

University of Westminster

Machine Learning and Data Mining

5DATA002W

Coursework Report

Name-R.H.A. Tharushi Nethma Ranasinghe

UOW Number-19564252

IIT Number-20221508

Table of Contents

1. Partition Clustering Part	5
1.1.1st Subtask Objectives	5
1.1.1.1. Preprocessing Task	5
1.1.2. Clustering using automated tools	12
1.1.3. K-means Clustering Investigation	20
1.2. 2nd Subtask Objectives	25
1.2.1. PCA Analysis	25
1.2.2. Find Eigenvalues and Eigenvectors	26
1.2.3. Find the cumulative Score for Principal Components	29
1.2.4. Selecting Principal Components with cumulative value>85%	30
1.2.5. The discussion about selected principal components	31
1.2.6. Finding the optimal number of clusters for PCA analyzed dataset using automated tools.	32
1.2.7. Perform K-means analysis using the most favored k value	36
1.2.8. Silhouette plot for the most favored k value	38
1.2.9. Calinski-Harabasz Index	39
2. Financial Forecasting Part	42
2.1. Multi-layer Perceptron Neural Network	42
2.1.1. Input Variables	42
2.1.2. Input Variables and I/O metrics for AR approach	42
2.1.3. Normalization	43
2.1.4. Implementation of different MLPs for the AR approach	47
2.1.5. Explanation of RMSE, MAE, MAPE, sMAPE	64
2.1.6. Comparison matrix for AR approach	66
2.1.7. Best one-hidden layer MLP and best two-hidden layer MLP in AR approach	67
3. Reference	69
4. Appendix	69
4.1. Appendix Part 1(Code for 1st question)	69
4.2. Appendix Part 2 (Code for 2nd question)	75

Figure 1: Access the Dataset	5
Figure 2: Code for boxplot	5
Figure 3: Plot before removing outliers	6
Figure 4: Code for calculating missing values of the dataset	6
Figure 5: Output after removing missing values	7
Figure 6: Code for outlier detection	8
Figure 7: Code for the boxplot of removing outliers in each entity	9
Figure 8: Boxplot for before outlier remove in each entity	9
Figure 9: Code for the boxplot after removing each entity	9
Figure 10: Boxplot for after outlier removal in each entity	10
Figure 11: Code for scaling the dataset	10
Figure 12: Plot for before and after scaling dataset	11
Figure 13: Code for summary statistics of before and after preprocessing	11
Figure 14: Summary statistics before preprocessing	12
Figure 15: Summary statistics after preprocessing	12
Figure 16: Code for NBClust method	13
Figure 17: Output of NbClust	14
Figure 18: Hubert index and D index plots	14
Figure 19: Plot for NbClust result	15
Figure 20: Code for Elbow method	15
Figure 21: Plot for Elbow method	16
Figure 22: Code for Gap Statistics method	17
Figure 23: Console output for Gap statistics	18
Figure 24: Plot for gap statistics	18
Figure 25: Code for Silhouette method	19
Figure 26: Plot for Silhouette method	19
Figure 27: Code for calculating BSS/TSS/WSS	20
Figure 28: Output for the calculated BSS/TSS/WSS values	21
Figure 29: Code for cluster plot for k=2	21
Figure 30: Graphical representation cluster plot for K=2	22
Figure 31: Code for the Average Silhouette width	23
Figure 32: Average silhouette width for 2 clusters	23
Figure 33: Plot average silhouette width for 2 clusters	24
Figure 34: Output for the calculation of the average Silhouette width	24
Figure 35: Code for applying PCA	25
Figure 36: Output of the PCA summary	26
Figure 37: Code of the Eigenvalues and Eigenvectors in PCA analysis	26
Figure 38: R output for the extracted Eigenvalues	27
Figure 39: R output for the extract Eigenvectors	27
Figure 40: Code for the scree plot	28
Figure 41: Scree plot (plot the eigenvalues of the principal components)	28
Figure 42: Code for find the cumulative score for principal components	29
Figure 43: R output for cumulative scores per principal components	29
Figure 44: Code for selecting Principal components with cumulative score>85%	30
Figure 45: R output for the Number of selected components	30

Figure 46: R output for the transformed after calculating the cumulative value>85%	31
Figure 47: Scree plot for PCA analysis.....	31
Figure 48: Plots for Hubert index and D index	32
Figure 49: Output of NbClust	33
Figure 50: Plot of NbClust result after PCA	33
Figure 51: Plot for the elbow method.....	34
Figure 52: Plot for the Silhouette method	34
Figure 53: Plot for the gap statistics method	35
Figure 54: R output for the gap statistics method	35
Figure 55: Graphical representation cluster plot for K=2	36
Figure 56: Details of the 2 cluster centers	37
Figure 57: Values for WSS,TSS, and BSS/TSS	37
Figure 58: Clusters silhouette plot	38
Figure 59: R output of the average silhouette width	39
Figure 60: Code for Calinski-Harbasz index.....	40
Figure 61: Calirski-Harabasz index plot	41
Figure 62: Code for normalization	44

1. Partition Clustering Part

1.1.1st Subtask Objectives

The first dataset comprises 2700 samples of white wine varieties sourced from a specific region in Portugal. Each sample is characterized by twelve attributes, with the final attribute being the quality score determined through sensory evaluation. The first eleven attributes represent the wines' diverse chemical characteristics, including chlorides, pH, sulphate, etc.

The dataset is imported to the project as a .xlsx file (Excel) using 'readxl' library and read the data from the file using read_excel() function. Figure 1 shows this.

```
#Sub Task 1

# Load Libraries
library("readxl")
library("dplyr")

# Read data from the file
whitewine_v6data <- read_excel("whitewine_v6.xlsx")
cat("Data loaded from Excel file successfully. Dimensions: ", dim(whitewine_v6data), "\n")
```

Figure 1: Access the Dataset

1.1.1.Preprocessing Task

Before proceeding with clustering, it's essential to preprocess the imported data from the Excel file. Due to the presence of missing values, incomplete data, duplicates, inconsistencies, and noisy data in the imported dataset, pre-processing is necessary to identify and eliminate them from the dataset. Data preprocessing stands as a critical starting step in data mining and analysis, where raw data undergoes conversion into a standardized format suitable for analysis and machine learning algorithms. This conversion guarantees that the data can be effectively interpreted and utilized by computer systems.

Figure 3 displays a boxplot illustrating the presence of outliers. On the x-axis are the attributes of the dataset, while the y-axis represents the number of data points.

```
par(mar=c(2, 2, 2, 2))
boxplot(whitewine_v6data)
```

Figure 2: Code for boxplot

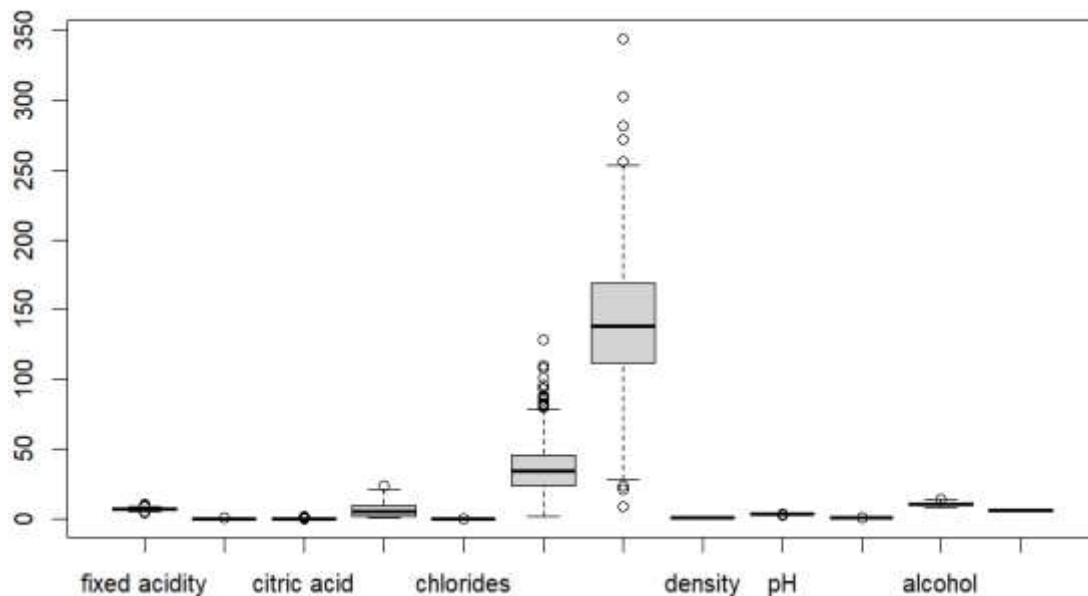


Figure 3: Plot before removing outliers

Before applying k-means, three preprocessing procedures must be completed since incorrect data points can seriously impair the dataset's ability to be used for analysis and processing.

- Checking and removing missing values
- Outliers detection/removal from the dataset
- Scaling the data

1.1.1.1. Calculating and removing missing values

```
# Calculate the total number of missing values in the dataset
missing_values <- sum(is.na(whitewine_v6data))

# Check if there are any missing values and if there missing values, remove rows with missing values
if (missing_values > 0) {
  whitewine_v6data <- whitewine_v6data[complete.cases(whitewine_v6data), ] # Remove rows with missing values
  cat("Missing values found and rows with missing values removed.", "\n")
} else {
  cat("No missing values found. No rows removed.", "\n")
}

# Confirm that all missing values have been removed
missing_values_after <- sum(is.na(whitewine_v6data))
cat("Total number of missing values after removal: ", missing_values_after, "\n")
```

Figure 4: Code for calculating missing values of the dataset

The code initially calculates the total number of missing values in the dataset using the `sum(is.na(data))` function. This variable, `missing_values`, captures the count of NA (missing) values across all variables in the dataset. Subsequently, an if statement evaluates whether there are any missing values present (`missing_values > 0`). If missing values are detected, the code proceeds to remove the corresponding rows containing these missing values. Within the conditional block, the `complete.cases()` function is employed to identify and retain rows without any missing values, effectively eliminating rows with incomplete data. A notification message, "Missing values found and rows with missing values removed," is then printed using `cat()` to inform the user of the performed action. Following the removal of rows with missing values, the variable `missing_values_after` is updated to reflect the total number of missing values in the dataset post-removal. This confirmation is accomplished by recalculating the sum of missing values using `sum(is.na(Whitewine_v6data))` and printing the result using `cat()`. This systematic handling of missing values ensures data integrity by verifying the successful removal of all missing values from the dataset.

The dataset hasn't any missing values. So that the output looks like the below Figure 5.

```
>
> # Calculate the total number of missing values in the dataset
> missing_values <- sum(is.na(whitewine_v6data))
>
> # Check if there are any missing values and if there missing values, remove rows with missing values
> if (missing_values > 0) {
+   whitewine_v6data <- whitewine_v6data[complete.cases(whitewine_v6data), ] # Remove rows with missing values
+   cat("Missing values found and rows with missing values removed","\n")
+ } else {
+   cat("No missing values found. No rows removed","\n" )
+ }
No missing values found. No rows removed.
>
> # Confirm that all missing values have been removed
> missing_values_after <- sum(is.na(whitewine_v6data))
> cat("Total number of missing values after removal: ", missing_values_after, "\n")
Total number of missing values after removal: 0
>
```

Figure 5: Output after removing missing values

1.1.1.2. Outlier detection from the dataset

Outlier detection entails identifying unusual or abnormal data points within a dataset. In this instance, outliers are being eliminated by employing the Interquartile Range (IQR) method.

```
cat("Starting outlier removal process...","\n")
# define outlier removal function
remove_outliers <- function(x) {
  qnt <- quantile(x, probs=c(.25, .75), na.rm = TRUE)
  iqr <- IQR(x, na.rm = TRUE)
  lower_bound <- qnt[1] - 1.5 * iqr
  upper_bound <- qnt[2] + 1.5 * iqr
  x_clipped <- pmax(pmin(x, upper_bound), lower_bound)
  return(x_clipped)
}

# Perform outlier removal
wine_data_clean <- as.data.frame(lapply(whitewine_v6data[, 1:11], remove_outliers))
cat("Successfully outlier removal.", "\n")
```

Figure 6: Code for outlier detection

The `remove_outliers` function is responsible for removing outliers and is contained within its function definition. It handles one vector (column) of data at a time. Inside the function, the `quantile()` function calculates the first and third quartiles, which are respectively the 25th and 75th percentiles of the data. These values are stored in a variable called `qnt`.

The interquartile range (IQR) is then calculated using the `IQR()` function, which measures the statistical spread of the middle 50% of the data. The lower and upper bounds for outliers are determined by extending 1.5 times the IQR below the first quartile and above the third quartile, respectively. The identified outliers are then clipped using `pmax()` and `pmin()` functions to ensure that no values fall below the lower bound or above the upper bound. This results in a "clipped" vector where all outliers have been adjusted to within the specified bounds.

The `remove_outliers` function is then applied to each of the first 11 columns of `Whitewine_v6data` using `lapply()`. This applies a function over a list or vector and returns a list. Each column is passed individually to the `remove_outliers` function, and the results are recombined into a new data frame called `wine_data_clean`. This approach assumes that the dataset has at least 11 columns and that all these columns contain numeric data suitable for outlier analysis and adjustment.

Figure 8 presents the boxplot before outlier removal for each of the 11 entities.


```
# Visualization Boxplot for each attribute before outlier removal

par(mfrow=c(3,4))
for (i in 1:11) {
  boxplot(whitewine_v6data[, i], main = names(whitewine_v6data)[i])
}
```

Figure 7: Code for the boxplot of removing outliers in each entity

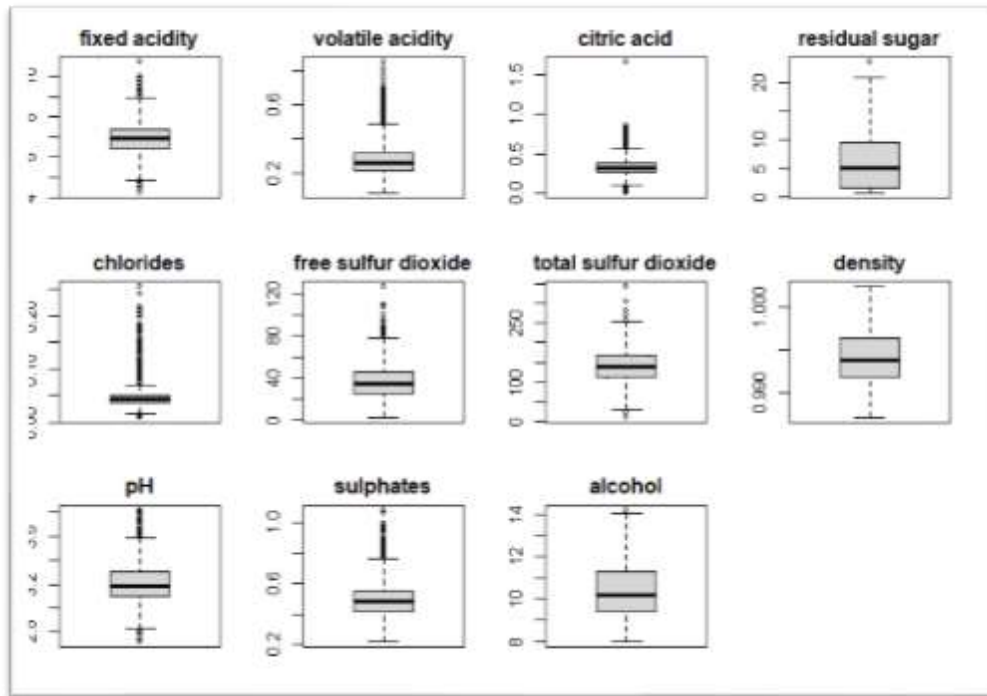


Figure 8: Boxplot for before outlier remove in each entity

Figure 10 illustrates the boxplot after outlier removal for each entity.

```
# Visualization Boxplot for each attribute after outlier removal

par(mfrow=c(3,4))
for (i in 1:11) {
  boxplot(scaled_data[, i], main = paste("Cleaned", names(whitewine_v6data)[i]))
}
```

Figure 9: Code for the boxplot after removing each entity

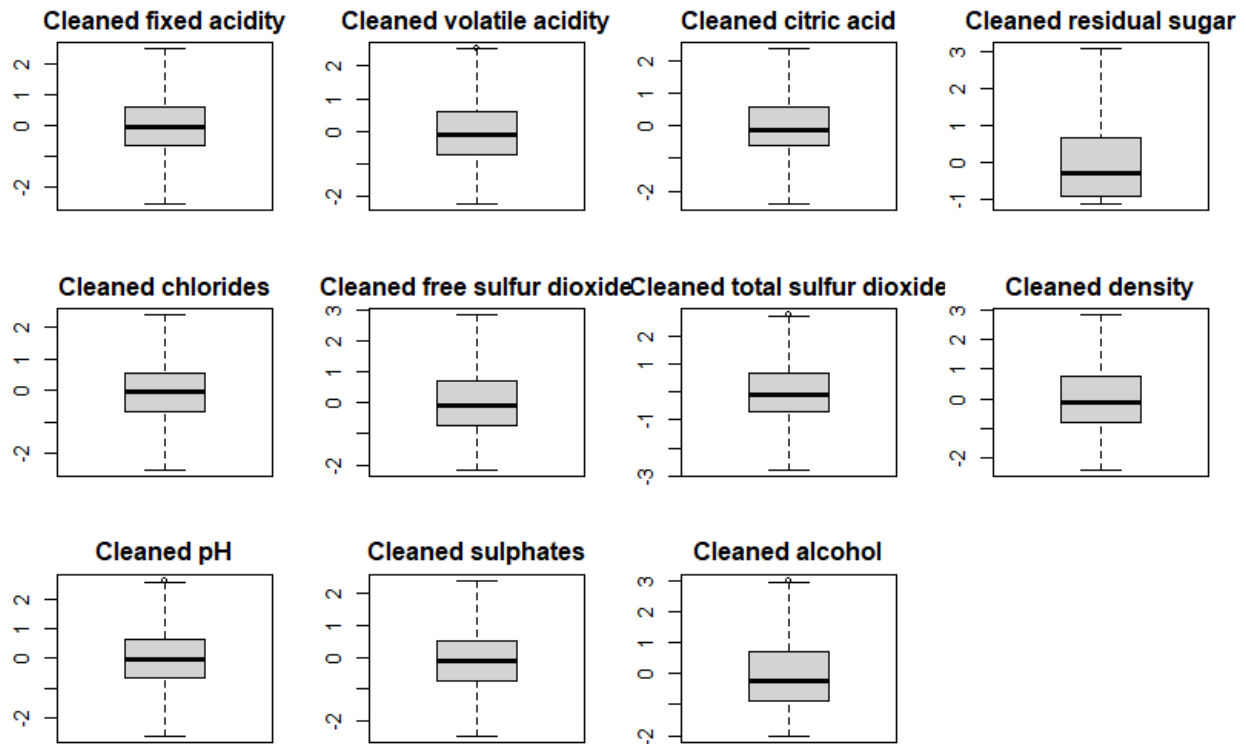


Figure 10: Boxplot for after outlier removal in each entity

1.1.1.3. Scaling the data

The process of standardizing the range of independent variables or data features is known as data scaling. In particular, when dealing with huge datasets, it attempts to standardize the data's range of features to avoid any feature from dominating the others. This is an important step in the preparation of data, especially for algorithms that are sensitive to the data's range.

```
# Perform scaling on cleaned data
cat("Scaling data...", "\n")
scaled_data <- scale(wine_data_clean)
cat("Data scaling completed.", "\n")
```

Figure 11: Code for scaling the dataset

The provided code segment is dedicated to scaling cleaned data. The function `scale()` is utilized to standardize the `wine_data_clean` dataset, where each variable undergoes adjustment by subtracting its mean and dividing by its standard deviation. This process guarantees that each variable attains a mean of zero and a standard deviation of one. The scaled data is then stored in the variable `scaled_data`. Figure 12 illustrates the appearance of the dataset both before and after the completion of the scaling process.

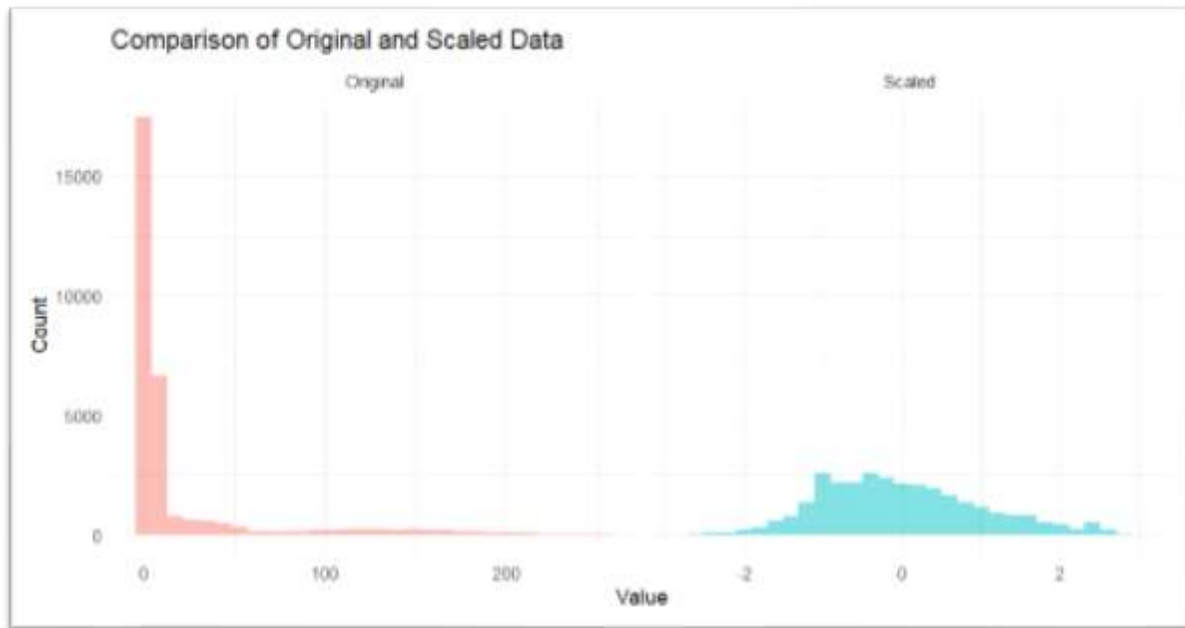


Figure 12: Plot for before and after scaling dataset

Now that the dataset preprocessing is completed, it's essential to assess the changes brought about by this process by comparing summaries of the data before and after preprocessing.

```
# Summary statistics before and after outlier removal
#before
summary_before <- apply(whitewine_v6data[, 1:11], 2, summary)
cat("Summary statistics before outlier removal:", "\n")
print(summary_before)

#after
summary_after <- apply(wine_data_clean, 2, summary)
cat("\nSummary statistics after outlier removal:", "\n")
print(summary_after)
```

Figure 13: Code for summary statistics of before and after preprocessing

The summary showcases how the statistics of each entity have changed after preprocessing.

```

> # Summary statistics before and after outlier removal
> #before
> summary_before <- apply(whitewine_v6data[, 1:11], 2, summary)
> cat("Summary statistics before outlier removal:", "\n")
Summary statistics before outlier removal:
> print(summary_before)
      fixed.acidity volatile.acidity citric.acid residual.sugar  chlorides free.sulfur.dioxide total.sulfur.dioxide
Min.      4.200000      0.0800000  0.0000000      0.600 0.00900000      2.00000      9.0000
1st Qu.    6.400000      0.2100000  0.2700000      1.700 0.03600000     24.00000     112.0000
Median    6.900000      0.2600000  0.3200000      4.900 0.04300000     34.00000     138.0000
Mean     6.935056      0.2731278  0.3373519      6.247 0.04573852     35.37852     141.4906
3rd Qu.    7.400000      0.3200000  0.3900000      9.600 0.05000000     46.00000     169.0000
Max.     10.700000      0.8500000  1.6600000     23.500 0.25500000    128.00000     344.0000

      density      pH sulphates  alcohol
Min.    0.9871100  2.720000  0.2200000   8.00000
1st Qu. 0.9918000  3.100000  0.4100000   9.40000
Median 0.9938300  3.190000  0.4800000  10.20000
Mean   0.9941502  3.200159  0.4953148  10.46157
3rd Qu. 0.9963000  3.300000  0.5500000  11.30000
Max.   1.0024100  3.820000  1.0800000  14.20000

```

Figure 14: Summary statistics before preprocessing

```

> #after
> summary_after <- apply(wine_data_clean[, 2, summary])
> cat("\nSummary statistics after outlier removal:", "\n")
Summary statistics after outlier removal:
> print(summary_after)
      fixed.acidity volatile.acidity citric.acid residual.sugar  chlorides free.sulfur.dioxide total.sulfur.dioxide
Min.      4.900000      0.0800000  0.0900000      0.600000 0.01500000      2.0000      26.500
1st Qu.    6.400000      0.2100000  0.2700000      1.700000 0.03600000     24.0000     112.000
Median    6.900000      0.2600000  0.3200000      4.900000 0.04300000     34.0000     138.000
Mean     6.927463      0.2705963  0.338556      6.246241 0.04367296     35.2863     141.432
3rd Qu.    7.400000      0.3200000  0.3900000      9.600000 0.05000000     46.0000     169.000
Max.     8.900000      0.4850000  0.5700000     21.450000 0.07100000     79.0000     254.500

      density      pH sulphates  alcohol
Min.    0.9871100  2.800000  0.2200000   8.00000
1st Qu. 0.9918000  3.100000  0.4100000   9.40000
Median 0.9938300  3.190000  0.4800000  10.20000
Mean   0.9941502  3.199315  0.4931963  10.46155
3rd Qu. 0.9963000  3.300000  0.5500000  11.30000
Max.   1.0024100  3.600000  0.7600000  14.15000

```

Figure 15: Summary statistics after preprocessing

1.1.2. Clustering using automated tools

Determining the ideal number of clusters in a data set is a crucial aspect of k-means clustering. The optimal number of clusters is subjective to some extent and depends on the chosen similarity measurement method and partitioning parameters. To address this, four automated tools are employed to determine the number of cluster centers, followed by utilizing the majority rule to decide on the best number of clusters.

To determine the optimal number of cluster centers, four automated tools are used in here.

- NBClust method
- Elbow method

- Gap Statistics method
- Silhouette method

1.1.2.1. NBClust Method

The NbClust method is employed to identify the optimal number of clusters within a dataset during the clustering process. With access to 30 indices for cluster validation, the NbClust function facilitates the simultaneous application of multiple indices. This allows users to select the best clustering scheme from the results obtained by varying all combinations of the number of clusters. To utilize this powerful functionality, one simply needs to utilize the "NbClust" library.

```
library(NbClust)
cat("Performing cluster analysis using NbClust...","\n")

set.seed(200) # for reproducibility
nbclust_results <- NbClust(scaled_data, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans")

# visualize the NbClust result
barplot(table(nbclust_results$Best.n[1,]),
        xlab = "Number of Clusters", ylab = "Number of Criteria",
        main = "NbClust Results", col = "lightgreen")

cat("NbClust analysis is successful.", "\n")
```

Figure 16: Code for NBClust method

In the above Figure 16, the NbClust library, a package in R used for identifying the most suitable number of clusters within a dataset. The seed value of 200 is established via set.seed() to ensure reproducibility in the results obtained from different executions of the code, as it guarantees that the same sequence of random numbers is generated each time. The NbClust function is then invoked with several parameters. It operates on the scaled_data dataset and utilizes the "euclidean" distance measure for clustering. The range of clusters considered spans from 2 to 15, specified by min.nc and max.nc. Moreover, the "method" parameter specifies the utilization of the k-means algorithm for clustering purpose.

In the NbClust function, the determination of the optimal number of clusters is achieved through the utilization of two specific indexes: the Hubert index and the D index. Figures 17 and 18

showcase the outputs and visualizations generated from employing these indexes to identify the most suitable number of clusters.

```

> set.seed(200) # for reproducibility
> nbclust_results <- NbClust(scaled_data, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans")
*** : The Hubert index is a graphical method of determining the number of clusters.
      In the plot of Hubert index, we seek a significant knee that corresponds to a
      significant increase of the value of the measure i.e the significant peak in Hubert
      index second differences plot.

*** : The D index is a graphical method of determining the number of clusters.
      In the plot of D index, we seek a significant knee (the significant peak in Dindex
      second differences plot) that corresponds to a significant increase of the value of
      the measure.

*****
* Among all indices:
* 10 proposed 2 as the best number of clusters
* 4 proposed 3 as the best number of clusters
* 2 proposed 4 as the best number of clusters
* 4 proposed 5 as the best number of clusters
* 1 proposed 11 as the best number of clusters
* 1 proposed 14 as the best number of clusters
* 2 proposed 15 as the best number of clusters

***** Conclusion *****

* According to the majority rule, the best number of clusters is 2

```

Figure 17: Output of NbClust

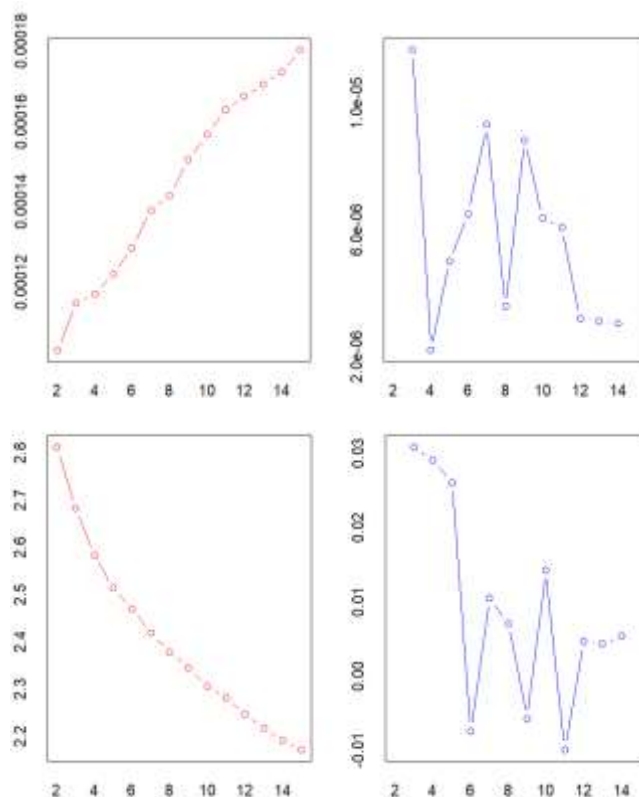


Figure 18: Hubert index and D index plots

The NbClust results are presented visually through a bar plot generated using the `barplot()` function. This plot illustrates the recommended number of clusters suggested by NbClust across various evaluation criteria. On the plot, the x-axis denotes the number of clusters, while the y-axis indicates the number of criteria considered. Figure 19 represent this.

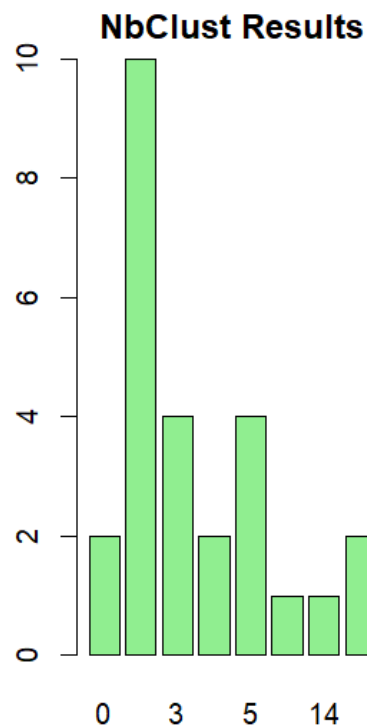


Figure 19: Plot for NbClust result

As per the NbClust method, the determined cluster number through the majority rule is two.

1.1.2.2. Elbow Method

The Elbow Method is a visual approach to determine the ideal 'K' (number of clusters) in K-means clustering. It calculates the Within-Cluster Sum of Squares (WSS), the total of the squared distances between data points and their cluster center.

```
library(factoextra)
cat("Performing cluster analysis using Elbow Method...", "\n")
set.seed(200) # for reproducibility
fviz_nbclust(scaled_data, kmeans, method = "wss") +
  geom_vline(xintercept = 2, linetype = 3, color = "red") +
  labs(title = "Elbow Method")
cat("Elbow analysis is successful.", "\n")
```

Figure 20: Code for Elbow method

This R code conducts cluster analysis utilizing the Elbow Method to identify the optimal number of clusters within the dataset. Initially, imports the factoextra package, which offers a diverse set of functions designed for visualizing the outcomes of multivariate analyses, including cluster analysis. The seed for generating random numbers is set to 200 using the `set.seed()` function. Establishing the seed guarantees the reproducibility of outcomes in scenarios involving randomization, like when centroids are initialized in the k-means clustering process. Then utilizes the use of the "factoextra" package's `fviz_nbclust()` function to generate a plot, which uses the Within-Cluster Sum of Squares (WSS) approach to determine the optimal number of clusters. It applies the k-means clustering method defined by k-means to the dataset given by the parameter `scaled_data`. The function is instructed to use the WSS approach to determine the ideal number of clusters by setting the `method` argument to "wss". The code includes the `geom_vline()` function, which adds a vertical dashed line to the plot at $x = 2$ to identify the "elbow point," which represents the optimal cluster count. Figure 21 shows the graphical view of the Elbow method.

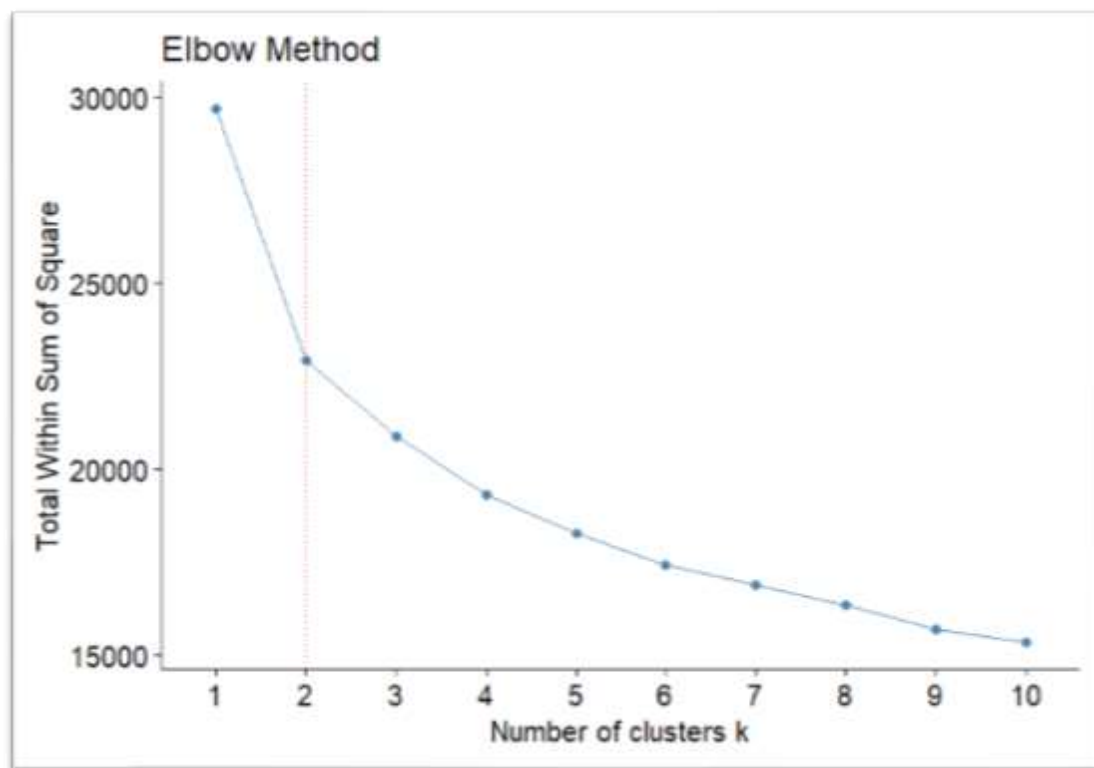


Figure 21: Plot for Elbow method

1.1.2.3. Gap Statistics method

The gap statistic is a statistical method that assesses the clustering structure by comparing the total within intra-cluster variation for various values of k to their expected values under the null reference distribution of the data. By selecting the value that maximizes the gap statistic, one can estimate the optimal clusters. This indicates that the clustering structure is significantly different from the random uniform distribution of points.

```
library(cluster)
cat("Performing cluster analysis using gap statistics Method...","\n")
set.seed(200) # for reproducibility
# Calculating the gap statistic
gap_stat <- clusGap(scaled_data, FUN = kmeans, nstart = 25, K.max = 15, B = 500)
# Visualizing the gap statistics
library(factoextra)
fviz_gap_stat(gap_stat)
cat("Gap statistics analysis is successful.", "\n")
```

Figure 22: Code for Gap Statistics method

This R code performs cluster analysis using the gap statistics method to determine the optimal number of clusters in the dataset. The "cluster" package, provides essential functions for clustering analysis, including the pivotal `clusGap()` function used to calculate the gap statistic. To ensure reproducibility, the random number generator seed is set to 200 using the `set.seed()` function. To determine the gap statistic, utilize the `clusGap()` function. The dataset being analyzed is represented by the `scaled_data` parameter. The clustering algorithm to be used is specified by the `FUN` parameter, which is set to k-means. `nstart` determines the number of random sets of initial clusters, `K.max` sets the maximum number of clusters to consider, and `B` specifies the number of bootstrap replicates for computing the reference distribution.

The output of this is below.

```

> library(cluster)
> cat("Performing cluster analysis using gap statistics Method...\n")
Performing cluster analysis using gap statistics Method...
> set.seed(200) # for reproducibility
> # Calculating the gap statistic
> gap_stat <- clusGap(scaled_data, FUN = kmeans, nstart = 25, K.max = 15, B = 500)
Clustering k = 1,2,..., K.max (= 15): .. done
Bootstrapping, b = 1,2,..., B (= 500) [one "." per sample]:
..... 50
..... 100
..... 150
..... 200
..... 250
..... 300
..... 350
..... 400
..... 450
..... 500
There were 50 or more warnings (use warnings() to see the first 50)
> # Visualizing the gap statistics
> library(factoextra)
> fviz_gap_stat(gap_stat)
> cat("Gap statistics analysis is successful.\n")
Gap statistics analysis is successful.
>

```

Figure 23: Console output for Gap statistics

To present the outcomes visually, the code incorporates the "factoextra" package, renowned for its comprehensive visualization capabilities in multivariate analysis. Subsequently, it employs the `fviz_gap_stat()` function to create a graphical depiction of the gap statistic.

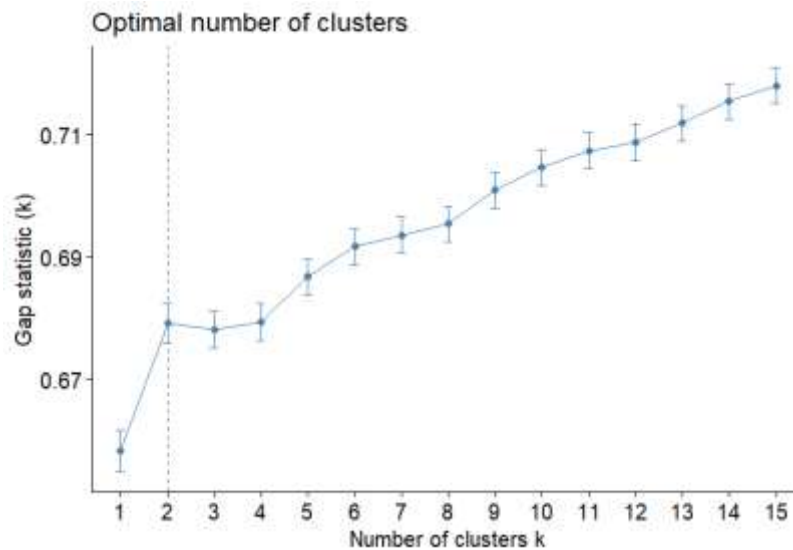


Figure 24: Plot for gap statistics

According to the gap statistics method, the optimal number of clusters is 2.

1.1.2.4. Silhouette Method

The silhouette score is a metric used to evaluate the quality of clustering performed by K-means. It essentially measures how well data points are grouped within their assigned clusters compared to data points in other clusters.

```
library(factoextra)
cat("Performing cluster analysis using silhouette Method...\n")
set.seed(200) # for reproducibility
# Calculating the best number of clusters using silhouette method
silhouette_results <- fviz_nbclust(scaled_data, kmeans, method = "silhouette")
# Plotting the silhouette analysis
print(silhouette_results)
cat("Silhouette analysis is successful.\n")
```

Figure 25: Code for Silhouette method

The silhouette approach is used in this R code to perform cluster analysis to determine the ideal number of clusters within a dataset. The "factoextra" package, which offers functions for visualizing the outcomes of multivariate analyses, including cluster analysis. The set.seed() function is employed to set the seed for random number generation to 200 in order to guarantee reproducibility of the results. then uses the fviz_nbclust() function to find the optimal number of clusters using the silhouette approach. Scaled_data represents the dataset that is being analyzed, and k-means indicates the clustering technique. The method = "silhouette" specifies the silhouette method. Finally, The print() function is utilized to display the plot representing the results of silhouette analysis.

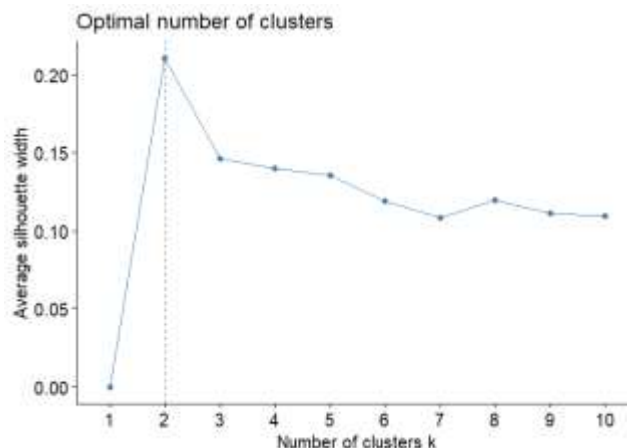


Figure 26: Plot for Silhouette method

According to the Silhouette method, the optimal number of clusters is 2.

1.1.3. K-means Clustering Investigation

Determining the optimal value for K in k-means clustering is a crucial step in the analysis. K represents the number of clusters the algorithm will identify in the dataset. Choosing the right K value is critical as it can have a significant impact on the clustering results and subsequent interpretations. Based on the previous results, it has been determined that the most suitable value for K is 2. Below Figure 27 showcases how the k-means implementation for the optimal k value.

1.1.3.1. Calculating BSS/TSS/WSS

```
# Performing k-means with k=2
set.seed(200) #for reproducibility
kmeans_2 <- kmeans(scaled_data, centers = 2, nstart = 50)

# Function to display results
display_kmeans_results <- function(km_result) {

  total_ss <- sum((scaled_data - colMeans(scaled_data))^2)
  bss <- sum(km_result$betweenss)
  tss <- total_ss
  bss_tss_ratio <- bss / tss

  cat("Total SS: ", total_ss, "\n")
  cat("BSS/TSS Ratio: ", bss_tss_ratio, "\n")
  cat("WSS (within-cluster sum of squares): ", km_result$tot.withinss, "\n\n")

}

# Displaying results for k = 2
cat("Results for k=2:", "\n")
display_kmeans_results(kmeans_2)
```

Figure 27: Code for calculating BSS/TSS/WSS

This R script executes k-means clustering with two clusters on the dataset `scaled_data`. It begins by setting the seed for random number generation to 200, ensuring consistent results for reproducibility. The `kmeans()` function is then utilized to perform the clustering, with the specified number of clusters set to 2 (`centers=2`). The `nstart` parameter determines the number of initial configurations to attempt, aiding in finding a better solution. Additionally, a function named `display_kmeans_results` is defined to present the results of the k-means clustering. This function calculates and prints various metrics, including:

1. TSS
2. BSS
3. WSS

The metric "total_ss" provides a comprehensive understanding of the data's dispersion by quantifying the total variation in the dataset.

The "bss" represents the variation between different clusters and highlights the extent to which the clusters are distinct.

"tss" is equivalent to total_ss, serving as a reference point for assessing the effectiveness of clustering by encompassing the entirety of the variation in the dataset.

The "bss_tss_ratio" compares the between-cluster sum of squares to the total sum of squares, indicating the degree of separation among the clusters and providing insight into their distinctiveness.

Lastly, "km_result\$tot.withinss" measures the within-cluster sum of squares, evaluating the compactness of the clusters and shedding light on how closely the data points within each cluster are grouped.

Figure 28 shows the output for the above metrics.

```
> # Displaying results for k = 2
> cat("Results for k=2:", "\n")
Results for k=2:
> display_kmeans_results(kmeans_2)
Total SS: 29689
BSS/TSS Ratio: 0.2285184
WSS (within-cluster sum of squares): 22904.52
```

Figure 28: Output for the calculated BSS/TSS/WSS values

1.1.3.2. Cluster plot for K=2

```
library(factoextra)
# visualizing clusters
fviz_cluster(list(data = scaled_data, cluster = kmeans_2$cluster), geom = "point", main = "k=2")
```

Figure 29: Code for cluster plot for k=2

To help visualize clustering results, the script employs the factoextra library. The script relies heavily on an essential function called `fviz_cluster()`. This function generates a scatter plot, where each point represents an observation from the dataset, and its color represents the cluster assignment. To generate the plot, the function uses a list containing the `scaled_data` and `kmeans_2$cluster`. The `kmeans_2$cluster` contains the cluster assignments for each observation resulting from a k-means clustering operation. The `kmeans_2` object includes the kmeans function's output applied to `scaled_data`, specifying two centers (clusters). The `fviz_cluster()` function call's arguments, including `geom = "point"` and `main = "k=2"`, dictate that the data points should be displayed as points on the plot, and the main title of the visualization will be "k=2," indicating that the displayed results show a clustering solution with two centers.

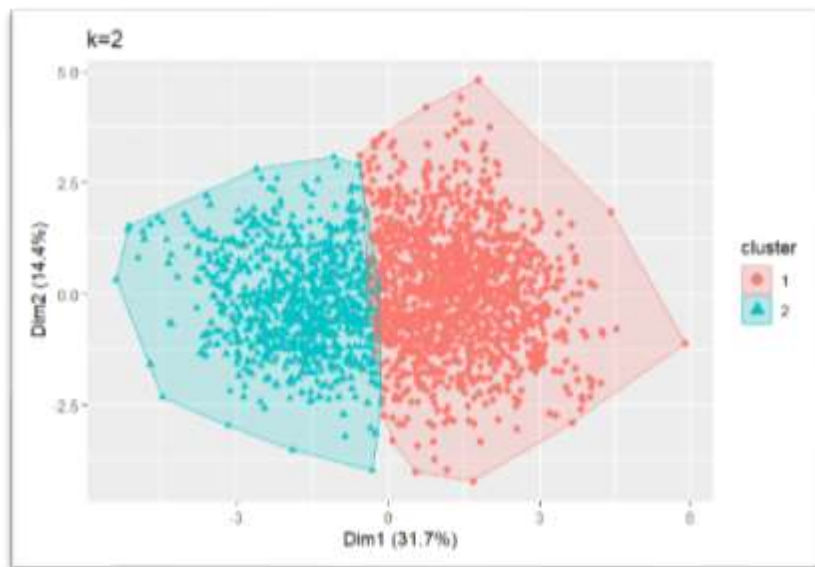


Figure 30: Graphical representation cluster plot for K=2

Figure 30, the visualization illustrates the grouping of data points into two distinct clusters. By analyzing this visual aid, we can accurately evaluate the efficacy of the clustering process. The visual separation serves as an indicator of the clustering algorithm's proficiency in grouping data based on the provided features. These visualizations play a crucial role in validating the assumptions of cluster analysis, as well as guiding further data exploration and decision-making processes.

1.1.3.3. Average Silhouette Width

Silhouette width is a way to measure how well an object fits in its assigned cluster. It considers how far the object is from other objects in the cluster, as well as from objects in nearby clusters. High Silhouette Width means that the object is a good match for its cluster and that the cluster is separated from nearby clusters. Low Silhouette Width means that the object might belong to a different cluster, or that the cluster may need to be examined more closely. Silhouette Width is important for data analysis and can help us better understand how objects are grouped in a dataset.

```
library(cluster)
library(factoextra)

# Compute silhouette information
sil_widths <- silhouette(kmeans_2$cluster, dist(scaled_data))

# Visualize the silhouette plot
silhouette_plot <- fviz_silhouette(sil_widths)
print(silhouette_plot)

# Calculate the average silhouette width
average_sil_width <- mean(sil_widths[, "sil_width"])
cat("Average silhouette width:", average_sil_width, "\n")
```

Figure 31: Code for the Average Silhouette width

The above script utilizes the silhouette approach to assess the results of clustering using the "cluster" and "factoextra" packages. First, silhouette widths for every observation are computed using the silhouette() function, which is based on the cluster assignments derived from the previous k-means clustering (kmeans_2\$cluster). The dist() method used to the scaled dataset (scaled_data) allows one to measure the distances between observations, which facilitates these computations.

```
> # Visualize the silhouette plot
> silhouette_plot <- fviz_silhouette(sil_widths)
  cluster size ave.sil.width
1         1 1578         0.21
2         2 1122         0.21
> print(silhouette_plot)
```

Figure 32: Average silhouette width for 2 clusters

The `fviz_silhouette()` function is then used to create a silhouette plot, which graphically depicts the silhouette widths and offers information on the quality of the clustering. The silhouette widths for each cluster as well as the total silhouette width for each observation are shown in this plot. The `print()` function is then used to print the resulting silhouette plot to the graphical display.

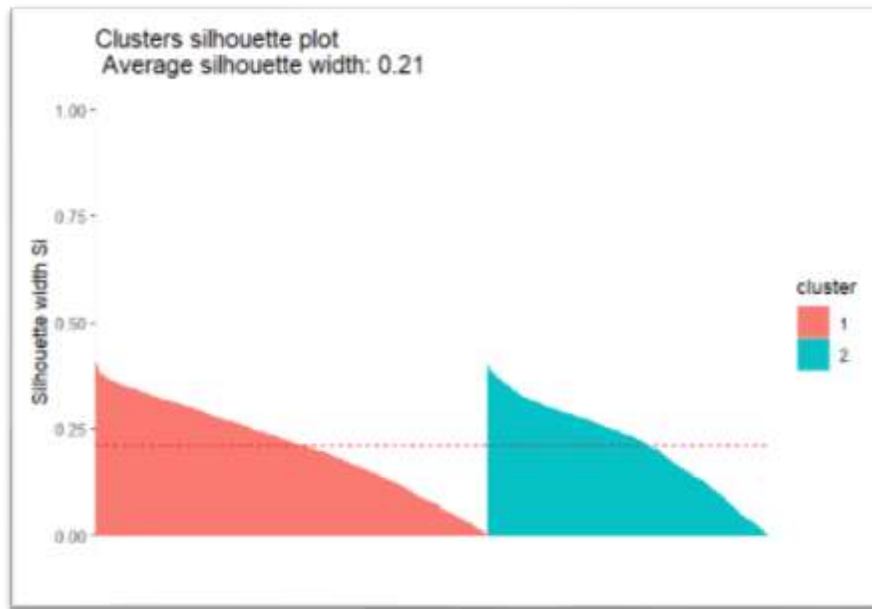


Figure 33: Plot average silhouette width for 2 clusters

Furthermore, the script determines the average silhouette width by calculating the mean of the silhouette widths returned by the `silhouette()` function. This measure provides a general evaluation of the quality of clustering. Figure 34 shows the average silhouette width.

```
# Calculate the average silhouette width
average_sil_width <- mean(sil_widths[, "sil_width"])
cat("Average silhouette width:", average_sil_width, "\n")
Average silhouette width: 0.2107027
```

Figure 34: Output for the calculation of the average Silhouette width

A silhouette width score of 0.21 is considered an average score, indicating moderate cluster quality. This means that the elements within each cluster are somewhat more similar to each other than to members of different clusters, but not distinctly so. While this level of score

suggests fair internal cohesion and moderate separation from each other, it may result in potential overlaps or ambiguities in cluster boundaries. While this score is satisfactory, there is scope for improvement in the clustering configuration. It may be worthwhile to adjust the number of clusters, the clustering algorithm, or the dataset features used.

1.2. 2nd Subtask Objectives

1.2.1. PCA Analysis

PCA, short for Principal Component Analysis, is a widely used technique for reducing the dimensionality of large data sets. By transforming a multitude of variables into a smaller set that retains most of the information from the original large set, PCA allows for simplified analysis. While reducing variables can compromise accuracy, dimensionality reduction is often a matter of balancing accuracy and simplicity. Smaller data sets are more manageable and easier to visualize, allowing machine learning algorithms to process data more quickly and efficiently without extraneous variables.

```
# Load libraries
library("factoextra")
library("cluster")

# Apply PCA to the scaled data
pca_result <- prcomp(scaled_data, scale. = TRUE)
pca_summary <- summary(pca_result)
print(pca_summary)
```

Figure 35: Code for applying PCA

This code performs Principal Component Analysis (PCA) on scaled data using R libraries. Firstly, the factoextra and cluster libraries are loaded, providing functions for visualizing multivariate analysis results and performing clustering, respectively. The scaled data is subsequently fed into the prcomp function, which then does the PCA. By ensuring that every variable has a mean of 0 and a standard deviation of 1, scaling makes it possible to compare variables in a meaningful way. scale. = TRUE denotes the need to scale the data before doing PCA. Then, with the summary

function, a summary of the PCA results is produced, combining details like the eigenvalues and the percentage of variance accounted for by each principal component. This summary directs decisions for additional analysis or visualization and helps interpret how each principal component affects the variation in the dataset. Following Figure 36 shows the summary.

```
> print(pca_summary)
Importance of components:
      PC1      PC2      PC3      PC4      PC5      PC6      PC7      PC8      PC9      PC10
Standard deviation 1.8671 1.2585 1.1112 1.02721 0.98428 0.88512 0.82705 0.73733 0.59750 0.53807
Proportion of Variance 0.3169 0.1440 0.1123 0.09592 0.08807 0.07122 0.06218 0.04942 0.03245 0.02632
Cumulative Proportion 0.3169 0.4609 0.5731 0.66906 0.75713 0.82835 0.89054 0.93996 0.97241 0.99873
      PC11
Standard deviation 0.11800
Proportion of Variance 0.00127
Cumulative Proportion 1.00000
```

Figure 36: Output of the PCA summary

1.2.2. Find Eigenvalues and Eigenvectors

Understanding the concepts of eigenvalues and eigenvectors is essential for developing efficient and high-performing machine learning models. An eigenvector is a vector that, when multiplied by a matrix, produces another vector that is parallel but scaled in size by a scalar multiple known as an eigenvalue. In simpler terms, eigenvalues represent the scaling factor by which a vector is transformed during a linear transformation. In essence, they are the values that scale eigenvectors when subjected to a linear transformation.

```
# Extract eigenvalues
eigenvalues <- pca_result$sdev^2
cat("Eigenvalues:", "\n")
print(eigenvalues)

# Extract eigenvectors
eigenvectors <- pca_result$rotation
cat("Eigenvectors:", "\n")
print(eigenvectors)
```

Figure 37: Code of the Eigenvalues and Eigenvectors in PCA analysis

The above code concentrates on extracting significant information from the preceding Principal Component Analysis (PCA) results. The eigenvalues, which represent the amount of variation explained by each primary component are first calculated. The eigenvalues are calculated by squaring the standard deviations of the main components that were retrieved from the

pca_result object. The code computes the eigenvalues and outputs the results to the console after printing a header stating that the output matches the eigenvalues.

The eigenvectors are then extracted from the PCA findings using the code in the second step. In essence, these eigenvectors show how each original feature contributes to the principal components by identifying the directions of the new feature space axes to the original features. The eigenvectors are obtained by accessing the rotation matrix kept in the pca_result object. The code publishes a header indicating that the output relates to eigenvectors and then shows the extracted eigenvectors on the console. Figures 38 and 39 show this.

```
> # Extract eigenvalues
> eigenvalues <- pca_result$sdev^2
> cat("Eigenvalues:", "\n")
Eigenvalues:
> print(eigenvalues)
[1] 3.48592520 1.58372958 1.23481797 1.05516524 0.96880534 0.78344137 0.68401665 0.54365122 0.35700453
[10] 0.28951996 0.01392294
```

Figure 38: R output for the extracted Eigenvalues

```
> #Extract eigenvectors
> eigenvectors <- pca_result$rotation
> cat("Eigenvectors:", "\n")
Eigenvectors:
> print(eigenvectors)
```

	PC1	PC2	PC3	PC4	PC5	PC6
fixed.acidity	-0.167222933	0.581766382	-0.07837124	0.08782062	0.244540913	0.12556870
volatile.acidity	-0.004245818	-0.054018935	0.66714092	0.19671990	0.497649018	-0.31466379
citric.acid	-0.115426208	0.358040455	-0.51157813	0.20547749	0.025088169	-0.29703956
residual.sugar	-0.409139181	-0.006499217	0.22937849	0.08296438	-0.070270313	0.47804484
chlorides	-0.312206182	-0.047829781	-0.05517313	-0.50728589	0.112069020	-0.57400288
free.sulfur.dioxide	-0.284942054	-0.269908676	-0.12033697	0.57512738	-0.265943475	-0.17478486
total.sulfur.dioxide	-0.395431313	-0.250416370	-0.01065505	0.34145145	0.024691344	-0.23678611
density	-0.498406888	-0.010947663	0.05232575	-0.15183260	0.035433532	0.30165361
pH	0.134545450	-0.581507369	-0.21767298	-0.17017242	-0.008397389	0.08398994
sulphates	-0.036456589	-0.249918964	-0.40772194	0.07501558	0.771982282	0.23266386
alcohol	0.434083864	0.037873205	0.02142389	0.37702259	0.076158438	0.01347007

	PC7	PC8	PC9	PC10	PC11
fixed.acidity	-0.26933104	-0.62918726	-0.167133735	-0.13495937	0.17204234
volatile.acidity	0.30546504	-0.07952925	0.106401206	-0.23571700	0.01014216
citric.acid	0.66388955	0.10212752	0.103564122	-0.03114881	0.01241032
residual.sugar	0.32385596	0.21860791	-0.391498659	0.13002984	0.46556360
chlorides	-0.10489359	0.08783697	-0.526211516	0.03799959	0.02745444
free.sulfur.dioxide	-0.24407411	0.05014801	-0.197926571	-0.14543174	-0.02591255
total.sulfur.dioxide	-0.13198759	-0.23924931	0.230098802	0.69835151	0.04314090
density	0.16288491	-0.10125700	0.021320636	-0.07321099	-0.76865485
pH	0.36273561	-0.61813385	-0.110585213	-0.12240171	0.13929770
sulphates	-0.18031820	0.28049915	0.007567217	-0.03736300	0.04012611
alcohol	0.08509695	-0.03290816	-0.644195977	0.31618482	-0.37183937

Figure 39: R output for the extract Eigenvectors

```
#plot of scree plot  
plot(pca_result,type="lines",main="Scree Plot")
```

Figure 40: Code for the scree plot

Using the `pca_result` object, the above code (Figure 40) snippet creates a scree plot based on the results of Principal Component Analysis (PCA). The plot is set up with particular parameters using R's `plot()` function. `type = "lines"` specifies a line plot, and `main = "Scree Plot"` sets the plot's main title.

A graphical representation of the variance explained by each principal component is provided by the scree plot. It assists in determining the importance of every primary component and the number of principal components required to properly represent the data. The plot typically shows the eigenvalues (variances) of each principal component plotted against the number of principal components. This visualization aids in identifying the optimal number of principal components to retain for further analysis by highlighting the point where the eigenvalues begin to stabilize. Figure 41 represents the scree plot.

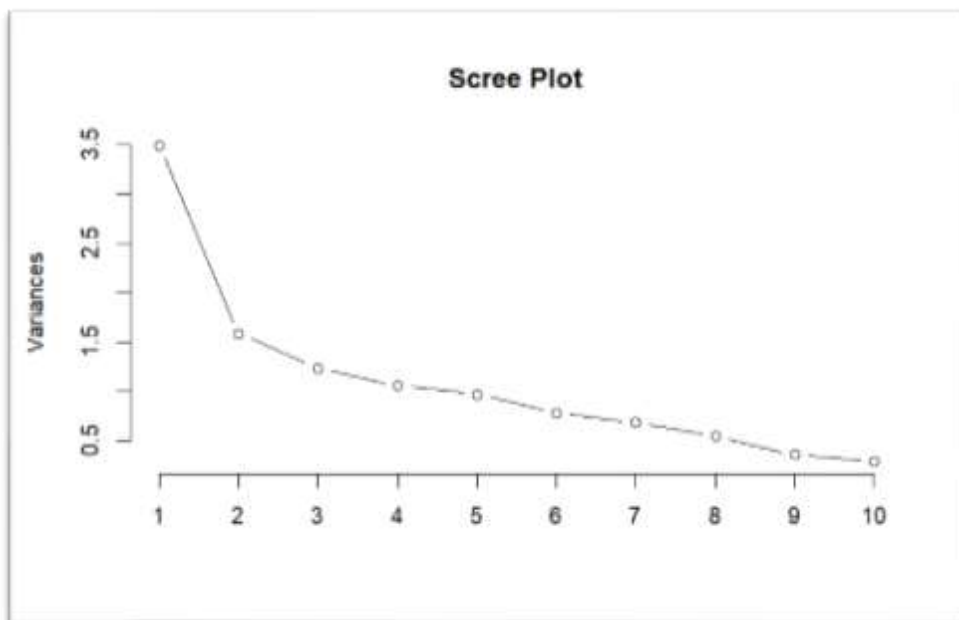


Figure 41: Scree plot (plot the eigenvalues of the principal components)

1.2.3. Find the cumulative Score for Principal Components

Principal components (PCs) that capture a certain portion of the variance in the original dataset. The eigenvalues of each PC represent the amount of variance captured. A scree plot is used to visualize the magnitude of each eigenvalue, which helps to determine the number of principal components to retain. However, it's essential to also consider the cumulative variance explained by each PC, which is the sum of all PC variances up to that point. This cumulative variance is an important metric for understanding the dataset's structure and deciding how many principal components are required to capture a significant portion of the variance, which can aid in dimensionality reduction and feature selection processes.

```
# Calculate cumulative score per principal component (PC)
cumulative_var <- cumsum(pca_result$sdev^2 / sum(pca_result$sdev^2))
cat("Cumulative Variance Explained:\n")
print(cumulative_var)
```

Figure 42: Code for find the cumulative score for principal components

This R code snippet uses a Principal Component Analysis (PCA) on a dataset to determine the cumulative variance explained by each principal component. The 'sdev' attribute of the PCA result object called 'pca_result' is where the standard deviations of the principal components are first accessed. Since variance is equal to the square of the standard deviation, each standard deviation is squared using the '^2' operator. Next, use 'sum(pca_result\$sdev^2)' to calculate the total variance across all principal components by summing these squared standard deviations. Next, the proportion of variance explained by each principal component is calculated by dividing each squared standard deviation by the total variance. Then,, the cumulative variance explained by successfully adding more principal components is revealed by computing the cumulative sum of these proportions using the 'cumsum()' function. Lastly, use cat() function to print the header and print() function use to display the cumulative variance values calculated earlier. Figure 43 illustrates the cumulative cores per principal components.

```
# Calculate cumulative score per principal component (PC)
cumulative_var <- cumsum(pca_result$sdev^2 / sum(pca_result$sdev^2))
cat("Cumulative Variance Explained:\n")
print(cumulative_var)
[1] 0.3169023 0.4608777 0.5731339 0.6690580 0.7571312 0.8283532 0.8905365 0.9399593 0.9724143 0.9987343 1.0000000
```

Figure 43: R output for cumulative scores per principal components

1.2.4. Selecting Principal Components with cumulative value>85%

```
# Create a new dataset with principal components as attributes
pc_data <- predict(pca_result)

# Choose principal components providing at least cumulative score > 85%
num_components <- which(cumulative_var > 0.85)[1]
cat("Number of components selected (85% variance):", num_components, "\n")

# Select the first 'num_components' principal components
pc_data_selected <- pc_data[, 1:num_components]
cat("Selected Principal Component Data:\n")
print(pc_data_selected)

# Determine the number of clusters for k-means on transformed dataset
fviz_eig(pca_result, addlabels = TRUE, main="Scree Plot")
```

Figure 44: Code for selecting Principal components with cumulative score>85%

By converting the original dataset into a new one where principal components act as attributes, this section of R code expands on the Principal Component Analysis (PCA) method. It begins by creating this altered dataset, called `pc_data`, based on the PCA result kept in `pca_result`, using the `predict()` function. Then it determines the number of principal components required to explain at least 85% of the cumulative variance. To accomplish this, find the index of the `cumulative_var` vector's first element where the cumulative variance is more than 85%. The `which()` function returns the index, and `[1]` is used to select the first element if multiple components meet the criterion. Then, it prints the number of selected components. The number of selected components is 7.

```
> num_components <- which(cumulative_var > 0.85)[1]
> cat("Number of components selected (85% variance):", num_components, "\n")
Number of components selected (85% variance): 7
# Select the first 'num_components' principal components
```

Figure 45: R output for the Number of selected components

It selects the first `num_components` principal components from the transformed dataset `pc_data` and stores them in `pc_data_selected`. This subset of principal components will be used for further analysis.

```

cat("Selected Principal Component Data:\n")
selected_principal_component_data:
print(pc_data_selected)

```

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
[1.]	-0.996583683	1.4464380832	1.576182e+00	-0.5894632495	-0.452687183	0.6778666939	0.0714893936
[2.]	0.818677326	1.6873305566	1.033146e+00	0.6179964292	1.372233514	-1.1189092693	1.4009678302
[3.]	-1.576068885	-0.2925788243	9.683993e-01	-1.4134158137	0.919831710	-1.7130376409	0.0007197959
[4.]	0.818677326	1.6873305566	1.033146e+00	0.6179964292	1.372233514	-1.1189092693	1.4009678302
[5.]	-0.996583683	1.4464380832	1.576182e+00	-0.5894632495	-0.452687183	0.6778666939	0.0714893936
[6.]	-2.321490055	1.6250334483	-3.085901e-01	0.9775440615	-0.444282189	-0.1987523341	1.3594616671
[7.]	0.049235519	1.3363099343	4.052727e-01	1.1035539204	-0.726447440	-0.3212307840	-1.5330995155
[8.]	-0.022622612	0.2957634740	-8.495758e-02	1.5176707797	-2.311342699	-0.0622612523	0.5000450289
[9.]	-4.289241105	0.8678646177	1.825028e-01	3.0142584623	2.858803638	0.2420158021	1.2865557701
[10.]	-1.069466933	3.0787398758	-2.714540e+00	-0.4277419878	1.100421144	-0.1792259477	-0.4494816490
[11.]	-0.095940606	3.5868044675	-8.675697e-01	0.3768395721	0.487600391	-0.7063259450	0.0477454963
[12.]	-0.718483822	2.3018926821	-1.097452e-01	-0.2888418780	1.279781981	-0.7519811429	-0.3501629715
[13.]	0.867622169	-0.5011640487	-1.747651e+00	-0.2421861944	0.497663801	0.3594441638	-0.5558278973
[14.]	-2.257433785	0.2644473836	-7.630858e-01	1.7713666135	-1.245419255	-1.1938113713	0.4794445580
[15.]	0.233097375	-2.3270318285	-1.478340e+00	-1.2522732982	0.7798185468	-1.3122454738	-0.9444325975
[16.]	-0.271571785	3.3006747931	-7.63696e-01	-1.0754786275	0.114570811	-0.4699172532	-0.9893993201
[17.]	-0.291332392	3.0815425602	-4.797321e-01	-0.8129481550	0.037281021	-0.4800643797	-1.0505987157
[18.]	-1.519358276	-1.3700050962	-8.084888e-01	0.2901427962	-0.413241720	-0.2366508893	-0.7083872080
[19.]	-1.519358276	-1.3700050962	-8.084888e-01	0.2901427962	-0.413241720	-0.2366508893	-0.7083872080
[20.]	1.114917620	1.3768949277	1.008591e+00	0.2883321378	0.591861382	-0.390210924	-0.1312698310
[21.]	-1.519358276	-1.3700050962	-8.084888e-01	0.2901427962	-0.413241720	-0.2366508893	-0.7083872080
[22.]	-0.691387375	-0.4536297087	-1.298830e+00	-0.4389685630	-1.321182554	0.2878630920	-0.8609550341
[23.]	0.832717854	2.0534625464	-7.034119e-01	0.9649864477	-0.730427623	-0.8227675590	-1.0363633827
[24.]	0.832717854	2.0534625464	-7.034119e-01	0.9649864477	-0.730427623	-0.8227675590	-1.0363633827
[25.]	-0.532851089	0.7711682250	5.905817e-01	-2.1711391850	0.246642091	-0.2026766538	-0.8601308715
[26.]	-3.611321599	1.3015130906	1.610498e+00	0.1034752556	1.478852992	-1.8173433896	0.9707280223
[27.]	0.577168568	-2.4264432165	-4.938672e-01	-1.1342813163	1.294773834	0.1994872237	0.7489653762
[28.]	0.748803479	4.1908016200	4.625013e-01	-0.2113625150	0.532780488	0.0467278353	0.6068613801
[29.]	2.031591951	-1.3491183947	2.604435e+00	-1.8398349207	-0.355367568	0.0810423086	-0.1128906086
[30.]	0.761868075	1.2658404805	-8.315449e-01	-0.0207710436	1.249775396	-1.4064814036	1.4201711436
[31.]	1.134448337	-1.5429280724	-5.149751e-01	-2.2014840053	-1.117250037	-0.2971407104	-0.4302573708
[32.]	0.692200505	1.0216773995	-1.373418e+00	0.9224874242	0.559576063	-0.9609598679	0.0978082384
[33.]	1.057921318	0.2344687580	-1.272103e+00	-1.7284114696	-0.414660103	-0.8845812098	1.3625513728
[34.]	1.057921318	0.2344687580	-1.272103e+00	-1.7284114696	-0.414660103	-0.8845812098	1.3625513728
[35.]	0.692200505	1.0216773995	-1.373418e+00	0.9224874242	0.559576063	-0.9609598679	0.0978082384
[36.]	-1.061322928	0.2298350337	6.958412e-01	-2.0166057896	0.010039519	1.5813983492	0.1726831372

Figure 46: R output for the transformed after calculating the cumulative value>85%

1.2.5. The discussion about selected principal components

The function called `fviz_eig()` to visualize the Scree Plot, which displays the eigenvalues associated with each principal component. The `pca_result` object is passed to this function, and `addlabels = TRUE` indicates that labels should be added to the plot. The main argument sets the main title of the plot as "Scree Plot". Figure 47 shows this.

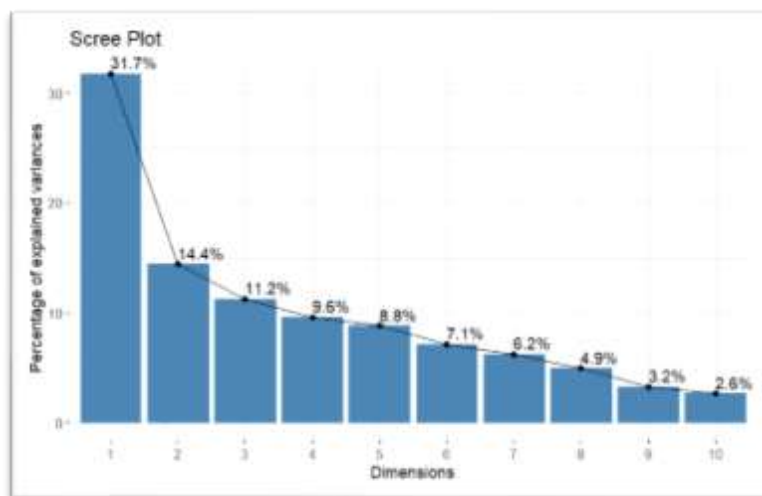


Figure 47: Scree plot for PCA analysis

When applying Principal Component Analysis (PCA) to the dataset, chose to decrease the number of principal components from 11 to 7. This approach guarantees that these components collectively represent over 85% of the variance. The decision was carefully considered to strike a balance between simplifying the data and retaining significant informational content. It's common for datasets with several correlated variables to experience issues related to complexity and overfitting, and reducing dimensionality helps address these challenges. By setting the cumulative variance retained at 85%, we ensure that the most influential aspects of the dataset are preserved, thus maintaining the integrity of any subsequent analyses, such as clustering or regression modeling. This threshold effectively simplifies the dataset, improving interpretability and reducing computational demands without sacrificing analytical validity. Our choice of retaining seven components also considers the diminishing returns on variance explained with additional components, optimizing the trade-off between accuracy and simplicity.

1.2.6. Finding the optimal number of clusters for PCA analyzed dataset using automated tools.

The same four automated techniques that were used in subtask 1 must be used here to identify the ideal number of clusters: Nbclust, elbow method, gap statistics, and silhouette method.

The output of NbClust() is displayed as a bar plot created with barplot(), which shows the ideal cluster count recommended by Nbclust for each of the evaluation criteria. The number of clusters is indicated by the x-axis, and the number of supporting criteria for each cluster count is indicated by the y-axis. The results and plots produced by choosing the optimal number of clusters using those indices are shown in Figures 48 and 49.

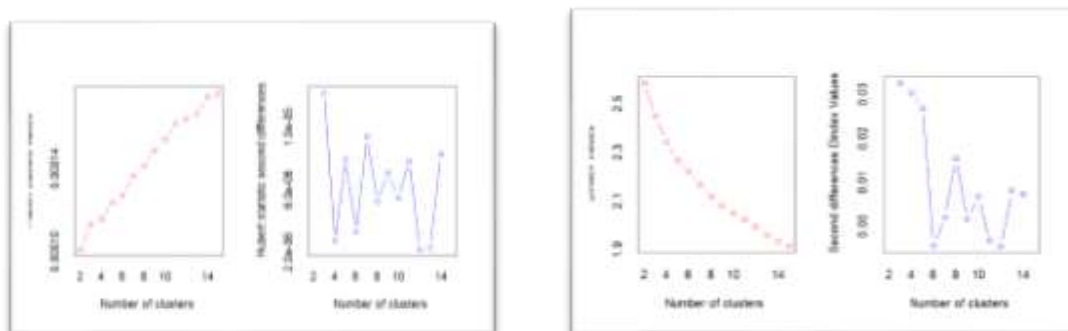


Figure 48: Plots for Hubert index and D index


```

> # NbClust
> set.seed(200)
> nbclust_results_pca <- NbClust(pc_data_selected, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans")
*** : The Hubert index is a graphical method of determining the number of clusters.
      In the plot of Hubert index, we seek a significant knee that corresponds to a
      significant increase of the value of the measure i.e the significant peak in Hubert
      Index second differences plot.

*** : The D index is a graphical method of determining the number of clusters.
      In the plot of D index, we seek a significant knee (the significant peak in D index
      second differences plot) that corresponds to a significant increase of the value of
      the measure.

*****
* Among all indices:
* 11 proposed 2 as the best number of clusters
* 5 proposed 3 as the best number of clusters
* 2 proposed 4 as the best number of clusters
* 2 proposed 5 as the best number of clusters
* 1 proposed 7 as the best number of clusters
* 1 proposed 13 as the best number of clusters
* 1 proposed 14 as the best number of clusters
* 1 proposed 15 as the best number of clusters

***** Conclusion *****

* According to the majority rule, the best number of clusters is: 2

*****
> cat("NbClust on PCA data suggests", which.max(table(nbclust_results_pca$best.n[1,])), "clusters.\n")
NbClust on PCA data suggests 2 clusters.

```

Figure 49: Output of NbClust

Below plot is the NbClust result after PCA.

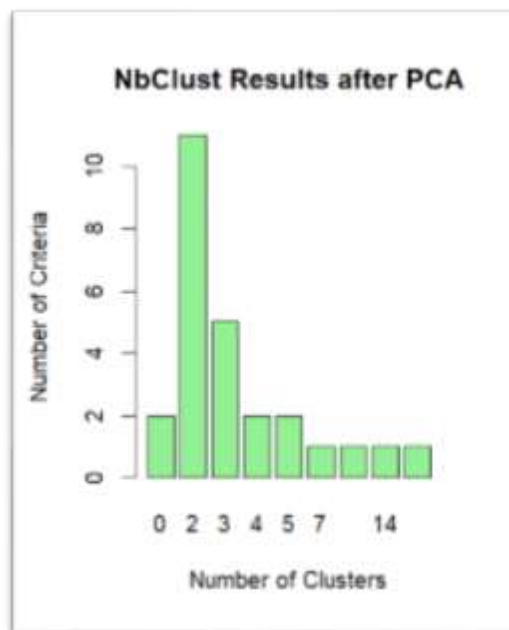


Figure 50: Plot of NbClust result after PCA

The factoextra package's `fviz_nbclust()` function creates a plot in the elbow method that uses the Within-Cluster Sum of Squares (WSS) method to get the ideal number of clusters.

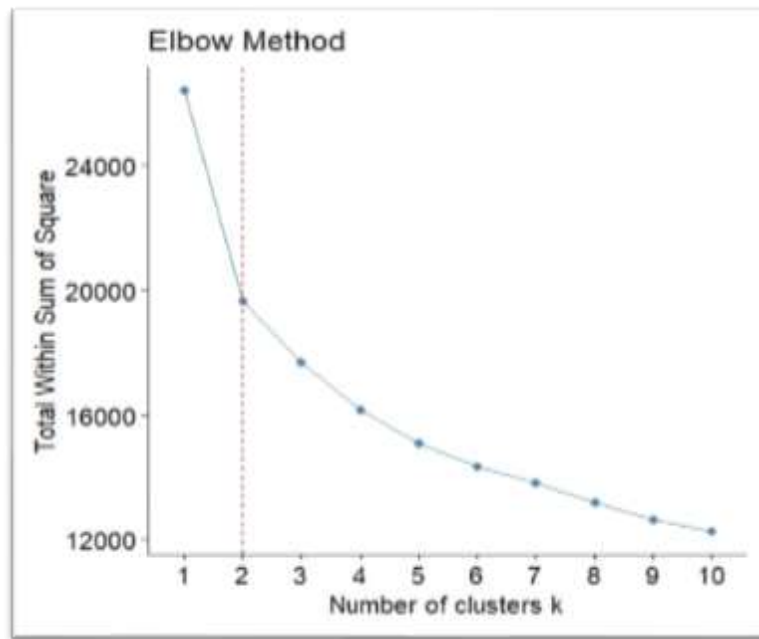


Figure 51: Plot for the elbow method

The silhouette approach is used to calculate the ideal number of clusters using the `fviz_nbclust()` function. Moreover, a graph with its peak on 2 is printed.

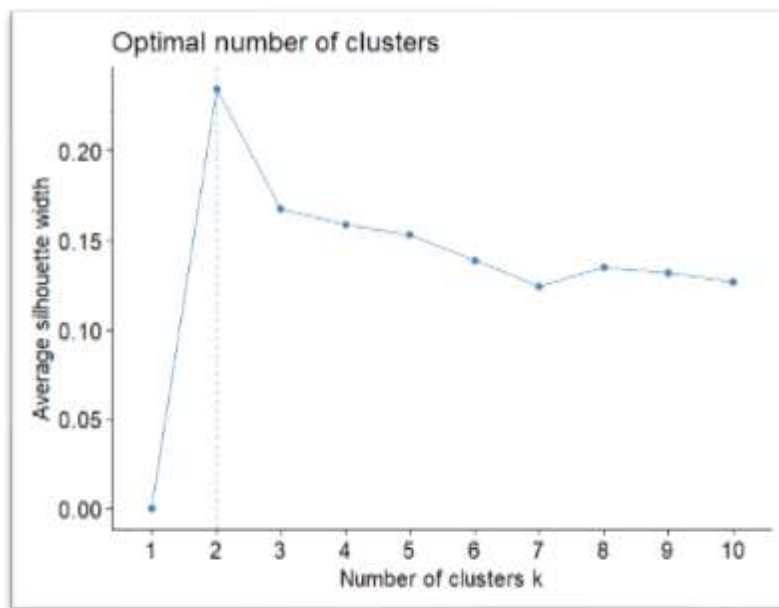


Figure 52: Plot for the Silhouette method

To visually represent the gap statistic values in the gap statistics approach, the `clusGap()` function computes the gap statistic values and obtains the optimal cluster number from the `fviz_gap_stat()` function.

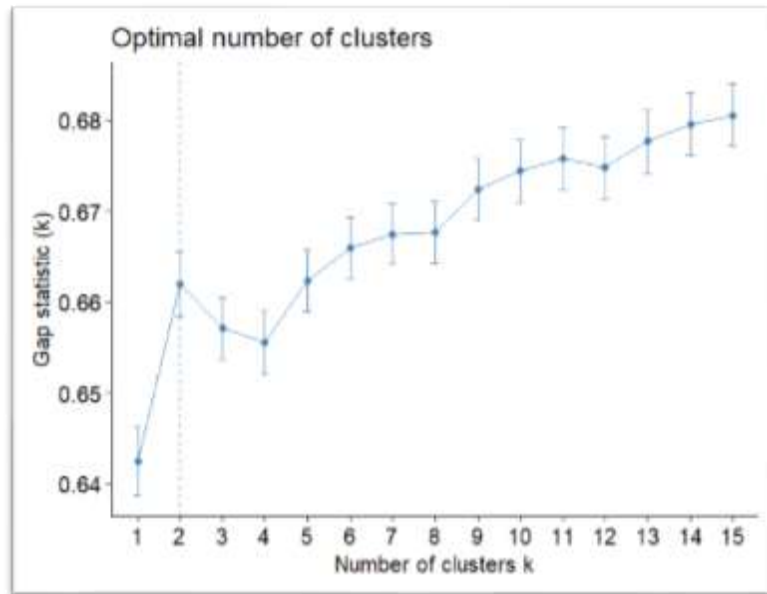


Figure 53: Plot for the gap statistics method

```
> # Gap statistics
> set.seed(200)
> gap_stat_pca <- clusGap(pc_data_selected, FUN = kmeans, nstart = 25, K.max = 15, B = 500)
Clustering k = 1,2,..., K.max (= 15): .. done
Bootstrapping, b = 1,2,..., B (= 500) [one "." per sample]:
..... 50
..... 100
..... 150
..... 200
..... 250
..... 300
..... 350
..... 400
..... 450
..... 500
There were 50 or more warnings (use warnings() to see the first 50)
> print(gap_stat_pca)
Clustering Gap statistic ["clusGap"] from call:
clusGap(x = pc_data_selected, FUNcluster = kmeans, K.max = 15, B = 500, nstart = 25)
B=500 simulated reference sets, k = 1..15; space0="scaledPCA"
--> Number of clusters (method "firstSEmax", SE.factor=1): 2
      loglik  E.loglik  gap      SE.sim
[1,] 7.957853 8.600334 0.6424807 0.003784595
[2,] 7.811352 8.473312 0.6619593 0.003488467
[3,] 7.757480 8.424592 0.6571125 0.003353574
[4,] 7.712014 8.367596 0.6555820 0.003537254
[5,] 7.678491 8.340843 0.6623526 0.003422463
[6,] 7.650271 8.316267 0.6659960 0.003390411
[7,] 7.627505 8.294986 0.6674816 0.003325204
[8,] 7.607102 8.274803 0.6677012 0.003424087
[9,] 7.586597 8.259008 0.6724110 0.003418071
[10,] 7.569441 8.243880 0.6744392 0.003442056
[11,] 7.553703 8.229457 0.6757535 0.003413760
[12,] 7.540080 8.215775 0.6747949 0.003390306
[13,] 7.526422 8.204318 0.6776961 0.003408759
[14,] 7.513579 8.193200 0.6796211 0.003442869
[15,] 7.502250 8.182853 0.6806029 0.003444603
> fviz_gap_stat(gap_stat_pca)
```

Figure 54: R output for the gap statistics method

According to all automated tools the best k value is 2.

1.2.7. Perform K-means analysis using the most favored k value

The k-means clustering on a dataset that has been transformed by Principal Component Analysis (PCA), with the aim of identifying unique groups within the data. To ensure reproducibility, the script sets a seed for random number generation, which is crucial given the random nature of the initial cluster center assignments in the k-means algorithm. The script then initiates the k-means clustering using `means()`, targeting two clusters (`centres = 2`), and enhances the likelihood of avoiding local optima by initializing the process 50 times (`start = 50`). This approach is intended to produce the most stable and accurate clustering outcome by selecting the best solution among multiple initializations based on the within-cluster sum of squares. After the clustering, the script employs `fviz_cluster()` from the `factoextra` package to visualize the results. This function generates a scatter plot in which each data point is colored based on its cluster assignment, providing a clear visual representation of how well the data segments into clusters. The visualization is useful for verifying the effectiveness of the clustering process and confirming whether the data points are grouped into coherent and distinct clusters as expected, thereby validating the choice of two clusters for this analysis.

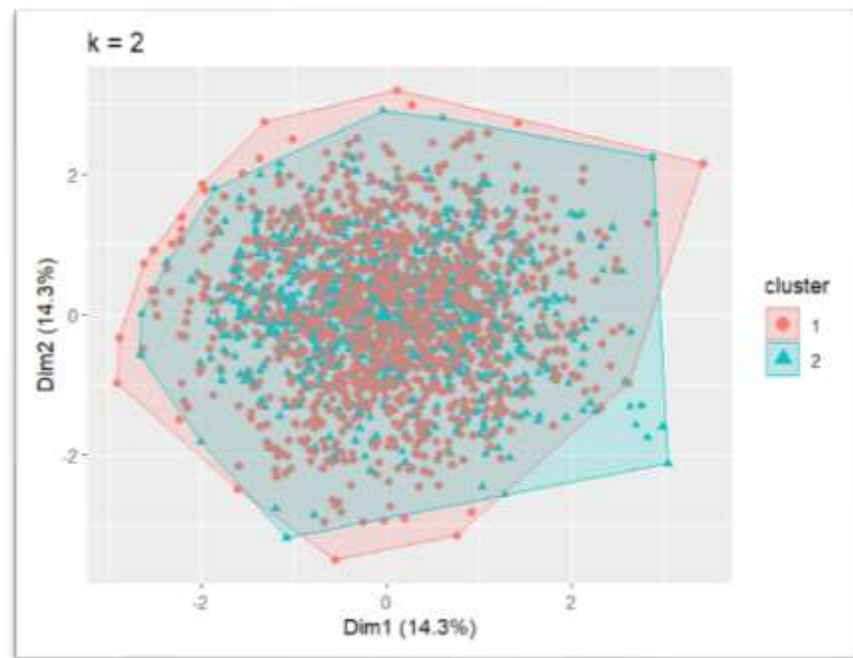


Figure 55: Graphical representation cluster plot for K=2

Details of the two cluster centers are printed in the console.

```

> # Display k-means results
> print(kmeans_pca_2$centers)
      PC1      PC2      PC3      PC4
1  1.332416  0.08863322 -0.07109089 -0.04036372
2 -1.868227 -0.12427577  0.09967904  0.05659539
      PC5      PC6      PC7
1  0.002061370 -0.009228952 -0.03760319
2 -0.002890319  0.012940239  0.05272476

```

Figure 56: Details of the 2 cluster centers

The next step is calculating the WSS and BSS.

Following the determination of the most favored k value, calculations for the Within-Cluster Sum of Squares (WSS), Between-Cluster Sum of Squares (BSS), Total Sum of Squares (TSS), and the BSS/TSS ratio are conducted. The BSS represents the summed distances between cluster centers and the overall mean, encapsulating the variation explained by clustering. TSS, on the other hand, quantifies the entire variation by summing the squared distances between each data point and the overall mean. The BSS/TSS ratio provides insight into the percentage of total variation attributed to clustering. WSS measures the unexplained variance by summing the squared distances between each data point and its assigned cluster center.

For the determined k value, the TSS is calculated as 26439.14, the BSS/TSS ratio as 0.25650, and the WSS as 19657.42. This console output summarizes the computations conducted to assess the efficacy of the clustering approach.

```

> cat("Total Sum of Squares (TSS):", total_ss_pca, "\n")
Total Sum of Squares (TSS): 26439.14
> cat("Between-cluster Sum of Squares (BSS)/TSS Ratio:", bss_tss_ratio_pca, "\n")
Between-cluster Sum of Squares (BSS)/TSS Ratio: 0.2565032
> cat("within-cluster Sum of Squares (WSS):", kmeans_pca_2$tot.withinss, "\n\n")
within-cluster Sum of Squares (WSS): 19657.42
>

```

Figure 57: Values for WSS, TSS, and BSS/TSS

1.2.8. Silhouette plot for the most favored k value

This task involves creating a silhouette plot for the new k-means clustering performed on the PCA-transformed dataset. The silhouette plot illustrates the proximity of each data point within its cluster to points in neighboring clusters. Figure 58 demonstrates the visualization of these cluster characteristics through the silhouette plot.

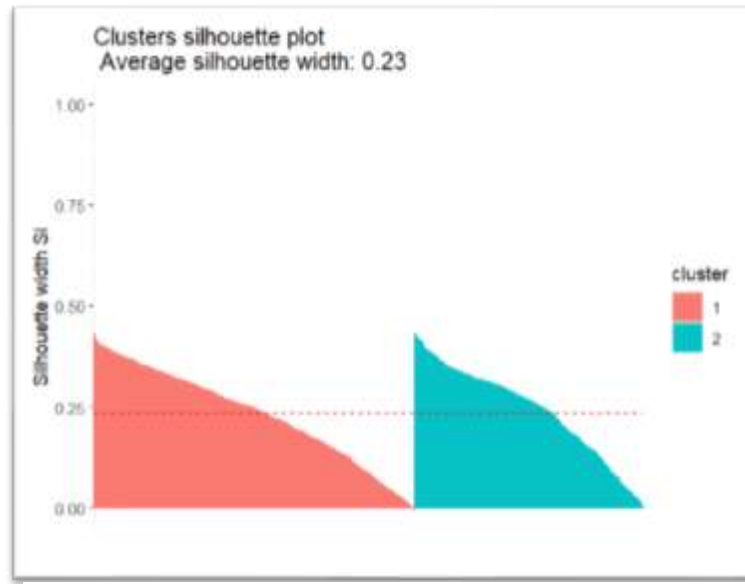


Figure 58: Clusters silhouette plot

The `fviz_silhouette` function from the `factoextra` package can be used to visualize the silhouette widths of clusters and assess a clustering algorithm's efficacy. After principal component analysis (PCA) and clustering on the converted data, this function produces a silhouette plot that shows the widths of the silhouettes for each data point. The silhouette widths for each data point are contained in the `sil_widths_pca_2` data frame or matrix, and the graphic helps determine how well each object fits within its cluster. A perfect fit is shown by silhouette values near +1, whereas a bad fit is indicated by values near -1. The `mean()` function yields an average width that is used to calculate the mean of the silhouette widths. The single score is essential for assessing the overall performance of the clustering process since it provides a measure of how tightly clustered the data are throughout the clustering solution. The average silhouette width is printed to the console.

```

>
> # Visualize the silhouette plot
> silhouette_plot_pca_2 <- fviz_silhouette(sil_widths_pca_2)
  cluster size ave.sil.width
1         1 1576         0.23
2         2 1124         0.24
> print(silhouette_plot_pca_2)
>
> # Calculate average silhouette width
> average_sil_width_pca_2 <- mean(sil_widths_pca_2[, "sil_width"])
> cat("Average silhouette width:", average_sil_width_pca_2, "\n")
Average silhouette width: 0.2347616
>

```

Figure 59: R output of the average silhouette width

The average silhouette width score of 0.23 provides insight into the similarity of objects within their respective clusters. This number, which goes from -1 to +1, represents how well an object fits within its cluster to nearby clusters. A lower number indicates a moderate degree of separation across clusters, whereas a larger value indicates that the object is well-matched to its cluster. In this instance, the dataset's score of 0.23 is on the lower end of the spectrum, suggesting that although there are categories within it, they are not very distinct or cohesive. This can be the result of significant overlaps across clusters or variations in individual data points within a cluster.

1.2.9. Calinski-Harabasz Index

For a given number of clusters, the Calinski-Harabasz index is a statistic used to evaluate the split quality by a K-Means clustering technique. The ratio of the total inter-cluster and between-cluster dispersion for each cluster is used to calculate the index. The algorithm performs better the higher the score. In essence, a high CH score denotes better clustering because the clusters themselves are more widely spaced apart and the observations within each cluster are more densely packed.

```

library(fpc) # for calinhara function

# Function to visualize Calinski-Harabasz Index
fviz_ch <- function(pc_data_selected, max_clusters = 10) {
  ch_scores <- numeric(max_clusters)
  for (i in 2:max_clusters) {
    km <- kmeans(pc_data_selected, centers = i, nstart = 25)
    ch_scores[i] <- calinhara(pc_data_selected, km$cluster, cn = max(km$cluster))
  }
  ch_scores <- ch_scores[2:max_clusters]
  k <- 2:max_clusters

  par(mar = c(5, 4, 4, 4) + 0.1)
  plot(k, ch_scores, xlab = "Cluster number k",
       ylab = "Calinski - Harabasz Score",
       main = "Calinski - Harabasz Plot", cex.main = 1,
       col = "dodgerblue", cex = 0.9,
       lty = 1, type = "o", lwd = 1, pch = 4,
       bty = "n", las = 1, cex.axis = 0.8, tcl = -0.2)
  abline(v = which.max(ch_scores) + 1, lwd = 1, col = "red", lty = "dashed")
}

# Use this function on the PCA-selected data
fviz_ch(pc_data_selected)

```

Figure 60: Code for Calinski-Harabasz index

The R code defines a function called `fviz_ch`, which has been designed for visualizing the quality of clustering in terms of separation and compactness, as indicated by the Calinski-Harabasz Index. To access the `calinhara()` function, which is used to compute the index, the code loads the necessary `fpc` library first. The `fviz_ch` function takes two arguments: `pc_data_selected`, representing the PCA-selected data, and `max_clusters`, which has a default value of 10 and sets the upper limit for the number of clusters. Within the function, a loop is created to iterate over a range of potential cluster numbers, beginning from 2 up to the specified maximum. For each iteration, k-means clustering is executed on the PCA-transformed data with the current cluster number, and the Calinski-Harabasz Index is computed based on the resulting cluster assignments. The function then stores the scores obtained and generates a plot that displays the Calinski-Harabasz Scores against the number of clusters, excluding the first score, which corresponds to a single cluster. The plot is customized with labels, colours, and line types, and a vertical dashed line is added at the cluster number with the highest score, indicating the optimal number of clusters. The `fviz_ch` function is an excellent tool for visualizing how the Calinski-Harabasz Index changes with different cluster numbers, which helps determine the best clustering configuration for the PCA-selected data.

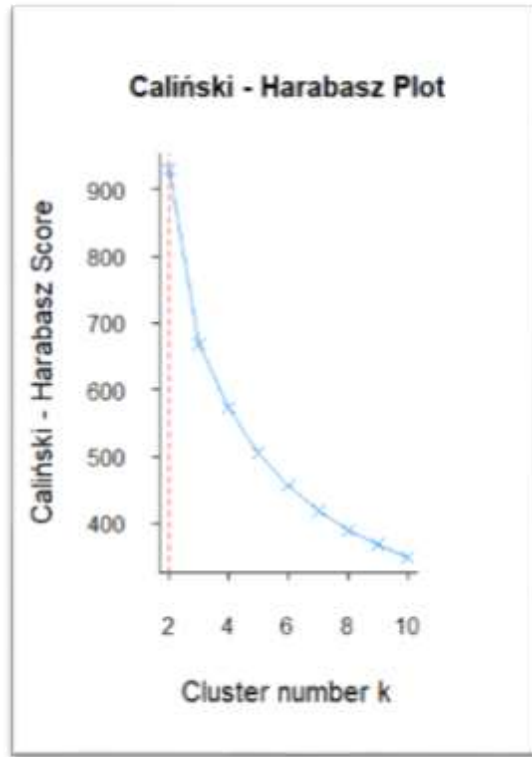


Figure 61: Calinski-Harabasz index plot

Based on the Calinski-Harabasz Plot, the number of clusters has a significant impact on the Calinski-Harabasz index values. The optimal Calinski-Harabasz index value is attained when there are two clusters, indicating that two clusters are the most suitable for discerning data. As the number of clusters grows, the Calinski-Harabasz index value decreases, implying that the clusters become less well-defined and may not be effective in isolating data.

2. Financial Forecasting Part

2.1. Multi-layer Perceptron Neural Network

2.1.1. Input Variables

One popular method is the use of Autoregressive (AR) Inputs, which involves using past values of the exchange rate to predict future values. This approach relies on the past behavior of the series to infer its future dynamics. For instance, the input vector might consist of lagged values of the exchange rate, such as $[x(t-1), x(t-2), \dots, x(t-p)]$, where p is the number of past periods included.

Moving Average and Exponential Smoothing methods smooth past data to reduce noise and highlight trends, which can be particularly useful in volatile markets like currency exchange. Input vectors might include the moving average of specific periods or values adjusted by exponential smoothing, providing a more stable input for the MLP to process.

Sentiment analysis has emerged as a useful tool for exchange rate forecasting. With advancements in text analytics, sentiment scores derived from financial news, social media, and other digital platforms can be used as inputs. These scores attempt to gauge the market sentiment, which can have a substantial short-term impact on currency movements.

Exchange rate forecasting using Multilayer Perceptrons (MLPs) requires careful selection of input variables to build effective models. There are several schemes and methods available for defining input vectors, each with its unique strengths and limitations.

2.1.2. Input Variables and I/O metrics for AR approach

To effectively use neural networks (NNs) for time-series analysis in the financial domain, especially in forecasting exchange rates, it is essential to define the input vector in a way that accurately captures the underlying patterns. The autoregressive (AR) approach is a widely accepted method for structuring these input vectors, but additional techniques can also be used to further supplement and enhance it. In this regard, let's discuss various methods to define the input vector, focusing on aspects specifically relevant to exchange rate forecasting.

Lagged Exchange Rates (AR approach):

This method involves using past exchange rate values as inputs to predict future values. The assumption here is that past behavior can provide insights into coming trends. For example, an NN model might use the exchange rate values of the previous ten days as inputs to predict the next day's rate.

Moving Averages:

A moving average input can smooth out short-term fluctuations and highlight longer-term trends in exchange rates. This can be particularly useful for eliminating noise and capturing the trend component in highly volatile data like exchange rates. For instance, inputs could include a 7-day or 30-day moving average of exchange rates.

Differencing:

To achieve stationarity in time-series data, first differencing (subtracting the previous value from the current value) or seasonal differencing can be used. For example, the input vector might include the first difference in the daily exchange rates to stabilize the mean of the time series.

2.1.3. Normalization

Normalization

Normalization of input data as preprocessing for artificial neural networks is an essential aspect of developing an effective model. By establishing a consistent scale across features, the neural network is equipped with a stable and efficient learning environment, which leads to more rapid convergence and an ability to generalize diverse datasets effectively. This practice also contributes to the stability of gradient descent and helps mitigate challenges related to weight initialization and activation function characteristics. Normalization is a crucial factor in enhancing the neural network's performance and has become a fundamental practice in modern deep-learning workflows. In essence, by normalizing the data to obtain a mean close to 0, the learning process is generally accelerated, resulting in faster convergence.

```
# Normalize function
normalize <- function(x) {
  x_min <- min(x, na.rm = TRUE)
  x_max <- max(x, na.rm = TRUE)
  (x - x_min) / (x_max - x_min)
}
```

Figure 62: Code for normalization

The above Figure describes the normalize function, which is intended to normalize a given vector. Taking into consideration any possible missing values (NA), the function first determines the minimum value (x_min) and the highest value (x_max) within the input vector x. Next, it applies the formula $(x - x_{\min}) / (x_{\max} - x_{\min})$ to normalize each member of the input vector x. To guarantee that the resultant values are scaled between 0 and 1, this method subtracts the minimum value from each element and divides by the vector's range.

```
original_data <- exchange_data$ExchangeRate
normalized_data <- normalize(exchange_data$ExchangeRate)
```

The variable "original_data" captures the raw exchange rates from your dataset, as retrieved via "exchange_data\$ExchangeRate". It does not undergo any transformations. On the other hand, "normalized_data" is created by applying the normalization function to the exchange rates. This function scales the data to a range usually between 0 and 1. The resulting normalized exchange rates are stored in the variable "normalized_data".

Denormalization

Denormalization is the process of intentionally adding redundancy to data to improve application performance and data integrity.

```
# Unnormalize function
unnormailize <- function(x, x_min, x_max) {
  x * (x_max - x_min) + x_min
}
```

In the above Figure, the included R code presents the unnormalized function, which is designed to return a normalized vector to its original scale once it has been normalized. Three parameters are accepted by this function: `x`, `x_min`, and `x_max`. In this case, `x` denotes the normalized vector, while `x_min` and `x_max` stand for the vector's minimum and maximum values before normalization, respectively. A single line captures the essence of the function: `x * (x_max - x_min) + x_min`. By multiplying each element of the normalized vector by the original range (the difference between `x_max` and `x_min`) and then adding the minimum value of the original vector back, this expression inversely transforms each element of the vector. It essentially returns the vector's values to their original scale by performing this.

The following figure showcases how the denormalization is used in the work.

```
# Test neural network model
test_nn <- function(nn_model, test_data, test_raw) {
  test_predictions <- compute(nn_model, test_data[, ncol(test_data)])
  predictions <- unnormailize(test_predictions$net.result, min(test_raw$OUTPUT), max(test_raw$OUTPUT))

  results <- data.frame(ACTUAL = test_raw$OUTPUT, PREDICTED = round(predictions, digits = 1))
  return(results)
}
```

This code introduces the `test_nn` function, designed to evaluate a neural network model's performance using test data. With three parameters `nn_model` for the trained neural network model, `test_data` for the dataset used in testing, and `test_raw` representing the original, unnormalized test data. The function proceeds to generate predictions using the `compute` function. This operation excludes the last column, typically holding the target variable. Following this, the function invokes the `unnormailize` function to convert predicted values from their normalized state back to the original scale, essential if the model's output was normalized during training. The unnormalized predictions are then rounded to one decimal place and compiled into a data frame alongside actual values from the raw test data. Ultimately, the function returns this

data frame, facilitating a direct comparison of actual and predicted values for assessing the model's predictive accuracy.

```
summary(original_data)
summary(normalized_data)
```

The above script prints a summary of the original data and is normalized in the R console.

```
> summary(original_data)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.206  1.287   1.308   1.305  1.327   1.417
> summary(normalized_data)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.0000 0.3853  0.4813  0.4702  0.5726  1.0000
```

The Figure, which provides summaries of the dataset before and after the normalization process, suggests changes in the dataset following normalization. Specifically, it indicates that the mean value of the dataset approaches 0 after normalization, while other statistics also undergo corresponding adjustments.



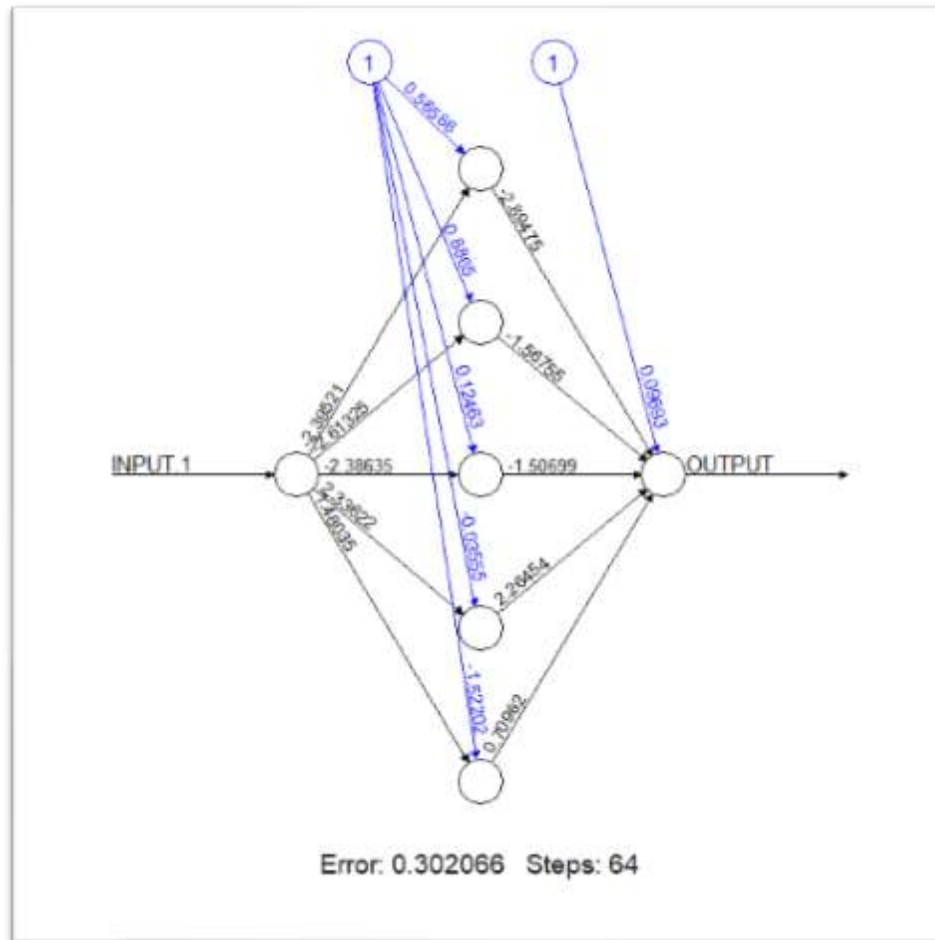
The boxplot comparing normalized and original data demonstrates the impact of normalization on the data distribution. It shows how normalization brings data points within the range of 0 to 1, ensuring uniformity across the dataset. With every data point standardized to the same scale, equal treatment is afforded to each, facilitating fair comparison and analysis.

2.1.4. Implementation of different MLPs for the AR approach

The below Figure contains the R code used for implementing different MLPs with different internal structures, input variables, and network parameters.

```
# Create models with different configurations
model_results <- list()
configurations <- list(
  list(lag=4, hidden=c(5)),
  list(lag=3, hidden=c(5)),
  list(lag=2, hidden=c(5)),
  list(lag=1, hidden=c(5)),
  list(lag=4, hidden=c(10,7)),
  list(lag=4, hidden=c(7)),
  list(lag=4, hidden=c(8,5)),
  list(lag=5, hidden=c(6)),
  list(lag=5, hidden=c(5,2)),
  list(lag=5, hidden=c(7,6)),
  list(lag=6, hidden=c(7)),
  list(lag=6, hidden=c(5,8)),
  list(lag=6, hidden=c(4,6)),
  list(lag=7, hidden=c(8)),
  list(lag=7, hidden=c(7, 5))
)
```

2.1.4.1. Implementing MLP for t-1 with one hidden layer with five neurons



The above MLP presents an enhanced performance summary of a Multilayer Perceptron (MLP) model. This model has undergone rigorous training to accurately predict a specific target variable from given input data. The model architecture consists of one input and one hidden layer with 5 neurons. The model assumes that the input data is lagged by a one-time step. This means that the model predicts the value of the target variable at a specific time, for instance, time t , based on the input data that was recorded at a previous time, in this case, time $t-1$.

The model's accuracy is stated to be 98.3393%, which indicates that 98.3393% of the time, the predictions made by the model come true. This is a performance indicator for the model that shows how well it can categorize the target variable using the input data.

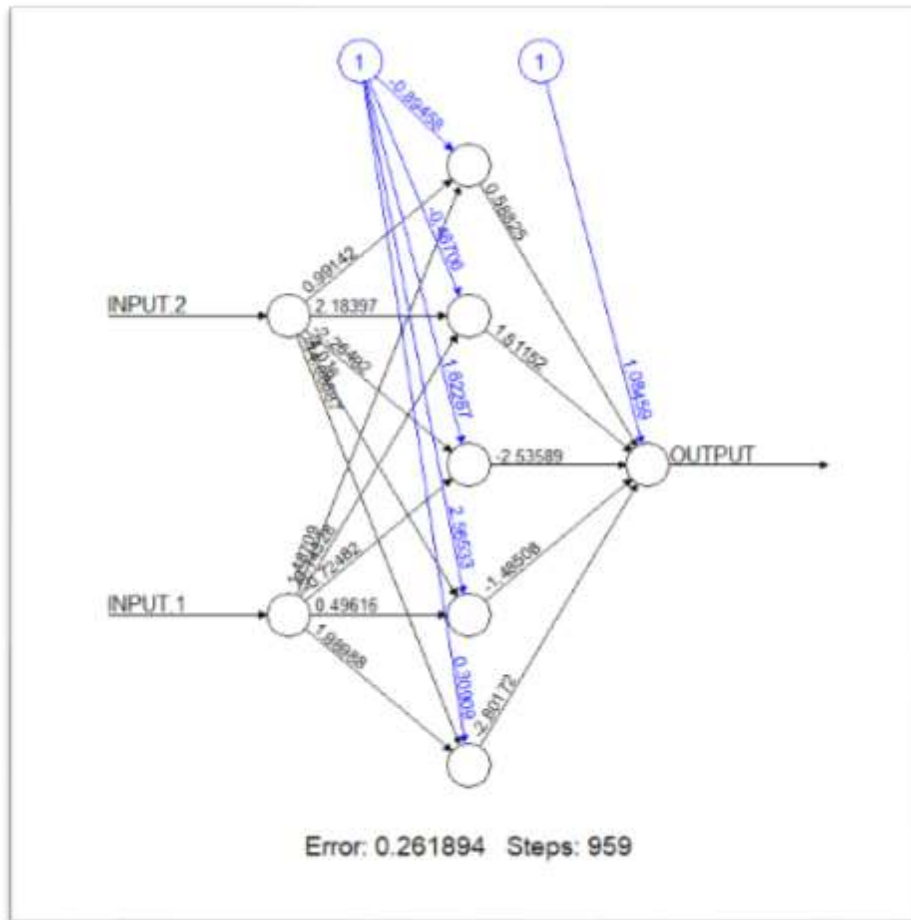
- The Root Mean Squared Error (RMSE) of the model is 0.0305

- The Mean Absolute Error (MAE) of the model is 0.0259
- The Mean Absolute Percentage Error (MAPE) of the model is 1.9424%
- The Symmetric Mean Absolute Percentage Error (SMAPE) of the model is 1.9665%

Based on the performance metrics, it appears that the MLP model with a single hidden layer and five neurons has performed admirably in forecasting the target variable using the input data. The model's notable accuracy and comparatively low RMSE, MAE, sMAPE, and MAPE values give confidence that the results are both accurate and dependable.

```
Model: Lag 1 Hidden 5  
RMSE: 0.0305  
MAE: 0.0259  
MAPE: 1.9424%  
sMAPE: 1.9665%  
Accuracy: 98.3393%
```

2.1.4.2. Implementing MLP for t-2 with one hidden layer with five neurons



The above MLP shows an enhanced performance summary of a Multilayer Perceptron (MLP) model. This model has undergone rigorous training to accurately predict a specific target variable from given input data. The model architecture consists of two inputs and one hidden layer with 5 neurons. The model assumes that the input data is lagged by a two-time step. This means that the model predicts the value of the target variable at a specific time, for instance, time t , based on the input data that was recorded at a previous time, in this case, time $t-2$.

The model's accuracy is stated to be 98.5408%, which indicates that 98.5408% of the time, the predictions made by the model come true. This is a performance indicator for the model that shows how well it can categorize the target variable using the input data.

- The Root Mean Squared Error (RMSE) of the model is 0.0299

- The Mean Absolute Error (MAE) of the model is 0.0256
- The Mean Absolute Percentage Error (MAPE) of the model is 1.9218%
- The Symmetric Mean Absolute Percentage Error (SMAPE) of the model is 1.9419%

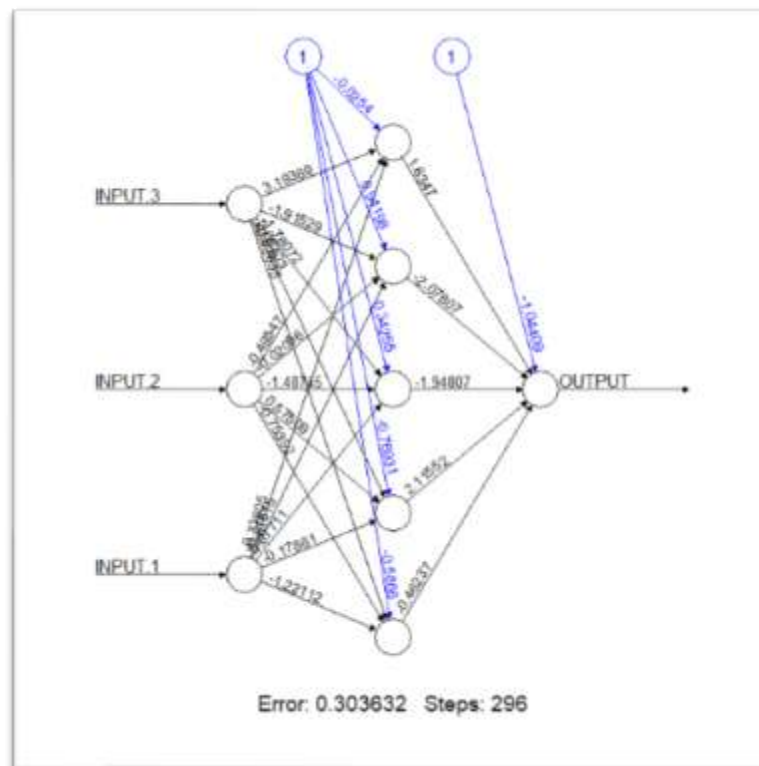
According to the performance metrics, the MLP model with one hidden layer and five neurons has performed admirably in forecasting the target variable using the input data. The model's high accuracy and comparatively low RMSE, MAE, MAPE, and sMAPE values, suggest that it can be considered highly accurate.

```

Model: Lag 2 Hidden 5
RMSE: 0.0299
MAE: 0.0256
MAPE: 1.9218%
sMAPE: 1.9419%
Accuracy: 98.5408%

```

2.1.4.3. Implementing MLP for t-3 with one hidden layer with five neurons

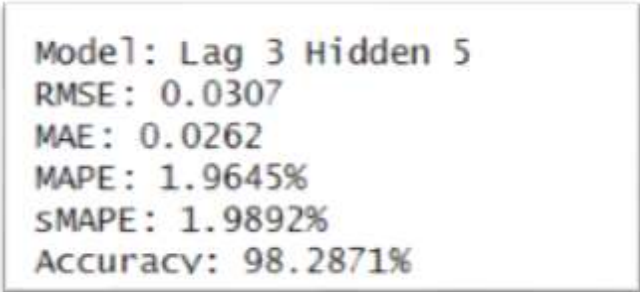


The provided figure illustrates the performance summary of an MLP (Multilayer Perceptron) model, diligently trained to predict a designated target variable from the given input data. This model structure encompasses three input nodes and a hidden layer containing 5 neurons. It operates under the assumption that input data is lagged by time steps, implying that the model forecasts the target variable's value at a time t based on input data recorded at time $t-3$.

The model's accuracy is reported as 98.2871%, signifying that its predictions align with the actual outcomes approximately 98.2871% of the time. This metric serves as a performance benchmark, reflecting the model's proficiency in classifying the target variable based on the provided input data.

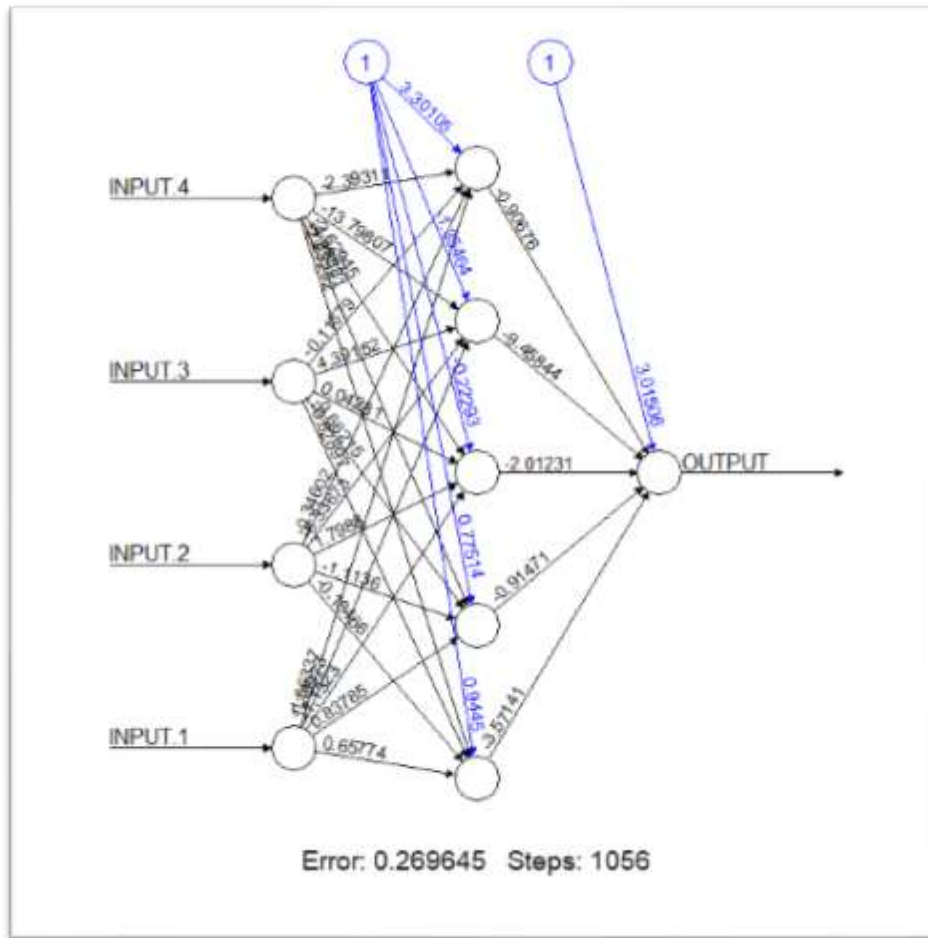
- The Root Mean Squared Error (RMSE) of the model is 0.0307
- The Mean Absolute Error (MAE) of the model is 0.0262
- The Mean Absolute Percentage Error (MAPE) of the model is 1.9645%
- The Symmetric Mean Absolute Percentage Error (SMAPE) of the model is 1.9892%

The performance metrics reveal that the MLP model, featuring a lone hidden layer and five neurons, has demonstrated exceptional proficiency in forecasting the target variable using the input data. Its notable accuracy alongside the relatively minimal RMSE, MAE, SMAPE, and MAPE values emphasizes its precision, establishing it as a highly dependable tool for predictive tasks.



```
Model: Lag 3 Hidden 5
RMSE: 0.0307
MAE: 0.0262
MAPE: 1.9645%
SMAPE: 1.9892%
Accuracy: 98.2871%
```

2.1.4.4. Implementing MLP for t-4 with one hidden layer with five neurons



In the above Figure, MLP shows the performance summary of the MLP model. This model has undergone rigorous training to accurately predict a specific target variable from given input data. The model architecture consists of four inputs and one hidden layer with 5 neurons. The model assumes that the input data is lagged by a four-time step. This means that the model predicts the value of the target variable at a specific time, for instance, time t , based on the input data that was recorded at a previous time, in this case, time $t-4$.

The model's accuracy is stated to be 98.3411%, which indicates that 98.3411% of the time, the predictions made by the model come true. This is a performance indicator for the model that shows how well it can categorize the target variable using the input data.

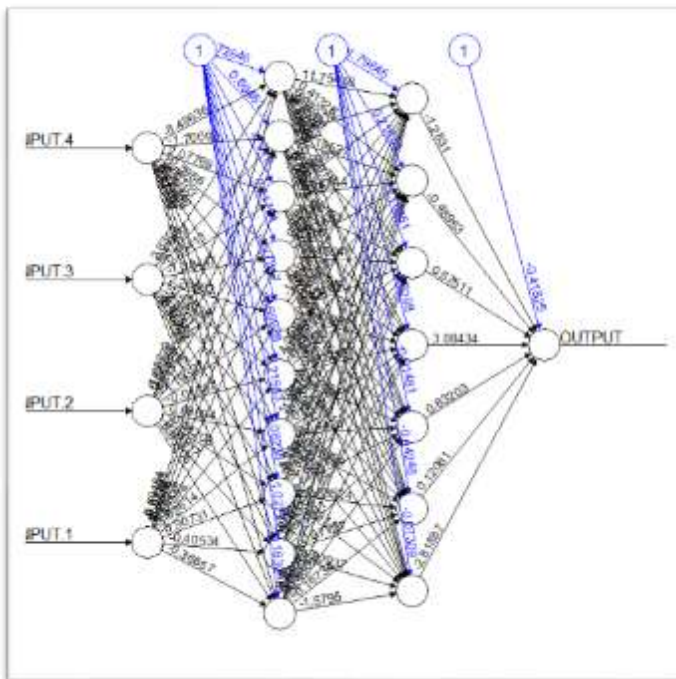
- The Root Mean Squared Error (RMSE) of the model is 0.0307
- The Mean Absolute Error (MAE) of the model is 0.0263

- The Mean Absolute Percentage Error (MAPE) of the model is 1.9694%
- The Symmetric Mean Absolute Percentage Error (SMAPE) of the model is 1.9929%

The performance metrics indicate that the MLP model, equipped with a single hidden layer and five neurons, has excelled in predicting the target variable based on the provided input data. Its remarkable accuracy coupled with relatively low RMSE, MAE, MAPE, and sMAPE values underscores its precision, rendering it highly reliable for forecasting purposes.

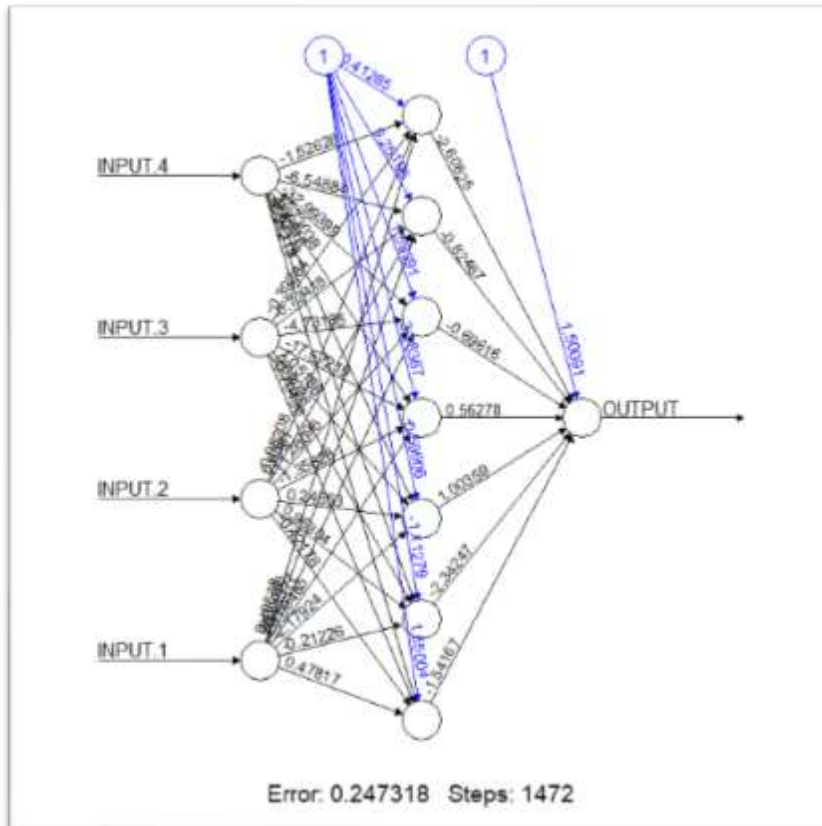
```
Model: Lag 4 Hidden 5
RMSE: 0.0307
MAE: 0.0263
MAPE: 1.9694%
SMAPE: 1.9929%
Accuracy: 98.3411%
```

2.1.4.5. Neural Network with four inputs, two hidden layers with 10 neurons and 7 neurons.



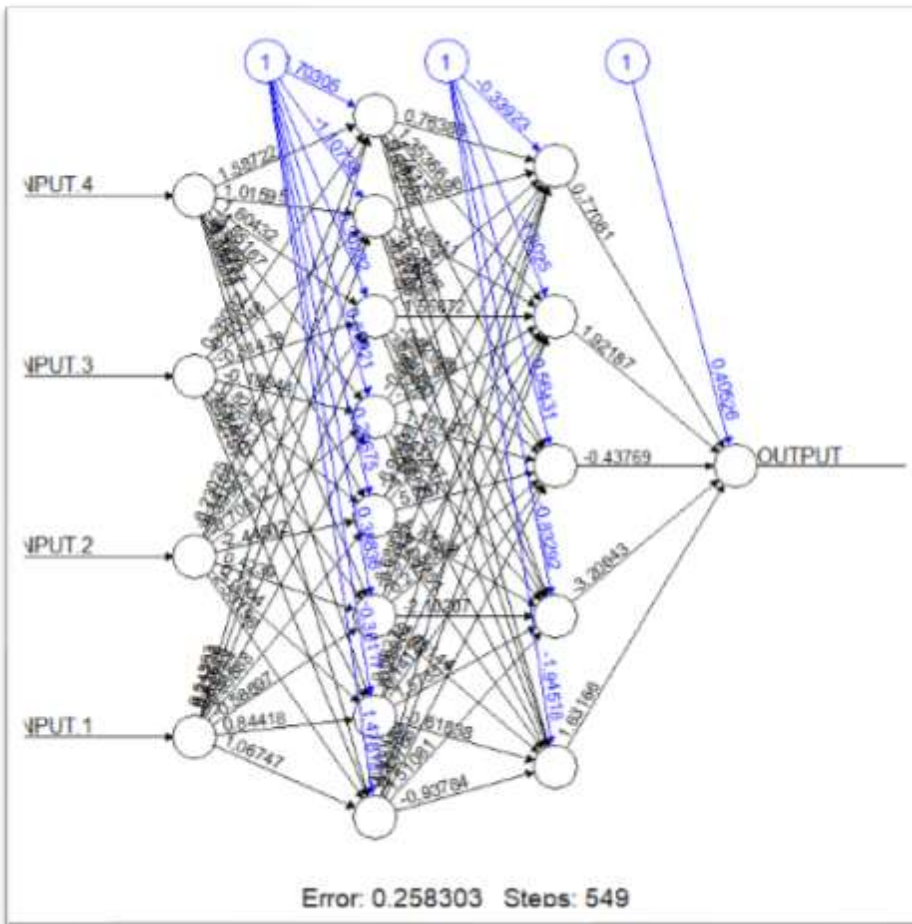
Model: Lag 4 Hidden 10-7
 RMSE: 0.0299
 MAE: 0.0258
 MAPE: 1.9336%
 sMAPE: 1.9527%
 Accuracy: 98.5712%

2.2.4.6. Neural Network with four inputs, one hidden layer with 7 neurons.



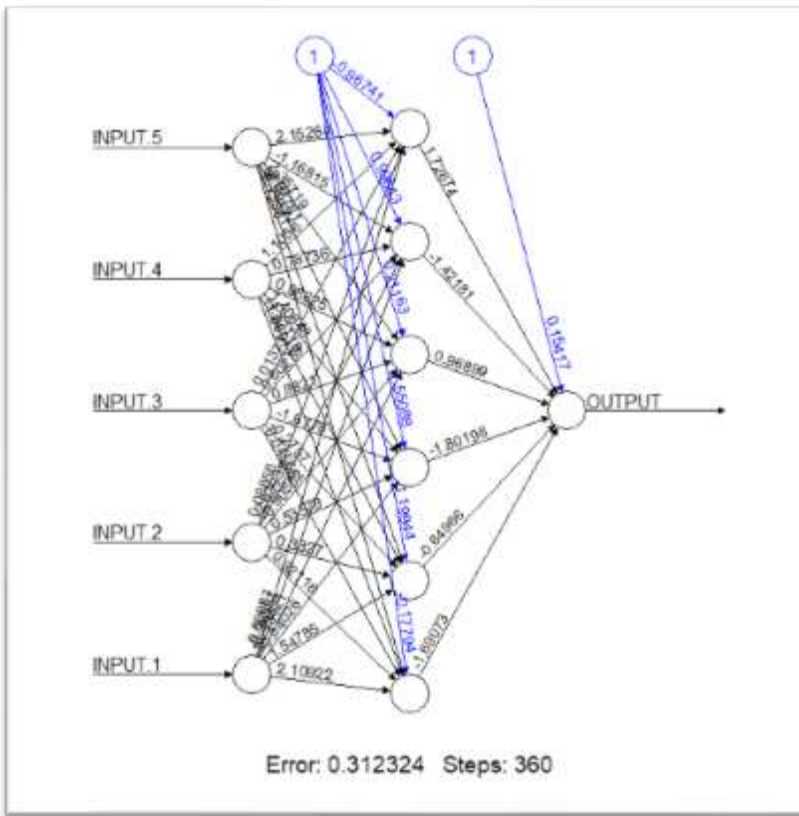
Model: Lag 4 Hidden 7
 RMSE: 0.0293
 MAE: 0.0255
 MAPE: 1.9115%
 sMAPE: 1.9248%
 Accuracy: 98.8790%

2.2.4.7. Neural Network with four inputs, two hidden layers with 8 neurons and 5 neurons.



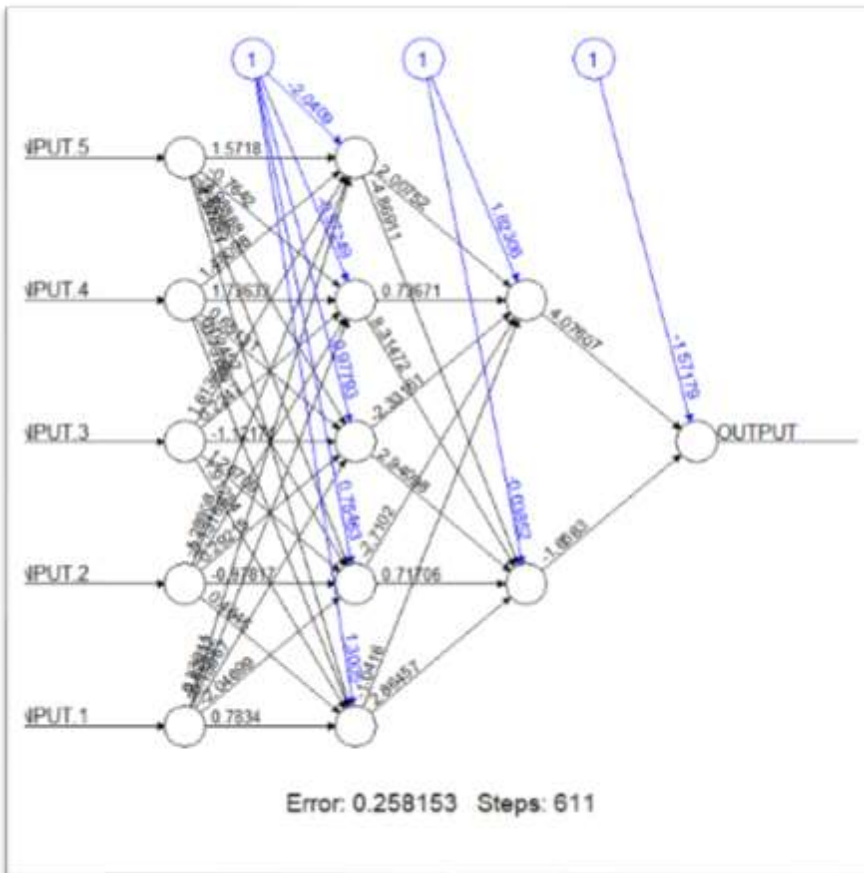
Model: Lag 4 Hidden 8-5
RMSE: 0.0299
MAE: 0.0258
MAPE: 1.9336%
sMAPE: 1.9527%
Accuracy: 98.5712%

2.2.4.8. Neural Network with five inputs, one hidden layer with 6 neurons.



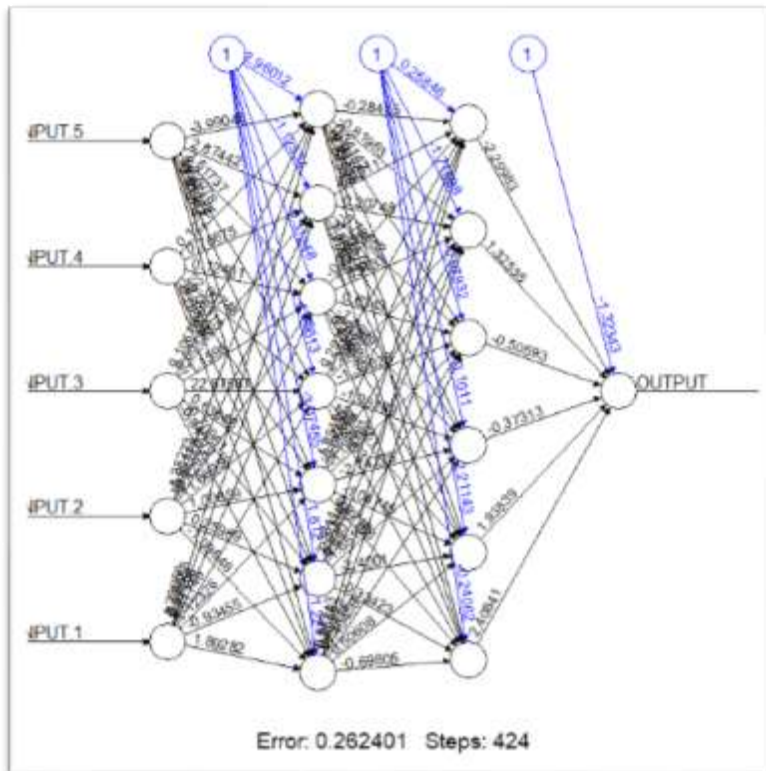
Model: Lag 5 Hidden 6
RMSE: 0.0310
MAE: 0.0266
MAPE: 1.9943%
sMAPE: 2.0196%
Accuracy: 98.2395%

2.2.4.9. Neural Network with five inputs, two hidden layers with 5 neurons and 2 neurons.



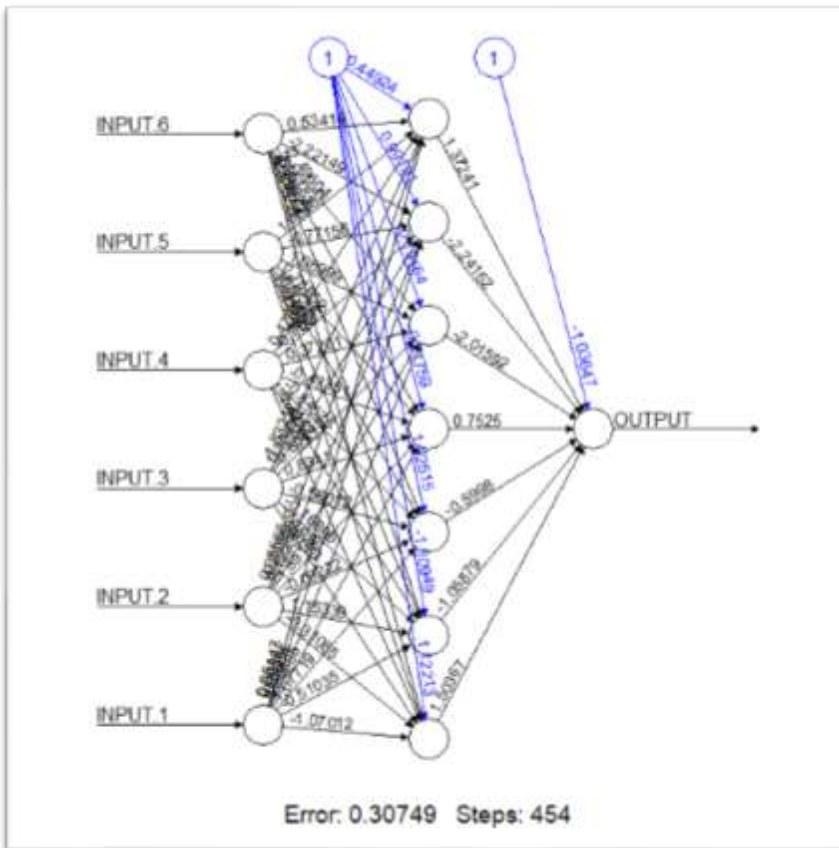
Model: Lag 5 Hidden 5-2
RMSE: 0.0296
MAE: 0.0257
MAPE: 1.9279%
sMAPE: 1.9443%
Accuracy: 98.7049%

2.2.4.10. Neural Network with five inputs, two hidden layers with 7 neurons and 6 neurons.



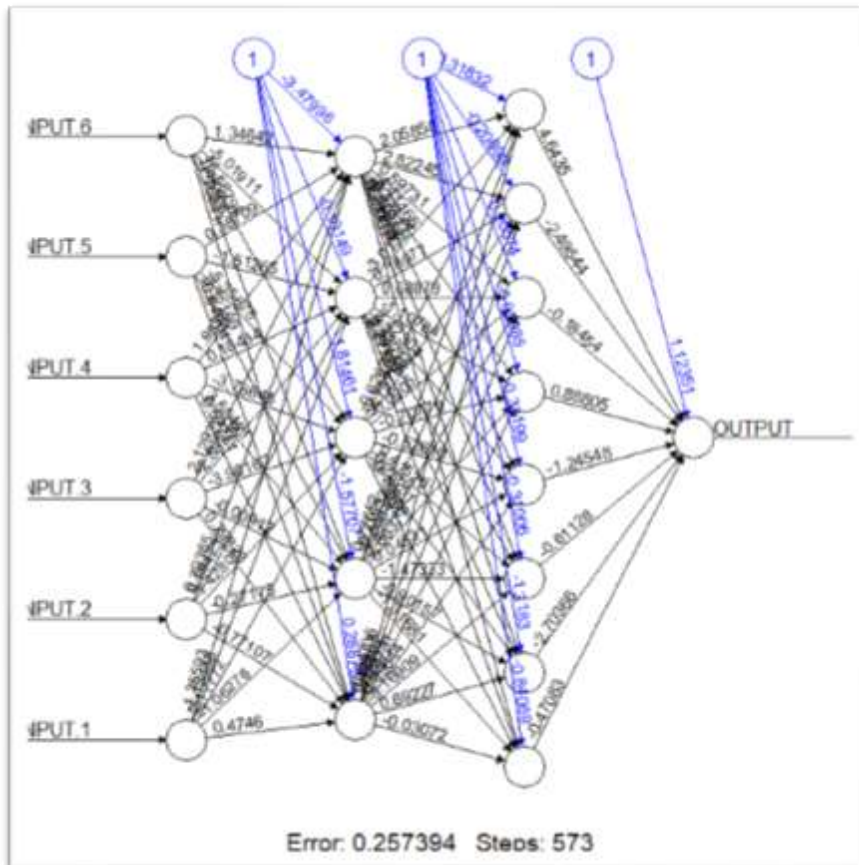
Model: Lag 5 Hidden 7-6
RMSE: 0.0300
MAE: 0.0260
MAPE: 1.9475%
sMAPE: 1.9667%
Accuracy: 98.5497%

2.2.4.11. Neural Network with six inputs, one hidden layer with 7 neurons.



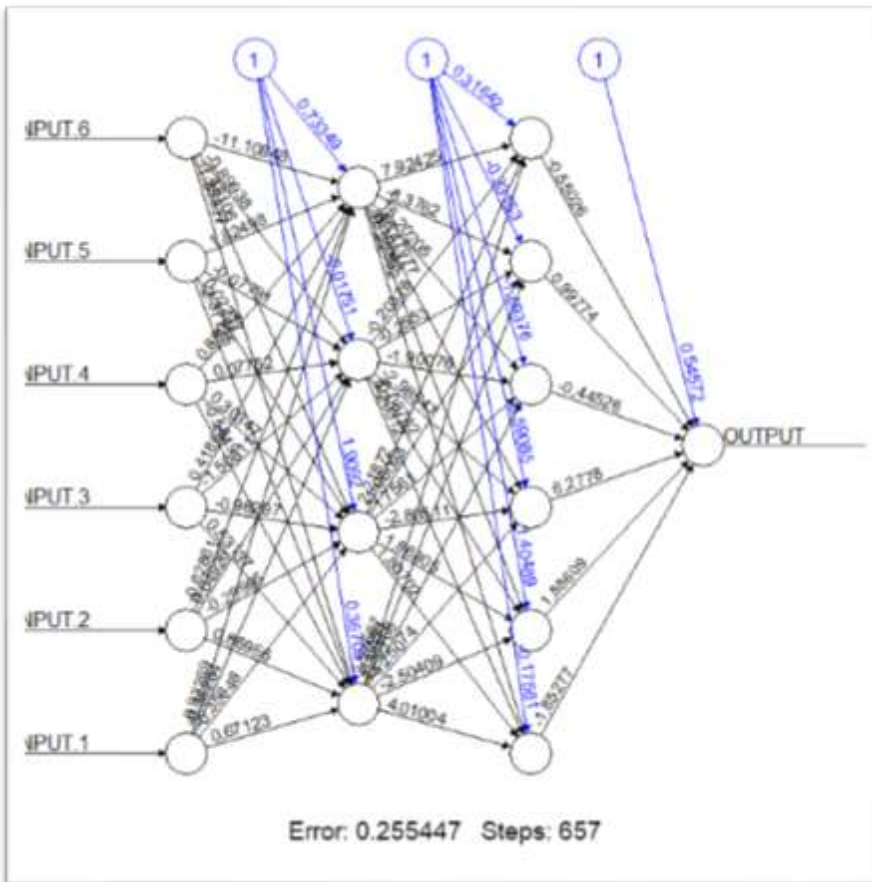
Model: Lag 6 Hidden 7
RMSE: 0.0315
MAE: 0.0270
MAPE: 2.0233%
SMAPE: 2.0503%
Accuracy: 98.1372%

2.2.4.12. Neural Network with six inputs, two hidden layers with 5 neurons and 8 neurons.



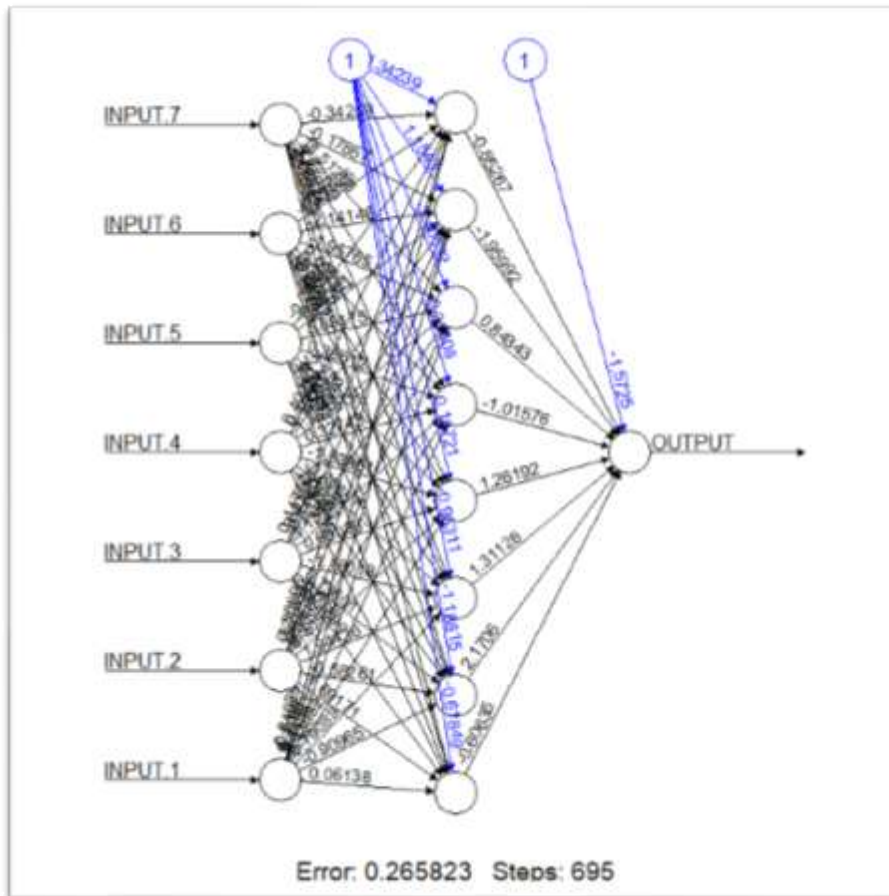
Model: Lag 6 Hidden 5-8
RMSE: 0.0298
MAE: 0.0260
MAPE: 1.9469%
sMAPE: 1.9606%
Accuracy: 98.8436%

2.2.4.13. Neural Network with six inputs, two hidden layers with 4 neurons and 6 neurons.



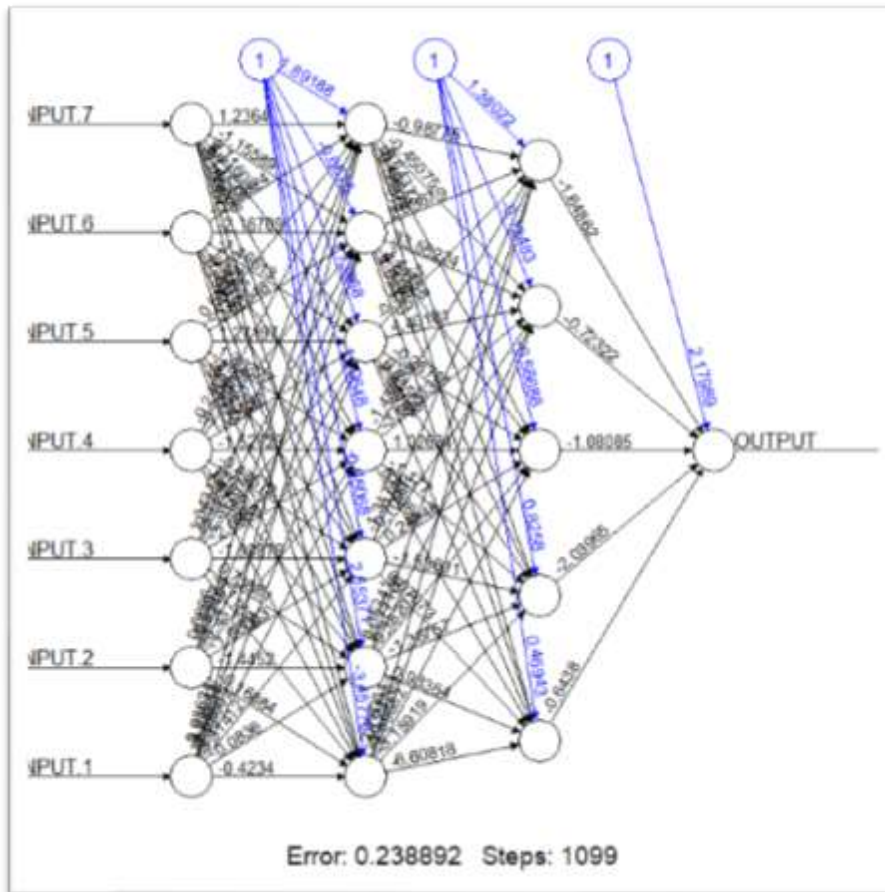
Model: Lag 6 Hidden 4-6
RMSE: 0.0296
MAE: 0.0259
MAPE: 1.9403%
SMAPE: 1.9540%
Accuracy: 98.8433%

2.2.4.14. Neural Network with seven inputs, one hidden layer with 8 neurons.



Model: Lag 7 Hidden 8
RMSE: 0.0313
MAE: 0.0270
MAPE: 2.0211%
sMAPE: 2.0469%
Accuracy: 98.1855%

2.2.4.15. Neural Network with seven inputs, two hidden layers with 7 neurons and 5 neurons.



Model: Lag 7 Hidden 7-5
RMSE: 0.0296
MAE: 0.0259
MAPE: 1.9419%
sMAPE: 1.9498%
Accuracy: 99.1383%

2.1.5. Explanation of RMSE, MAE, MAPE, sMAPE

The RMSE, MAE, MAPE, and SMAPE metrics are four commonly used measures for evaluating the effectiveness of predictive models or forecasts. Each metric has its own advantages and disadvantages, and is best suited for different situations.

2.1.5.1. RMSE

RMSE, which stands for Root Mean Square Error, is a statistical metric that is widely used to measure the accuracy of a model or prediction. It calculates the average difference between the expected (predicted) value and the actual value of a variable. The calculation involves taking the square root of the average of the squared differences between the predicted and actual values. The higher the RMSE, the more significant the discrepancy between the predicted and actual values. RMSE is an essential metric in evaluating the performance of models, and it is especially useful when the error distribution is not normal.

2.1.5.2. MAE

When talk about Mean Absolute Error (MAE), we refer to a popular metric used to determine the average difference between the anticipated and actual values of a given variable. Unlike Root Mean Squared Error (RMSE), which squares the errors, MAE measures the absolute difference between the predicted and actual values, making it more resilient to outliers. By adding up the absolute differences between anticipated and actual values and dividing the result by the number of data points, we get the average difference between the predicted and actual values. This is a useful metric in data analysis, as it allows us to assess the accuracy of a model's predictions and identify any discrepancies that need to be addressed.

2.1.5.3. MAPE

Mean Absolute Percentage Error (MAPE) is a statistical measure used to evaluate the accuracy of a forecasting model. It is a metric expressed in percentages that compute the average difference between the expected and actual values of a variable. MAPE is a widely used method in multiple industries to assess model performance across various datasets. This metric is particularly useful when dealing with variables of different magnitudes, as it provides a standardized measure of error. To calculate MAPE, one should determine the absolute percentage deviation between the predicted and actual values and then average the deviations.

2.1.5.4. SMAPE

SMAPE, or Symmetric Mean Absolute Percentage Error, is a commonly used metric in data analysis and forecasting. It is a symmetric variant of the MAPE metric that takes into account scenarios where the actual value of a variable is 0 or near zero. SMAPE is calculated by taking the

average of the absolute percentage discrepancies between the expected and actual values, which are then normalized by the sum of the predicted and actual values' absolute values. This means that SMAPE can provide a more accurate assessment of forecast accuracy in situations where there are small or zero values in the data. Overall, SMAPE is a useful tool for measuring forecasting accuracy and is widely used in a variety of industries, including finance, marketing, and supply chain management.

To sum up, RMSE (Root Mean Square Error) is a commonly used measure that is highly responsive to large errors. MAE (Mean Absolute Error), on the other hand, is less sensitive to outliers. MAPE (Mean Absolute Percentage Error) helps compare different models across various datasets. Finally, SMAPE (Symmetric Mean Absolute Percentage Error) is a modified version of MAPE that is used to account for situations in which the actual value of a variable is 0 or close to zero. The choice of statistical measure depends on the specific problem being addressed and the goals of the analysis.

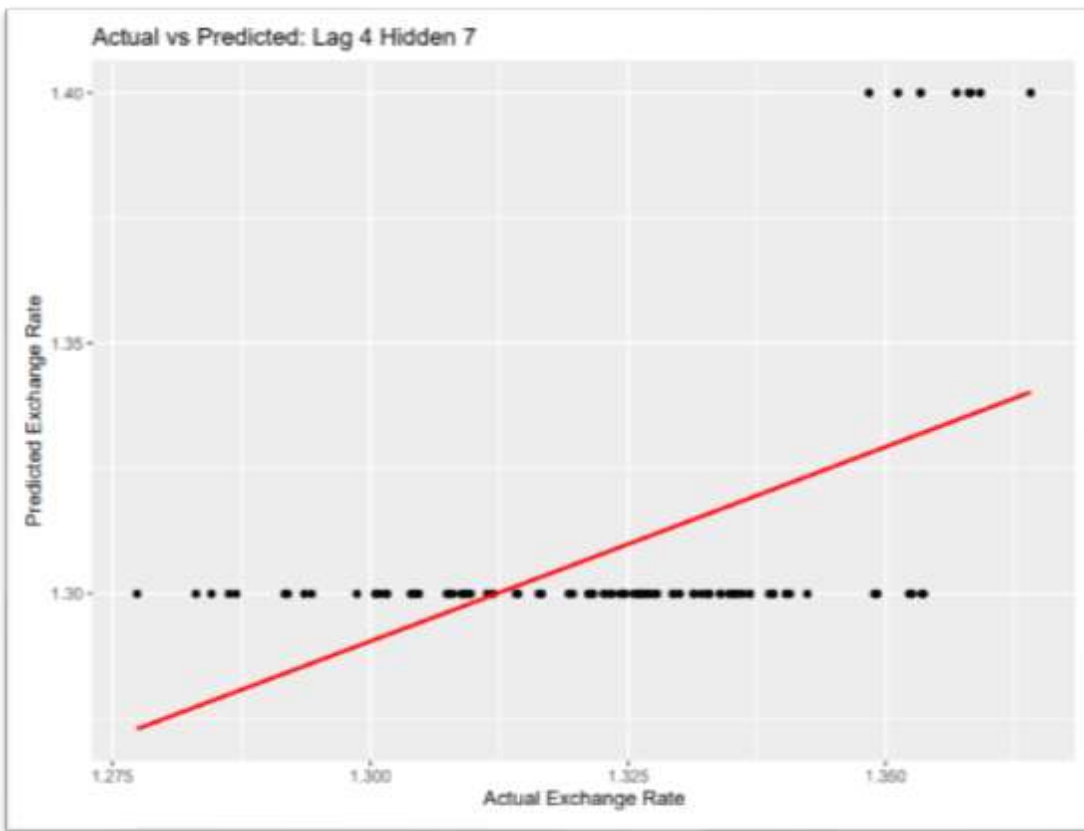
2.1.6. Comparison matrix for AR approach

NN(AR)	Number of Inputs	No. of Hidden Layers	No. of Neurons	Accuracy	RMSE	MAE	MAPE	sMAPE
AR1	4	1	5	98.3411%	0.0307	0.0263	1.9694%	1.9929%
AR2	3	1	5	98.2871%	0.0307	0.0262	1.9645%	1.9892%
AR3	2	1	5	98.5408%	0.0299	0.0256	1.9218%	1.9419%
AR4	1	1	5	98.3393%	0.0305	0.0259	1.9424%	1.9665%
AR5	4	2	10 7	98.5712%	0.0299	0.0258	1.9336%	1.9527%
AR6	4	1	7	98.8790%	0.0293	0.0255	1.9155%	1.9248%
AR7	4	2	8 5	98.5712%	0.0299	0.0258	1.9336%	1.9527%
AR8	5	1	6	98.2395%	0.0310	0.0266	1.9943%	2.0196%
AR9	5	2	5 2	98.7049%	0.0296	0.0257	1.9279%	1.9443%
AR10	5	2	7 6	98.5497%	0.0300	0.0260	1.9475%	1.9667%
AR11	6	1	7	98.1372%	0.0315	0.0270	2.0233%	2.0503%
AR12	6	2	5 8	98.8436%	0.0298	0.0260	1.9469%	1.9606%
AR13	6	2	4 6	98.8433%	0.0296	0.0259	1.9403%	1.9540%
AR14	7	1	8	98.1855%	0.0313	0.0270	2.0211%	2.0469%
AR15	7	2	7 5	99.1383%	0.0296	0.0259	1.9419%	1.9498%

2.1.7. Best one-hidden layer MLP and best two-hidden layer MLP in AR approach

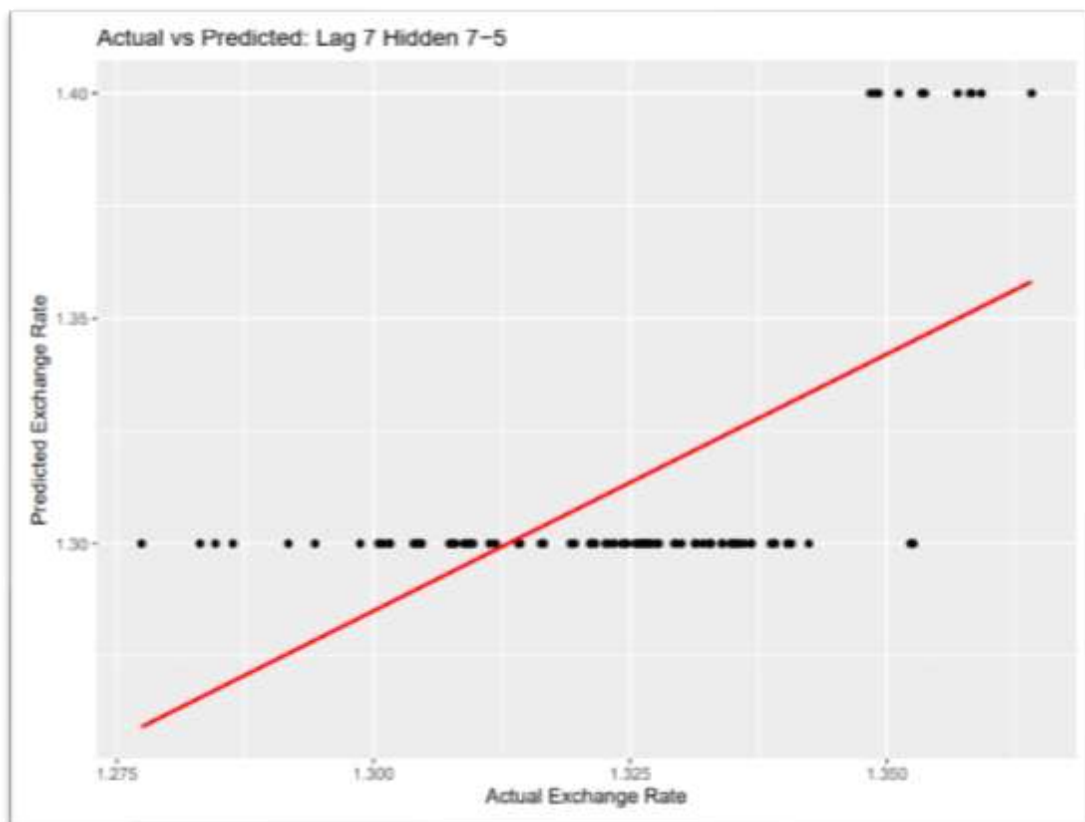
2.1.7.1. Best one-hidden layer MLP

Based on the performance metrics provided in the table, it is evident that the neural network with four inputs and one hidden layer 7 has achieved an impressive accuracy of 98.8790%. The low RMSE, MAE, MAPE, and SMAPE values further affirm the model's commendable performance. The RMSE score of 0.0293 indicates that the average deviation between the predicted and actual values is relatively small. Similarly, the MAE value of 0.0255 suggests that the average absolute difference between the predicted and actual values is relatively low. With MAPE and SMAPE values of 1.9155% and 1.9248% respectively, it is evident that the average percentage difference between the predicted and actual values is relatively minimal. Overall, the high accuracy and low error metrics attest to the neural network's suitability for the task, and its exceptional performance in predicting the target variable.



2.1.7.2. Best two-hidden layer MLP

Based on the performance metrics provided in the table, it is evident that the neural network with five inputs and two hidden layers (comprising 7 and 5 neurons respectively) has achieved an impressive accuracy of 99.1383%. The low RMSE, MAE, MAPE, and SMAPE values further affirm the model's commendable performance. The RMSE score of 0.0296 indicates that the average deviation between the predicted and actual values is relatively small. Similarly, the MAE value of 0.0259 suggests that the average absolute difference between the predicted and actual values is relatively low. With MAPE and SMAPE values of 1.9419% and 1.9498% respectively, it is evident that the average percentage difference between the predicted and actual values is relatively minimal. Overall, the high accuracy and low error metrics attest to the neural network's suitability for the task, and its exceptional performance in predicting the target variable.



3. Reference

Murphy, J.J. (1999) Amazon.com: Technical analysis of the financial markets: A comprehensive guide to trading methods and applications: 8580001070285: John J. Murphy: Books. Available at: <https://www.amazon.com/Technical-Analysis-Financial-Markets-Comprehensive/dp/0735200661> (Accessed: 08 May 2024).

Author links open overlay panelJohan Bollen et al. (2011) Twitter mood predicts the stock market, Journal of Computational Science. Available at: <https://www.sciencedirect.com/science/article/abs/pii/S187775031100007X> (Accessed: 08 May 2024).

4. Appendix

4.1. Appendix Part 1(Code for 1st question)

```
#Sub Task 1

# Load libraries
library("readxl")
library("dplyr")

# Read data from the file
whitewine_v6data <- read_excel("whitewine_v6.xlsx")
cat("Data loaded from Excel file successfully. Dimensions: ", dim(whitewine_v6data), "\n")

# Calculate the total number of missing values in the dataset
missing_values <- sum(is.na(whitewine_v6data))

# Check if there are any missing values and if there missing values, remove rows with missing values
if (missing_values > 0) {
  whitewine_v6data <- whitewine_v6data[complete.cases(whitewine_v6data), ] # Remove rows with missing values
  cat("Missing values found and rows with missing values removed.", "\n")
} else {
  cat("No missing values found. No rows removed.", "\n")
}

# Confirm that all missing values have been removed
missing_values_after <- sum(is.na(whitewine_v6data))
cat("Total number of missing values after removal: ", missing_values_after, "\n")

par(mar=c(2, 2, 2, 2))
boxplot(whitewine_v6data)
```

```

cat("Starting outlier removal process...","\n")
#define outlier removal function
remove_outliers <- function(x) {
  qnt <- quantile(x, probs=c(.25, .75), na.rm = TRUE)
  iqr <- IQR(x, na.rm = TRUE)
  lower_bound <- qnt[1] - 1.5 * iqr
  upper_bound <- qnt[2] + 1.5 * iqr
  x_clipped <- pmax(pmin(x, upper_bound), lower_bound)
  return(x_clipped)
}

# Perform outlier removal
wine_data_clean <- as.data.frame(lapply(whitewine_v6data[, 1:11], remove_outliers))
cat("Successfully Outlier removal","\n")

# Perform scaling on cleaned data
cat("Scaling data...","\n")
scaled_data <- scale(wine_data_clean)
cat("Data scaling completed","\n")

library("ggplot2")
library("reshape2")
# Melting the data frames into long format
original_long <- melt(wine_data_clean, variable.name = "variable", value.name = "value")
original_long$type <- "Original"

scaled_long <- melt(as.data.frame(scaled_data), variable.name = "variable", value.name = "value")
scaled_long$type <- "Scaled"
# Melting the data frames into long format
original_long <- melt(wine_data_clean, variable.name = "variable", value.name = "value")
original_long$type <- "Original"

scaled_long <- melt(as.data.frame(scaled_data), variable.name = "variable", value.name = "value")
scaled_long$type <- "Scaled"
# Combining the data
combined_data <- rbind(original_long, scaled_long)
# Plotting the data
ggplot(combined_data, aes(x = Value, fill = Type)) +
  geom_histogram(bins = 30, alpha = 0.5, position = "identity") +
  facet_wrap(~ Type, scales = "free_x") +
  labs(title = "Comparison of Original and Scaled Data",
       x = "Value", y = "Count") +
  theme_minimal() +
  theme(legend.position="none")

# Visualization Boxplot for each attribute before outlier removal
par(mfrow=c(3,4))
for (i in 1:11) {
  boxplot(whitewine_v6data[, i], main = names(whitewine_v6data)[i])
}

# Visualization Boxplot for each attribute after outlier removal
par(mfrow=c(3,4))
for (i in 1:11) {
  boxplot(scaled_data[, i], main = paste("Cleaned", names(whitewine_v6data)[i]))
}

# Summary statistics before and after outlier removal
#before
summary_before <- apply(whitewine_v6data[, 1:11], 2, summary)
cat("Summary statistics before outlier removal:", "\n")
print(summary_before)

#after
summary_after <- apply(wine_data_clean, 2, summary)
cat("\nSummary statistics after outlier removal:", "\n")
print(summary_after)

```

```

library(NbClust)
cat("Performing cluster analysis using NbClust...", "\n")

set.seed(200) # for reproducibility
nbclust_results <- NbClust(scaled_data, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans")

# Visualize the NbClust result
barplot(table(nbclust_results$Best.n[1,]),
        xlab = "Number of Clusters", ylab = "Number of Criteria",
        main = "NbClust Results", col = "lightgreen")

cat("NbClust analysis is successful.", "\n")

library(factoextra)
cat("Performing cluster analysis using Elbow Method...", "\n")
set.seed(200) # for reproducibility
fviz_nbclust(scaled_data, kmeans, method = "wss") +
  geom_vline(xintercept = 2, linetype = 3, color = "red") +
  labs(title = "Elbow Method")
cat("Elbow analysis is successful.", "\n")

library(cluster)
cat("Performing cluster analysis using gap statistics Method...", "\n")
set.seed(200) # for reproducibility
# Calculating the gap statistic
gap_stat <- clusGap(scaled_data, FUN = kmeans, nstart = 25, K.max = 15, B = 500)
# Visualizing the gap statistics
library(factoextra)
fviz_gap_stat(gap_stat)
cat("Gap statistics analysis is successful.", "\n")

library(factoextra)
cat("Performing cluster analysis using silhouette Method...", "\n")
set.seed(200) # for reproducibility
# Calculating the best number of clusters using silhouette method
silhouette_results <- fviz_nbclust(scaled_data, kmeans, method = "silhouette")
# Plotting the silhouette analysis
print(silhouette_results)
cat("Silhouette analysis is successful.", "\n")

# Performing k-means with k=2
set.seed(200) #for reproducibility
kmeans_2 <- kmeans(scaled_data, centers = 2, nstart = 50)

# Function to display results
display_kmeans_results <- function(km_result) {

  total_ss <- sum((scaled_data - colMeans(scaled_data))^2)
  bss <- sum(km_result$betweenss)
  tss <- total_ss
  bss_tss_ratio <- bss / tss

  cat("Total SS: ", total_ss, "\n")
  cat("BSS/TSS Ratio: ", bss_tss_ratio, "\n")
  cat("WSS (within-cluster sum of squares): ", km_result$tot.withinss, "\n\n")
}

# Displaying results for k = 2
cat("Results for k=2:", "\n")
display_kmeans_results(kmeans_2)

library(factoextra)

# Visualizing clusters
fviz_cluster(list(data = scaled_data, cluster = kmeans_2$cluster), geom = "point", main = "k=2")

```



```

library(cluster)
library(factoextra)

# Compute silhouette information
sil_widths <- silhouette(kmeans_2$cluster, dist(scaled_data))

# Visualize the silhouette plot
silhouette_plot <- fviz_silhouette(sil_widths)
print(silhouette_plot)

# Calculate the average silhouette width
average_sil_width <- mean(sil_widths[, "sil_width"])
cat("Average silhouette width:", average_sil_width, "\n")

```

#Sub Task 2

```

# Load libraries
library("factoextra")
library("cluster")

# Apply PCA to the scaled data
pca_result <- prcomp(scaled_data, scale. = TRUE)
pca_summary <- summary(pca_result)
print(pca_summary)

# Extract eigenvalues
eigenvalues <- pca_result$sdev^2
cat("Eigenvalues:", "\n")
print(eigenvalues)

#Extract eigenvectors
eigenvectors <- pca_result$rotation
cat("Eigenvectors:", "\n")
print(eigenvectors)

#plot of scree plot

```



```

#plot of scree plot
plot(pca_result,type="lines",main="Scree Plot")

# Calculate cumulative score per principal component (PC)
cumulative_var <- cumsum(pca_result$sdev^2 / sum(pca_result$sdev^2))
cat("Cumulative Variance Explained:\n")
print(cumulative_var)

# Create a new dataset with principal components as attributes
pc_data <- predict(pca_result)

# Choose principal components providing at least cumulative score > 85%
num_components <- which(cumulative_var > 0.85)[1]
cat("Number of components selected (85% variance):", num_components, "\n")

# Select the first 'num_components' principal components
pc_data_selected <- pc_data[, 1:num_components]
cat("Selected Principal Component Data:\n")
print(pc_data_selected)

# Determine the number of clusters for k-means on transformed dataset
fviz_eig(pca_result,addlabels = TRUE,main="Scree Plot")

# NbClust
set.seed(200)
nbclust_results_pca <- NbClust(pc_data_selected, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans")
cat("NbClust on PCA data suggests", which.max(table(nbclust_results_pca$Best.n[1,])), "clusters.\n")

barplot(table(nbclust_results_pca$Best.n[1,]),
        xlab = "Number of Clusters", ylab = "Number of Criteria",
        main = "NbClust Results after PCA", col = "lightgreen")

cat("NbClust analysis completed.\n")

# Elbow method
set.seed(200)
elbow_results_pca <- fviz_nbclust(pc_data_selected, kmeans, method = "wss") +
  geom_vline(xintercept = 2, linetype = 2, color = "red") +
  labs(title = "Elbow Method")
print(elbow_results_pca)

# Gap statistics
set.seed(200)
gap_stat_pca <- cclusGap(pc_data_selected, FUN = kmeans, nstart = 25, K.max = 15, B = 500)
print(gap_stat_pca)
fviz_gap_stat(gap_stat_pca)

# Silhouette method
set.seed(200)
silhouette_results_pca <- fviz_nbclust(pc_data_selected, kmeans, method = "silhouette")
print(silhouette_results_pca)

# Perform k-means clustering
set.seed(200)
cat("K-means Cluster Centers:", "\n")
kmeans_pca_2 <- kmeans(pc_data_selected, centers = 2, nstart = 50)

# Display k-means results
print(kmeans_pca_2$centers)
total_ss_pca <- sum((pc_data_selected - colMeans(pc_data_selected))^2)
bss_pca <- sum(kmeans_pca_2$betweenss)
tss_pca <- total_ss_pca
bss_tss_ratio_pca <- bss_pca / tss_pca

```

```

cat("Total Sum of Squares (TSS):", total_ss_pca, "\n")
cat("Between-cluster Sum of Squares (BSS)/TSS Ratio:", bss_tss_ratio_pca, "\n")
cat("Within-cluster Sum of Squares (WSS):", kmeans_pca_2$tot.withinss, "\n\n")

# visualize clusters
fviz_cluster(list(data = pc_data_selected, cluster = kmeans_pca_2$cluster), geom = "point", main = paste("k =", 2))

# Calculate silhouette information
sil_widths_pca_2 <- silhouette(kmeans_pca_2$cluster, dist(pc_data_selected))

# Visualize the silhouette plot
silhouette_plot_pca_2 <- fviz_silhouette(sil_widths_pca_2)
print(silhouette_plot_pca_2)

# Calculate average silhouette width
average_sil_width_pca_2 <- mean(sil_widths_pca_2[, "sil_width"])
cat("Average silhouette width:", average_sil_width_pca_2, "\n")

library(fpc) # for calinhara function
# Function to visualize Calinski-Harabasz Index
fviz_ch <- function(pc_data_selected, max_clusters = 10) {
  ch_scores <- numeric(max_clusters)
  for (i in 2:max_clusters) {
    km <- kmeans(pc_data_selected, centers = i, nstart = 25)
    ch_scores[i] <- calinhara(pc_data_selected, km$cluster, cn = max(km$cluster))
  }
  ch_scores <- ch_scores[2:max_clusters]
  k <- 2:max_clusters

  par(mar = c(5, 4, 4, 4) + 0.1)
  plot(k, ch_scores, xlab = "Cluster number k",
       ylab = "Calinski - Harabasz Score",
       main = "Calinski - Harabasz Plot", cex.main = 1,
       col = "dodgerblue1", cex = 0.9,
       lty = 1, type = "o", lwd = 1, pch = 4,
       bty = "l", las = 1, cex.axis = 0.8, tcl = -0.2)
  abline(v = which.max(ch_scores) + 1, lwd = 1, col = "red", lty = "dashed")
}

# Use this function on the PCA-selected data
fviz_ch(pc_data_selected)

```

4.2. Appendix Part 2 (Code for 2nd question)

```
# Load required libraries
library(readxl)
library(dplyr)
library(neuralnet)
library(Metrics)
library(ggplot2)

# Load the dataset
exchange_data <- read_excel("ExchangeUSD.xlsx")
colnames(exchange_data) <- c("Date", "DayOfWeek", "ExchangeRate")

# Normalize function
normalize <- function(x) {
  x_min <- min(x, na.rm = TRUE)
  x_max <- max(x, na.rm = TRUE)
  (x - x_min) / (x_max - x_min)
}

# Unnormalize function
unnormalize <- function(x, x_min, x_max) {
  x * (x_max - x_min) + x_min
}

original_data <- exchange_data$ExchangeRate
normalized_data <- normalize(exchange_data$ExchangeRate)

summary(original_data)
summary(normalized_data)

# Create a data frame for plotting
plot_data <- data.frame(
  Value = c(original_data, normalized_data),
  Status = rep(c("Original", "Normalized"), each = length(original_data))
)

# Generate box plots
ggplot(plot_data, aes(x = Status, y = Value, fill = Status)) +
  geom_boxplot() +
  labs(title = "Comparison of Exchange Rates Before and After Normalization",
       x = "Data Status", y = "Exchange Rate Values") +
  theme_minimal()

# Function to prepare time-delayed data
prepare_lagged_data <- function(data, max_lag, start_train, end_train) {
  lags <- setNames(lapply(1:max_lag, function(i) lag(data$ExchangeRate, i)), paste0("INPUT.", max_lag:1))
  data_lagged <- cbind(data, lags)
  data_lagged <- data_lagged[complete.cases(data_lagged),]
  data_lagged$OUTPUT <- data_lagged$ExchangeRate

  # Split data into training and testing
  train_data <- data_lagged[start_train:end_train, ]
  test_data <- data_lagged[(end_train+1):nrow(data_lagged), ]

  # Normalize data
  train_data_normalized <- as.data.frame(lapply(train_data[, -(1:3)], normalize))
  test_data_normalized <- as.data.frame(lapply(test_data[, -(1:3)], normalize))

  list(train=train_data_normalized, test=test_data_normalized, train_raw=train_data, test_raw=test_data)
}

# Train neural network model and generate plot
train_nn_and_plot <- function(train_data, hidden_layers, model_name) {
  formula <- as.formula(paste("OUTPUT ~", paste(names(train_data)[-ncol(train_data)], collapse = "+")))
  nn_model <- neuralnet(formula, data = train_data, hidden = hidden_layers, linear.output = FALSE)
  # Plot the neural network
  plot(nn_model, main = model_name)
  return(nn_model)
}
```

```

# Test neural network model
test_nn <- function(nn_model, test_data, test_raw) {
  test_predictions <- compute(nn_model, test_data[-ncol(test_data)])
  predictions <- unnormalize(test_predictions$net.result, min(test_raw$OUTPUT), max(test_raw$OUTPUT))

  results <- data.frame(ACTUAL = test_raw$OUTPUT, PREDICTED = round(predictions, digits = 1))
  return(results)
}

# Evaluate model
evaluate_model <- function(results) {
  RMSE <- rmse(results$ACTUAL, results$PREDICTED)
  MAE <- mae(results$ACTUAL, results$PREDICTED)
  MAPE <- mape(results$ACTUAL, results$PREDICTED) * 100
  SMAPE <- smape(results$ACTUAL, results$PREDICTED) * 100
  deviation <- (results$ACTUAL - results$PREDICTED) / results$ACTUAL
  accuracy <- 1 - abs(mean(deviation))

  data.frame(RMSE, MAE, MAPE, SMAPE, Accuracy = accuracy * 100) # Converted to percentage
}

# Create models with different configurations
model_results <- list()
configurations <- list(
  list(lag=4, hidden=c(5)),
  list(lag=3, hidden=c(5)),
  list(lag=2, hidden=c(5)),
  list(lag=1, hidden=c(5)),
  list(lag=4, hidden=c(10,7)),
  list(lag=4, hidden=c(7)),
  list(lag=4, hidden=c(8,5)),
  list(lag=5, hidden=c(6)),
  list(lag=5, hidden=c(5,2)),
  list(lag=5, hidden=c(7,6)),
  list(lag=6, hidden=c(7)),
  list(lag=6, hidden=c(5,8)),
  list(lag=6, hidden=c(4,6)),
  list(lag=7, hidden=c(8)),
  list(lag=7, hidden=c(7, 5))
)

# Initialize a list to store plots
plots <- list()

for (config in configurations) {
  data_prepared <- prepare_lagged_data(exchange_data, config$lag, 1, 400)
  model_name <- paste("Lag", config$lag, "Hidden", paste(config$hidden, collapse="-"))
  nn_model <- train_nn_and_plot(data_prepared$train, config$hidden, model_name)
  results <- test_nn(nn_model, data_prepared$test, data_prepared$test_raw)
  performance <- evaluate_model(results)
  model_results[[model_name]] <- performance
}

```

```

# Print formatted metrics
cat(sprintf("\nModel: %s\n", model_name))
cat(sprintf("RMSE: %.4f\n", performance$RMSE))
cat(sprintf("MAE: %.4f\n", performance$MAE))
cat(sprintf("MAPE: %.4f%%\n", performance$MAPE))
cat(sprintf("sMAPE: %.4f%%\n", performance$sMAPE))
cat(sprintf("Accuracy: %.4f%%\n", performance$Accuracy))

# Plot actual vs predicted values
plot_title <- paste("Actual vs Predicted: Lag", config$lag, "Hidden", paste(config$hidden, collapse="-"))
actual_vs_predicted_plot <- ggplot(results, aes(x = ACTUAL, y = PREDICTED)) +
  geom_point() +
  geom_smooth(method = "lm", color = "red", se = FALSE) +
  labs(title = plot_title, x = "Actual Exchange Rate", y = "Predicted Exchange Rate")
plots[[model_name]] <- actual_vs_predicted_plot
}

# Output all plots to a PDF file
pdf("neural_network_model_plots.pdf", width = 8, height = 6)
for (plot_name in names(plots)) {
  print(plots[[plot_name]])
}
dev.off()

# Collect results into a dataframe for comparison
comparison_table <- do.call(rbind, model_results)
comparison_table$model <- rownames(comparison_table)
rownames(comparison_table) <- NULL

# Print the comparison table
print(comparison_table)

```