Raphael Suarez

MATH 6350 Final Exam

12/13/2020

Q0: The Data

This data was recorded in order to attempt to classify subjects' body positions. There were monitors placed on the subjects' hands, wrists, and head which monitored the velocity and acceleration of these specific body parts every 10 milliseconds. The results are 32 numerical features and 9873 samples. There is one more feature called "Phase" which gives us the position of the person. In order to change these variables into numerical values, I assigned D to the number 1, H to 2, P to 3, R to 4, and S to 5. These categorical variables will be our classes:

Class 'D': 2741 cases

Class 'H': 998 cases

Class 'P': 2097 cases

Class 'R': 1087 cases

Class 'S': 2950 cases

The size of the classes are relatively balanced, and we will therefore not need to regroup, clone, or subsample any of them.

There are negative and possible values, we imagine the negative values could stand for deceleration of limbs, and positive values for acceleration.

This project could prove to be useful in detecting specific human movements, interpreting sign language, and many other reasons. This is why we take the opportunity to try to classify each case to its respective class using KMEANS clustering, random forest algorithms, and linear SVM models. In addition to this, we will revise and analyze our performance and use other techniques such as cloning the features or running models on specific clusters of data to improve our results.

In order to perform multiple of the tests, we had to split the data randomly into TEST and TRAIN sets. The specifics of the test will be explained later, but here are the sizes of the TEST and TRAIN sets for the whole data set:

TEST set: 1977 cases (approximately 20% of the data)

TRAIN set: 7896 cases (approximately 80% of the data)

Q1: PCA Analysis

Principal component analysis (PCA) is a technique for processing sets of data with many predictor variables (features). The main objective of PCA is to reduce the number of variables used to estimate dependent variables, often called data compression or dimension reduction. Reducing the number of variables can drastically reduce the computation time of a statistical learning algorithm such as K-nearest neighbor (KNN), K-means clustering, or random forest. However, information can also be lost in the reduction. Therefore, we usually want to select the principal components that explain roughly 95% of the variance in the dataset.

To perform a PCA analysis, first we had to standardize the data. In order to do this, we used the scale() function in R, which centers and scales each column in the data frame. Our standardized set is called SDATA. Next we create a correlation matrix for all of SDATA and apply the eigen() function which allows us to extract the eigenvalues and eigenvectors.

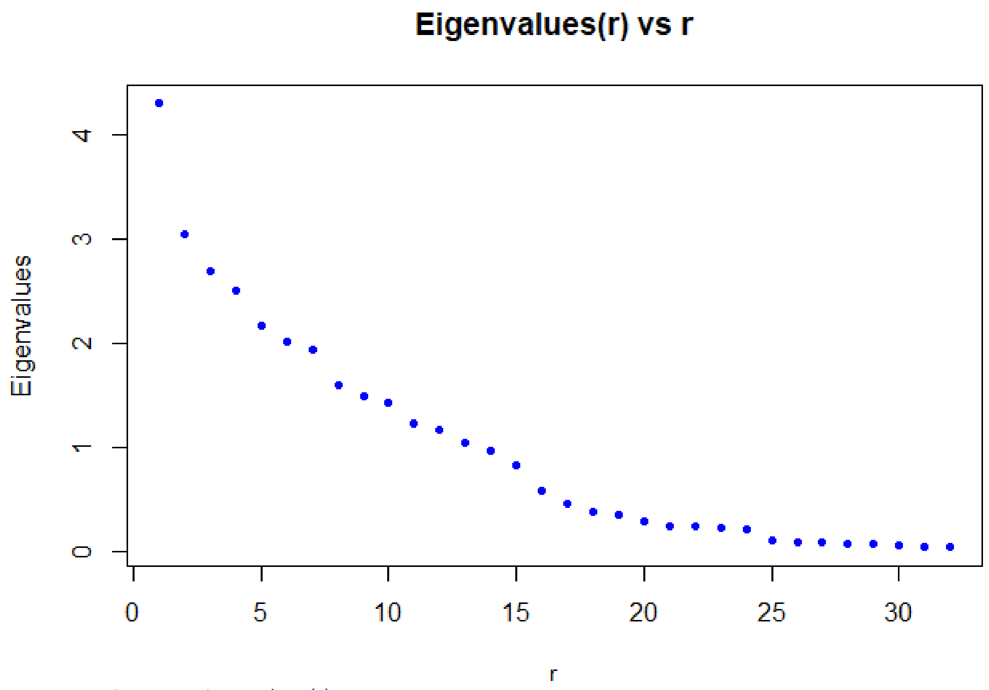


Figure 1: Eigenvalues(r) vs r

Above is the plot for eigenvalue(r) vs. r. This plot gives us an idea of how much variance is explained by the first couple of eigenvalues and how important they are. We can find the variance explained as a proportion and replot the information to get a different look at the eigenvalues.

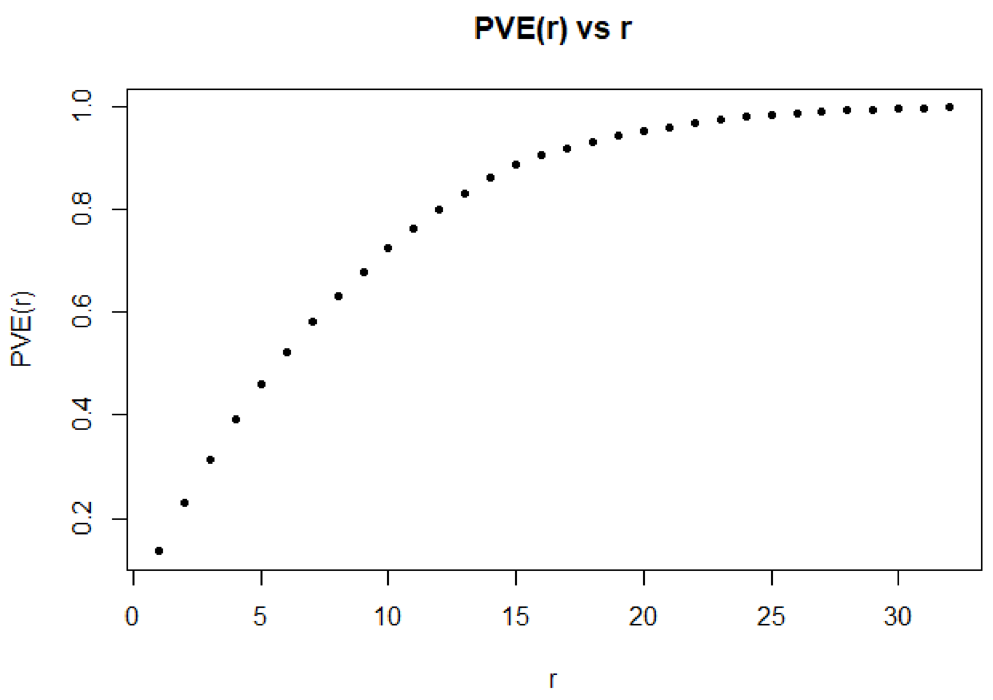


Figure 2: PVE(r) vs r

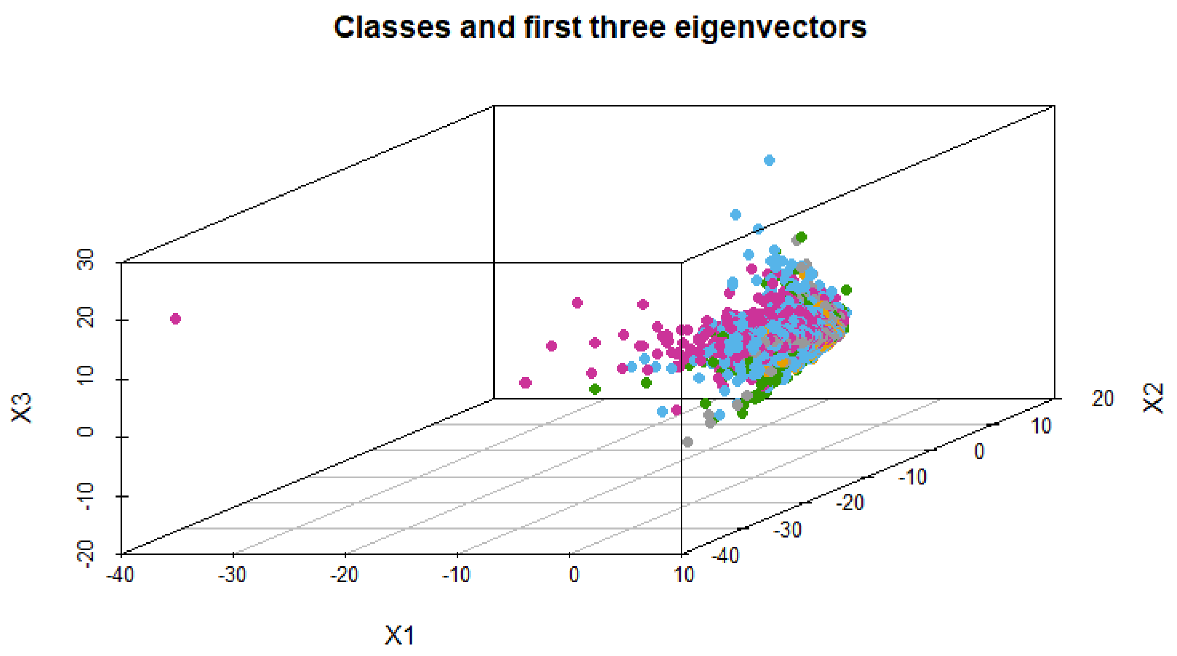
We can see from the graph that as the r increases and gets closer to 30, the percent of variance explained almost reaches 100%. For efficient dimension reduction, we would look at the first 20 eigenvectors, which appear to explain around 95% of the variance in the dataset.

First three eigenvalues and % of total variance:

1. 4.30 13.45%
2. 3.04 22.96%
3. 2.69 31.38%

These 3 eigenvalues and their respective eigenvectors carry 31.38% of the variance in the whole dataset. It is evident that PCA analysis can reduce the amount of data we're working with at an efficient level.

We created a 3D plot of the 5 classes vs the first three eigenvectors and separated them by color. This can give us a brief overview of how we can separate and classify the data into the 5 classes and how PCA analysis could help us with this task.



It appears as though our data isn't going to be easily separated using just the first three eigenvectors. The cases for the 5 classes are all accumulated on a range of coordinates with a small number of outliers.

In order to separate the data, we're going to need more than 31% of the explained variance and some other methods. We will explore K-means clustering in the next section.

Q2: K-means

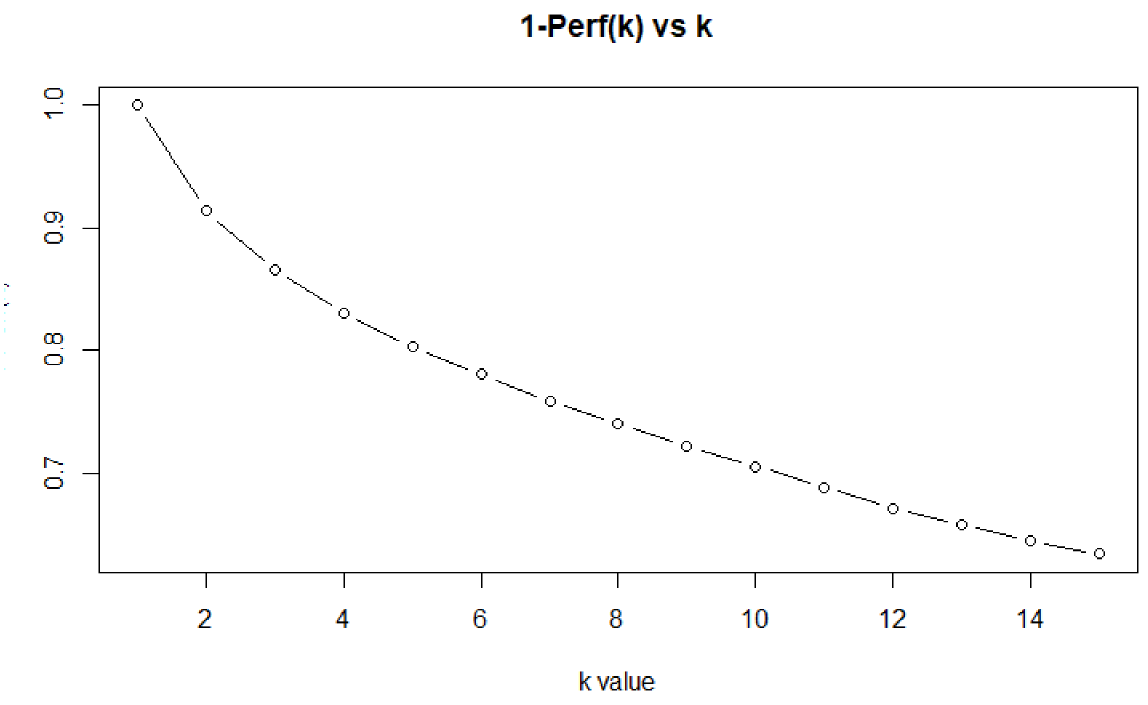
K-means clustering is an unsupervised method for machine learning for partitioning the data. Unsupervised means it doesn't take into account the class of each case. K-means can be used to assign an observation to one and only one of a number k of clusters. The assignment to a group is made on the idea that observations within a cluster tend to be similar, and different to the observations of another cluster. By performing K-means classification we can get insights about the structure of the data set and subsequently draw conclusions.

It is important to choose a k value with good performance, low impurity, and also a relatively quick computing time. We run the kmeans() function on SDATA with a k range of 1-15. We calculate the Gini and the performance for each K in order to help us choose the best number of clusters. To get an insight into the computing time for each run, we calculated how long it took for k=3, k=10, and k=15:

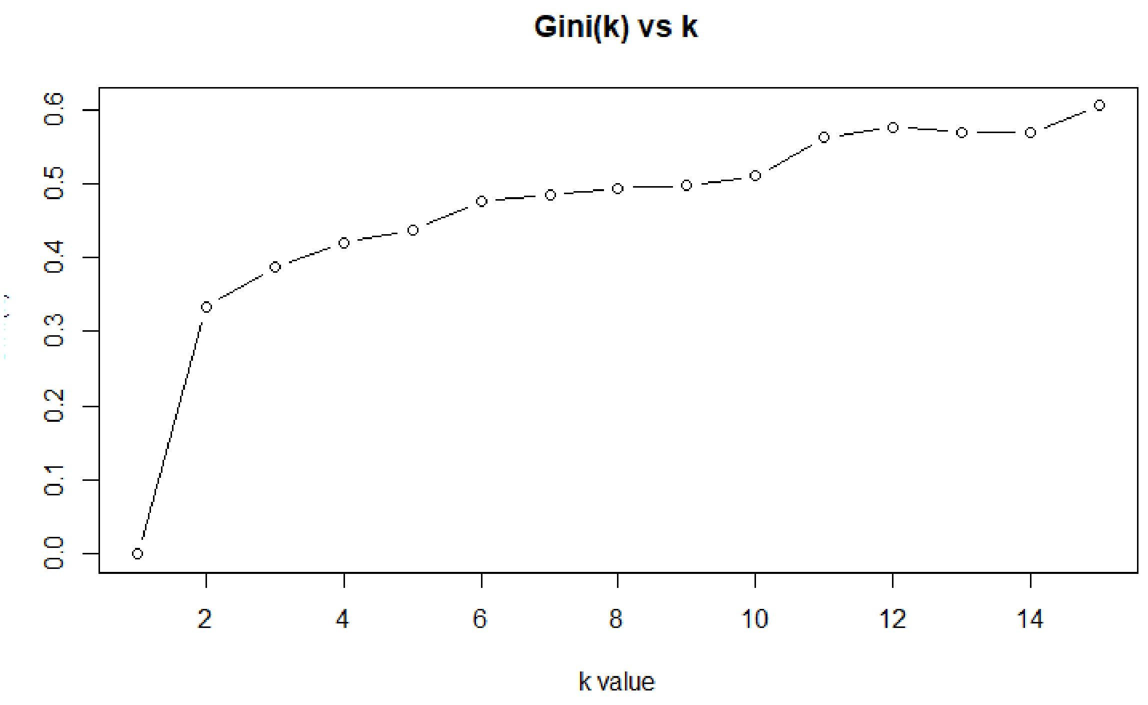
K=3 : 1.79 seconds

K=10 : 4.49 seconds

K=15 : 5.29 seconds

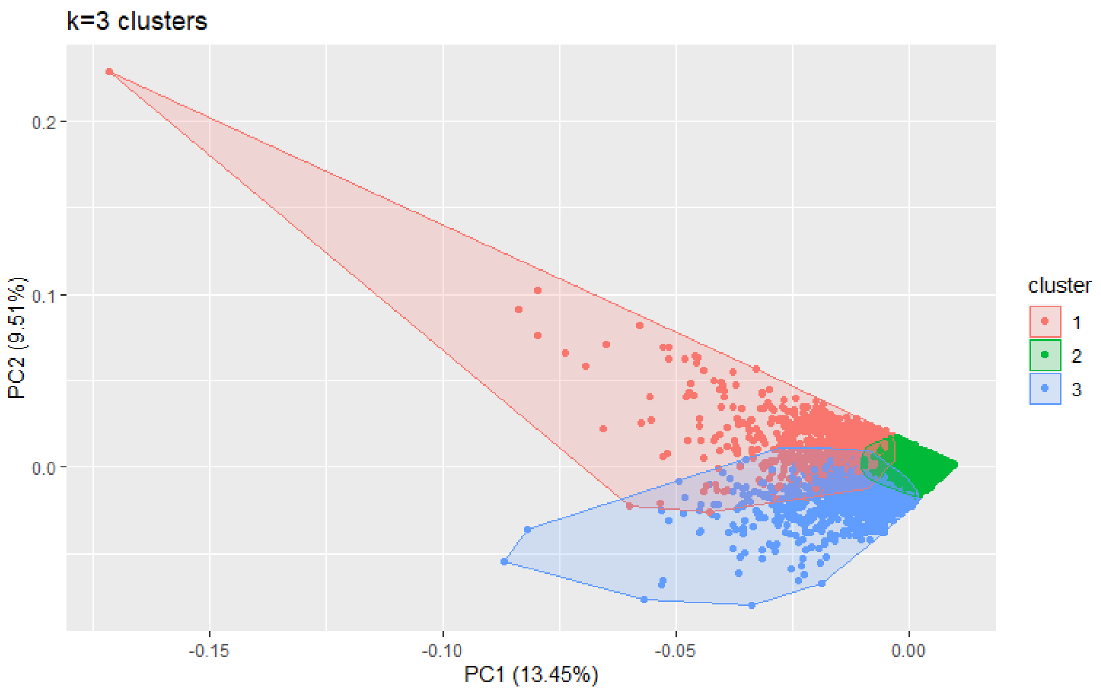


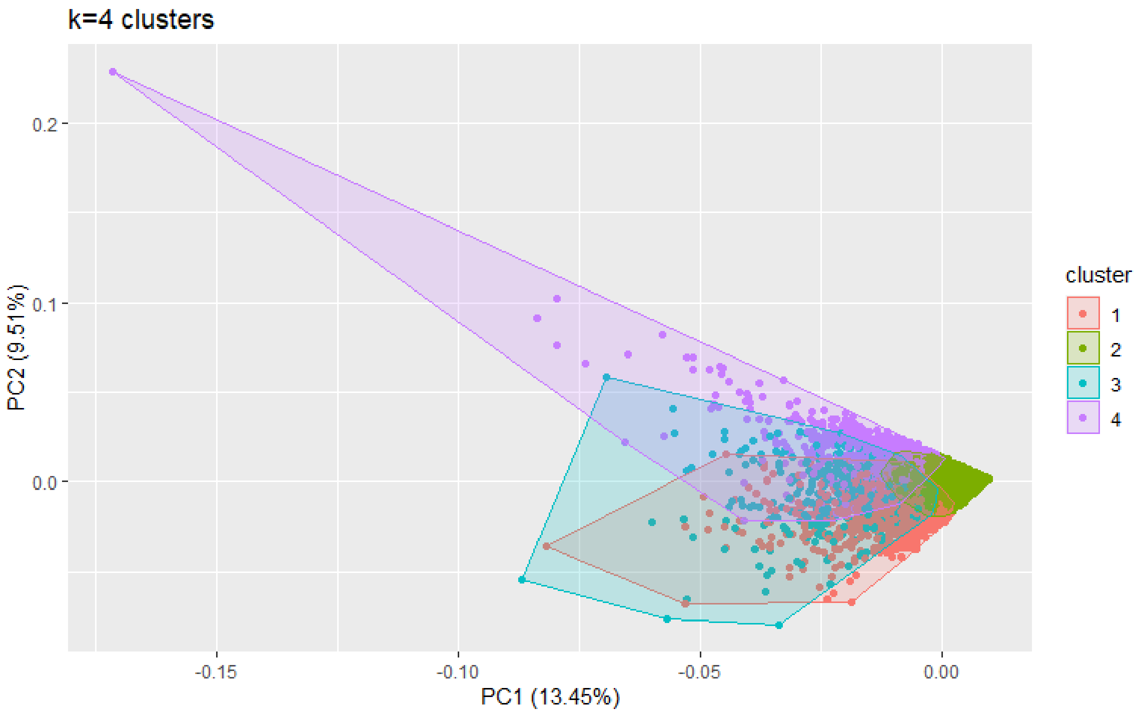
We calculated the performance for every k value in our chosen range and plotted it vs. the number of clusters. The "elbow rule" tells us that an ideal K value shows us a more horizontal angle on this graph. It is not obvious, but I believe after k=4, the difference in Y values decreases and the graph becomes more linear. Therefore, I think k=4 can be a good candidate for our "best K" value. Next, we look at the graph of Gini values vs. number of clusters:

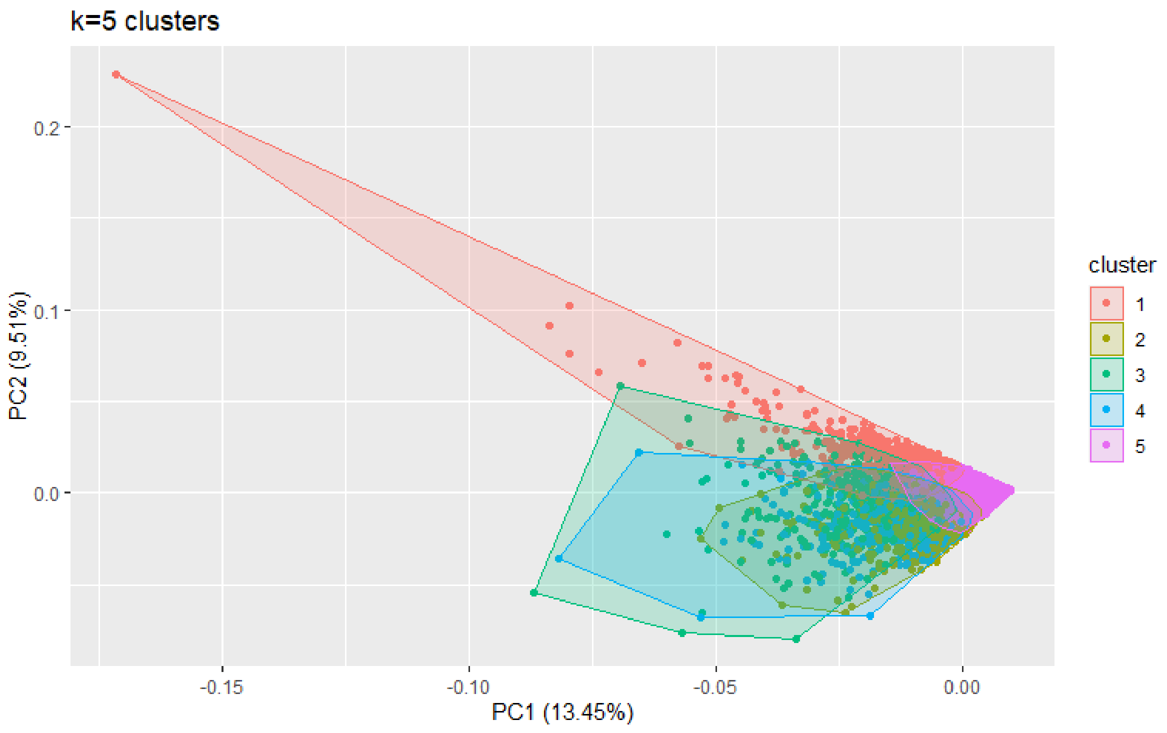


The Gini values measure the impurity of each set by taking into account the number of cases within each cluster. If a cluster only contains one case, it is said to have maximum purity and a Gini index of 0. If a cluster has the same number of all the cases, it is said to have maximum impurity and a Gini index of 1-1/s, s being the number of classes in the data set. In our case, since we have 5 classes, maximum impurity would mean a Gini value of 0.8. Based on the graph above, a good k can be in the range of 2 to 5. The Gini indexes for these k values stay in the 0.3 to 0.4 range.

Since it's not obvious which k gives the best results, we plotted k=3,4, and 5 to see how they separate the data:







The graphs above give us a better look into how the data is being separated and where the data points lie. For k=4 and k=5, there seems to be a large amount of overlapping between certain clusters and there's not a clear separation. However, for k=3, we can easily see separation with just a little bit of overlapping in the middle of the three clusters. Therefore, we will pick k=3 as our best K value.

Q3: Cluster characteristics

The kmeans() function in R allows us to look at certain characteristics of our model very easily. We made vectors and/or data frames for the centers, size, dispersion, and Gini values for each cluster:

Size:

Cluster 1=1015 cases

Cluster 2=7543 cases

Cluster 3=1315 cases

Centers:

```
            x1          x2          x3          x4           x5          x6           x7           x8          x9          x10
1  1.022745760 -0.09380205 -0.61810228 -0.87567816  0.840611179 -0.50791918  1.040300330 -0.081442064 -0.61669688 -0.867730625
2  0.009194564 -0.00414543 -0.02067963  0.01078385  0.003754823 -0.02567264  0.008135685 -0.006585418 -0.02367795  0.008697001
3 -0.842160871  0.09618103  0.59571121  0.61404617 -0.670374886  0.53930546 -0.849636735  0.100636883  0.61182518  0.619882212
           x11         x12         x13         x14          x15          x16         x17         x18         x19         x20
1  0.840042100 -0.51464940  0.57061828 -0.02905873 -0.326443146 -0.523023521  0.59338457 -0.36596484  0.59089828 -0.05473152
2  0.001777054 -0.02522837  0.01202445 -0.01076197  0.002651877 -0.001672405  0.01244548  0.03393543  0.01038732 -0.01033118
3 -0.658590910  0.54195189 -0.50941289  0.08416134  0.236757938  0.413295682 -0.52940047  0.08781699 -0.51567553  0.10150616
           x21          x22          x23        x24        x25        x26        x27        x28        x29        x30        x31
1 -0.319382157 -0.538703328  0.551337721 -0.3593820  1.1781727  1.2561556  1.1670026  1.2424313  0.9010694  0.9371711  0.8662377
2  0.001849994  0.004105274  0.009368501  0.0294590 -0.3540389 -0.3580047 -0.3515389 -0.3640217 -0.2784874 -0.2625337 -0.2740166
3  0.235907516  0.392256880 -0.479296113  0.1084133  1.1214221  1.0839781  1.1157039  1.1290860  0.9019353  0.7825574  0.9031757
         x32
1  0.9651734
2 -0.2815975
3  0.8702960
```

The centers data frame tells us the mean value for each feature in every cluster. For example, in cluster 1, the X1 feature mean is 1.02 while in cluster 2 it's 0.01 and in cluster 3 -0.84. This gives us an important insight into how the kmeans() function separated our 32 features into 3 clusters and how the features differ from each other.

Dispersion:

Cluster 1 = 97867.75

Cluster 2 = 62737.13

Cluster 3 = 112662.94

The dispersions are a measure of how far the data points are from its respective cluster center. As we can see from the resulting numbers, our dispersions are quite high. However, this makes sense since we only have 3 clusters and a lot of data points (>9500). Therefore, there will be many cases in which the data point is not close to its cluster center, increasing the dispersion.

Frequencies of each class in clusters:

Cluster 1:

Class 1	Class 2	Class 3	Class 4	Class 5
52	27	304	129	503

Cluster 2:

Class 1	Class 2	Class 3	Class 4	Class 5
2599	941	1531	695	1777

Cluster 3:

Class 1	Class 2	Class 3	Class 4	Class 5
90	30	262	263	670

We have to keep in mind that kmeans() is an unsupervised algorithm, so it makes sense that the clusters aren't separated based on class. We can see from the tables and the frequencies that cluster 2 has the majority of the data points. If we take a look at the plot provided before, the cluster 2 data points are very similar and would be difficult to separate using clustering. However, we can see that cluster 2 has a great majority of Class 1 and Class 2 cases, meaning it could be useful to classify between those two classes specifically. The other three classes are more balanced and we're going to need other techniques to classify them.

Gini values:

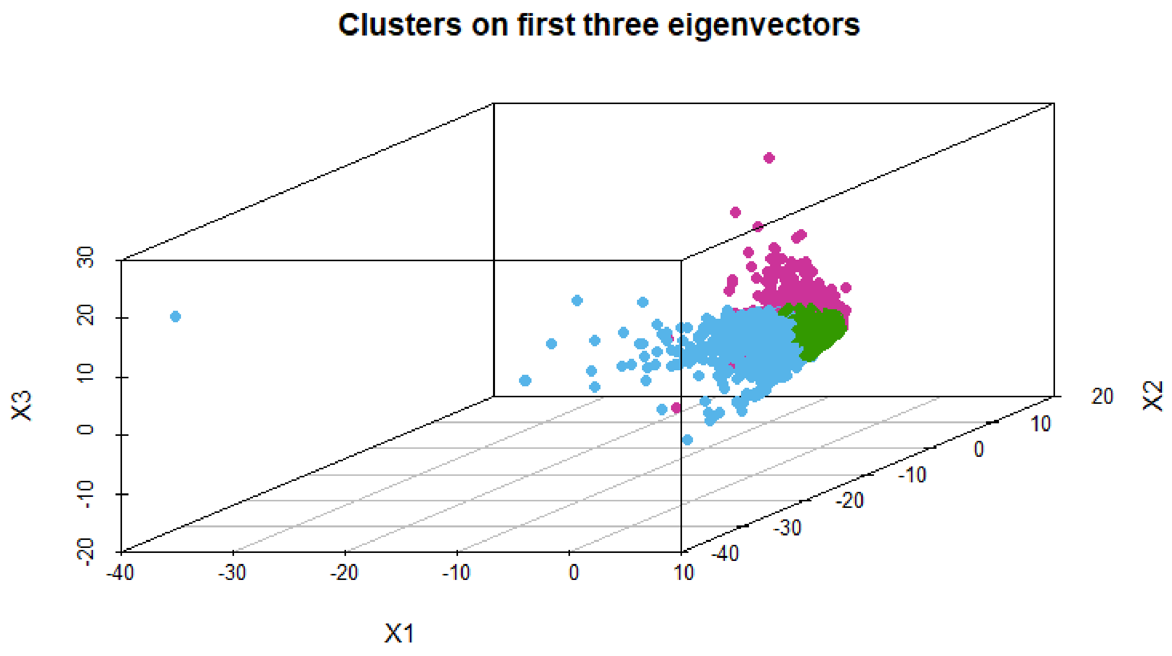
Cluster 1 = 0.64

Cluster 2 = 0.76

Cluster 3 = 0.66

The calculated Gini indexes reflect what we saw in the frequency tables above. Cluster 2 has high impurity because it has a large majority of the cases (>7000), while clusters 1 and 3 are more balanced throughout. Therefore, k-means clustering does not appear to be the best technique for this specific data set since the majority of the points are extremely similar.

Next, we projected the clusters on the first three eigenvectors calculated before:



The 3D projection shows promising results, even though it's hard to interpret since there are so many data points joined together. We know that the green (Cluster 2) dots are the grand majority. However, the plot suggests that using PCA analysis and dimension reduction could have improved the separation of the clusters.

Q4: TRAIN/TEST Sets

In order to apply the random forest technique to our data, we must first split it into TRAIN and TEST sets. This is done randomly and with an 80:20 split. We randomly select 80% of the cases for each class and combine the results to create a large TRAINSET. The other 20% of the data are also combined to create the TESTSET.

TRAINSET: 7896 cases

Class 1	Class 2	Class 3	Class 4	Class 5
2192	798	1677	869	2360

TESTSET: 1977 cases

Class 1	Class 2	Class 3	Class 4	Class 5
549	200	420	218	590

The data sets are fairly balanced, with a large number of every class in each of them. Therefore, there is no need to clone or under-sample.

Q5: Random forest technique

In the random forest technique, a random independent sample is gathered from our TRAINSET and assigned to one of a number of decision trees. The trees perform classification and splits the data sample into decision nodes. One branch contains the instances below a threshold for the mean response and the other branch contains the instances below the threshold. These branches are then further split based on the mean responses. There is one more final split done for regions that yield the lowest residual sum of squares (RSS). These final regions are known as the leaves. The decision tree provides a vote for each case to go into one of the classes, and this gives the resulting classification.

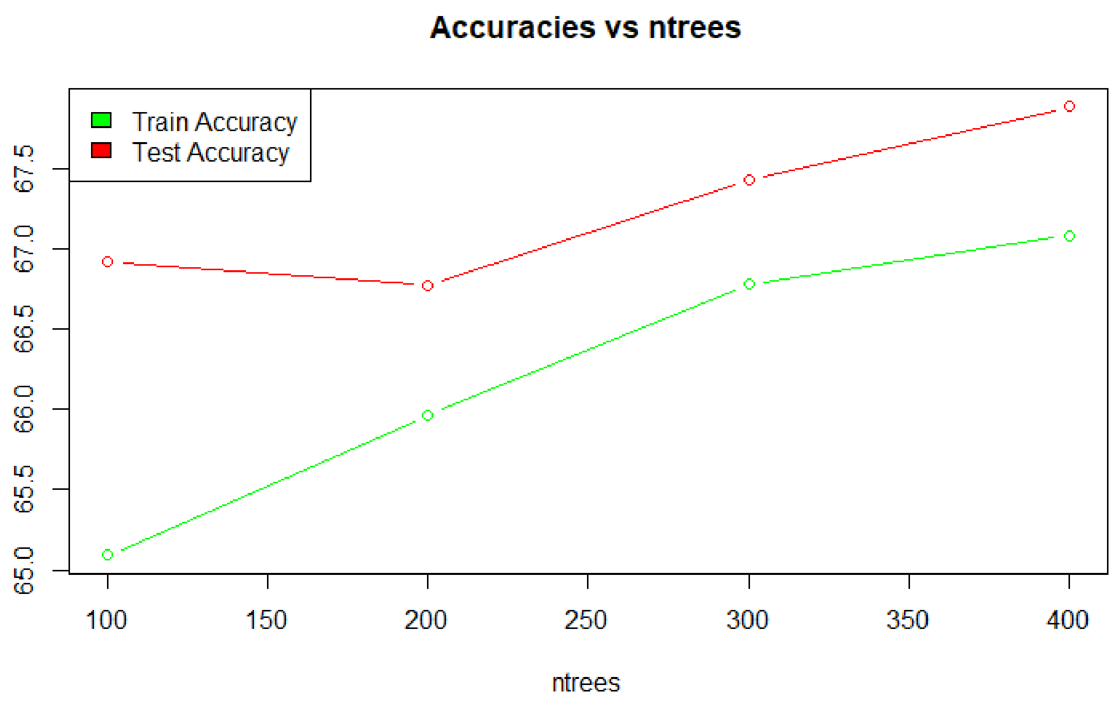
We use the randomForest() function in R to create a model based on the TRAINSET. We're training the model and use the out of bag (OOB) error rate and confusion matrix. This OOB evaluation gives us an estimate of the ability of the model to classify new data. This is why in our case, the TRAIN accuracy is sometimes lower than the TEST accuracy.

We then use the predict() function to classify the TESTSET. Predict() uses the trained model we made to classify each case into one of the 5 classes.

This is the function we used for the following problem:

randomForest(Class ~., data= TRAINSET, ntree=[100,200,300,400], mtry=sqrt(32))

We want to see which number of trees results in better performance, so we ran the model for 4 different values and analyzed the results.



The plot above shows us the OOB TRAIN accuracies and the TEST accuracies for different numbers of trees. Surprisingly, our TEST accuracies are slightly higher than our TRAIN accuracies. However, as mentioned before, this is because the OOB accuracies are sometimes pessimistic and underestimate the classification power of the model on new cases. Regardless, the difference in accuracies is only about 1%. We can see that the accuracy of the model only increases to ~ 68% as the number of trees goes up towards 400. Therefore, this seems like our best option. We also calculated the confusion matrices and computing time for the 4 TEST models:

NTREES = 100

Computing time = 3.31 seconds

Overall accuracy = 66.92%

```
          test.labels
rf.test100           D           H           P           R           S
         D 0.894353370 0.365000000 0.150000000 0.155963303 0.106779661
         H 0.009107468 0.370000000 0.016666667 0.013761468 0.001694915
         P 0.030965392 0.095000000 0.519047619 0.077981651 0.076271186
         R 0.012750455 0.040000000 0.030952381 0.380733945 0.040677966
         S 0.052823315 0.130000000 0.283333333 0.371559633 0.774576271
```

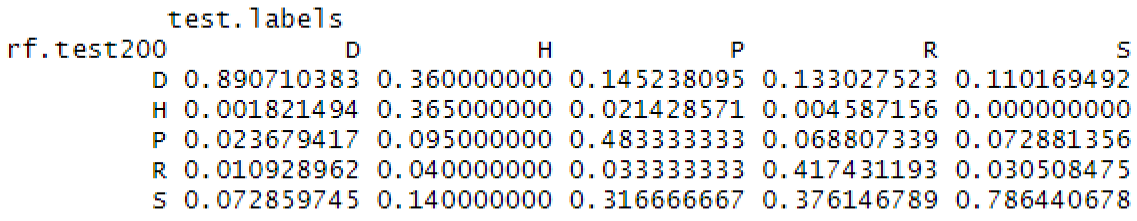
The confusion matrix gives us an insight into how the model classified each specifically. The diagonals give these values. For example, the [1,1] diagonal gives us the accuracy for Class D, [2,2] for Class H, and so on. It is evident from the confusion matrix that the model did a great job classifying Class D and Class S cases with accuracies of 0.89 and 0.77 respectively. Meanwhile, the other classes three lie in the 30%

and 50% range. We can also see that the model classified many Class H cases as Class D, and many Class R cases as Class S.

NTREES = 200

Computing time = 5.61 seconds

Overall accuracy = 66.77%



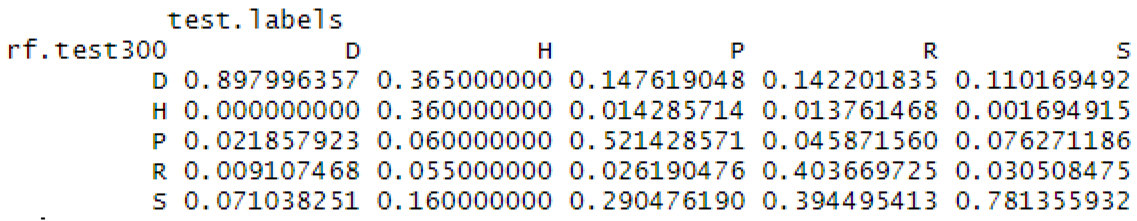
```
          test.labels
rf.test200           D           H           P           R           S
         D 0.890710383 0.360000000 0.145238095 0.133027523 0.110169492
         H 0.001821494 0.365000000 0.021428571 0.004587156 0.000000000
         P 0.023679417 0.095000000 0.483333333 0.068807339 0.072881356
         R 0.010928962 0.040000000 0.033333333 0.417431193 0.030508475
         S 0.072859745 0.140000000 0.316666667 0.376146789 0.786440678
```

For 200 trees, the confusion matrix looks very similar to the previous one. The only difference is that Classes P and H’s accuracies decreased by 2%, while Classes R and S’s accuracies increased by 1%. We start to see a pattern for our model.

NTREES = 300

Computing time = 9.33 seconds

Overall accuracy = 67.43%



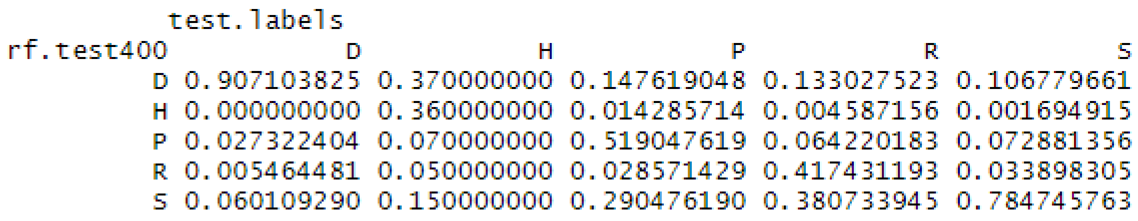
```
          test.labels
rf.test300           D           H           P           R           S
         D 0.897996357 0.365000000 0.147619048 0.142201835 0.110169492
         H 0.000000000 0.360000000 0.014285714 0.013761468 0.001694915
         P 0.021857923 0.060000000 0.521428571 0.045871560 0.076271186
         R 0.009107468 0.055000000 0.026190476 0.403669725 0.030508475
         S 0.071038251 0.160000000 0.290476190 0.394495413 0.781355932
```

For 300 trees, most of the accuracies remain the same as for 200 trees. However, Class P was classified with ~4% more accuracy.

NTREES = 400

Computing time = 11.66 seconds

Overall accuracy = 67.88%



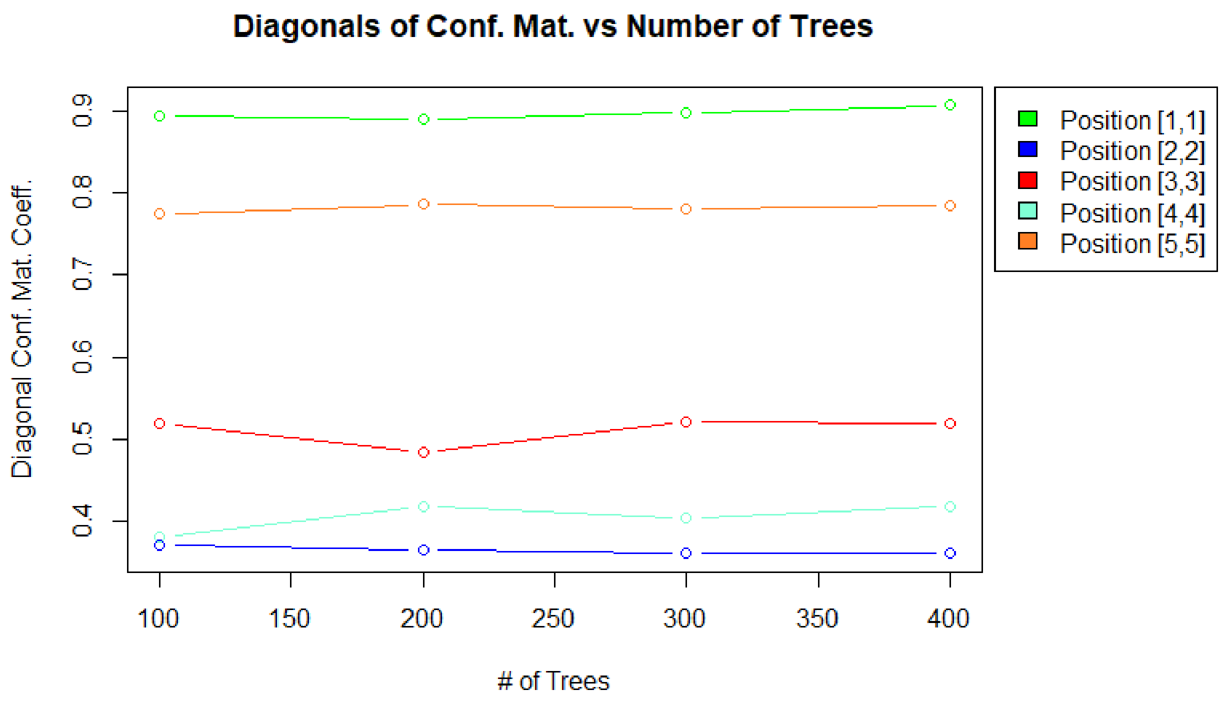
```
          test.labels
rf.test400           D           H           P           R           S
         D 0.907103825 0.370000000 0.147619048 0.133027523 0.106779661
         H 0.000000000 0.360000000 0.014285714 0.004587156 0.001694915
         P 0.027322404 0.070000000 0.519047619 0.064220183 0.072881356
         R 0.005464481 0.050000000 0.028571429 0.417431193 0.033898305
         S 0.060109290 0.150000000 0.290476190 0.380733945 0.784745763
```

Our final model with ntree=400 gives us the best classification accuracy for Class D and R, while remaining steady with the other 3 classes. We can see why it had the best overall accuracy.

Q6: Diagonal coefficients

In order to help us select our best value for number of trees, we look more closely at the diagonal coefficients of the confusion matrices shown above. The diagonal coefficients give us the specific accuracy of classification for each class. For example, if the [3,3] coefficient is 60%, it tells us that the model correctly predicted 60% of the Class 3 cases.

We plotted 5 curves, one for each diagonal coefficient, vs the number of trees:



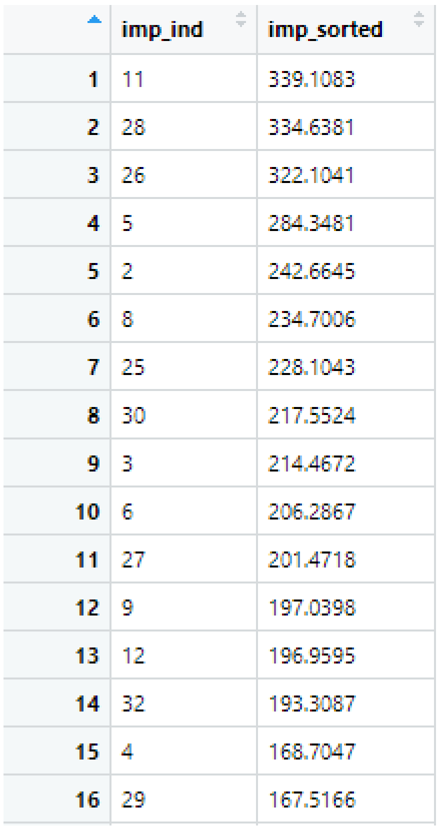
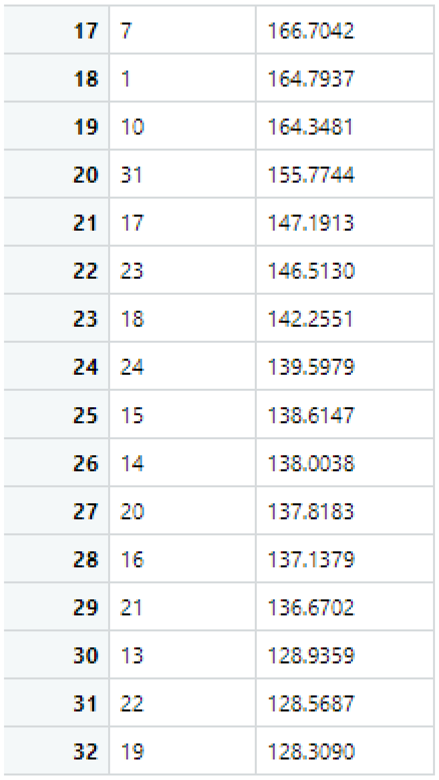
The plot shows us that there isn't much change between the diagonal coefficients when we increase the number of trees. There seems to be a small decrease for the [3,3] position at 200 trees, but most of the other diagonals increase a little bit as well. Other than that, they remain steady at their corresponding accuracies with the overall highest being at 400 trees. Therefore we select 400 as our BNT (best number of trees).

Q7: BestRF Classifier

Using our new BNT value of 400, and ntry=sqrt(32), we create our “BestRF” model. After training it using our TRAINSET, we will look at the importance values for each feature. These importance values are calculated to see which variables had the strongest predictive value. The features with high importance values have a significant impact on the results of the prediction. In addition, features with low importance values could be dropped from the model to make it quicker and simpler.

The randomForest function calculates these importance values by computing the prediction error with and without the variable of interest and comparing the difference. The difference is then averaged over all trees.

Importance features and values:

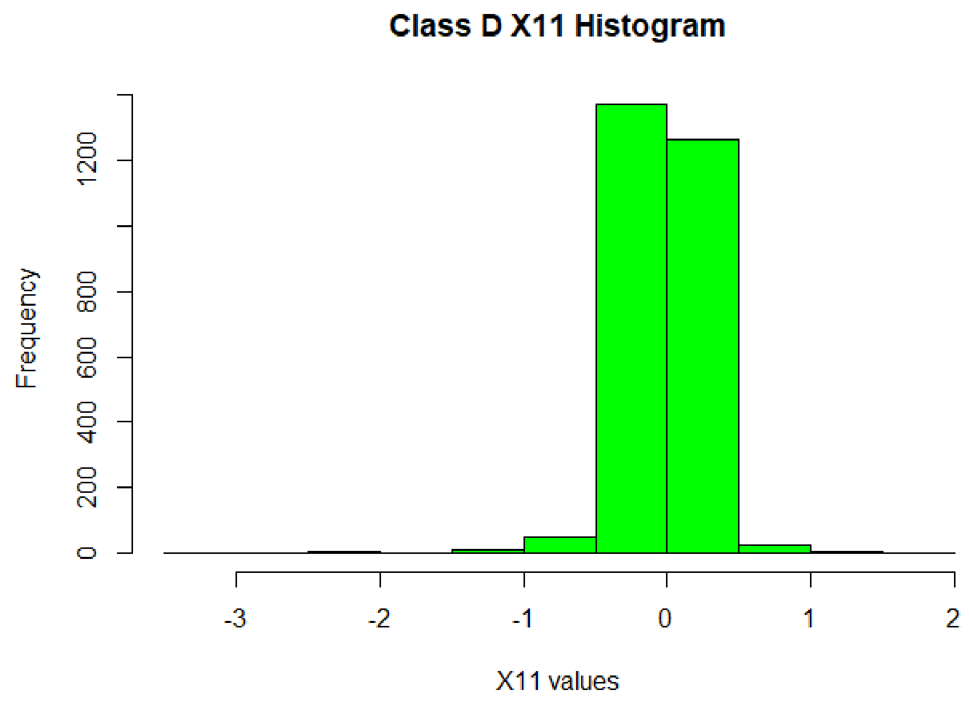



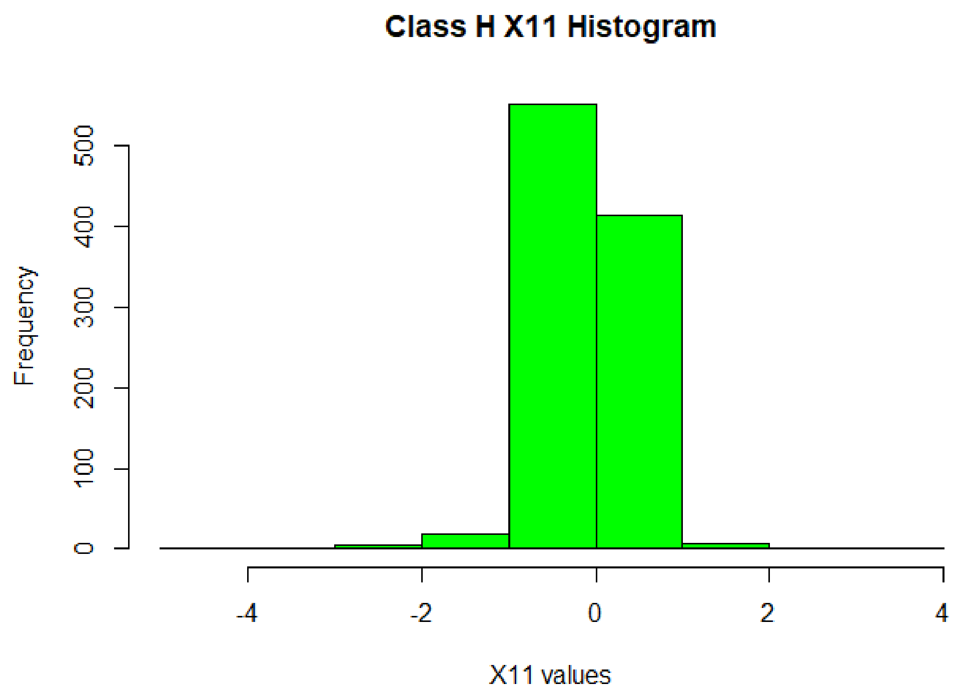
	imp_ind	imp_sorted
1	11	339.1083
2	28	334.6381
3	26	322.1041
4	5	284.3481
5	2	242.6645
6	8	234.7006
7	25	228.1043
8	30	217.5524
9	3	214.4672
10	6	206.2867
11	27	201.4718
12	9	197.0398
13	12	196.9595
14	32	193.3087
15	4	168.7047
16	29	167.5166
17	7	166.7042
18	1	164.7937
19	10	164.3481
20	31	155.7744
21	17	147.1913
22	23	146.5130
23	18	142.2551
24	24	139.5979
25	15	138.6147
26	14	138.0038
27	20	137.8183
28	16	137.1379
29	21	136.6702
30	13	128.9359
31	22	128.5687
32	19	128.3090

The feature with the highest importance value is X11 and the feature with the lowest is X19.

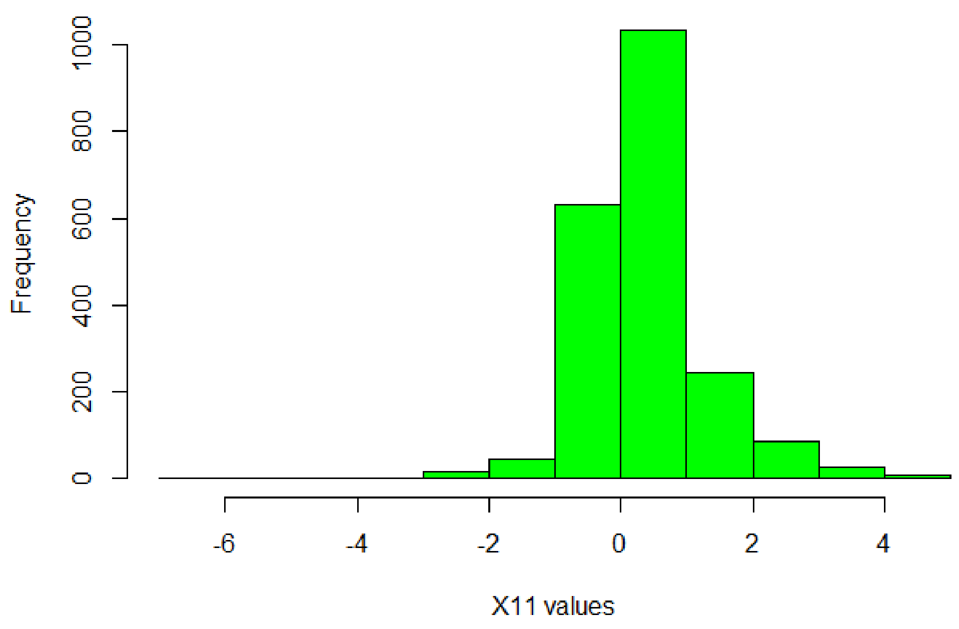
Q8: Histograms of important and non-important features

We will compare the histograms per class of the feature with the highest importance, X11. This will be done using the Kolmogorov-Smirnov test which tests whether two samples are significantly different from each other. In other words, we want to see if the feature X11 is significantly different in cases with different classes.

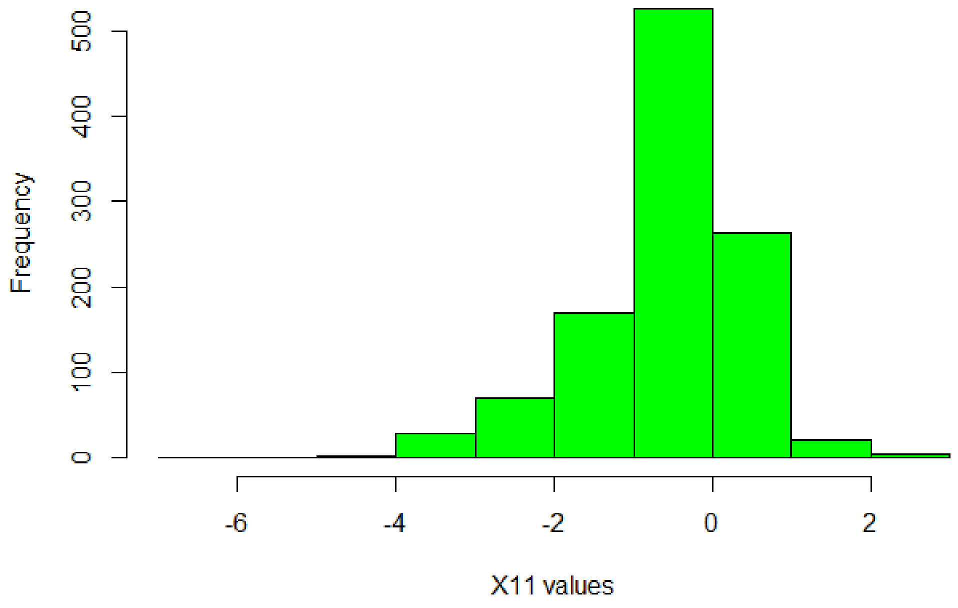


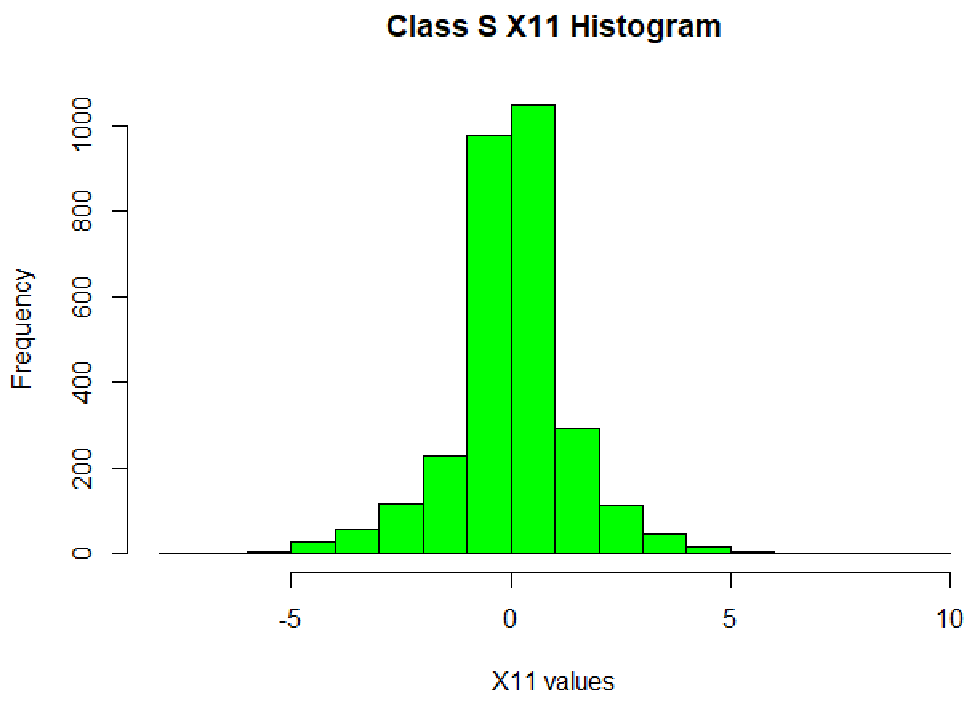


Class P X11 Histogram



Class R X11 Histogram





We run 10 different KS tests to compare every possible pair of histograms. Our hypothesis for every test is Ho: There is no difference between the two distributions and Ha: There is a difference between the two distributions. We set an alpha of 0.05, so if the returned p-value is less than that, we reject the null hypothesis.

P-values:

Class D vs. Class H = 5.55e-15

Class D vs. Class P ≈ 0

Class D vs Class R ≈ 0

Class D vs Class S ≈ 0

Class H vs Class P ≈ 0

Class H vs Class R ≈ 0

Class H vs Class S ≈ 0

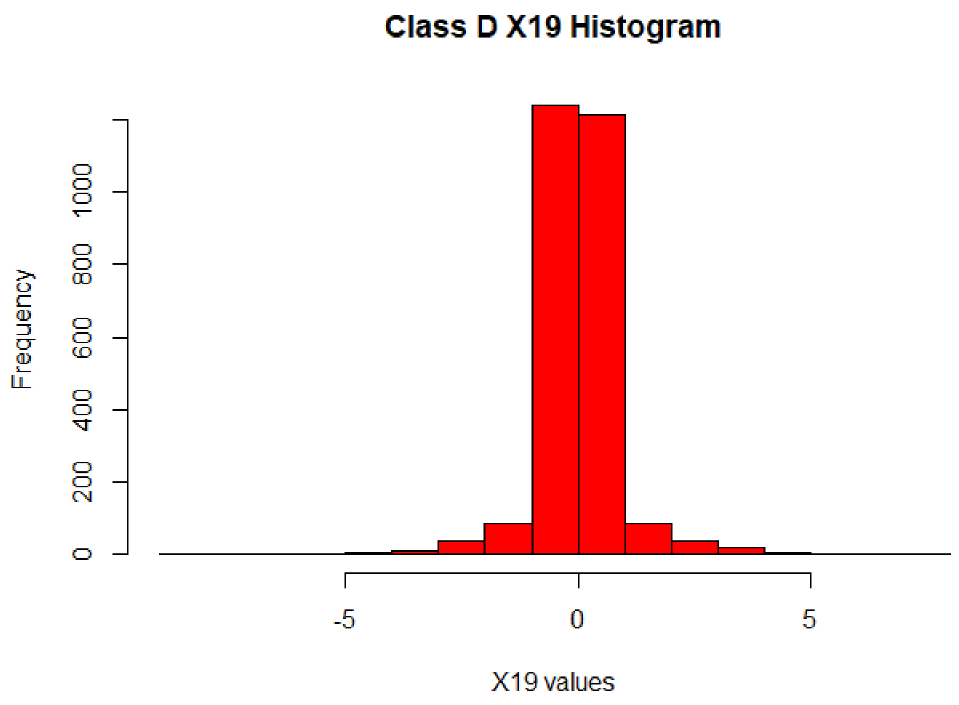
Class P vs Class R ≈ 0

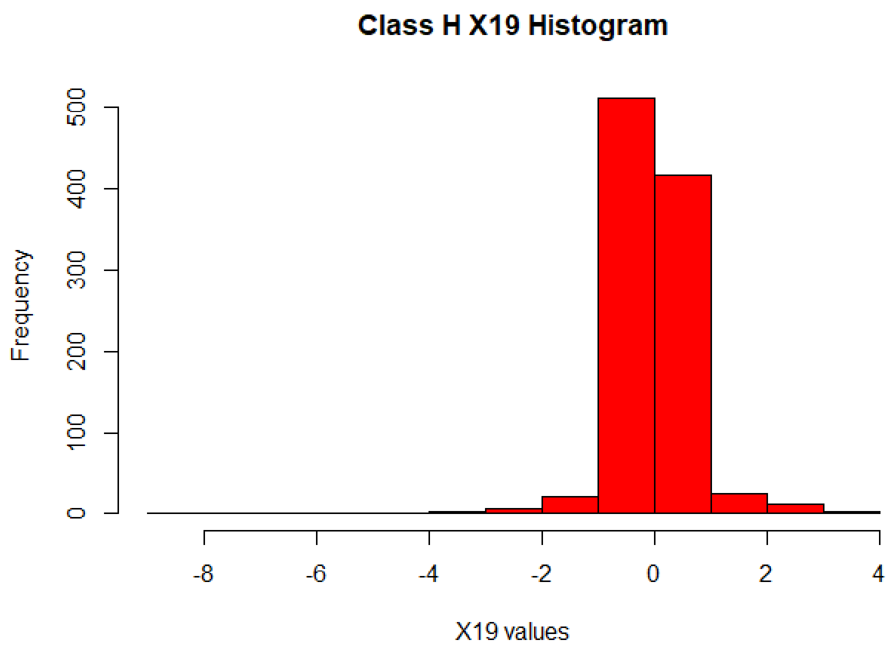
Class P vs Class S ≈ 0

Class R vs Class S ≈ 0

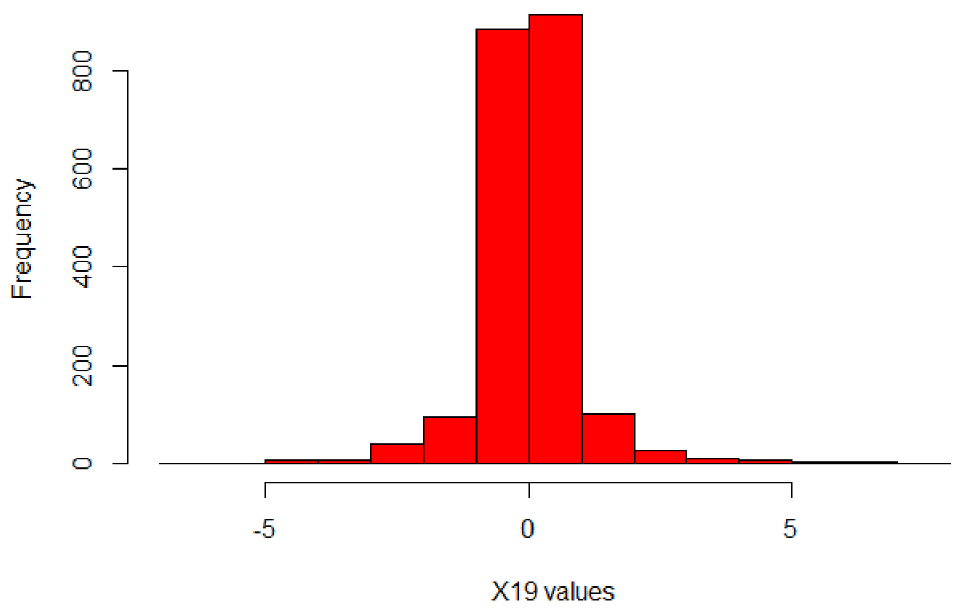
The KS tests resulted in a grand majority of p-values being extremely close to zero, meaning that all of the samples have different distributions. This is not too surprising since the histograms all have different ranges and shapes (skewed left, right, symmetric).

Next, we run the same KS tests comparing the high importance X11 feature to the low importance X19 feature. This will allow us to see if the importance truly holds great significance.

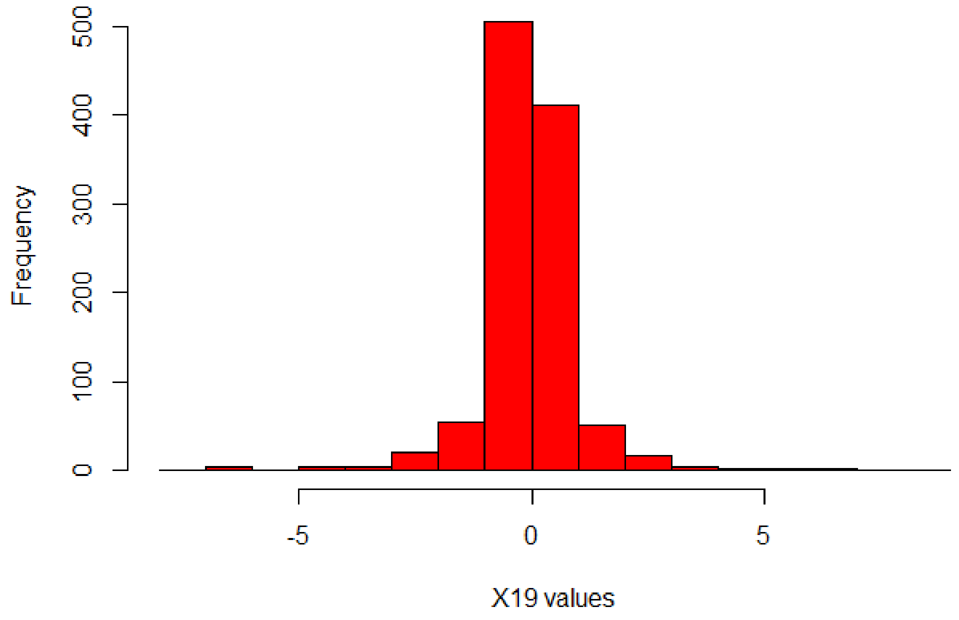


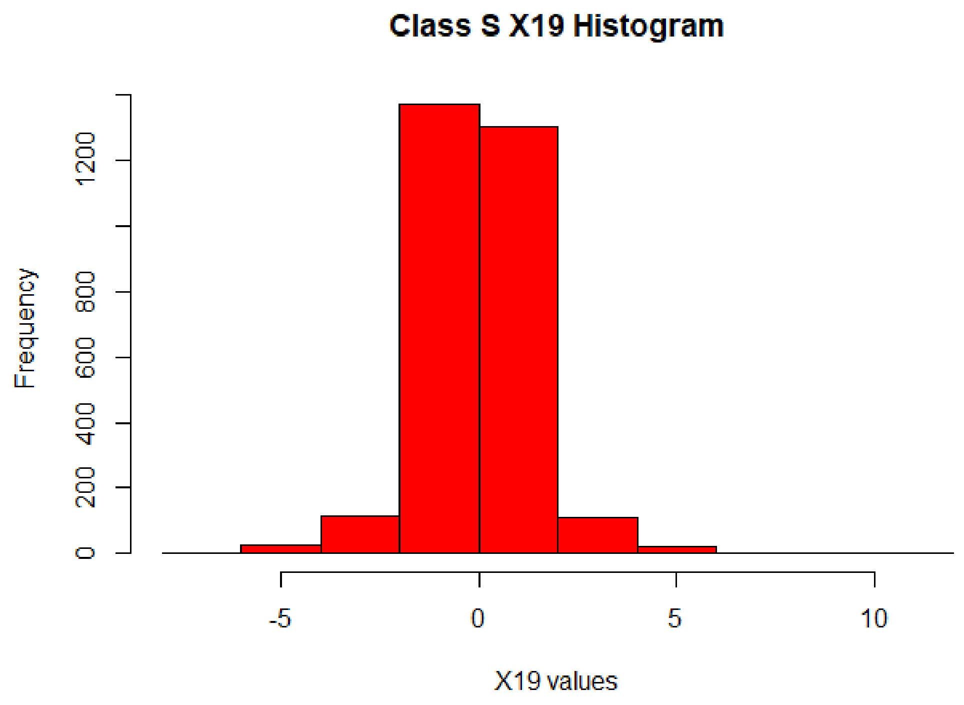


Class P X19 Histogram



Class R X19 Histogram





Once again, the hypothesis for these tests will be as follows: Ho= There is no difference between the two distributions and Ha= There is a difference between the two distributions. We set an alpha of 0.05, so if the returned p-value is less than that, we reject the null hypothesis.

P-values:

Class D X11 vs Class D X19 = 6.17e-13

Class H X11 vs Class H X19 = 0.0001

Class P X11 vs Class P X19 ≈ 0

Class R X11 vs Class R X19 ≈ 0

Class S X11 vs Class S X19 ≈ 0

The p-values for these 5 tests are all less than our alpha level of 0.05. Therefore, we reject the null hypothesis and assume that there is a difference between the distribution of the compared samples. This tells us that the importance values are significant in separating the features with the strongest predictive value.

Q9: Cluster with minimum Gini RF classifier

For this task, we choose the cluster with the lowest Gini value calculated in Q3. This was cluster 1 with a Gini index of 0.65. We want to train a new random forest classifier using only this cluster, so we must first split the data in cluster 1 into TRAIN and TEST sets using the same method as Q4. We take a look at the number of cases per class in Cluster 1:

Class 1	Class 2	Class 3	Class 4	Class 5
52	27	304	129	503

There are very few cases for Class 1 and Class 2, so we clone each of them in order to receive reliable results. We decide to clone Class 1 once and Class 2 4 times in order to reach a number of cases close to 100. In order to do this, we copy each row and multiply it by a random small number so that it changes by about 1% randomly. The formula we used is: x*(1+(runif(1)-0.5)/10000). These are the resulting numbers:

Class 1	Class 2	Class 3	Class 4	Class 5
104	108	304	129	503

Now, we can train a random forest model using a TRAIN set from just Cluster 1 and predict the classification of a TEST set:

OOB TRAIN accuracy = 75.71%

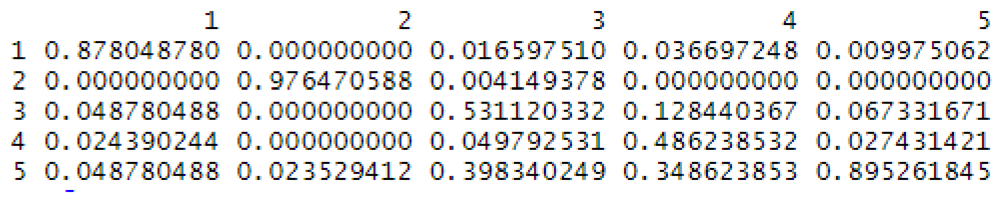
TEST accuracy = 79.57%

Q10: Confusion Matrix of Cluster 1 RF

We calculated the confusion matrix for the results of Cluster 1 RF, which only used the data in cluster 1. Our accuracy improved from 67.88% when using all the data, to 79.57% when using just cluster 2. There definitely seems to be an advantage to train a random forest separately for each cluster. Whether the other two clusters would have better results is yet to be seen, especially with the unbalanced number of classes in them.

True Values

Predicted



```
            1           2           3           4           5
1 0.878048780 0.000000000 0.016597510 0.036697248 0.009975062
2 0.000000000 0.976470588 0.004149378 0.000000000 0.000000000
3 0.048780488 0.000000000 0.531120332 0.128440367 0.067331671
4 0.024390244 0.000000000 0.049792531 0.486238532 0.027431421
5 0.048780488 0.023529412 0.398340249 0.348623853 0.895261845
```

The confusion matrix suggests to us that Cluster 1 together with the random forest model worked very well at classifying Class 1 (D), Class 2 (H), and Class 5 (S), with accuracies of 87.80%, 97.65%, and 89.53% respectively. However, the accuracies for Classes 3 and 4 seem unchanged when compared to our initial model. The results can be more trustworthy with more detailed cloning of classes, maybe using a different method since they were unbalanced in the beginning.

Q11: Linear SVM

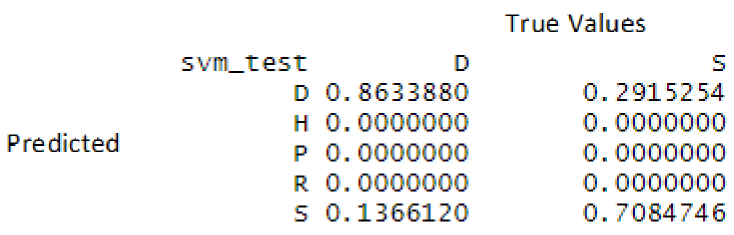
Our last task is to classify only two classes using a linear SVM model. The idea for SVM is to separate the data into the classes by creating a hyperplane which attempts to split the data points into groups. A hyperplane is a subset of an Euclidian space that divides the space into two disconnected parts.

In r, it works very similarly to the randomForest() function. First, we train the model using a TRAINSET, and then use the result to predict the TESTSET.

We randomly chose two classes using sample() and they are Class 1 (D) and Class 5 (S). We use the already defined trainsetD, testsetD, trainsetS, and testset S to create larger TRAINSET_SVM and TESTSET_SVM.

We use the svm(Class ~., data=TRAINSET_SVM) function to train our model and the predict() function to classify the TEST set. Here are our results:

Overall accuracy: 78.31%



```
                                True Values
            svm_test         D                S
                   D 0.8633880        0.2915254
                   H 0.0000000        0.0000000
Predicted          P 0.0000000        0.0000000
                   R 0.0000000        0.0000000
                   S 0.1366120        0.7084746
```

From the confusion matrix above, we can see that the SVM model was relatively accurate. It correctly predicted 86.34% of Class D cases while correctly predicting 70.85% of the Class S cases. The random forest did slightly better using the whole data set, however there are still ways to improve the SVM model making slight changes to the data and the arguments.

Conclusion:

After running a number of classification tests and models on our data, it is clear that this specific dataset is not the best for classification. However, in real-life applications, no data set will be perfect. We achieved decent accuracies when using only one cluster to train the data. Therefore, this could be a technique worth exploring. We could run three different models for each of the clusters and this could increase the accuracy. However, we also know that our data was extremely clumped together, so our clusters aren't as balanced as we would like.

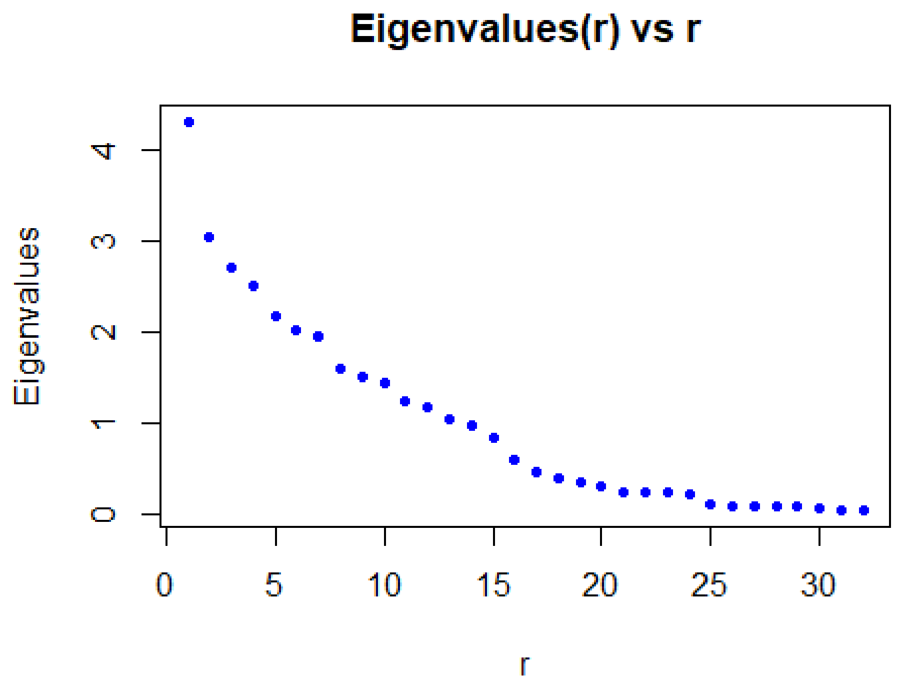
There is also the idea of changing the way randomForest() predicts ties. There are ways to make that process more specific and ultimately accurate than randomly selecting between the tying votes.

Overall, there are certain techniques that could be attempted in order to more accurately classify this data set into its correct classes. With more time, they can be explored and assessed.

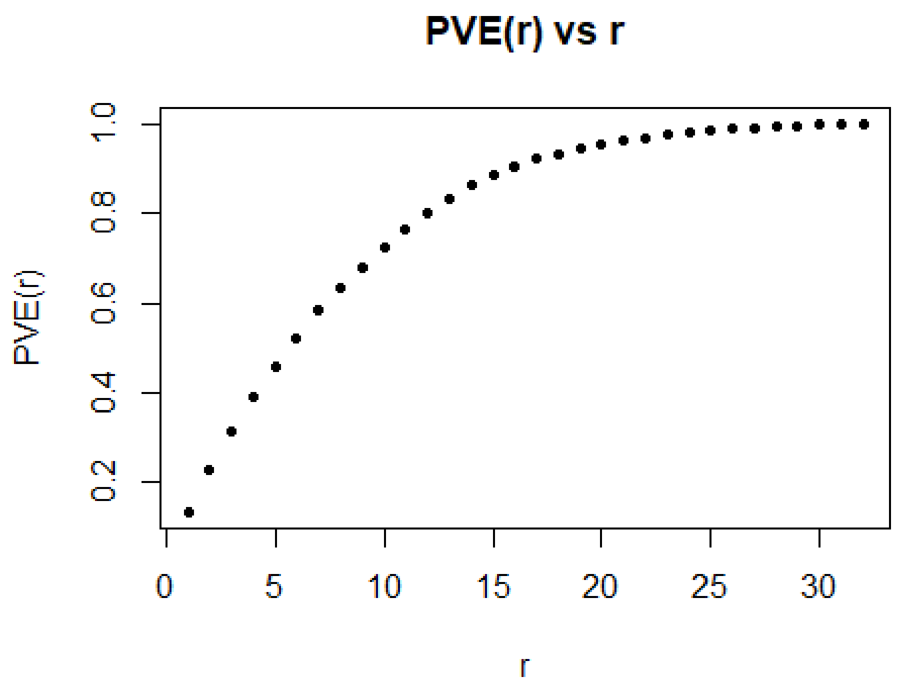
Code:

```
data1<-read.csv("C:\\Users\\rafoc\\Desktop\\a1_va3.csv",header=TRUE)
data2<-read.csv("C:\\Users\\rafoc\\Desktop\\a2_va3.csv",header=TRUE)
data3<-read.csv("C:\\Users\\rafoc\\Desktop\\a3_va3.csv",header=TRUE)
data4<-read.csv("C:\\Users\\rafoc\\Desktop\\b1_va3.csv",header=TRUE)
data5<-read.csv("C:\\Users\\rafoc\\Desktop\\b3_va3.csv",header=TRUE)
data6<-read.csv("C:\\Users\\rafoc\\Desktop\\c1_va3.csv",header=TRUE)
data7<-read.csv("C:\\Users\\rafoc\\Desktop\\c3_va3.csv",header=TRUE)

data<-rbind(data1,data2,data3,data4,data5,data6,data7)

sum(is.na(data))

## [1] 0

library(class)

## Warning: package 'class' was built under R version 3.6.3

classes<-data$Phase
table(classes)

## classes
##    D    H    P    R    S
## 2741  998 2097 1087 2950

#standardize data
SDATA<-data[,1:32]*10000
SDATA<-scale(SDATA)
SDATA<-data.frame(SDATA)
#SDATA$Phase<-data$Phase

# First 10 correlation values
cor(SDATA[1:10])

##              X1           X2          X3           X4          X5           X6
## X1   1.00000000  0.019970898 -0.09726804 -0.234864186  0.09890754 -0.038567277
## X2   0.01997090  1.000000000  0.06827546 -0.010015892  0.09961126  0.025230193
## X3  -0.09726804  0.068275459  1.00000000  0.054960108  0.04861029  0.141779759
## X4  -0.23486419 -0.010015892  0.05496011  1.000000000 -0.14281241  0.001778668
## X5   0.09890754  0.099611260  0.04861029 -0.142812406  1.00000000  0.075490898
## X6  -0.03856728  0.025230193  0.14177976  0.001778668  0.07549090  1.000000000
## X7   0.85490906  0.007856284 -0.09765338 -0.242669404  0.10705008 -0.039812266
## X8   0.01855043  0.930148622  0.04552120 -0.006973486  0.09754913  0.022404909
## X9  -0.10517827  0.087782454  0.88537550  0.062560114  0.04399675  0.112404605
## X10 -0.25077543 -0.029061138  0.04910312  0.860104565 -0.13243959 -0.001447376
##               X7           X8          X9          X10
## X1   0.854909060  0.018550425 -0.10517827 -0.250775426
## X2   0.007856284  0.930148622  0.08778245 -0.029061138
## X3  -0.097653377  0.045521199  0.88537550  0.049103125
## X4  -0.242669404 -0.006973486  0.06256011  0.860104565
## X5   0.107050078  0.097549130  0.04399675 -0.132439595
## X6  -0.039812266  0.022404909  0.11240461 -0.001447376
## X7   1.000000000 -0.008165162 -0.11608352 -0.256272745
## X8  -0.008165162  1.000000000  0.07762390 -0.023655603
## X9  -0.116083520  0.077623902  1.00000000  0.050334257
## X10 -0.256272745 -0.023655603  0.05033426  1.000000000

cor_sdata<- cor(SDATA)
eig<-eigen(cor_sdata)
```

```r
eigvalues<-eig$values
eigvectors<-eig$vectors

plot(seq(1,32,1),eigvalues,pch=20,col="blue",xlab='r',ylab='Eigenvalues',main='Eigenvalues(r)
vs r')
```

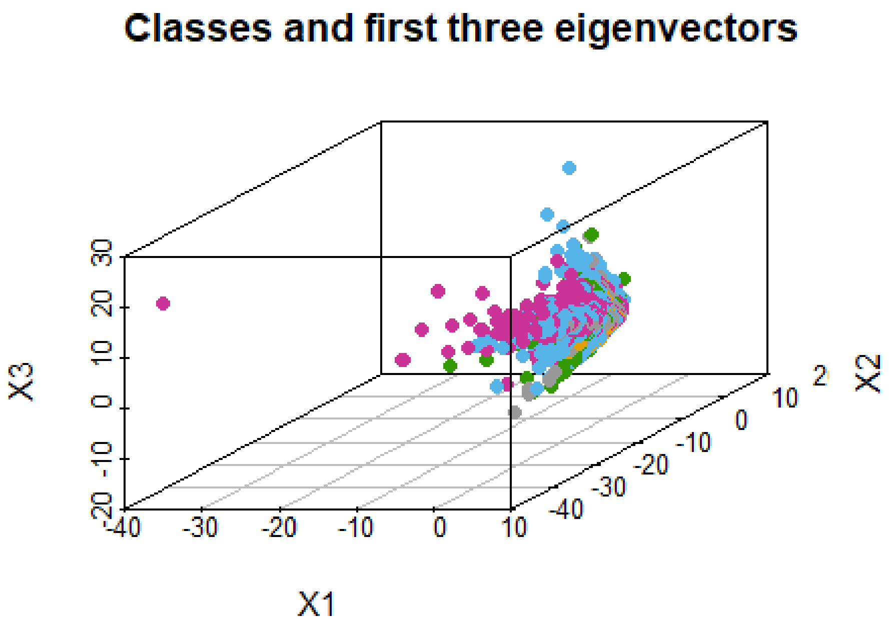


```r
pve<-rep(0,32)
sum_eig<-0
for (i in seq(1,32,1)){
  sum_eig<-sum_eig+eigvalues[i]
  pve[i]<-sum_eig/length(eigvalues)
}

plot(seq(1,32,1),pve,xlab='r',ylab='PVE(r)',main='PVE(r) vs r',pch=20)
```

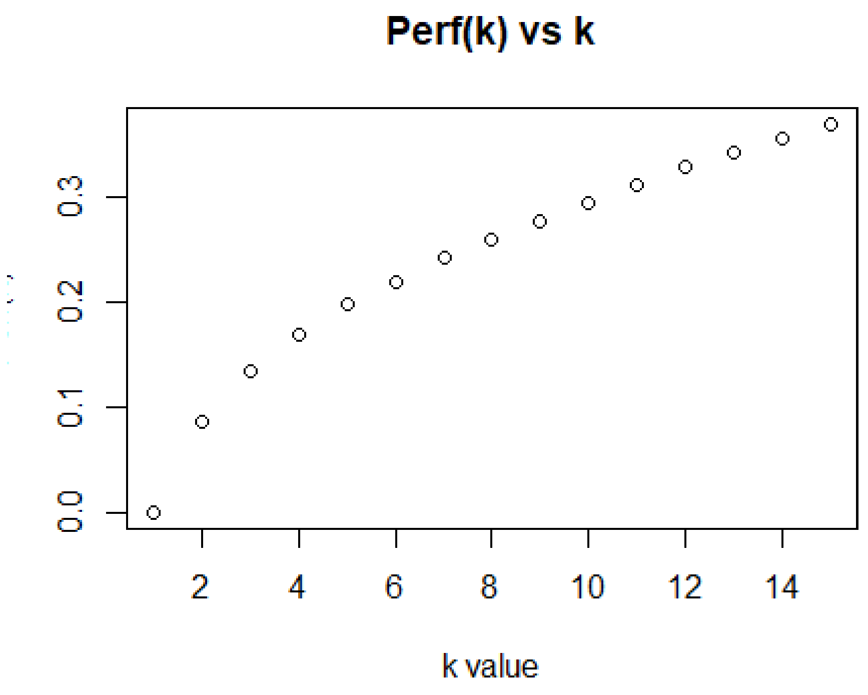


```r
# The first three eigenvalues
print(eigvalues[1:3])

## [1] 4.304586 3.042939 2.693231

# Their respective % total variance
print(pve[1:3])

## [1] 0.1345183 0.2296101 0.3137736

for (i in seq(1,32,1)){
  if (pve[i]>=0.95){
    print(i)
  } else{
    next
  }
  break
}

## [1] 20

# i = 20
eigvec_95<-eigvectors[,1:20]

eigvec_95<-data.matrix(eigvec_95)
w<-t(eigvec_95)

m.SDATA<-data.matrix(SDATA)
y.n<-m.SDATA %*% eigvec_95
y.n<-data.frame(y.n)

eigvec_3<-y.n[,1:3]
```

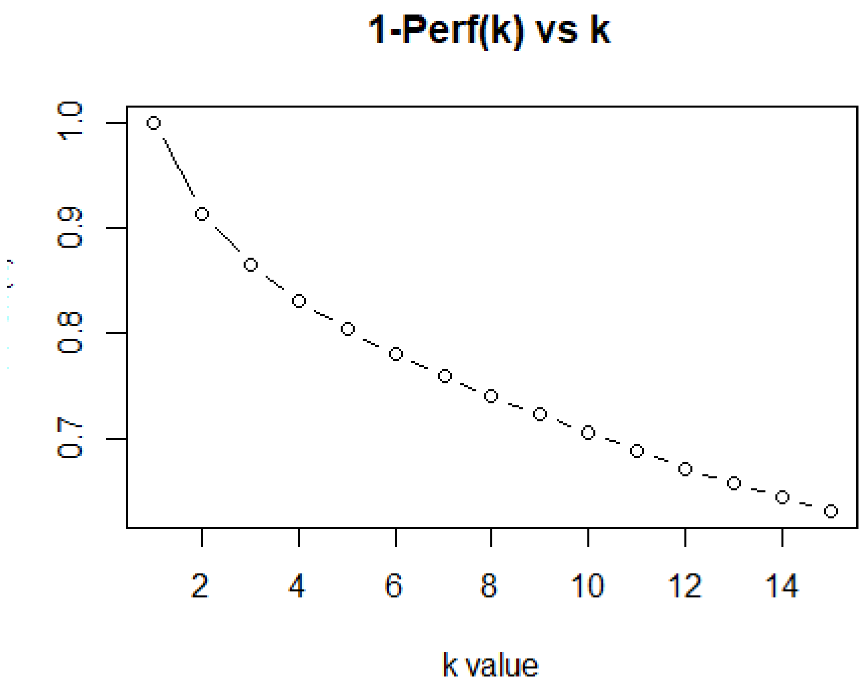
```r
library('scatterplot3d')

data$class[data$Phase == "D"]<-1
data$class[data$Phase == "H"]<-2
data$class[data$Phase == "P"]<-3
data$class[data$Phase == "R"]<-4
data$class[data$Phase == "S"]<-5

colors <- c("#999999", "#E69F00", "#56B4E9",'#339900','#CC3399')
colors <- colors[as.numeric(data$Phase)]
scatterplot3d(eigvec_3,pch=16,color=colors,main='Classes and first three eigenvectors')
```



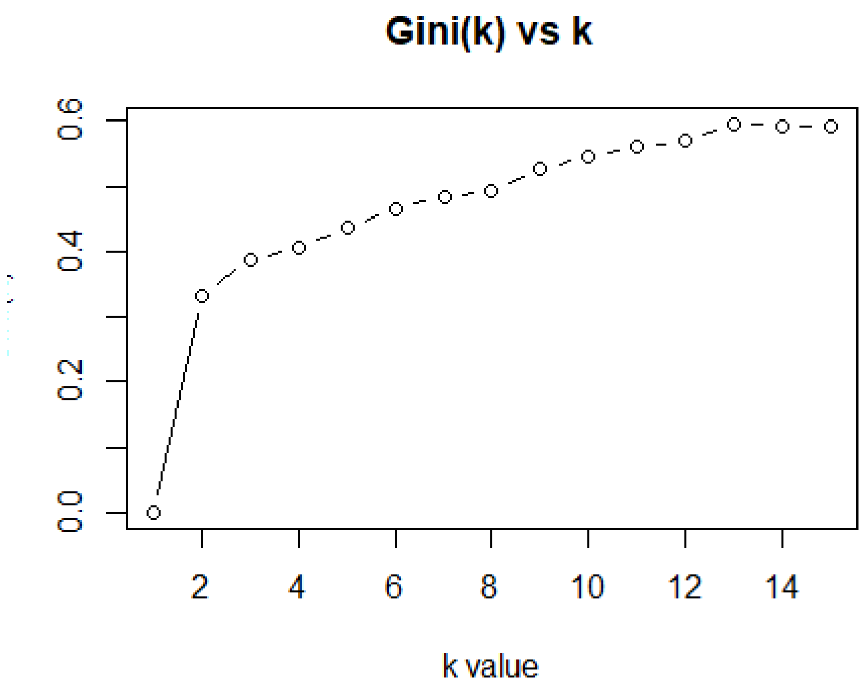
```r
#2
sws<-seq(1,15,1)
perf<-seq(1,15,1)
gini<-seq(1,15,1)
N<-nrow(SDATA)
k.range<-1:15
for (k in k.range){
  clust<-kmeans(SDATA,k,nstart=20,iter.max=30)
  sws[k]<-sum(clust$withinss)
  perf[k]<-1-(sws[k]/sws[1])
  gini[k]<-sum((clust$size/N)*(1-clust$size/N))
}

plot(k.range,perf,xlab='k value',ylab='Perf(k)',main='Perf(k) vs k')
```



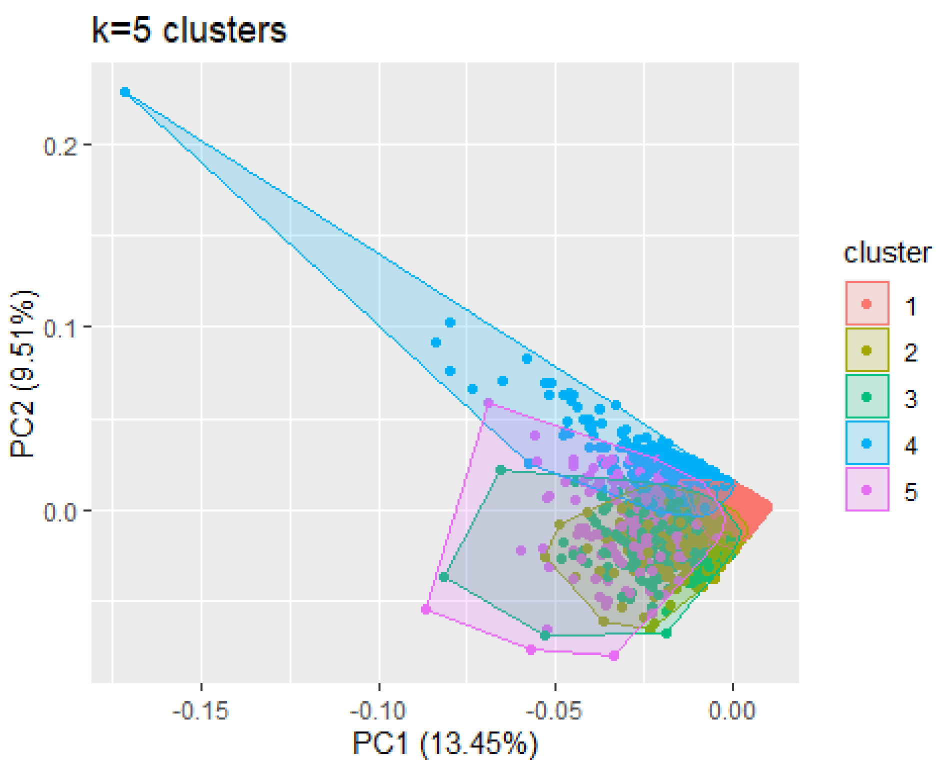
```
# made perf.2 to plot it vs the k values and apply 'elbow' rule
perf.2<-1-perf
plot(k.range,perf.2,xlab='k value',ylab='1-Perf(k)',type='b',main='1-Perf(k) vs k')
```



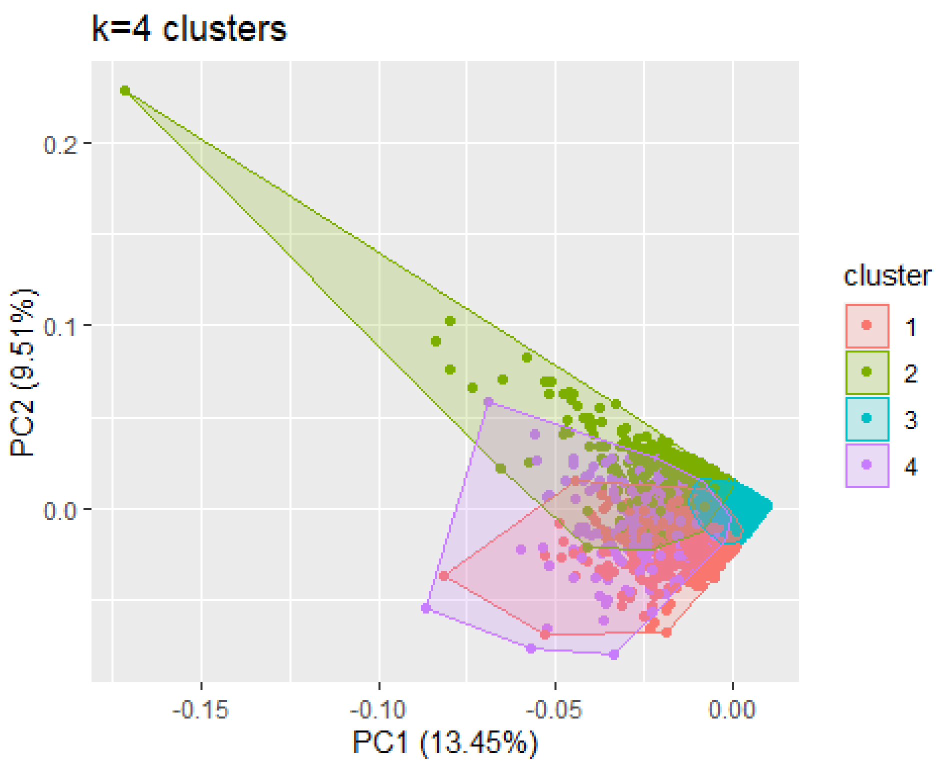
```
plot(k.range,gini,xlab='k value',ylab='Gini(k)',type='b',main='Gini(k) vs k')


k5 <- kmeans(SDATA, centers=5, nstart=50,iter.max=30)
k4 <- kmeans(SDATA, centers=4, nstart=50,iter.max=30)

s<-Sys.time()
k3 <- kmeans(SDATA, centers=3, nstart=50,iter.max=30)
e<-Sys.time()
d<-e-s
# 1.79 sec

s<-Sys.time()
k15<-kmeans(SDATA, centers=15, nstart=50,iter.max=30)

## Warning: Quick-TRANSfer stage steps exceeded maximum (= 493650)

e<-Sys.time()
d<-e-s
# 5.29 sec

s<-Sys.time()
k15<-kmeans(SDATA, centers=10, nstart=50,iter.max=30)
e<-Sys.time()
d<-e-s
d

## Time difference of 4.287873 secs

# 4.49 sec

#3

library('ggplot2')

## Warning: package 'ggplot2' was built under R version 3.6.3
```



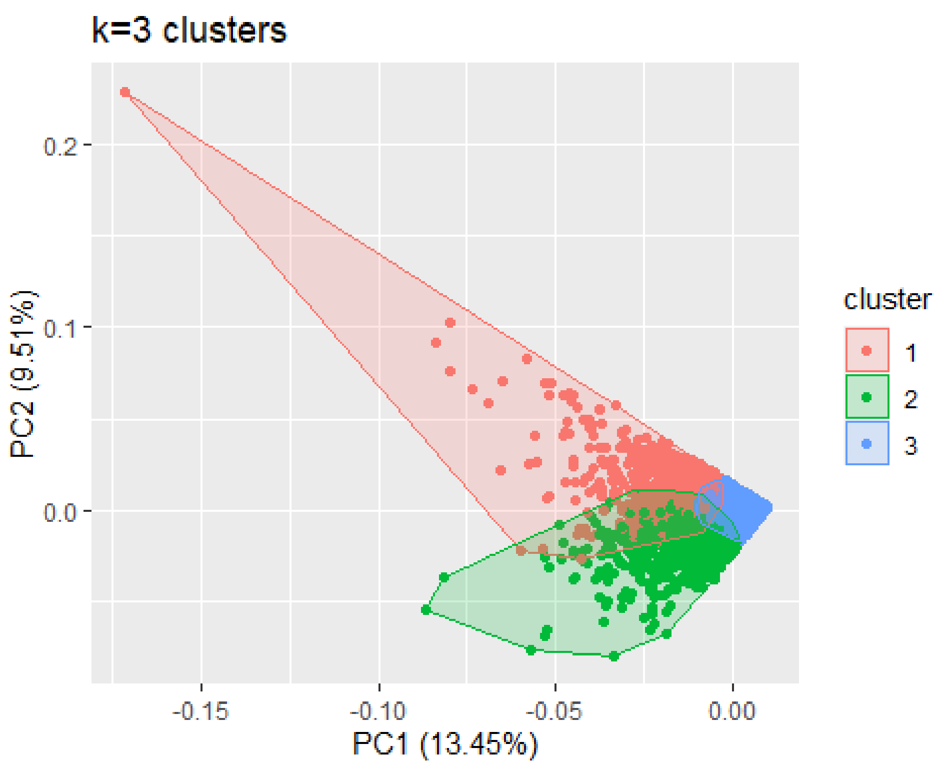
```
library('ggfortify')

## Warning: package 'ggfortify' was built under R version 3.6.3

autoplot(k5,SDATA,frame=TRUE,main='k=5 clusters')

## Warning: `select_()` is deprecated as of dplyr 0.7.0.
## Please use `select()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.

## Warning: `group_by_()` is deprecated as of dplyr 0.7.0.
## Please use `group_by()` instead.
## See vignette('programming') for more help
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```



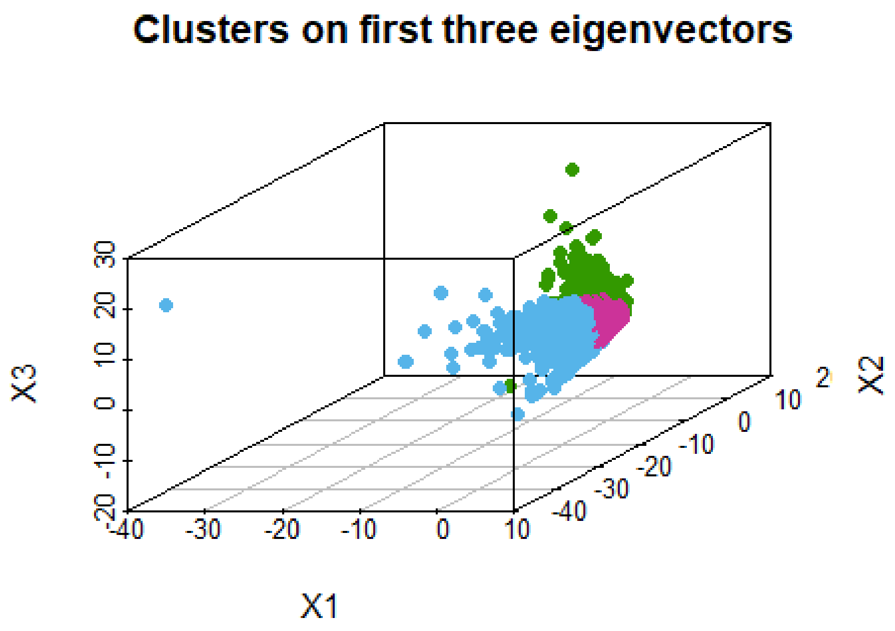
```r
autoplot(k4,SDATA,frame=TRUE,main='k=4 clusters')
```



```r
autoplot(k3,SDATA,frame=TRUE,main='k=3 clusters')
```



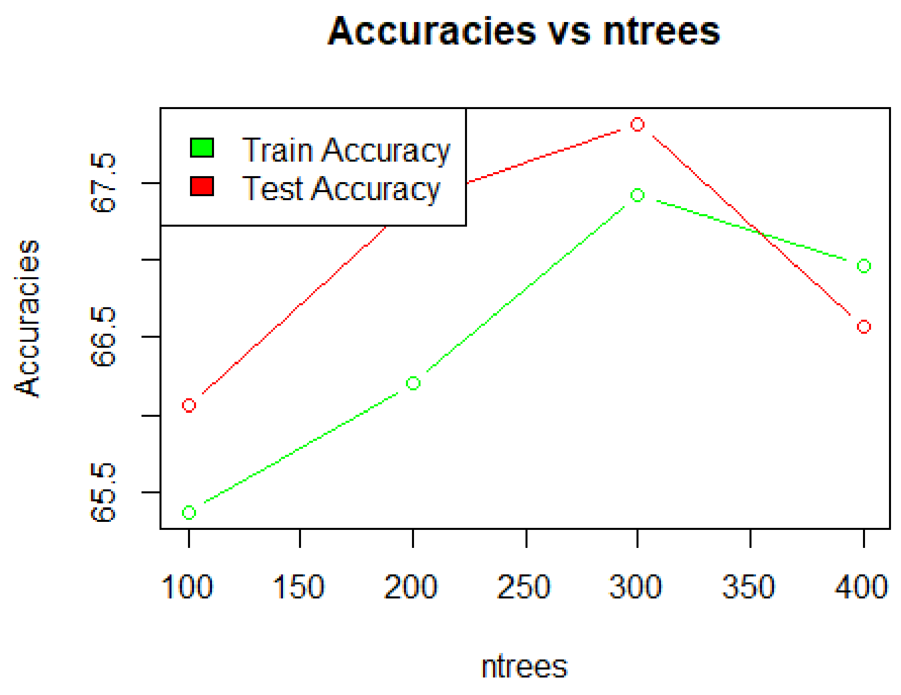
```
size_3<-k3$size
centers_3<-k3$centers
disp_3<-k3$withinss
library('plyr')

## Warning: package 'plyr' was built under R version 3.6.3

y<-1:9873
a<-1
b<-1
c<-1

clu.1<-vector()
clu.2<-vector()
clu.3<-vector()

for (i in y){
  if (k3$cluster[i]==1){
    clu.1[a]<-i
    a<-a+1
  } else if (k3$cluster[i]==2){
    clu.2[b]<-i
    b<-b+1
  } else if (k3$cluster[i]==3){
    clu.3[c]<-i
    c<-c+1
  }
}

data.clu1<-SDATA[clu.1,]
data.clu2<-SDATA[clu.2,]
data.clu3<-SDATA[clu.3,]
data.clu1<-cbind(data.clu1,data$class[clu.1])
data.clu2<-cbind(data.clu2,data$class[clu.2])
data.clu3<-cbind(data.clu3,data$class[clu.3])


tab1<-table(data.clu1$`data$class[clu.1]`)
tab2<-table(data.clu2$`data$class[clu.2]`)
tab3<-table(data.clu3$`data$class[clu.3]`)


n1<-nrow(data.clu1)
n2<-nrow(data.clu2)
n3<-nrow(data.clu3)


gini_2<-(2599/n2)*(1-(2599/n2))+(941/n2)*(1-(941/n2))+(1531/n2)*(1-(1531/n2))+(695/n2)*(1-(695
/n2))+(1777/n2)*(1-(1777/n2))
gini_1<-(52/n1)*(1-(52/n1))+(27/n1)*(1-(27/n1))+(304/n1)*(1-(304/n1))+(129/n1)*(1-(129/n1))+(5
03/n1)*(1-(503/n1))
gini_3<-(90/n3)*(1-(90/n3))+(30/n3)*(1-(30/n3))+(262/n3)*(1-(262/n3))+(263/n3)*(1-(263/n3))+(6
70/n3)*(1-(670/n3))


colors <- c("#56B4E9",'#339900','#CC3399')
colors <- colors[k3$cluster]
scatterplot3d(eigvec_3,pch=16,color=colors,main='Clusters on first three eigenvectors')
```



```
#4
data_m<-cbind(SDATA,data$Phase)
classD<-data_m[which(data_m$`data$Phase`=="D"),]
classH<-data_m[which(data_m$`data$Phase`=="H"),]
classP<-data_m[which(data_m$`data$Phase`=="P"),]
```

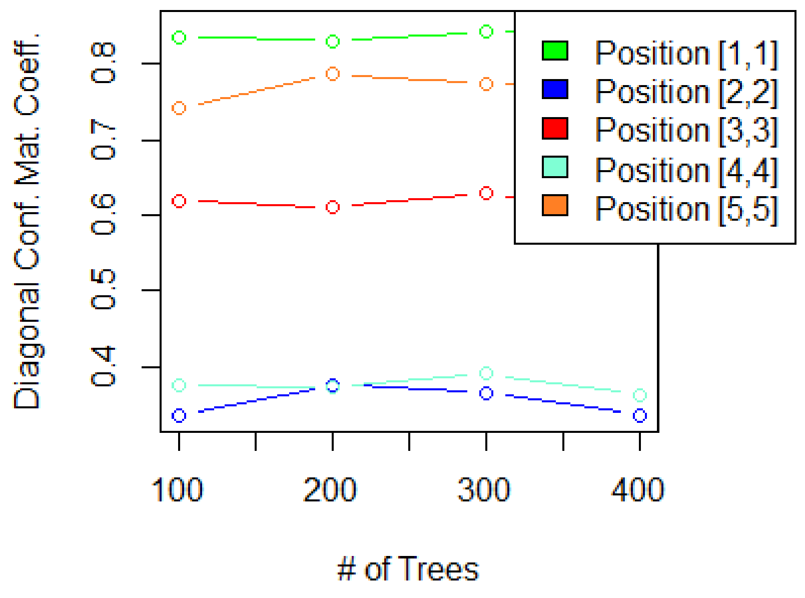
```
classR<-data_m[which(data_m$`data$Phase`=="R"),]
classS<-data_m[which(data_m$`data$Phase`=="S"),]

set.seed(18)
indexD1<-sample(1:nrow(classD),size=.8*nrow(classD))
trainD<-classD[indexD1,]
indexD2<-setdiff(1:nrow(classD),indexD1)
testD<-classD[indexD2,]

indexH1<-sample(1:nrow(classH),size=.8*nrow(classH))
trainH<-classH[indexH1,]
indexH2<-setdiff(1:nrow(classH),indexH1)
testH<-classH[indexH2,]

indexP1<-sample(1:nrow(classP),size=.8*nrow(classP))
trainP<-classP[indexP1,]
indexP2<-setdiff(1:nrow(classP),indexP1)
testP<-classP[indexP2,]

indexR1<-sample(1:nrow(classR),size=.8*nrow(classR))
trainR<-classR[indexR1,]
indexR2<-setdiff(1:nrow(classR),indexR1)
testR<-classR[indexR2,]

indexS1<-sample(1:nrow(classS),size=.8*nrow(classS))
trainS<-classS[indexS1,]
indexS2<-setdiff(1:nrow(classS),indexS1)
testS<-classS[indexS2,]

TRAINSET<-rbind(trainD,trainH,trainP,trainR,trainS)
TESTSET<-rbind(testD,testH,testP,testR,testS)

train.labels<-TRAINSET$`data$Phase`
test.labels<-TESTSET$`data$Phase`

names(TRAINSET)[names(TRAINSET) == 'data$Phase'] <- 'Class'
names(TESTSET)[names(TESTSET) == 'data$Phase'] <- 'Class'

#5
library(randomForest)

## Warning: package 'randomForest' was built under R version 3.6.3

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin

library('caret')

## Warning: package 'caret' was built under R version 3.6.3

## Loading required package: lattice
```

```r
rf100 <- randomForest(Class ~ ., data=TRAINSET, ntree=100, mtry=sqrt(32))
rf200 <- randomForest(Class ~ ., data=TRAINSET, ntree=200, mtry=sqrt(32))
rf300 <- randomForest(Class ~ ., data=TRAINSET, ntree=300, mtry=sqrt(32))
rf400 <- randomForest(Class ~ ., data=TRAINSET, ntree=400, mtry=sqrt(32))

rf.test100 <- predict(rf100, TESTSET)
rf.test200 <- predict(rf200, TESTSET)
rf.test300 <- predict(rf300, TESTSET)
rf.test400 <- predict(rf400, TESTSET)

conf.mat.train100 <- table(rf100$predicted, train.labels)
conf.mat.test100 <- table(rf.test100, test.labels)

conf.mat.train200 <- table(rf200$predicted, train.labels)
conf.mat.test200 <- table(rf.test200, test.labels)

conf.mat.train300 <- table(rf300$predicted, train.labels)
conf.mat.test300 <- table(rf.test300, test.labels)

conf.mat.train400 <- table(rf400$predicted, train.labels)
conf.mat.test400 <- table(rf.test400, test.labels)

conf100<-prop.table(conf.mat.test100,margin=2)
conf200<-prop.table(conf.mat.test200,margin=2)
conf300<-prop.table(conf.mat.test300,margin=2)
conf400<-prop.table(conf.mat.test400,margin=2)

accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}

acc.train.100<-accuracy(rf100$confusion)
acc.test.100<-accuracy(conf.mat.test100)

acc.train.200<-accuracy(rf200$confusion)
acc.test.200<-accuracy(conf.mat.test200)

acc.train.300<-accuracy(rf300$confusion)
acc.test.300<-accuracy(conf.mat.test300)

acc.train.400<-accuracy(rf400$confusion)
acc.test.400<-accuracy(conf.mat.test400)

acc.train<-c(acc.train.100,acc.train.200,acc.train.300,acc.train.400)
acc.test<-c(acc.test.100,acc.test.200,acc.test.300,acc.test.400)

ntrees<- c(100,200,300,400)
par(mar=c(5.1,4.1,4.1,2.1))
plot(ntrees,acc.train,col='green',type='b',ylim = range(c(acc.train, acc.test)),ylab='Accuraci
es',main='Accuracies vs ntrees')
lines(ntrees,acc.test,col='red',type='b')
legend("topleft", c("Train Accuracy", "Test Accuracy"), fill = c("green",'red'))
```

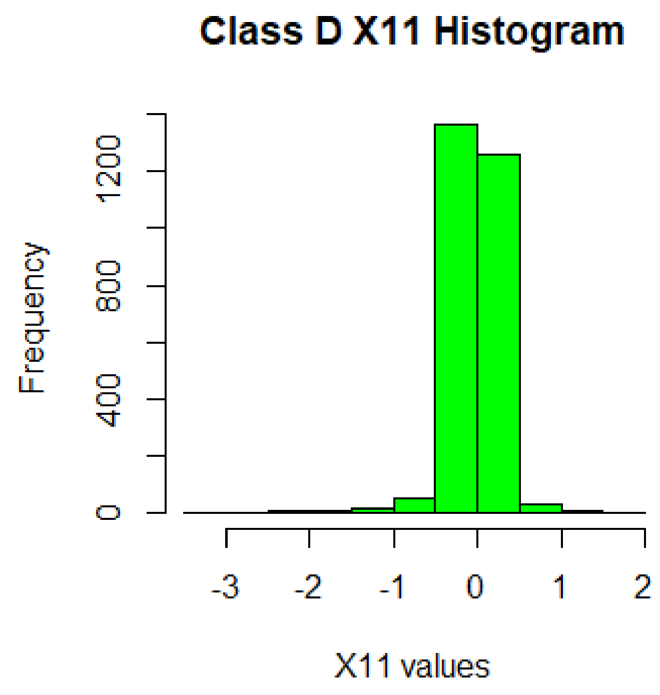


```
#6

diag1 <- c(conf100[1,1], conf200[1,1], conf300[1,1], conf400[1,1])
diag2 <- c(conf100[2,2], conf200[2,2], conf300[2,2], conf400[2,2])
diag3 <- c(conf100[3,3], conf200[3,3], conf300[3,3], conf400[3,3])
diag4 <- c(conf100[4,4], conf200[4,4], conf300[4,4], conf400[4,4])
diag5 <- c(conf100[5,5], conf200[5,5], conf300[5,5], conf400[5,5])

par(mar=c(5.1, 4.1, 4.1, 8.1), xpd=TRUE)
plot(ntrees, diag1, type = "b", ylim = range(c(diag1, diag2,diag3,diag4,diag5)), col = "green"
, xlab = "# of Trees", ylab = "Diagonal Conf. Mat. Coeff.", main = "Diagonals of Conf. Mat. vs
Number of Trees")
lines(ntrees, diag2, type = "b", col = "blue")
lines(ntrees, diag3, type="b", col = "red")
lines(ntrees, diag4, type="b", col = "aquamarine")
lines(ntrees, diag5, type="b", col = "chocolate1")
legend("topright",inset=c(-0.275,0), c("Position [1,1]", "Position [2,2]", "Position [3,3]","P
osition [4,4]",'Position [5,5]'), fill = c("green", "blue", "red",'aquamarine','chocolate1'))
```

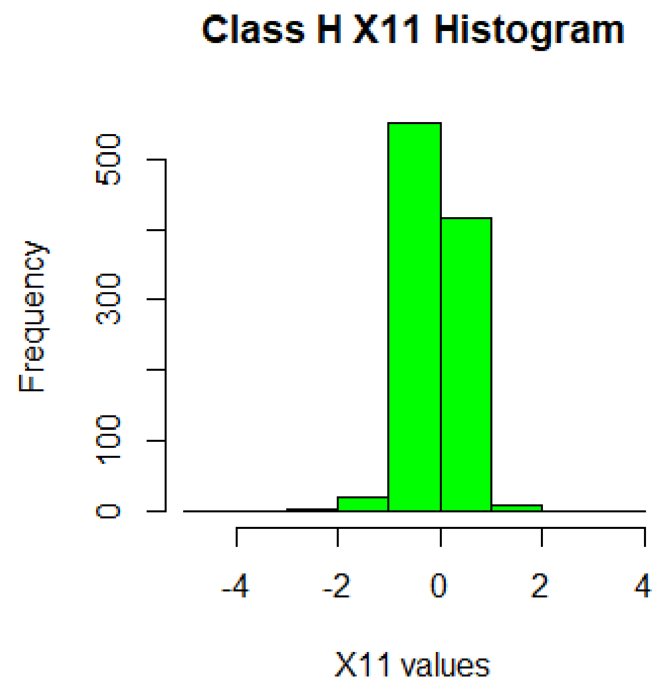
Diagonals of Conf. Mat. vs Number of Trees



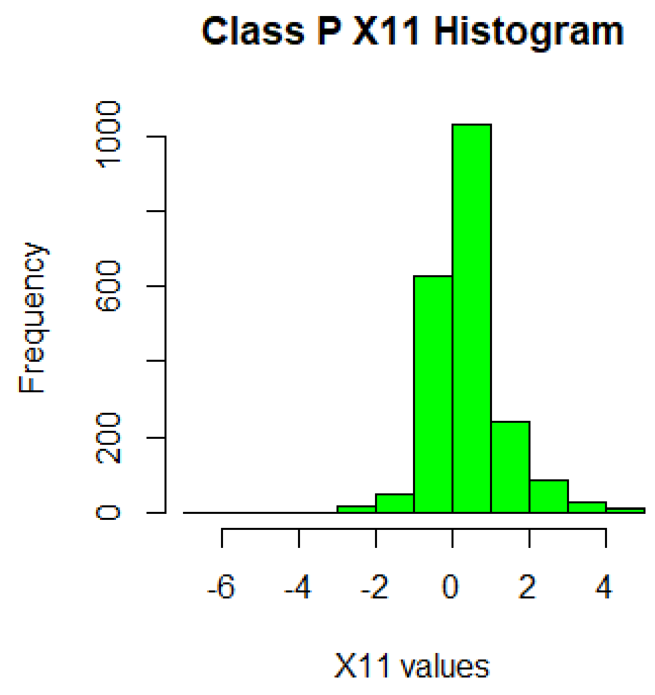
```
BNT=400


#7
bestRF<-randomForest(Class ~ ., data=TRAINSET, ntree=BNT, mtry=sqrt(32))
imp<-bestRF$importance
imp_sorted<-sort(imp,decreasing=TRUE)
imp_ind<-order(-imp)
imp_final<-cbind(imp_ind,imp_sorted)


#8
# Most important feature 11
hist(classD$X11,main='Class D X11 Histogram',xlab='X11 values',col='green')
```



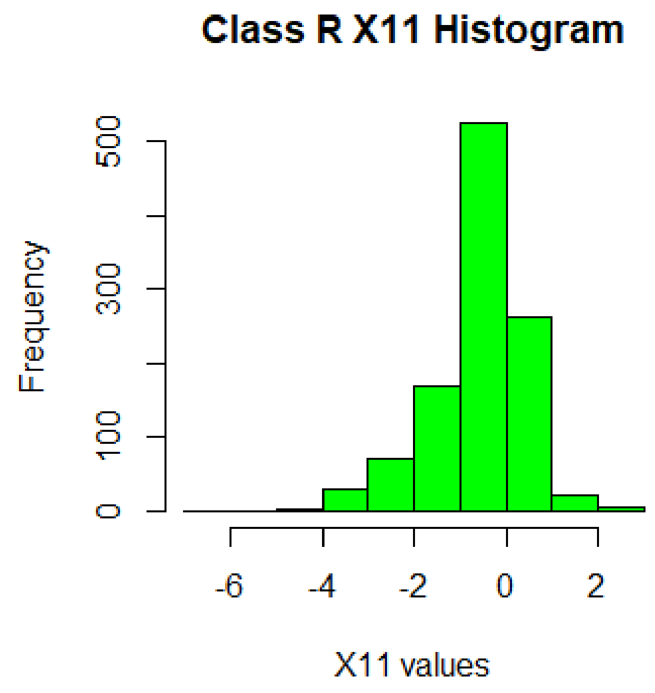
```r
hist(classH$X11,main='Class H X11 Histogram',xlab='X11 values',col='green')
```



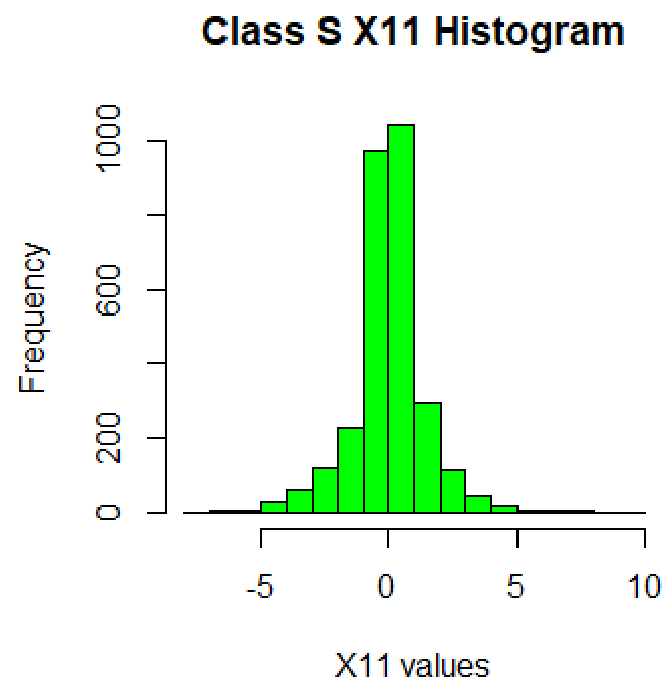
```r
hist(classP$X11,main='Class P X11 Histogram',xlab='X11 values',col='green')
```



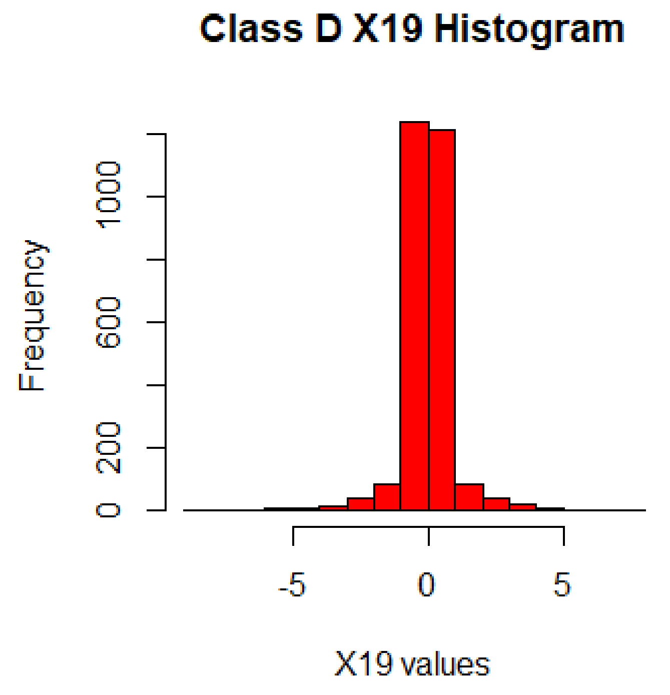
```
hist(classR$X11,main='Class R X11 Histogram',xlab='X11 values',col='green')
```



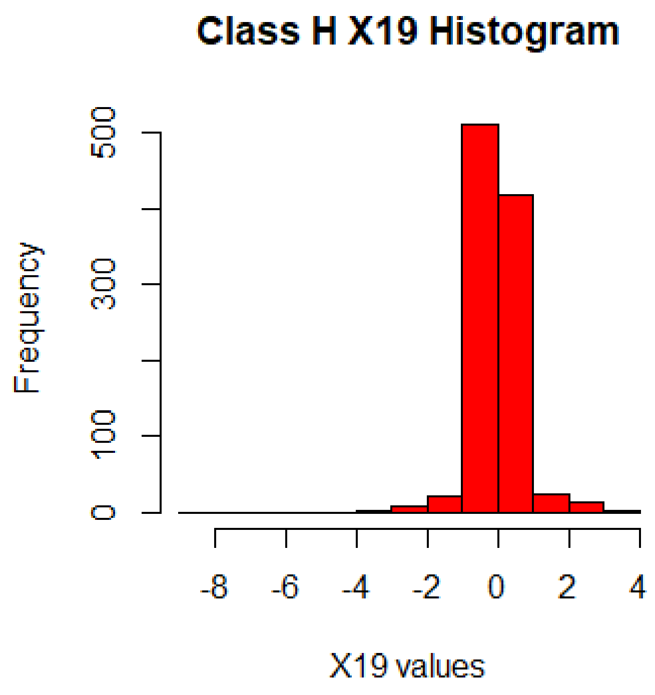
```
hist(classS$X11,main='Class S X11 Histogram',xlab='X11 values',col='green')
```



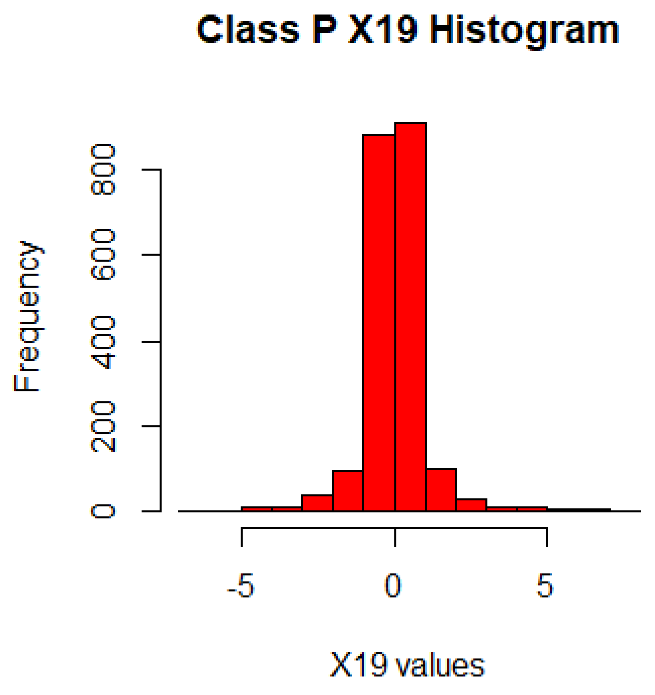
```r
# Least important feature 19
hist(classD$X19,main='Class D X19 Histogram',xlab='X19 values',col='red')
```



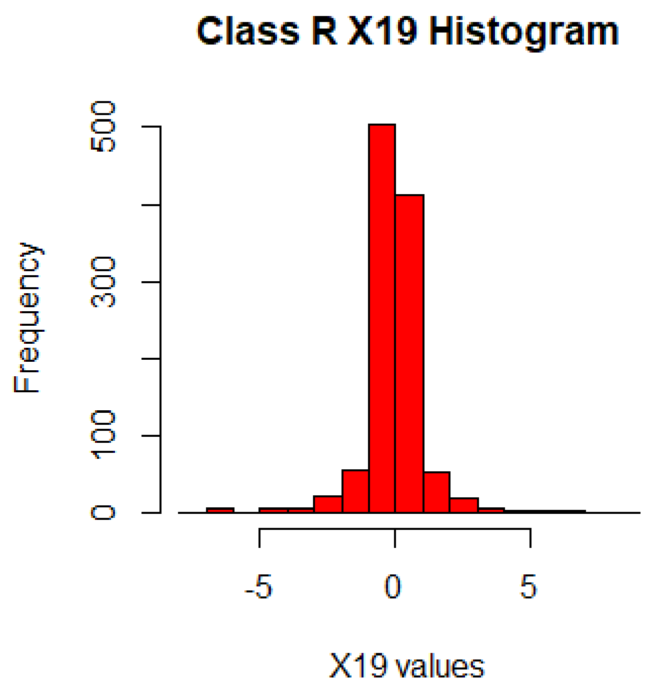
```r
hist(classH$X19,main='Class H X19 Histogram',xlab='X19 values',col='red')
```



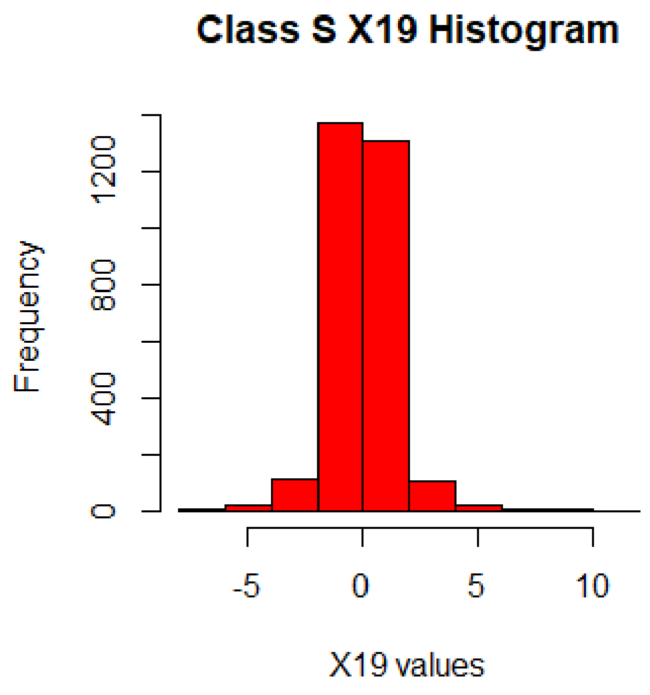
```r
hist(classP$X19,main='Class P X19 Histogram',xlab='X19 values',col='red')
```



```r
hist(classR$X19,main='Class R X19 Histogram',xlab='X19 values',col='red')
```



```r
hist(classS$X19,main='Class S X19 Histogram',xlab='X19 values',col='red')
```



```r
ks.test(classD$X11,classH$X11)
```

```
## Warning in ks.test(classD$X11, classH$X11): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classD$X11 and classH$X11
## D = 0.15133, p-value = 5.551e-15
## alternative hypothesis: two-sided
```

```
ks.test(classD$X11,classP$X11)
```

```
## Warning in ks.test(classD$X11, classP$X11): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classD$X11 and classP$X11
## D = 0.43635, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classD$X11,classR$X11)
```

```
## Warning in ks.test(classD$X11, classR$X11): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classD$X11 and classR$X11
## D = 0.51386, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classD$X11,classS$X11)
```

```
## Warning in ks.test(classD$X11, classS$X11): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classD$X11 and classS$X11
## D = 0.32536, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classH$X11,classP$X11)
```

```
## Warning in ks.test(classH$X11, classP$X11): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classH$X11 and classP$X11
## D = 0.38449, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classH$X11,classR$X11)
```

```
## Warning in ks.test(classH$X11, classR$X11): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classH$X11 and classR$X11
## D = 0.42096, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classH$X11,classS$X11)
```

```
## Warning in ks.test(classH$X11, classS$X11): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classH$X11 and classS$X11
## D = 0.28714, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classP$X11,classR$X11)
```

```
## Warning in ks.test(classP$X11, classR$X11): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classP$X11 and classR$X11
## D = 0.45878, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classP$X11,classS$X11)
```

```
## Warning in ks.test(classP$X11, classS$X11): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classP$X11 and classS$X11
## D = 0.19708, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classR$X11,classS$X11)
```

```
## Warning in ks.test(classR$X11, classS$X11): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classR$X11 and classS$X11
## D = 0.27912, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
ks.test(classD$X11,classD$X19)
```

```
## Warning in ks.test(classD$X11, classD$X19): p-value will be approximate in the
## presence of ties
```

```
##
##  Two-sample Kolmogorov-Smirnov test
```

```
##
## data:  classD$X11 and classD$X19
## D = 0.10252, p-value = 6.168e-13
## alternative hypothesis: two-sided

ks.test(classH$X11,classH$X19)

## Warning in ks.test(classH$X11, classH$X19): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classH$X11 and classH$X19
## D = 0.099198, p-value = 0.0001086
## alternative hypothesis: two-sided

ks.test(classP$X11,classP$X19)

## Warning in ks.test(classP$X11, classP$X19): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classP$X11 and classP$X19
## D = 0.25608, p-value < 2.2e-16
## alternative hypothesis: two-sided

ks.test(classR$X11,classR$X19)

## Warning in ks.test(classR$X11, classR$X19): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classR$X11 and classR$X19
## D = 0.31371, p-value < 2.2e-16
## alternative hypothesis: two-sided

ks.test(classS$X11,classS$X19)

## Warning in ks.test(classS$X11, classS$X19): p-value will be approximate in the
## presence of ties

##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  classS$X11 and classS$X19
## D = 0.1139, p-value < 2.2e-16
## alternative hypothesis: two-sided

library(dgof)

##
## Attaching package: 'dgof'

## The following object is masked from 'package:stats':
##
##     ks.test
```

```r
#9
# Cluster 2 had the minimum gini
names(data.clu1)[names(data.clu1) == "data$class[clu.1]"] <- "Class"
data.clu1$Class<-as.factor(data.clu1$Class)

cl1_class1<-data.clu1[data.clu1$Class==1,]
for (i in 1:nrow(cl1_class1)){
  cl1_class1[i,1:32]<-cl1_class1[i,1:32]*(1+(runif(1)-0.5)/10000)
}

cl1_class2<-data.clu1[data.clu1$Class==2,]
cl1_class2<-rbind(cl1_class2,cl1_class2,cl1_class2)
for (i in 1:nrow(cl1_class2)){
  cl1_class2[i,1:32]<-cl1_class2[i,1:32]*(1+(runif(1)-0.5)/10000)
}

data.clu1<-rbind(data.clu1,cl1_class1,cl1_class2)
set.seed(18)
indexCL1<-sample(1:nrow(data.clu1),size=.8*nrow(data.clu1))
trainCL1<-data.clu1[indexCL1,]
indexCL1_2<-setdiff(1:nrow(data.clu1),indexCL1)
testCL1<-data.clu1[indexCL1_2,]

trainCL1$Class<-as.factor(trainCL1$Class)
rf_CL1 <- randomForest(Class ~ .,data=trainCL1, ntree=BNT, mtry=sqrt(32))
rf.testCL1<-predict(rf_CL1,testCL1)

confCL1_train<-prop.table(table(rf_CL1$predicted,trainCL1$Class),margin=2)
accuracy(table(rf_CL1$predicted,trainCL1$Class))

## [1] 77.01525

confCL1_test<-prop.table(table(rf.testCL1,testCL1$Class),margin=2)
accuracy(table(rf.testCL1,testCL1$Class))

## [1] 80.86957

table(data.clu1$Class)

##
##   1   2   3   4   5
## 104 108 304 129 503

#11
# Selected class 1(D) and 5(S)

TRAINDATA_11<-rbind(trainD,trainS)
TESTDATA_11<-rbind(testD,testS)

library('e1071')

## Warning: package 'e1071' was built under R version 3.6.3

names(TRAINDATA_11)[names(TRAINDATA_11) == 'data$Phase'] <- 'Class'
names(TESTDATA_11)[names(TESTDATA_11) == 'data$Phase'] <- 'Class'
classifier<-svm(Class ~.,data=TRAINDATA_11,type='C-classification',kernel='linear')
svm_test<-predict(classifier,TESTDATA_11)

conf_svm<-prop.table(table(svm_test,TESTDATA_11$Class),margin=2)
accuracy(table(svm_test,TESTDATA_11$Class))
```

```
## [1] 77.61194
```