CL2)

svm_model = svm(as.factor(as.character(newTRAIN[,1]))~.,
                data = newTRAIN[,-1],
                kernel = "linear",
                scale = TRUE,
                cost = 1)

# when cost is small, margins will be wide, more tolerence for misclassification
summary(svm_model)

pred = predict(svm_model, newTEST[,-1])
conf = table(pred, true_test = as.factor(as.character(newTEST[,1])))
print(conf)
sum(diag(conf))/sum(conf)



sx = (rbind(newTRAIN, newTEST))[1:9]
plot(sx[,-1], col = as.factor(sx[,1]))

t11_stop = proc.time()
t11 = t11_stop - t11_start
print(t11)
```