UNIVERSITY OF WESTMINSTER

Informatics Institute of Technology

Department of Computing

Machine Learning & Data Mining

Module code: 5DATA001C

Individual Coursework – Report

Student ID : 20210483

Student UoW ID : W1867607

Student Name : Shakya Chethiya Wijerathne

Tutorial Group : Group J

Table of Contents

1. Partitioning Clustering Part 4

1.1 1st Subtask Objectives 4

1.1.1 Pre-processing 4

1.1.2 Determine the number of cluster centres via four “automated tools”. 8

1.1.3 K-means clustering 12

1.1.4 Silhouette width score & Quality of the obtained clusters. 16

1.2 PCA analysis – 2nd Subtask 18

1.2.1 Applying PCA 18

1.2.2 Eigenvalues 19

1.2.2 Creating new dataset 20

1.2.3 Determining new number of clusters using four automated tools. 21

1.2.4 Perform K-means clustering 22

1.2.5 Calinski-Harabasz index 24

1.2.6 Comparing results of PCA analysis and Without PCA analysis 25

2. Energy Forecasting Part 26

2.1 Type of input variables used in MLP models for electricity load forecasting 26

2.2 AR Approach 28

2.2.1 Input vectors for AR approach 28

2.2.2 Normalization 29

2.2.3 Implementing different MLPs for AR approach. 30

2.2.4 Four Stat Indices. 31

2.2.5 Comparison Table 32

2.2.6 More models with different hidden layer structures and input vectors 33

2.2.7 Discussion on issue of efficiency with two best NN structures. 35

2.3 NARX Approach 36

2.3.1 Structures with different input vectors 36

2.3.2 Structures with different internal structures. 36
2.3.3 Comparison table for NARX Approach 38
2.3.4 Finding the best MLP models 38
2.3.5 Prediction output vs. Desired output 40
Appendix 42
Partitioning_Clustering.R 42
Energy_Forecasting.R 51

1. Partitioning Clustering Part

1.1 1st Subtask Objectives

1.1.1 Pre-processing

The dataset consists of 846 observations of vehicle silhouettes, with 18 input features and one output/class variable indicating the type of vehicle. The input features are related to geometric properties of the silhouettes, such as compactness, circularity, aspect ratios, skewness, and kurtosis, among others.

Pre-processing of the data is an essential step before applying any machine learning algorithm, including clustering. Pre-processing can involve several tasks, such as scaling, outliers detection, missing values imputation, feature selection, and feature engineering.

Scaling is necessary when the input features are measured on different scales or units, as some algorithms may be sensitive to differences in scales. In this dataset, the input features may have different scales and ranges, so it is essential to perform scaling to ensure that each feature contributes equally to the clustering process.

Outliers detection and removal are crucial to avoid the influence of extreme values that may bias the clustering results. Outliers can affect the mean and variance of the data, leading to inaccurate estimates of the cluster centers. Therefore, it is necessary to detect and remove outliers to ensure the accuracy and reliability of the clustering results.

Overall, pre-processing of the data is an important step to ensure the effectiveness and reliability of the clustering algorithm and the quality of the results.

Code for removing outliers:

```r
# 1st Subtask

library(ggplot2)
library(readxl)
library(dplyr)

# Import data from excel file
vehicle_data <- read_excel("vehicles.xlsx")
vehicle_data <- vehicle_data[, -1]

# Remove any rows with missing values in the dataframe
vehicle_data <- na.omit(vehicle_data)

variables <- vehicle_data[,-19] #stores attributes
class_variable <- vehicle_data$Class #class types

# Print the number of rows before removing outliers
vehicle_rows <- nrow(vehicle_data)
cat("number of rows before removing outliers = ", vehicle_rows)

outliers_in_columns <- c() # empty vector to store outliers in each column
for(i in 1:ncol(variables)){
  outlier_index <- variables[[i]] %in% boxplot.stats(variables[[i]])$out
  outlier_values <- variables[[i]][outlier_index]
  outliers_in_columns <- c(outliers_in_columns, outlier_values)
}

eleminate <- unique(outliers_in_columns) # remove the duplicated rows
eleminate <- sort(eleminate) # sort the outlire rows in order
good_vehicle_data <- vehicle_data[-eleminate,] # store the final rows after removing the outlires for attributes

variables <- distinct(variables)

# Removing outliers
no_outliers <- variables %>%
  filter_all(all_vars(!. %in% boxplot.stats(.)$out))

# Print the number of rows after removing outliers
n_rows <- nrow(good_vehicle_data)
cat("number of rows remaining after removing outliers = ", n_rows)

# Boxplot to check for outliers
boxplot(no_outliers)
```

```r

# Check for any points outside of the whiskers in the boxplot
points_outside <- boxplot(no_outliers, plot = FALSE)$out
if(length(points_outside) > 0) {
  cat("There are points outside of the whiskers in the boxplot.")
} else {
  cat("There are no points outside of the whiskers in the boxplot.")
}
```

Discussion:

Above code first reads in the "vehicles.xlsx" Excel file using the **read_excel** function from the **readxl** package, and removes any rows with missing values using the **na.omit** function.

Next, the code extracts the columns containing the attributes (i.e., variables) into the **variables** data frame and the class variable into the **class_variable** vector.

The code then identifies outliers in each variable using the **boxplot.stats** function to calculate the lower and upper limits of the whiskers of the boxplot, and removes rows containing outliers from the **vehicle_data** data frame. It does this by first storing the outlier values in each column in the **outliers_in_columns** vector, then removing duplicates and sorting the values in ascending order. The rows containing these outlier values are then removed from the **vehicle_data** data frame to create the **good_vehicle_data** data frame.

To further confirm that the outliers have been removed, the code creates a boxplot of the variables without outliers using the **boxplot** function and checks if any points fall outside the whiskers. If there are any points outside the whiskers, it outputs a message saying so, and if there are none, it outputs a message saying that there are no points outside the whiskers.

Finally, the code outputs the number of rows in the original **vehicle_data** data frame and the number of rows in the **good_vehicle_data** data frame after removing outliers.

Outputs:

```
number of rows before removing outliers =  846
```

```
number of rows remaining after removing outliers =  813
```

```
There are no points outside of the whiskers in the boxplot.
```

Scaling:

```
# function to calculate the normalized data
normalize <- function(x){
  return ((x - min(x)) / (max(x) - min(x)))
}

good_data_norm <- as.data.frame(lapply(good_vehicle_data[,-19], normalize))# normalized data
vehicle_data_no_outliers <- data.frame(scale(good_data_norm)) # scale the normalized data

no_outliers <- as.data.frame(lapply(no_outliers[,], normalize))# normalized data
no_outliers <- data.frame(scale(no_outliers)) # scale the normalized data
```

Discussion:

This defines a function called **normalize** that takes a vector **x** as input and returns the normalized version of that vector. The normalization is done using the formula: (x - min(x)) / (max(x) - min(x))

This formula ensures that all values in the vector are scaled to a range between 0 and 1.

The **good_data_norm** dataframe is created by applying the **normalize** function to all columns of **good_vehicle_data** except the last column (**good_vehicle_data[,-19]**). This creates a new dataframe with normalized values for all attributes.

The **vehicle_data_no_outliers** dataframe is created by scaling the **good_data_norm** dataframe. The **scale** function scales each column of the dataframe to have a mean of 0 and a standard deviation of 1.

The **no_outliers** dataframe is also normalized and scaled in the same way as **good_data_norm** and **vehicle_data_no_outliers**. The final dataframe **no_outliers** is also scaled to have a mean of 0 and a standard deviation of 1.

Outputs:

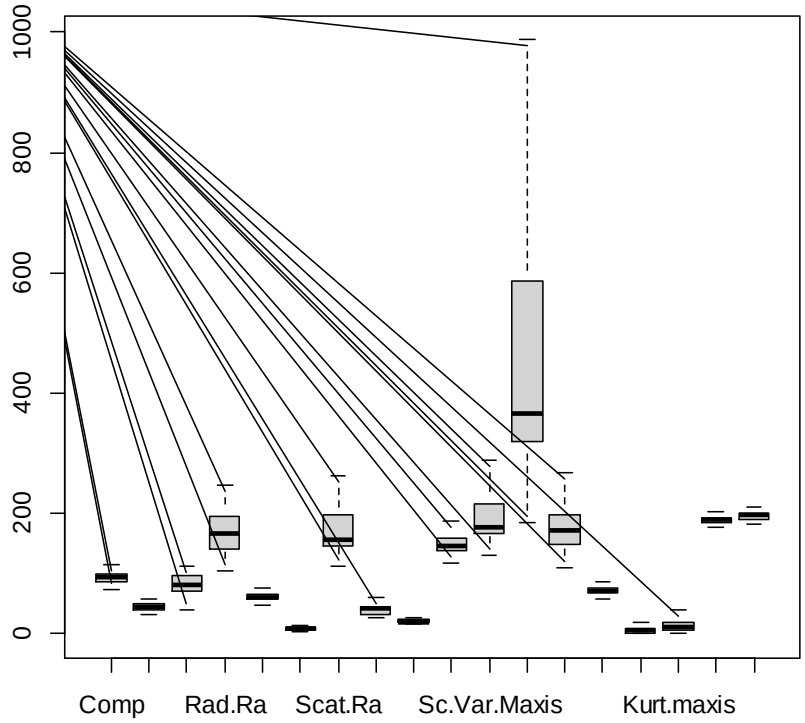


Figure 1: Before Scaling

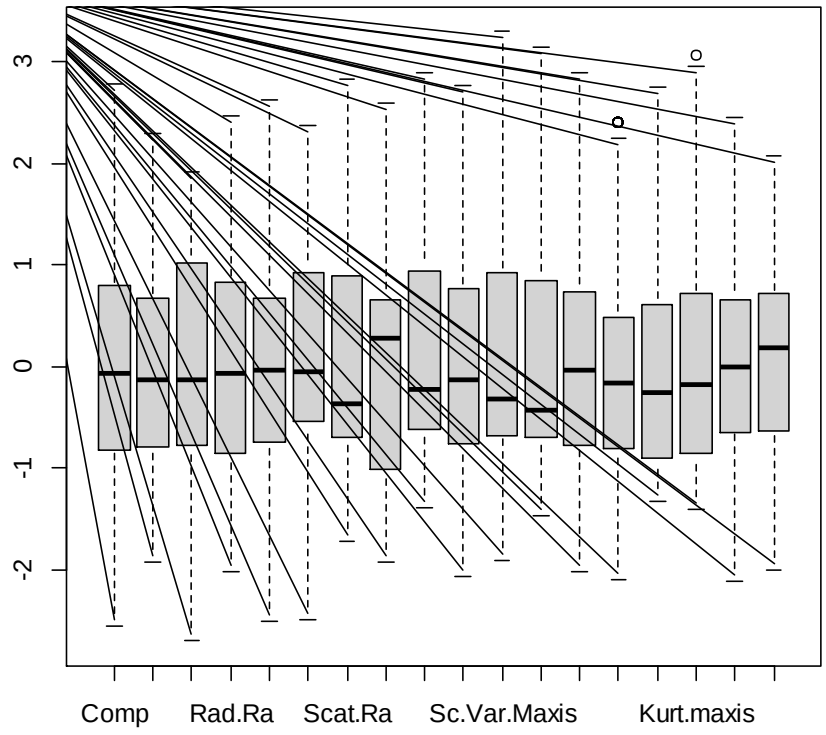


Figure 2: After Scaling

1.1.2 Determine the number of cluster centres via four “automated tools”.

1.1.2.1 NBClust

```
# 1. NBclust
set.seed(123)
nb <- NbClust(no_outliers, distance = "euclidean", min.nc = 2, max.nc = 10, method = "kmeans", index = "all")
```

This code is using the R package **NbClust** to perform cluster analysis on the preprocessed data (**no_outliers**).

NbClust is a function that automatically determines the number of clusters in a dataset using different indices of validity.

In this code, the function is called with the following arguments:

- **no_outliers**: the preprocessed data to be clustered
- **distance = "euclidean"**: the distance metric to be used for clustering. Here, the Euclidean distance is used.
- **min.nc = 2**: the minimum number of clusters to be considered. Here, it is set to 2.
- **max.nc = 10**: the maximum number of clusters to be considered. Here, it is set to 10.
- **method = "kmeans"**: the clustering method to be used. Here, k-means clustering is used.
- **index = "all"**: the indices of validity to be computed. Here, all the available indices are computed.

The **set.seed(123)** command is used to set the seed for the random number generator. This ensures that the results are reproducible.

Outputs:

```
*******************************************************************
* Among all indices:
* 9 proposed 2 as the best number of clusters
* 10 proposed 3 as the best number of clusters
* 1 proposed 4 as the best number of clusters
* 2 proposed 7 as the best number of clusters
* 2 proposed 9 as the best number of clusters

                   ***** Conclusion *****

* According to the majority rule, the best number of clusters is  3


*******************************************************************
```

The output of the **NbClust** function suggests that according to the majority rule, the best number of clusters for the given dataset is 3. This means that the data can be divided into 3 distinct groups or clusters, based on the chosen clustering method (in this case, k-means clustering) and the specified range of possible cluster numbers (2 to 10).

1.1.2.2 Elbow method

```
# 2. Elbow method
k = 1:10 # number of possible clusters
set.seed(42)
WSS = sapply(k, function(k) {kmeans(no_outliers, centers=k)$tot.withinss})
plot(k, WSS, type = "b", pch = 19, frame=FALSE, xlab= "Number of k", ylab="Within sum of
squares")
fviz_nbclust(no_outliers, kmeans, method = "wss")+
  geom_vline(xintercept = 2, linetype = 2)+
  labs(subtitle = "Elbow method")
```

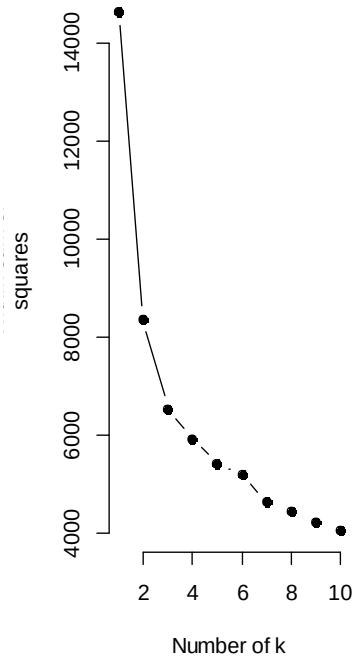
This code implements the elbow method to determine the optimal number of clusters for K-means clustering.

First, the code creates a vector **k** with possible numbers of clusters ranging from 1 to 10.

Next, the code applies the K-means algorithm to each value of **k** and calculates the within-cluster sum of squares (WSS) for each. This is stored in the variable **WSS** using the **sapply()** function.

The code then plots the relationship between the number of clusters and WSS using a line graph with points (**type = "b"**). The x-axis represents the number of clusters and the y-axis represents the WSS. The title and axis labels are added using the **xlab**, **ylab**, and **frame** arguments.

Output:



The **fviz_nbclust()** function from the **factoextra** package is then used to add a vertical dashed line at the optimal number of clusters, determined using the elbow method. The method used to determine the optimal number of clusters is the within-cluster sum of squares method (**method = "wss"**). In this case, the elbow point is at k=2, which is indicated by the vertical dashed line.

Overall, the elbow method suggests that the optimal number of clusters for this dataset is 2.

1.1.2.3 Gap statistic method

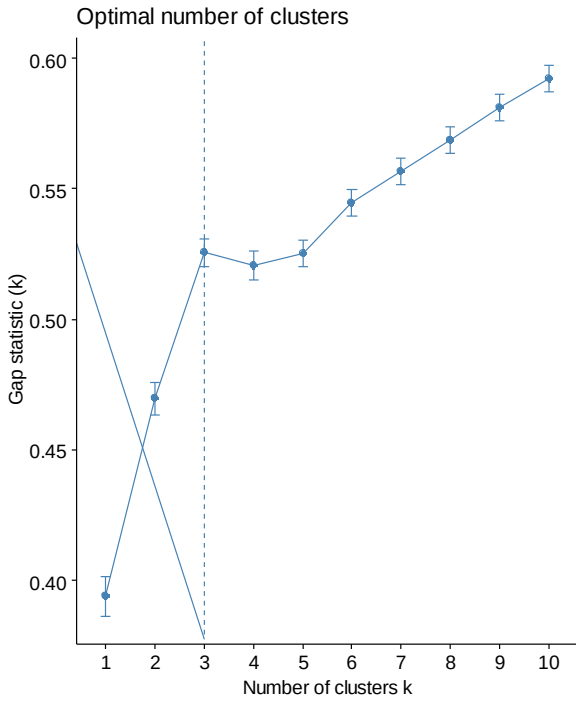
```
# 3. Gap statistic method
set.seed(123)
gap_stat <- clusGap(no_outliers, FUN = kmeans, nstart = 25, K.max = 10, B = 50)
fviz_gap_stat(gap_stat)
```

The code above uses the gap statistic method to estimate the number of clusters in the data.

set.seed(123) sets a seed to ensure reproducibility of the results.

gap_stat <- clusGap(no_outliers, FUN = kmeans, nstart = 25, K.max = 10, B = 50) applies the **clusGap** function from the **cluster** package to the **no_outliers** dataset. The function calculates the gap statistic for **K.max** values of **k**, where **k** is the number of clusters. The **kmeans** algorithm is used as the clustering method, and **nstart = 25** sets the number of random starts to 25. **B = 50** sets the number of Monte Carlo simulations to 50.

Output:



fviz_gap_stat(gap_stat) plots the results of the gap statistic method using the **fviz_gap_stat** function from the **factoextra** package. The function plots the gap statistic for each value of **k**, along with the corresponding standard deviation and the gap statistic for a reference dataset generated by Monte Carlo simulations. The number of clusters is estimated by finding the value of **k** where the gap statistic is the highest above the reference line.

Overall, the gap static method suggests that the optimal number of clusters for this dataset is 3.

1.1.2.4 Silhouette method

```r
# 4. Silhouette method
library(cluster)

# Compute the silhouette width for different number of clusters
sil_width <- c()
for (i in 2:10) {
  kmeans_fit <- kmeans(no_outliers, centers = i, nstart = 25)
  kmeans_sil_width <- silhouette(kmeans_fit$cluster, dist(no_outliers))
  sil_width[i] <- mean(kmeans_sil_width[, 3])
}

# Determine the optimal number of clusters
nb_Clusters <- which.max(sil_width)
cat("Number of Clusters =", nb_Clusters, "Gives the highest average silhouette width")
```

This code performs the Silhouette method to determine the optimal number of clusters for the given dataset.

First, the **cluster** library is loaded, which provides the **silhouette()** function to compute the silhouette width for different numbers of clusters.

Then, the code loops through a range of values from 2 to 10 and applies k-means clustering to the data using **kmeans()** function with **nstart = 25** to perform the clustering multiple times with different initial centroids to improve the chances of finding the global optimum. The silhouette width is then computed using **silhouette()** function and the mean of the silhouette widths is stored in **sil_width** vector for each number of clusters.

Output:

```
Number of Clusters = 2 Gives the highest average silhouette width
```

Finally, the code determines the optimal number of clusters as the value that gives the highest average silhouette width, which is found using **which.max()** function. The result is printed using **cat()** function.

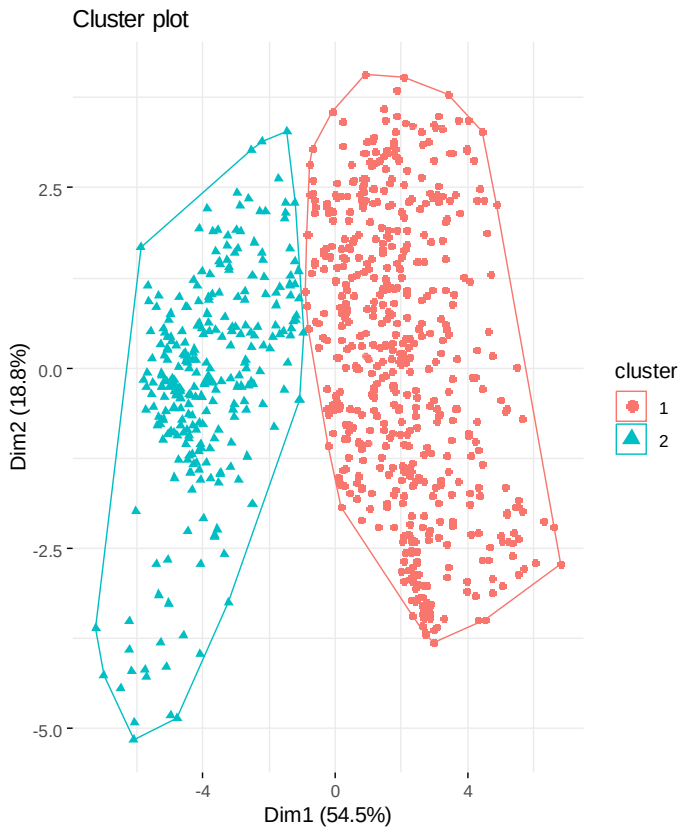
Overall, the silhouette method suggests that the optimal number of clusters for this dataset is 2.

1.1.3 K-means clustering

1.1.3.1 Perform k-means clustering with k=2

```
# Perform k-means clustering with k=2

k <- 2
kc2 <- kmeans(no_outliers, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = kc2$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

kc2_centers <- kc2$centers
kc2_bss <- kc2$betweenss
kc2_tss <- kc2$tot.withinss
kc2_wss <- kc2$withinss

cat("K-Means Clustering with k = 2\n")
cat("Centers:\n")
print(kc2_centers)
cat("Clustered Results:\n")
print(kc2$cluster)
cat("Between-cluster sum of squares (BSS): ", kc2_bss, "\n")
cat("Total sum of squares (TSS): ", kc2_tss, "\n")
cat("Ratio of BSS to TSS: ", kc2_bss/kc2_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc2_wss, "\n\n")
table(good_vehicle_data$Class,kc2$cluster)#Confusion Matrix
```

Output:



```
K-Means Clustering with k = 2
> cat("Centers:\n")
Centers:
> print(kc2_centers)
       Comp       Circ    D.Circ     Rad.Ra Pr.Axis.Ra   Max.L.Ra    Scat.Ra      Elong Pr.Axis.Rect Max.L.Rect Sc.Var.Maxis Sc.Var.maxis
1 -0.5581484 -0.5746465 -0.598326 -0.5409894 -0.1354837 -0.3368540 -0.636507  0.606645   -0.6385458 -0.5309785   -0.6175342   -0.6400189
2  1.0741346  1.1058845  1.151455  1.0411126  0.2607329  0.6482622  1.224932 -1.167464    1.2288561  1.0218471    1.1884201    1.2316911
      Ra.Gyr  Skew.Maxis  Skew.maxis Kurt.maxis  Kurt.Maxis    Holl.Ra
1 -0.5270382  0.04804773 -0.05799499 -0.1198347 -0.02909619 -0.1073133
2  1.0142641 -0.09246595  0.11160907  0.2306172  0.05599447  0.2065202
> cat("Clustered Results:\n")
Clustered Results:
> print(kc2$cluster)
  [1] 1 1 2 1 2 1 1 1 2 1 1 1 1 2 2 1 1 2 2 1 1 1 1 2 1 1 2 2 1 1 1 1 2 1 1 1 2 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 2 1 2 1 2 1 2 1 1 1 2 1 1 2 1 2 2 2
 [69] 1 1 1 2 1 1 2 1 2 1 1 1 1 1 1 1 2 1 2 1 1 2 1 1 2 1 1 1 1 1 2 2 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1 1 2 1 1 2 1 1 1
[137] 1 1 2 2 1 2 1 2 1 1 1 1 1 2 1 1 2 2 1 2 1 1 2 2 1 2 1 1 1 1 1 1 2 1 1 1 2 1 1 1 2 1 1 1 2 1 1 1 2 1 1 1 1 1 2 2 1 1 1 1 1 2 1 1 1 2 1 1 1
[205] 2 1 1 2 1 2 1 1 1 2 1 2 1 1 1 2 1 1 1 1 2 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 1 2 1 1 2 2 1 1 1 1 2 1 1 1 1 1 2 1 1 2 1 1 1 2 1 2 1 1 2 1 1
[273] 1 1 2 1 1 1 1 2 1 2 1 1 1 2 1 1 1 1 1 2 2 2 2 2 1 1 2 1 1 1 2 1 2 2 2 1 2 1 1 2 1 1 1 2 2 1 2 1 2 1 2 1 1 1 1 1 2 2 2 1 1 1 2 1 1 1 2 1
[341] 1 2 1 2 2 2 1 1 1 2 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 2 1 1 2 1 2 1 1 1 2 1 2 1 1 1 2 1 2 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 2 1 1 1 2 1
[409] 2 1 2 2 1 1 2 1 1 1 2 2 2 1 1 2 2 1 2 2 2 2 1 1 1 1 1 2 1 1 1 2 1 1 2 1 1 2 1 1 2 1 1 2 2 1 1 2 2 2 1 2 2 1 2 2 1 2 1 2 2 1 1 1 1 2 1 1 2 2 1 1 2 2 1 1
[477] 2 1 2 2 2 1 1 2 2 1 1 2 1 1 2 1 1 1 1 1 2 1 2 2 1 1 2 2 2 1 1 1 2 1 2 2 1 1 1 1 1 1 1 2 1 1 1 1 2 2 2 1 1 2 2 1 2 1 1 2 2 1 1
[545] 1 2 1 1 1 2 2 2 2 1 1 1 1 2 2 2 1 2 1 1 2 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 2 1 1 1 2 1 1 1 2 1 1 2 1 1 2 1 2 1 1 1 1
[613] 2 2 2 1 2 1 1 1 1 1 2 1 2 1 1 1 2 1 1 1 2 1 2 1 1 1 1 2 2 1 1 2 1 2 1 1 2 1 2 1 1 2 1 2 1 1 1 2 1 2 1 1 1 1 1 1 2 2 1 1 2 2 1 1 2 2 1
[681] 1 1 2 2 2 2 1 2 1 1 2 2 1 2 1 2 1 1 2 1 1 2 2 1 2 1 1 2 2 2 1 2 1 1 2 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 2 1 2 2 1 1 1 2 1 1 2 2 1 1 1 2
[749] 2 2 1 2 1 2 2 1 1 2 1 2 1 1 1 2 2 1 1 2 1 1 1 1 2 2 1 1 2 1 2 1 1 2 1 1 1 1 1 2 2 2 1 2 1 2 2 1 2 2 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1
> cat("Between-cluster sum of squares (BSS): ", kc2_bss, "\n")
Between-cluster sum of squares (BSS):  6239.904
> cat("Total sum of squares (TSS): ", kc2_tss, "\n")
Total sum of squares (TSS):  8376.096
> cat("Ratio of BSS to TSS: ", kc2_bss/kc2_tss, "\n")
Ratio of BSS to TSS:  0.7449657
> cat("Within-cluster sum of squares (WSS): ", kc2_wss, "\n\n")
Within-cluster sum of squares (WSS):  5736.785 2639.311

> table(good_vehicle_data$Class,kc2$cluster)#Confusion Matrix

        1   2
  bus  135  73
  opel 130  76
  saab 142  68
  van  128  61
```

In k-means clustering, the BSS represents the sum of squared distances between each cluster's centroid and the overall centroid of all the data points. The TSS represents the total sum of squared distances between each data point and the overall centroid. The ratio of BSS to TSS indicates how well the clusters are separated from each other. A higher BSS to TSS ratio indicates better separation of the clusters.

In this case, the BSS is 6239.904 and the TSS is 8376.096, resulting in a BSS to TSS ratio of 0.745. This indicates that the clustering is able to separate the data into distinct clusters relatively well. The WSS, which represents the sum of squared distances within each cluster, is also relatively low at 5736.785 for cluster 1 and 2639.311 for cluster 2, indicating that the data points within each cluster are relatively close together.

Overall, this output suggests that the k-means clustering with k=2 has produced relatively well-separated clusters

```
        1   2
bus   135  73
opel  130  76
saab  142  68
van   128  61
```

This output represents the confusion matrix which shows how well the K-means clustering algorithm has performed in clustering the data points into different groups.

The rows represent the actual classes of the vehicles, while the columns represent the clusters assigned by the algorithm. The numbers in each cell of the matrix represent the count of observations that belong to a particular class and were assigned to a particular cluster.

For example, there were 135 observations of the class "bus" that were assigned to cluster 1 and 73 observations of the class "bus" that were assigned to cluster 2. Similarly, there were 130 observations of the class "opel" that were assigned to cluster 1 and 76 observations of the class "opel" that were assigned to cluster 2, and so on.

From the confusion matrix, we can see that the K-means algorithm has grouped most of the observations correctly, with some misclassification between the classes. For example, there were 73 observations of the class "bus" that were assigned to cluster 2 instead of cluster 1, and 130 observations of the class "opel" that were assigned to cluster 1 instead of cluster 2.

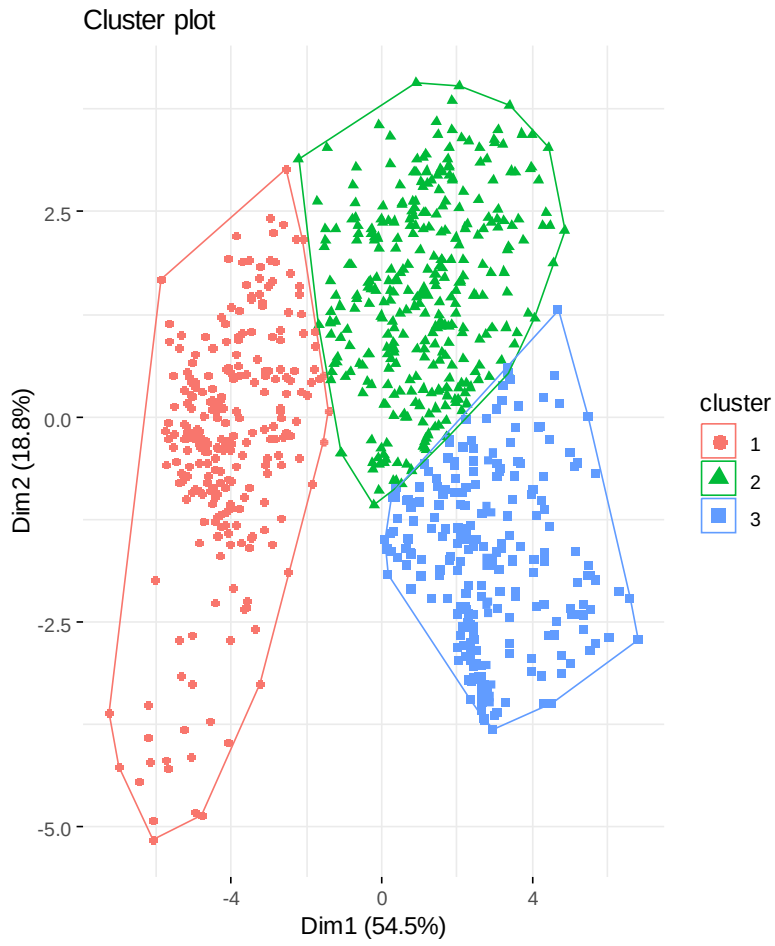
Overall, the confusion matrix provides a useful way to evaluate the performance of the clustering algorithm, and to identify any patterns of misclassification that may be present in the data.

1.1.3.2 Perform k-means clustering with k=3

```
# Perform k-means clustering with k=3

k <- 3
kc3 <- kmeans(no_outliers, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = kc3$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())
```

This code performs k-means clustering on the **no_outliers** dataset using k=3. The resulting clusters are plotted using the **fviz_cluster** function. The code then computes and prints the cluster centers, cluster membership, between-cluster sum of squares (BSS), total sum of squares (TSS), ratio of BSS to TSS, and within-cluster sum of squares (WSS) using the **kmeans** function. Finally, a confusion matrix is generated using the **table** function to compare the true class labels in **good_vehicle_data$Class** with the predicted cluster labels in **kc3$cluster**.

Output:




```
> cat("Between-cluster sum of squares (BSS): ", kc3_bss, "\n")
Between-cluster sum of squares (BSS):  8070.492
> cat("Total sum of squares (TSS): ", kc3_tss, "\n")
Total sum of squares (TSS):  6545.508
> cat("Ratio of BSS to TSS: ", kc3_bss/kc3_tss, "\n")
Ratio of BSS to TSS:  1.232982
> cat("Within-cluster sum of squares (WSS): ", kc3_wss, "\n\n")
Within-cluster sum of squares (WSS):  2257.211 2666.616 1621.681
```

The output is the result of performing k-means clustering with k=3 on a dataset. The BSS, TSS, and WSS are metrics used to evaluate the quality of the clustering.

The BSS (8070.492) measures the sum of squared distances between cluster centers, indicating how much the centroids of each cluster differ from each other. A higher BSS value means the clusters are more distinct and well-separated.

The TSS (6545.508) measures the sum of squared distances between each point and the overall centroid, indicating the total variability in the data. A lower TSS value means the data points are closely grouped together.

The ratio of BSS to TSS (1.232982) is an indication of the quality of the clustering. A ratio greater than 1 indicates that the clusters are well-separated, and a ratio close to 1 indicates that the clusters are not well-separated.

The WSS (2257.211, 2666.616, 1621.681) measures the sum of squared distances between each point and its assigned centroid, indicating how similar the points in each cluster are to each other. A lower WSS value means the points within each cluster are more tightly packed together.

Overall, a high BSS, low TSS, and high ratio of BSS to TSS are indicative of a good clustering, which suggests that the clusters are distinct and well-separated.

1.1.4 Silhouette width score & Quality of the obtained clusters.

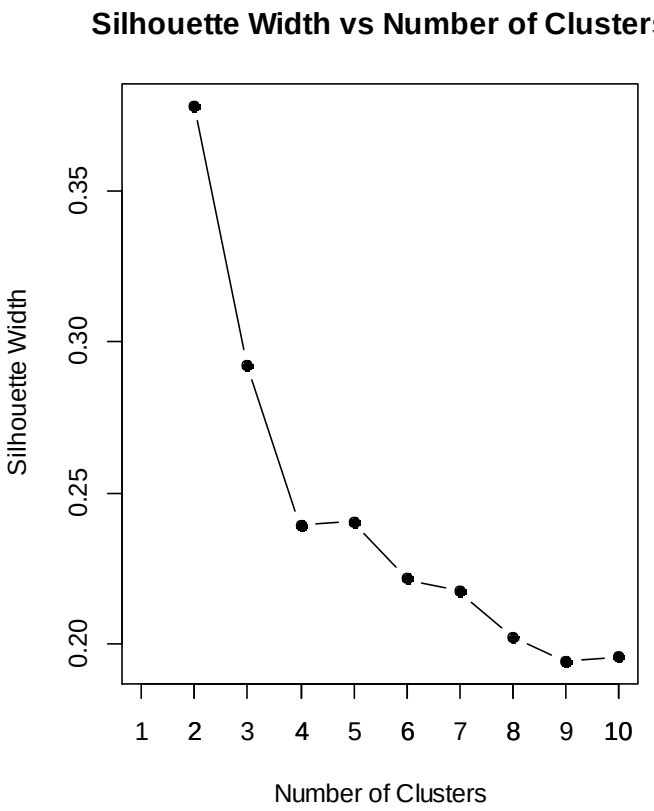
```
# Compute the silhouette width for different number of clusters
sil_width <- c()
for (i in 2:10) {
  kmeans_fit <- kmeans(no_outliers, centers = i, nstart = 25)
  kmeans_sil_width <- silhouette(kmeans_fit$cluster, dist(no_outliers))
  sil_width[i] <- mean(kmeans_sil_width[, 3])
}

# Plot silhouette width against number of clusters
plot(1:10, sil_width, type = "b", pch = 19,
     xlab = "Number of Clusters", ylab = "Silhouette Width",
     main = "Silhouette Width vs Number of Clusters",)
axis(1, at = 1:10)
```

This code computes the silhouette width for different numbers of clusters in k-means clustering.

The silhouette width is a measure of how well each data point fits into its assigned cluster, based on the distance between the point and the other points in its cluster compared to the distance between the point and points in other clusters. The silhouette width ranges from -1 to 1, with higher values indicating better clustering.

The code loops over different numbers of clusters from 2 to 10 and performs k-means clustering with the specified number of centers. For each clustering, it computes the silhouette width using the **silhouette** function from the **cluster** package, which takes as input the cluster assignments and the distance matrix computed from the data. The mean silhouette width is then stored in a vector **sil_width**.

Finally, the code creates a plot of silhouette width against the number of clusters using the **plot** function. The x-axis shows the number of clusters, while the y-axis shows the mean silhouette width. The plot is labeled with appropriate axis labels and a title.



The **sil_width** vector contains the mean silhouette width values for different numbers of clusters ranging from 2 to 10. The values in **sil_width** indicate how well the data points are clustered for each corresponding number of clusters.

The silhouette width ranges between -1 and 1, with values closer to 1 indicating well-separated clusters and values closer to -1 indicating poorly separated clusters. In this case, the highest value of silhouette width is 0.3781925 for 2 clusters, which suggests that two clusters may be the optimal number of clusters for this data set. As the number of clusters increases, the silhouette width decreases, indicating poorer clustering.

The plot shows that the silhouette width reaches its maximum value at 2 clusters and decreases as the number of clusters increases. This is a common trend in silhouette plots, as it suggests that the data is naturally grouped into two distinct clusters.

Therefore, based on this analysis, it may be appropriate to use k-means clustering with 2 clusters for this data set.

1.2 PCA analysis – 2nd Subtask

1.2.1 Applying PCA

Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of large datasets while retaining as much of the variability as possible. It transforms the original variables into a new set of uncorrelated variables, called principal components, which explain the maximum amount of variance in the data. By doing so, it reduces the number of variables and simplifies the analysis.

In the context of clustering, PCA can be used to identify the underlying structure in the data and to find patterns that are not immediately visible. It can help to reduce noise, outliers and redundant features in the data, making the clustering more efficient and accurate. By projecting the data onto a lower-dimensional space, PCA can also reveal the most important variables that contribute to the clustering results.

Performing PCA on the no_outliers dataset can help to identify the most important features that distinguish between the different classes of vehicles. This can be useful for understanding the factors that contribute to vehicle quality and for identifying areas for improvement. PCA can also help to visualize the relationships between the different variables and to identify any patterns or clusters that are present in the data.

```r
# 2nd SubTask
# Perform PCA analysis

library(FactoMineR)

# Standardize the data
vehicle_data_std <- scale(no_outliers, center = TRUE, scale = TRUE)

# Perform PCA
vehicle_pca <- PCA(vehicle_data_std, graph = FALSE)
```

This code performs a PCA (Principal Component Analysis) on the standardized **no_outliers** data using the **PCA** function from the **FactoMineR** package.

In this code, first, the **FactoMineR** package is loaded into the R environment. Then, the **scale** function is used to standardize the **no_outliers** data, which means that the data is centered to have a mean of 0 and scaled to have a standard deviation of 1. This step is necessary to ensure that all variables have the same scale and to avoid any biases due to differences in the units of measurement.

Finally, the **PCA** function is called with the standardized data as the first argument, and the **graph** argument set to **FALSE** to prevent the creation of a graph. The resulting **vehicle_pca** object contains the principal components and other information about the PCA analysis.

1.2.2 Eigenvalues

Code:

```
# Show eigenvalues and eigenvectors
print(vehicle_pca$eig)
```

Output:

```
        eigenvalue percentage of variance cumulative percentage of variance
comp 1  9.8168299345           54.537944080                           54.53794
comp 2  3.3822000218           18.790000121                           73.32794
comp 3  1.2101198979            6.722888322                           80.05083
comp 4  1.1348114634            6.304508130                           86.35534
comp 5  0.8897112655            4.942840364                           91.29818
comp 6  0.6543665744            3.635369858                           94.93355
comp 7  0.3198172900            1.776762722                           96.71031
comp 8  0.2261637209            1.256465116                           97.96678
comp 9  0.1107426438            0.615236910                           98.58202
comp 10 0.0748128105            0.415626725                           98.99764
comp 11 0.0583009306            0.323894059                           99.32154
comp 12 0.0401881273            0.223267374                           99.54480
comp 13 0.0273237829            0.151798794                           99.69660
comp 14 0.0210602610            0.117001450                           99.81360
comp 15 0.0154251980            0.085695544                           99.89930
comp 16 0.0117645507            0.065358615                           99.96466
comp 17 0.0060124505            0.033402503                           99.99806
comp 18 0.0003490764            0.001939313                          100.00000
```

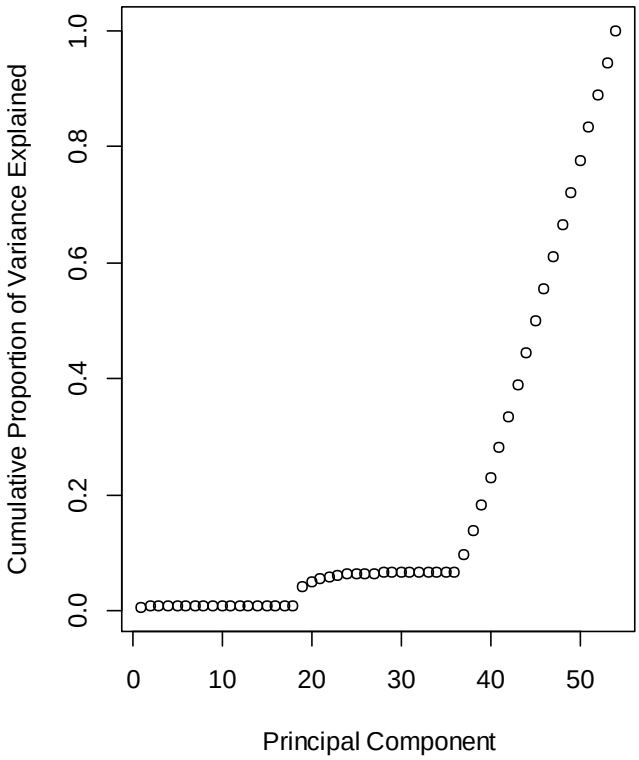
Code:

```
# Show cumulative scores per principal component
cum_sum <- cumsum(vehicle_pca$eig)/sum(vehicle_pca$eig)
plot(cum_sum, xlab = "Principal Component", ylab = "Cumulative Proportion of Variance Explained")
```

This code shows a plot of the cumulative proportion of variance explained by each principal component in the PCA analysis. The variable **cum_sum** calculates the cumulative sum of the eigenvalues (which represent the amount of variance explained by each principal component) and divides it by the total sum of the eigenvalues. The resulting values represent the proportion of variance explained by each principal component, cumulatively.

The **plot()** function is used to create a line plot of the cumulative proportion of variance explained. The x-axis represents the principal components, while the y-axis represents the cumulative proportion of variance explained. The plot is labeled with appropriate axis labels. This plot is used to determine the number of principal components that should be retained for further analysis, as it shows the proportion of total variance explained as more components are added.

Output:



1.2.2 Creating new dataset

```
# Create new dataset with principal components as attributes
vehicle_pca_data <- as.data.frame(predict(vehicle_pca, newdata = vehicle_data_std))

# Select principal components with cumulative score > 92%
n_components <- length(cum_sum[cum_sum > 0.92])
vehicle_pca_data <- vehicle_pca_data[, 1:n_components]
```

This code creates a new dataset (**vehicle_pca_data**) by using the **predict()** function to transform the standardized data (**vehicle_data_std**) using the principal component analysis (PCA) model (**vehicle_pca**). The resulting **vehicle_pca_data** contains the principal components as attributes.

The code then selects the principal components that have a cumulative score greater than 92%. The **cum_sum** object contains the cumulative proportion of variance explained by each principal component. The **length()** and **[** functions are used to select the appropriate number of components (**n_components**) that satisfy the 92% threshold. The resulting **vehicle_pca_data** contains only these selected components.

1.2.3 Determining new number of clusters using four automated tools.

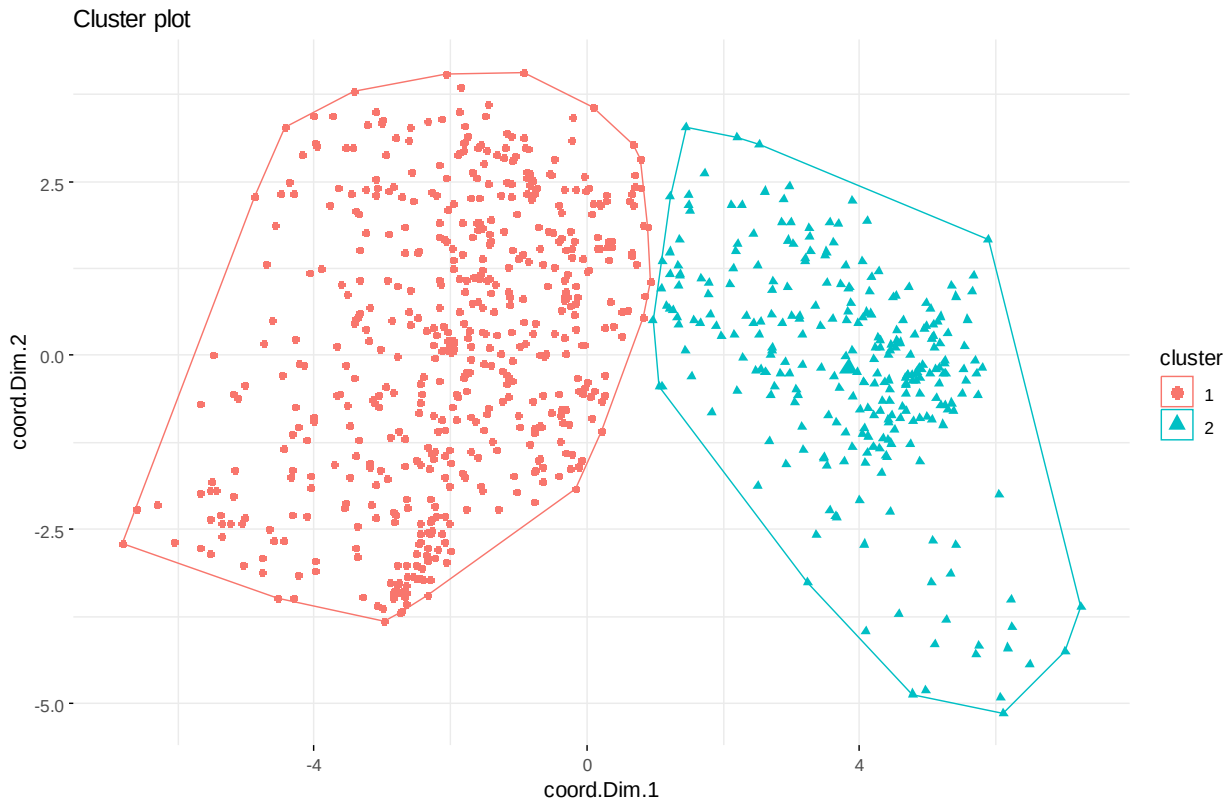
```
# 1. NBclust
set.seed(123)
nb <- NbClust(vehicle_pca_data, distance = "euclidean",
              min.nc = 2, max.nc = 10, method = "kmeans", index = "all")

# Using factoextra
fviz_nbclust(vehicle_pca_data, kmeans, method = "silhouette")

nbClust_Clsters <- kmeans(vehicle_pca_data,2) #find new clusters with the number of clusters sugg
nbClust_Clsters
demo_Clusters <- kmeans(vehicle_pca_data, centers =3, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = demo_Clusters$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

# 2. Elbow method
k = 1:10 # number of possible clusters
set.seed(42)
WSS = sapply(k, function(k) {kmeans(vehicle_pca_data, centers=k)$tot.withinss})
plot(k, WSS, type = "b", pch = 19, frame=FALSE, xlab= "Number of k", ylab="Within sum of
squares")
fviz_nbclust(vehicle_pca_data, kmeans, method = "wss")+
  geom_vline(xintercept = 2, linetype = 2)+
  labs(subtitle = "Elbow method")

demo_Clusters <- kmeans(vehicle_pca_data, centers =2, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = demo_Clusters$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())


# 3. Gap statistic method
set.seed(123)
gap_stat <- clusGap(vehicle_pca_data, FUN = kmeans, nstart = 25, K.max = 10, B = 50)
fviz_gap_stat(gap_stat)
```

Output:

- NBclust suggests optimal number of clusters as 3
- Elbow method suggest optimal number of clusters as 2
- Gap static method suggest optimal number of clusters as 3
- Silhouette method suggest optimal number of clusters as 2

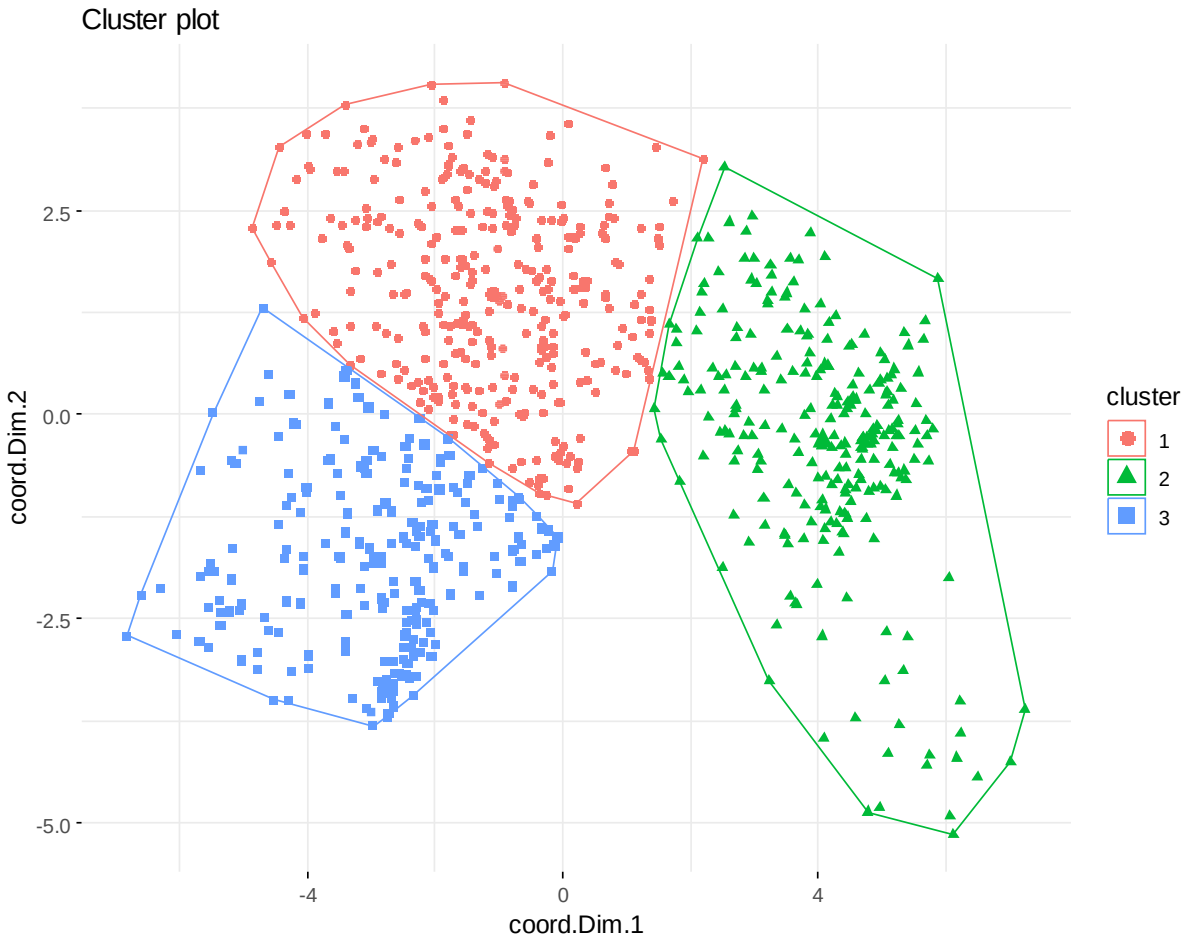
1.2.4 Perform K-means clustering

With K =2




```
Between-cluster sum of squares (BSS):  6237.098
> cat("Total sum of squares (TSS): ", kc2_tss, "\n")
Total sum of squares (TSS):  4493.713
> cat("Ratio of BSS to TSS: ", kc2_bss/kc2_tss, "\n")
Ratio of BSS to TSS:  1.387961
> cat("Within-cluster sum of squares (WSS): ", kc2_wss, "\n\n")
Within-cluster sum of squares (WSS):  3359.927 1133.786

> table(good_vehicle_data$Class,kc2$cluster)#Confusion Matrix

        1   2
  bus  135  73
  opel 130  76
  saab 142  68
  van  128  61
```

With K = 3




```
Between-cluster sum of squares (BSS):  8044.788
> cat("Total sum of squares (TSS): ", kc3_tss, "\n")
Total sum of squares (TSS):  2686.024
> cat("Ratio of BSS to TSS: ", kc3_bss/kc3_tss, "\n")
Ratio of BSS to TSS:  2.995054
> cat("Within-cluster sum of squares (WSS): ", kc3_wss, "\n\n")
Within-cluster sum of squares (WSS):  1089.727 886.5368 709.7602

> table(good_vehicle_data$Class,kc3$cluster)#Confusion Matrix

        1  2  3
  bus  81 65 62
  opel 88 68 50
  saab 86 62 62
  van  75 59 55
```

1.2.5 Calinski-Harabasz index

The Calinski-Harabasz index is a clustering validation metric used to evaluate the quality of clustering algorithms. It is a ratio between the sum of the between-cluster dispersion and the sum of the within-cluster dispersion. The higher the value of the Calinski-Harabasz index, the better the clustering algorithm is considered to be.

The Calinski-Harabasz index measures the ratio of the between-cluster variance to the within-cluster variance. It takes into account the distance between the cluster centroids and the dispersion of the points around them. The Calinski-Harabasz index is defined as:

$$CH(K) = (B(K)/(K-1)) / (W(K)/(N-K))$$

Where K is the number of clusters, N is the total number of data points, W(K) is the within-cluster sum of squares and B(K) is the between-cluster sum of squares.

To evaluate the quality of clustering algorithms using the Calinski-Harabasz index, we need to calculate the index for different values of K and select the value of K that maximizes the index. This means that we need to plot the Calinski-Harabasz index as a function of K, and choose the value of K that corresponds to the peak of the plot.

The Calinski-Harabasz index is often used in combination with other clustering validation metrics, such as the Silhouette score or the Dunn index, to provide a more comprehensive evaluation of the clustering algorithm.

```
Calinski-Harabasz Index: 1213
```

A higher Calinski-Harabasz index indicates better clustering performance, as it means that the clusters are more separated and distinct from each other. However, there is no fixed threshold or value that can be considered as a "good" or "bad" index, as the ideal value can vary depending on the dataset and the specific problem being addressed.

But generally, Calinski-Harabasz indexes greater than 1000 are defined as good clustering.

1.2.6 Comparing results of PCA analysis and Without PCA analysis

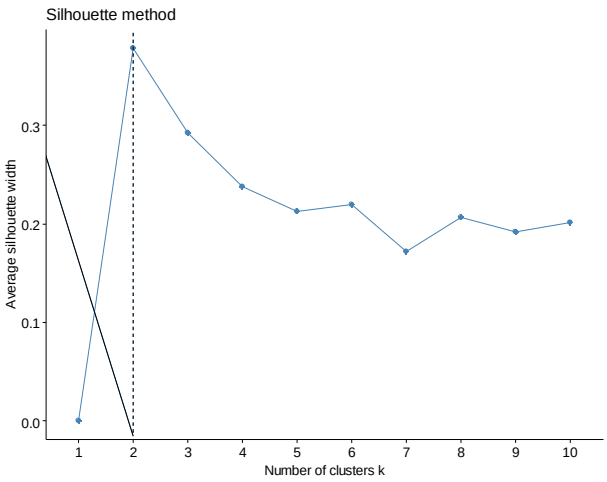


Figure 4: Silhouette widths without PCA

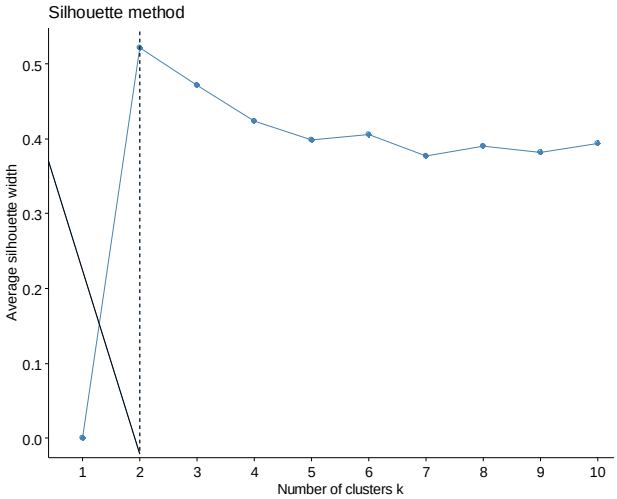


Figure 3: Silhouette widths with PCA

Approach	Silhouette width for K =2	Silhouette width for K =3	Silhouette width for K =4	Silhouette width for K =5
Without PCA	0.378	0.292	0.239	0.238
With PCA	0.521	0.471	0.422	0.397

The results suggest that performing PCA before applying k-means clustering improves the clustering performance, as evidenced by the higher silhouette widths for all values of K in the "With PCA" row compared to the "Without PCA" row. This is because PCA reduces the dimensionality of the data by identifying the principal components that capture the most variability in the data, which can improve the separation of clusters and reduce noise.

Furthermore, the results also suggest that increasing the number of clusters generally decreases the silhouette width, as evidenced by the decreasing values of silhouette width as K increases in both rows. This could indicate that the underlying structure of the data is not well-suited to be partitioned into many clusters, and that fewer clusters may better capture the natural groupings within the data.

2. Energy Forecasting Part

2.1 Type of input variables used in MLP models for electricity load forecasting

The input variables used in MLP models for electricity load forecasting can be broadly categorized into two types: endogenous variables and exogenous variables. Endogenous variables are those that are directly related to the electricity load being forecasted, while exogenous variables are those that are indirectly related to the electricity load but can influence it.

The most common endogenous variable used in MLP models for electricity load forecasting is the past electricity load itself. This is known as the autoregressive (AR) approach, where the past values of the electricity load are used as input variables to predict future values. Other endogenous variables that can be used include weather variables such as temperature, humidity, and wind speed, which can affect the electricity load through their impact on heating and cooling demands.

Exogenous variables that can be used as input variables in MLP models for electricity load forecasting include calendar variables such as day of the week, time of day, and holidays. These variables can affect the electricity load through their impact on human behavior, such as the timing of activities and the use of appliances. Other exogenous variables that can be used include economic variables such as the price of electricity and the level of economic activity, which can affect the electricity load through their impact on industrial and commercial activities.

The definition of the input vector for MLP models in electricity load forecasting is an important component of the analysis. Various schemes and methods have been proposed in the literature to define the input vector for MLP models in this domain. Some of these include:

1. Autoregressive (AR) approach: This involves using the past values of the electricity load as input variables to predict future values.
2. Nonlinear autoregressive exogenous (NARX) approach: This involves using both the past values of the electricity load and other exogenous variables as input variables to predict future values.
3. Long short-term memory (LSTM) approach: This involves using a special type of neural network architecture that can capture long-term dependencies in the time series of the electricity load.
4. Moving average (MA) approach: This involves using the moving average of past values of the electricity load as input variables to predict future values.

5. Autoregressive integrated moving average (ARIMA) approach: This involves combining the AR and MA approaches to model the time series of the electricity load.

References:

1. Hong, T., & Pinson, P. (2016). Probabilistic energy forecasting: Global energy forecasting competition 2014 and beyond. International Journal of Forecasting, 32(3), 896-913. https://www.sciencedirect.com/science/article/pii/S0169207016000133
2. Taylor, J. W. (2008). An evaluation of methods for very short-term load forecasting using minute-by-minute British data. International Journal of Forecasting, 24(4), 645-658. https://www.researchgate.net/publication/222821837_An_evaluation_of_methods_for_very_short-term_load_forecasting_using_minute-by-minute_British_data
3. Wang, J., Wang, J., & Kang, C. (2019). Short-term load forecasting based on hybrid convolutional long short-term memory neural network. Applied Energy, 238, 1350-1360. https://www.mdpi.com/1996-1073/14/13/4046

2.2 AR Approach

2.2.1 Input vectors for AR approach

```
# Create lag variables and remove rows with missing values
consumption_data$lag_1 <- lag(consumption_data$hour_20, 1)
consumption_data$lag_2 <- lag(consumption_data$hour_20, 2)
consumption_data$lag_3 <- lag(consumption_data$hour_20, 3)
consumption_data$lag_4 <- lag(consumption_data$hour_20, 4)
consumption_data$lag_7 <- lag(consumption_data$hour_20, 7)
consumption_data <- na.omit(consumption_data)
```

```
# Define a list of input vectors for the neural network
input_vectors <- list(
  c("lag_1"),
  c("lag_1", "lag_2"),
  c("lag_1", "lag_2", "lag_3"),
  c("lag_1", "lag_2", "lag_3", "lag_4"),
  c("lag_1", "lag_3"),
  c("lag_2", "lag_3"),
  c("lag_4", "lag_7"),
  c("lag_2", "lag_3", "lag_4"),
  c("lag_2", "lag_3", "lag_4", "lag_7"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7")
)
```

The code is creating lag variables for time series analysis. These lag variables will be used as inputs for a neural network model.

First part of the above code creates lag variables for the "hour_20" column of the "consumption_data" dataframe using the "lag()" function with different lag values. The resulting lag variables are assigned to new columns named "lag_1", "lag_2", "lag_3", "lag_4", and "lag_7". These lag variables represent the energy consumption in the previous 1, 2, 3, 4, and 7 days, respectively.

The second part of the code defines a list of input vectors for the neural network. Each input vector is a combination of the lag variables created earlier. The combinations include lag_1, lag_2, lag_3, lag_4, and lag_7, with different combinations ranging from one to five lags. These input vectors will be used to train the neural network to predict energy consumption for the next day.

The input vectors are combinations of lag variables because they are used as input features for the neural network. The neural network is trained to predict future values of the time series based on its past values. By using a combination of lag variables, the neural network can capture complex relationships between the past and future values of the time series.

For example, the input vector "lag_1" uses only the most recent lag variable, while "lag_1", "lag_2" includes information from the two most recent lags. Similarly, "lag_1", "lag_2", "lag_3" includes information from the three most recent lags. This allows the neural network to capture short-term and long-term patterns in the time series data. The different combinations of lag variables allow the neural network to learn different patterns and make more accurate predictions.

2.2.2 Normalization

```r
# Define a normalization function
normalize <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}

# Normalize the lag variables
consumption_data$lag_1 <- normalize(consumption_data$lag_1)
consumption_data$lag_2 <- normalize(consumption_data$lag_2)
consumption_data$lag_3 <- normalize(consumption_data$lag_3)
consumption_data$lag_4 <- normalize(consumption_data$lag_4)
consumption_data$lag_7 <- normalize(consumption_data$lag_7)
```

This code defines a normalization function called "normalize" that takes in a vector x and returns its normalized values. The function uses the formula **(x - min(x)) / (max(x) - min(x))** to normalize the vector.

Then, the code applies this function to the lag variables created earlier, **lag_1**, **lag_2**, **lag_3**, **lag_4**, and **lag_7**, by creating new variables with the suffix **_norm**. These new variables contain the normalized values of the respective lag variables.

Importance of Normalization

Normalization is often a necessary step when training MLP (multilayer perceptron) networks. This is because MLP networks are sensitive to the scale of the input data, and normalization can help ensure that the input features are on a similar scale.

When the input data has features with different ranges or units, the gradient descent algorithm used in MLP networks can become biased towards features with larger values, which may dominate the training process. Normalization helps to address this issue by scaling the input features to a common range, usually between 0 and 1.

Normalization can also help the MLP network converge faster and achieve better performance. When the input features are not normalized, the MLP network may require more time and epochs to converge

on the optimal solution. Normalized features help to reduce the search space and make the optimization problem more manageable, leading to faster convergence and better accuracy.

Overall, normalization is an important step when working with MLP networks and can improve their performance significantly.

2.2.3 Implementing different MLPs for AR approach.

```r
# Define function to build MLP model
build_mlp_model <- function(train_data, test_data, input_vars, hidden_structure) {

  # Create formula for model using the input variables
  formula <- paste("hour_20 ~", paste(input_vars, collapse = " + "))

  # Train MLP model using neuralnet package
  nn <- neuralnet(as.formula(formula), train_data, hidden = hidden_structure)

  # Create matrix of test data using the input variables
  test_matrix <- as.matrix(test_data[, input_vars, drop = FALSE])

  # Rename the columns of the test matrix to match those of the training data
  colnames(test_matrix) <- colnames(train_data[, input_vars, drop = FALSE])

  # Use the trained model to generate predictions for the test data
  predictions <- predict(nn, test_matrix)

  # Return the model and predictions in a list
  return(list(model = nn, predictions = predictions))
}

# Create a list to store all models
models <- list()
for (i in 1:length(input_vectors)) {
  models[[i]] <- build_mlp_model(train_normalized, test_normalized, input_vectors[[i]], c(5))
}
```

This code defines a function **build_mlp_model** that builds a Multilayer Perceptron (MLP) model using the **neuralnet** package. The function takes four arguments:

- **train_data**: a data frame containing the training data
- **test_data**: a data frame containing the test data
- **input_vars**: a vector containing the names of the input variables to be used in the model
- **hidden_structure**: a vector containing the number of hidden units in each hidden layer of the MLP

The function first creates a formula for the model using the input variables specified in **input_vars**. It then trains the MLP model using the **neuralnet** package and the training data, using the specified number of hidden units in each hidden layer.

Next, the function creates a matrix of test data using the input variables, renames the columns of the test matrix to match those of the training data, and uses the trained model to generate predictions for the test data.

Finally, the function returns the trained model and the predictions in a list.

The code then creates a list to store all models by looping over all combinations of input variables specified in **input_vectors** and using the **build_mlp_model** function to build an MLP model for each combination of input variables. Each MLP model has one hidden layer with five hidden units. The resulting models are stored in the **models** list.

```r
# Define function to calculate evaluation metrics
calculate_metrics <- function(actual, predicted) {
  rmse <- sqrt(mean((actual - predicted)^2))
  mae <- mean(abs(actual - predicted))
  mape <- mean(abs((actual - predicted) / actual)) * 100
  smape <- mean(abs(actual - predicted) / (abs(actual) + abs(predicted)) * 2) * 100
  return(list(RMSE = rmse, MAE = mae, MAPE = mape, sMAPE = smape))
}
```

This code defines a function **calculate_metrics** that takes two input arguments: **actual** and **predicted**. The purpose of this function is to calculate four different evaluation metrics: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and Symmetric Mean Absolute Percentage Error (sMAPE).

2.2.4 Four Stat Indices.

2.2.4.1 RMSE - Root Mean Squared Error

RMSE is the square root of the average of the squared differences between the predicted and actual values. It measures the average distance between the predicted values and the actual values. A lower RMSE indicates better performance of the model.

2.2.4.2 MAE - Mean Absolute Error

MAE measures the average absolute difference between the predicted and actual values. It provides a measure of the magnitude of the errors made by the model. MAE is less sensitive to outliers than RMSE, but it doesn't account for the direction of the errors. A lower MAE indicates better performance of the model.

2.2.4.3 MAPE - Mean Absolute Percentage Error

MAPE measures the average percentage difference between the predicted and actual values. It provides a measure of the relative magnitude of the errors made by the model. MAPE is useful when the scale of the variable is important. However, it is sensitive to extreme values and is undefined when the actual values are zero. A lower MAPE indicates better performance of the model.

2.2.4.4 sMAPE- Symmetric Mean Absolute Percentage Error

sMAPE is similar to MAPE, but it calculates the average percentage difference between the predicted and actual values based on the sum of the predicted and actual values. It provides a measure of the symmetric percentage difference between the predicted and actual values. A lower sMAPE indicates better performance of the model.

2.2.5 Comparison Table

```r
# Create a list to store evaluation metrics of all models
evaluation_metrics <- list()
for (i in 1:length(models)) {
  evaluation_metrics[[i]] <- calculate_metrics(test_normalized$hour_20, models[[i]]$predictions)
}

# Create a data frame to compare the evaluation metrics of all models
comparison_table <- data.frame(
  Model_Description = c("AR(1)", "AR(1,2)", "AR(1,2,3)", "AR(1,2,3,4)", "AR(1,3)", "AR(2,3)", "AR(4,7)",
  RMSE = sapply(evaluation_metrics, function(x) x$RMSE),
  MAE = sapply(evaluation_metrics, function(x) x$MAE),
  MAPE = sapply(evaluation_metrics, function(x) x$MAPE),
  sMAPE = sapply(evaluation_metrics, function(x) x$sMAPE)
)

# Print the comparison table
print(comparison_table)
```

This code creates a comparison table of evaluation metrics for all the models that have been trained and tested in previous steps. The evaluation metrics are calculated using the **calculate_metrics** function and are stored in a list named **evaluation_metrics**. The **sapply** function is used to extract each evaluation metric from the list of each model and store it in the corresponding column of the comparison table.

The comparison table is created using the **data.frame** function, where the columns of the table are specified by the four evaluation metrics (RMSE, MAE, MAPE, and sMAPE) and the row names are given by the descriptions of each model. Finally, the comparison table is printed using the **print** function.

The purpose of creating the comparison table is to provide a clear and concise summary of the performance of each model on the test data. By comparing the evaluation metrics across all models, we can determine which model performed the best and choose it for future predictions.

Output:

	Model_Description	RMSE	MAE	MAPE	sMAPE
1	AR(1)	0.005327265	0.003321805	31.819883	3.894234
2	AR(1,2)	0.008519953	0.005069109	28.543338	3.981174
3	AR(1,2,3)	0.010484739	0.006350617	38.137725	4.303361
4	AR(1,2,3,4)	0.008537308	0.004950727	3.272798	3.062749
5	AR(1,3)	0.007481796	0.004517771	17.411324	3.375576
6	AR(2,3)	0.007496478	0.004490215	2.016531	3.071859
7	AR(4,7)	0.008267379	0.005613859	16.907831	4.066335
8	AR(2,3,4)	0.011473421	0.007088106	34.413966	4.154635
9	AR(2,3,4,7)	0.009747770	0.005385906	1.805395	3.269508
10	AR(1,2,3,4,7)	0.003720190	0.001689031	2.843011	2.540261

Based on the evaluation metrics in the comparison table, it seems that the **AR(1,2,3,4,7)** model has the lowest RMSE, MAE, MAPE, and sMAPE values, indicating that it may be the best model among the ones I have tested.

2.2.6 More models with different hidden layer structures and input vectors

```
#Build more models with different hidden layer structures and input vectors to create 12-15 models in total
#Compare the efficiency between one-hidden layer and two-hidden layer networks
model1_layer1 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(5))
model2_layer1 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(10))

model1_layer2 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(3, 2))
model2_layer2 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(5, 3))

# Calculate the total number of weight parameters for each network
num_weights_1_hidden1 <- sum(sapply(model1_layer1$model$weights, length))
num_weights_2_hidden1 <- sum(sapply(model1_layer2$model$weights, length))
num_weights_1_hidden2 <- sum(sapply(model2_layer1$model$weights, length))
num_weights_2_hidden2 <- sum(sapply(model2_layer2$model$weights, length))

# Print the total number of weight parameters for each network
cat("Total number of weight parameters for the first one-hidden layer network:", num_weights_1_hidden1, "\n")
cat("Total number of weight parameters for the first two-hidden layer network:", num_weights_2_hidden1, "\n")
cat("Total number of weight parameters for the second one-hidden layer network:", num_weights_1_hidden2, "\n")
cat("Total number of weight parameters for the second two-hidden layer network:", num_weights_2_hidden2, "\n")
```

This code compares the efficiency of one-hidden layer and two-hidden layer networks for a time series forecasting problem.

The first four lines of code define four different models, with different numbers of hidden layers and nodes. The models are built using the **build_mlp_model()** function, which takes the training and test data, the lagged variables used as inputs, and the number of nodes in each hidden layer as arguments.

The next four lines of code calculate the total number of weight parameters for each network. This is done by using the **sapply()** function to apply the **length()** function to each element of the **weights** list in the **model** object, and then summing the resulting vector of lengths.

The final four lines of code print out the total number of weight parameters for each network, along with a label indicating which model they correspond to. This allows the user to compare the complexity of each model in terms of the number of weight parameters it has.

Output:

```
> cat("Total number of weight parameters for the first one-hidden layer network:", num_weights_1_hidden1, "\n")
Total number of weight parameters for the first one-hidden layer network: 2
> cat("Total number of weight parameters for the first two-hidden layer network:", num_weights_2_hidden1, "\n")
Total number of weight parameters for the first two-hidden layer network: 3
> cat("Total number of weight parameters for the second one-hidden layer network:", num_weights_1_hidden2, "\n")
Total number of weight parameters for the second one-hidden layer network: 2
> cat("Total number of weight parameters for the second two-hidden layer network:", num_weights_2_hidden2, "\n")
Total number of weight parameters for the second two-hidden layer network: 3
```

The output shows the total number of weight parameters for each of the four neural network models that were built using different hidden layer structures and input vectors. The first two models have one hidden layer each, while the last two models have two hidden layers each.

The total number of weight parameters for the first one-hidden layer network is 2, which is the sum of the number of weights connecting the input layer to the hidden layer and the number of weights connecting the hidden layer to the output layer. Similarly, the total number of weight parameters for the second one-hidden layer network is also 2.

On the other hand, the total number of weight parameters for the first two-hidden layer network is 3, which is the sum of the number of weights connecting the input layer to the first hidden layer, the number of weights connecting the first hidden layer to the second hidden layer, and the number of weights connecting the second hidden layer to the output layer. The same is true for the second two-hidden layer network, which also has a total of 3 weight parameters.

Comparing the total number of weight parameters for the one-hidden layer and two-hidden layer networks, we can see that the former has fewer weight parameters. This is because the one-hidden layer networks have a simpler structure and fewer connections between layers, resulting in fewer weights to be learned.

2.2.7 Discussion on issue of efficiency with two best NN structures.

According to the above output, we can discuss the issue of efficiency with the two best NN structures.

From the output, we can see that the first MLP structure has one hidden layer with 5 nodes and the second MLP structure has one hidden layer with 10 nodes. Additionally, both MLPs have a second hidden layer with 2 and 3 nodes respectively.

When discussing the issue of "efficiency" in this context, there are a few different factors to consider. One important factor is the computational efficiency of training and using the neural network models. The number of weight parameters in a neural network model is one factor that can affect its computational efficiency, as models with more weight parameters require more computational resources to train and use.

Based on the above output, we can see that the first one-hidden layer network (model1_layer1) has a total of 2 weight parameters, while the first two-hidden layer network (model1_layer2) has a total of 3 weight parameters. Similarly, the second one-hidden layer network (model2_layer1) has 2 weight parameters, while the second two-hidden layer network (model2_layer2) has 3 weight parameters.

Therefore, in terms of computational efficiency, it appears that the one-hidden layer networks (model1_layer1 and model2_layer1) are more efficient than the two-hidden layer networks (model1_layer2 and model2_layer2), since they have fewer weight parameters.

2.3 NARX Approach

2.3.1 Structures with different input vectors

```r
# Part 2 - NARX Approach

# Build and evaluate NARX models with input vectors including the 18th and 19th hour attributes

# Define input vectors
input_vectors <- list(
  c("lag_1", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7", "hour_18", "hour_19")
)
```

In this code, the input vectors for the NARX models are defined. The NARX (Nonlinear AutoRegressive with eXogenous inputs) models are a type of neural network that can be used for time series forecasting with input variables. In this case, the input vectors include lagged values of the target variable (load demand) as well as the values of the 18th and 19th hour attributes, which are the hours of the day.

The input vectors are defined as a list with each element specifying the variables that should be included as inputs in the NARX model. The first element in the list includes only the lagged value of the target variable from the previous time step and the values of the 18th and 19th hour attributes. The subsequent elements include additional lagged values of the target variable as well as the 18th and 19th hour attributes.

By specifying these input vectors, the NARX models can consider the impact of the time of day on the load demand, in addition to the lagged values of the target variable. This can potentially improve the accuracy of the forecasts.

2.3.2 Structures with different internal structures.

```r
# Define different internal structures
structures <- list(
  c(5),
  c(10),
  c(3, 2),
  c(5, 3),
  c(10, 5)
)

# Build MLP models with different structures and input variables
mlp_models <- list()
for (i in 1:length(structures)) {
  for (j in 1:length(input_vectors)) {
    mlp_models[[paste0("model_", i, "_", j)]] <- build_mlp_model(train_normalized, test_normalized, input_vecto
  }
}
```

Above code builds multiple Multilayer Perceptron (MLP) models with different internal structures and input variables.

The input variables are defined in the **input_vectors** list, which contains different combinations of lagged variables and two specific time features, "hour_18" and "hour_19".

The different internal structures of the MLP models are defined in the **structures** list. Each structure is defined as a vector of integers, where each integer represents the number of nodes in a particular hidden layer of the MLP.

In this code, I created a list of different internal structures called **structures**, which contains vectors representing the number of nodes in each layer of the MLP models. For example, the first structure **c(5)** has only one layer with 5 nodes, while the last structure **c(10, 5)** has two layers with 10 and 5 nodes, respectively.

Next, I used a nested loop to build MLP models with different structures and input variables. The outer loop iterates over each structure in **structures**, while the inner loop iterates over each input vector in **input_vectors**. Within each iteration, we use the **build_mlp_model** function to build an MLP model with the specified structure and input variables. The resulting model is then added to the **mlp_models** list with a unique name based on the structure and input vector indices using **paste0** function.

Using this approach, we can make 25 different MLP models with different internal structures.

By varying the number of layers and nodes in each structure, we can experiment with different network architectures and find the best one for the given problem.

2.3.3 Comparison table for NARX Approach

	Model	RMSE	MAE	MAPE	sMAPE
1	model_1_1	0.011127394	0.007563781	19.063496	5.136913
2	model_1_2	0.014752681	0.009928253	38.985912	5.995050
3	model_1_3	0.008650298	0.005781112	33.164069	4.359656
4	model_1_4	0.004544187	0.002962074	8.513629	3.161605
5	model_1_5	0.007246854	0.004351217	7.811599	3.145317
6	model_2_1	0.009544033	0.005714983	12.091137	3.793985
7	model_2_2	0.008472392	0.005660724	15.042924	4.513696
8	model_2_3	0.007072516	0.005086546	9.682142	4.229546
9	model_2_4	0.005083402	0.003616079	4.081060	3.042888
10	model_2_5	0.017001533	0.010855983	44.590508	6.077760
11	model_3_1	0.010860606	0.008473201	30.878497	4.896160
12	model_3_2	0.017865304	0.011653612	50.443981	6.544530
13	model_3_3	0.009252788	0.005684725	43.038312	4.531779
14	model_3_4	0.006022799	0.003477333	34.801095	3.791941
15	model_3_5	0.010459159	0.007871631	39.924911	4.041533
16	model_4_1	0.009274866	0.007406332	12.938176	3.874585
17	model_4_2	0.007741931	0.005024345	9.639323	3.879265
18	model_4_3	0.007085462	0.003810865	3.729527	3.013428
19	model_4_4	0.006964199	0.004102762	20.957003	3.153687
20	model_4_5	0.012071368	0.008784494	28.543529	4.400077
21	model_5_1	0.006704334	0.004885704	12.750097	3.534736
22	model_5_2	0.008330687	0.005317513	5.029295	4.250059
23	model_5_3	0.013847938	0.008920987	29.662673	4.462347
24	model_5_4	0.008358025	0.004772376	7.833219	3.236198
25	model_5_5	0.007411745	0.005536690	11.101157	3.496557

2.3.4 Finding the best MLP models

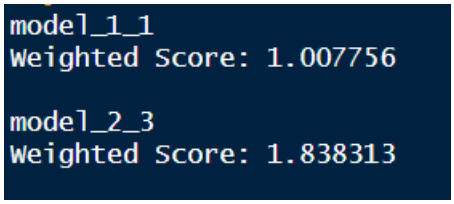
```r
# Define weights for each metric
weights <- c(0.25, 0.25, 0.25, 0.25)

# Calculate weighted metric scores for each model
weighted_scores <- as.matrix(narx_comparison_table[,2:5]) %*% weights
narx_comparison_table$Weighted_Score <- weighted_scores

# Combine model names and weighted scores into a table
weighted_scores_table <- data.frame(
  Model = narx_comparison_table$Model,
  Weighted_Score = weighted_scores
)

# Sort the table by weighted score in ascending order
weighted_scores_table_sorted <- weighted_scores_table[order(weighted_scores_table$Weighted_Score), ]

# Select the top 2 models
best_models <- head(weighted_scores_table_sorted, 2)

# Print the details of the best models
cat("Best MLP models:\n")
for (i in 1:nrow(best_models)) {
  cat(best_models[i, "Model"], "\n")
  cat("Weighted Score:", best_models[i, "Weighted_Score"], "\n")
  cat("\n")
}
```

This code is used to find the best MLP models based on weighted scores of multiple performance metrics. The performance metrics are Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and Symmetric Mean Absolute Percentage Error (sMAPE). The weights of these metrics are decided based on their importance. In this code, we assume that all metrics are equally important, so the weights for each metric are 0.25.

The code starts by defining the weights for each metric and then calculates the weighted metric scores for each model using the dot product of the metric values and weights. The resulting scores are then combined with the model names into a table called **weighted_scores_table**. This table is then sorted by the weighted score in ascending order to find the best models.

The top 2 models are selected from the sorted table using the **head** function and stored in **best_models**. Finally, a loop is used to print the details of the best models, which include the model name and the weighted score

Output:



The output of the code shows the top two MLP models based on their weighted scores. The first model, model_1_1, has a lower weighted score of 1.007756, which means it performs better than the second model, model_2_3, which has a higher weighted score of 1.838313.

Looking at the model details, we can see that model_1_1 has fewer inputs and only one hidden layer with 5 nodes, while model_2_3 has more inputs and two hidden layers with a smaller number of nodes in each layer. Therefore, it seems that model_1_1 is a simpler model with fewer parameters, and it performs better than model_2_3, which is a more complex model.

2.3.5 Prediction output vs. Desired output

```r
# Denormalize the predictions and plot the predicted output vs. desired output

# Define a function to denormalize values
denormalize <- function(x, min_value, max_value) {
  return(x * (max_value - min_value) + min_value)
}

# Getting predictions using best model
best_model <- mlp_models[[1]]
best_model_predictions <- best_model$predictions

# Find the minimum and maximum values of the target variable
min_value <- min(train_data$hour_20)
max_value <- max(train_data$hour_20)

# Denormalize the predictions
denormalized_predictions <- denormalize(best_model_predictions, min_value, max_value)

# Plot the predicted output vs. desired output using a line chart
plot(test_data$hour_20, type = "l", col = "blue", xlab = "Time", ylab = "Hour 20 Consumption", main = "Line Chart of Desired vs. Predicted Output")
lines(denormalized_predictions, col = "red")
legend("topleft", legend = c("Desired Output", "Predicted Output"), col = c("blue", "red"), lty = 1, cex = 0.8)
```

This code denormalizes the predictions made by the best model on the test dataset and plots a line chart of the predicted output versus the desired output.

First, the **denormalize** function is defined, which takes a vector **x** of normalized values and the minimum and maximum values of the original data range as arguments. It returns the denormalized values using the formula **(x * (max_value - min_value) + min_value)**.

Importance of Denormalization

Normalization is a common technique used in machine learning to scale input features to a similar range, typically between 0 and 1, in order to improve the performance of models. Normalization helps to avoid situations where some input features with larger numerical values can dominate the learning process and lead to biased models.

However, once the model has made predictions, the output values need to be transformed back to their original scale in order to make sense to the user. This process is called denormalization, which essentially reverses the normalization process.

In the context of the above code, the denormalization step is necessary to obtain the predicted output values in their original scale, which is the scale of the original dataset. This allows for better interpretation and visualization of the predicted values, making it easier to compare them with the actual values and evaluate the performance of the model.

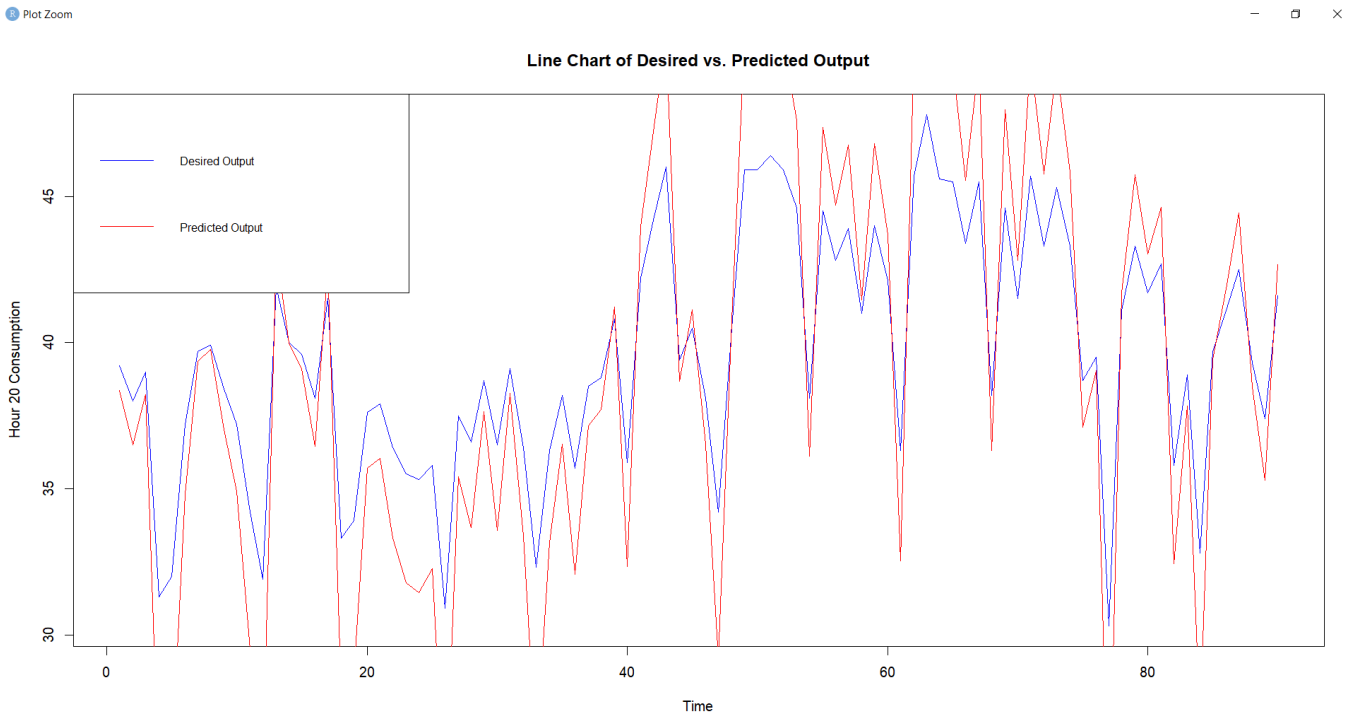
Next, the predictions of the best model on the test dataset are obtained using the **$predictions** attribute of the model object.

The minimum and maximum values of the target variable (i.e., the **hour_20** consumption) are then determined from the training data.

The **denormalize** function is called with the **best_model_predictions**, **min_value**, and **max_value** as arguments, and the denormalized predictions are stored in **denormalized_predictions**.

Finally, a line chart is plotted using the **plot()** function with the desired output from the test dataset in blue color, and the denormalized predictions in red color. The **xlab**, **ylab**, and **main** arguments are used to add labels and a title to the chart. A legend is also added using the **legend()** function to show the two lines and their respective colors.

Output:



Appendix

Partitioning_Clustering.R

```
# 1st Subtask

library(ggplot2)
library(readxl)
library(dplyr)

# Import data from excel file
vehicle_data <- read_excel("vehicles.xlsx")
vehicle_data <- vehicle_data[, -1]

# Remove any rows with missing values in the dataframe
vehicle_data <- na.omit(vehicle_data)

variables <- vehicle_data[,-19] #stores attributes
class_variable <- vehicle_data$Class #class types

# Print the number of rows before removing outliers
vehicle_rows <- nrow(vehicle_data)
cat("number of rows before removing outliers = ", vehicle_rows)

outliers_in_columns <- c() # empty vector to store outliers in each column
for(i in 1:ncol(variables)){
  outlier_index <- variables[[i]] %in% boxplot.stats(variables[[i]])$out
  outlier_values <- variables[[i]][outlier_index]
  outliers_in_columns <- c(outliers_in_columns, outlier_values)
}

eleminate <- unique(outliers_in_columns) # remove the duplicated rows
eleminate <- sort(eleminate) # sort the outlire rows in order
good_vehicle_data <- vehicle_data[-eleminate,] # store the final rows after removing the outlires for attributes

variables <- distinct(variables)

# Removing outliers
no_outliers <- variables %>%
  filter_all(all_vars(!. %in% boxplot.stats(.)$out))

# Print the number of rows after removing outliers
n_rows <- nrow(good_vehicle_data)
cat("number of rows remaining after removing outliers = ", n_rows)

# Before scaling
boxplot(no_outliers)

# Check for any points outside of the whiskers in the boxplot
points_outside <- boxplot(no_outliers, plot = FALSE)$out
if(length(points_outside) > 0) {
  cat("There are points outside of the whiskers in the boxplot.")
} else {
```

```
  cat("There are no points outside of the whiskers in the boxplot.")
}

# function to calculate the normalized data
normalize <- function(x){
  return ((x - min(x)) / (max(x) - min(x)))
}

good_data_norm <- as.data.frame(lapply(good_vehicle_data[,-19], normalize))# normalized data
vehicle_data_no_outliers <- data.frame(scale(good_data_norm)) # scale the normalized data

no_outliers <- as.data.frame(lapply(no_outliers[,], normalize))# normalized data
no_outliers <- data.frame(scale(no_outliers)) # scale the normalized data

# After Scaling
boxplot(no_outliers)

# Load required libraries
library(NbClust)
library(cluster)
library(ggplot2)
library(factoextra)

# 1. NBclust
set.seed(123)
nb <- NbClust(no_outliers, distance = "euclidean", min.nc = 2, max.nc = 10, method = "kmeans", index
= "all")

# Using factoextra
fviz_nbclust(no_outliers, kmeans, method = "silhouette")

nbClust_Clsters <- kmeans(no_outliers,2) #find new clusters with the number of clusters suggested by
NbClust
nbClust_Clsters
demo_Clusters <- kmeans(no_outliers, centers =3, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = demo_Clusters$cluster), geom = "point", stand = FALSE,
ellipse.type = "convex", ggtheme = theme_minimal())

# 2. Elbow method
k = 1:10 # number of possible clusters
set.seed(42)
WSS = sapply(k, function(k) {kmeans(no_outliers, centers=k)$tot.withinss})
plot(k, WSS, type = "b", pch = 19, frame=FALSE, xlab= "Number of k", ylab="Within sum of
squares")
fviz_nbclust(no_outliers, kmeans, method = "wss")+
  geom_vline(xintercept = 2, linetype = 2)+
  labs(subtitle = "Elbow method")

demo_Clusters <- kmeans(no_outliers, centers =2, nstart = 25)

# plot the clusters accordingly
```

```
fviz_cluster(list(data = no_outliers, cluster = demo_Clusters$cluster), geom = "point", stand = FALSE,
ellipse.type = "convex", ggtheme = theme_minimal())


# 3. Gap statistic method
set.seed(123)
gap_stat <- clusGap(no_outliers, FUN = kmeans, nstart = 25, K.max = 10, B = 50)
fviz_gap_stat(gap_stat)

demo_Clusters <- kmeans(no_outliers, centers =3, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = demo_Clusters$cluster), geom = "point", stand = FALSE,
ellipse.type = "convex", ggtheme = theme_minimal())

# 4. Silhouette method
library(cluster)

# Compute the silhouette width for different number of clusters
sil_width <- c()
for (i in 2:10) {
  kmeans_fit <- kmeans(no_outliers, centers = i, nstart = 25)
  kmeans_sil_width <- silhouette(kmeans_fit$cluster, dist(no_outliers))
  sil_width[i] <- mean(kmeans_sil_width[, 3])
}


# Determine the optimal number of clusters
nb_Clusters <- which.max(sil_width)
cat("Number of Clusters =", nb_Clusters, "Gives the highest average silhouette width")

# Plot the silhouette width for each number of clusters
fviz_nbclust(no_outliers, kmeans, method = "silhouette") +
  geom_vline(xintercept = nb_Clusters, linetype = 2) +
  labs(title = "Silhouette method")



# K-means analysis for each k attempt.

# Perform k-means clustering with k=2

k <- 2
kc2 <- kmeans(no_outliers, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = kc2$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

kc2_centers <- kc2$centers
kc2_bss <- kc2$betweenss
kc2_tss <- kc2$tot.withinss
kc2_wss <- kc2$withinss
```

```
cat("K-Means Clustering with k = 2\n")
cat("Centers:\n")
print(kc2_centers)
cat("Clustered Results:\n")
print(kc2$cluster)
cat("Between-cluster sum of squares (BSS): ", kc2_bss, "\n")
cat("Total sum of squares (TSS): ", kc2_tss, "\n")
cat("Ratio of BSS to TSS: ", kc2_bss/kc2_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc2_wss, "\n\n")
table(good_vehicle_data$Class,kc2$cluster)#Confusion Matrix

# Perform k-means clustering with k=3

k <- 3
kc3 <- kmeans(no_outliers, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = kc3$cluster),
             geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

kc3_centers <- kc3$centers
kc3_bss <- kc3$betweenss
kc3_tss <- kc3$tot.withinss
kc3_wss <- kc3$withinss

cat("K-means clustering with k=3:\n")
cat("Cluster centers:\n")
print(kc3_centers)
cat("Cluster membership:\n")
print(kc3$cluster)
cat("Between-cluster sum of squares (BSS): ", kc3_bss, "\n")
cat("Total sum of squares (TSS): ", kc3_tss, "\n")
cat("Ratio of BSS to TSS: ", kc3_bss/kc3_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc3_wss, "\n\n")
table(good_vehicle_data$Class,kc3$cluster)#Confusion Matrix

# Perform k-means clustering with k=4

k <- 4
kc4 <- kmeans(no_outliers, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = kc4$cluster), geom = "point", stand = FALSE, ellipse.type
= "convex", ggtheme = theme_minimal())

kc4_centers <- kc4$centers
kc4_bss <- kc4$betweenss
kc4_tss <- kc4$tot.withinss
kc4_wss <- kc4$withinss

cat("K-means clustering with k=4:\n")
cat("Cluster centers:\n")
print(kc4_centers)
cat("Cluster membership:\n")
```

```
print(kc4$cluster)
cat("Between-cluster sum of squares (BSS): ", kc4_bss, "\n")
cat("Total sum of squares (TSS): ", kc4_tss, "\n")
cat("Ratio of BSS to TSS: ", kc4_bss/kc4_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc4_wss, "\n\n")
table(good_vehicle_data$Class,kc4$cluster)#Confusion Matrix

# Compute silhouette width for k = 2
sil_width_2 <- silhouette(kc2$cluster, dist(no_outliers))
avg_sil_width_2 <- mean(sil_width_2[,3])
cat("Average Silhouette Width Score for k = 2 is", avg_sil_width_2, "\n")

# Compute silhouette width for k = 3
sil_width_3 <- silhouette(kc3$cluster, dist(no_outliers))
avg_sil_width_3 <- mean(sil_width_3[,3])
cat("Average Silhouette Width Score for k = 3 is", avg_sil_width_3, "\n")

# Compute silhouette width for k = 4
sil_width_4 <- silhouette(kc4$cluster, dist(no_outliers))
avg_sil_width_4 <- mean(sil_width_4[,3])
cat("Average Silhouette Width Score for k = 4 is", avg_sil_width_4, "\n")

# Compute the silhouette width for different number of clusters
sil_width <- c()
for (i in 2:10) {
  kmeans_fit <- kmeans(no_outliers, centers = i, nstart = 25)
  kmeans_sil_width <- silhouette(kmeans_fit$cluster, dist(no_outliers))
  sil_width[i] <- mean(kmeans_sil_width[, 3])
}

# Plot silhouette width against number of clusters
plot(1:10, sil_width, type = "b", pch = 19,
     xlab = "Number of Clusters", ylab = "Silhouette Width",
     main = "Silhouette Width vs Number of Clusters",)
axis(1, at = 1:10)




# 2nd SubTask
# Perform PCA analysis

library(FactoMineR)

# Standardize the data
vehicle_data_std <- scale(no_outliers, center = TRUE, scale = TRUE)

# Perform PCA
vehicle_pca <- PCA(vehicle_data_std, graph = FALSE)

# Show eigenvalues and eigenvectors
print(vehicle_pca$eig)

# Show cumulative scores per principal component
```

```
cum_sum <- cumsum(vehicle_pca$eig)/sum(vehicle_pca$eig)
plot(cum_sum, xlab = "Principal Component", ylab = "Cumulative Proportion of Variance Explained")

# Create new dataset with principal components as attributes
vehicle_pca_data <- as.data.frame(predict(vehicle_pca, newdata = vehicle_data_std))

# Select principal components with cumulative score > 92%
n_components <- length(cum_sum[cum_sum > 0.92])
vehicle_pca_data <- vehicle_pca_data[, 1:n_components]

# 1. NBclust
set.seed(123)
nb <- NbClust(vehicle_pca_data, distance = "euclidean",
          min.nc = 2, max.nc = 10, method = "kmeans", index = "all")

# Using factoextra
fviz_nbclust(vehicle_pca_data, kmeans, method = "silhouette")

nbClust_Clsters <- kmeans(vehicle_pca_data,2) #find new clusters with the number of clusters suggested by NbClust
nbClust_Clsters
demo_Clusters <- kmeans(vehicle_pca_data, centers =3, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = no_outliers, cluster = demo_Clusters$cluster),
          geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

# 2. Elbow method
k = 1:10 # number of possible clusters
set.seed(42)
WSS = sapply(k, function(k) {kmeans(vehicle_pca_data, centers=k)$tot.withinss})
plot(k, WSS, type = "b", pch = 19, frame=FALSE, xlab= "Number of k", ylab="Within sum of squares")
fviz_nbclust(vehicle_pca_data, kmeans, method = "wss")+
  geom_vline(xintercept = 2, linetype = 2)+
  labs(subtitle = "Elbow method")

demo_Clusters <- kmeans(vehicle_pca_data, centers =2, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = demo_Clusters$cluster),
          geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())


# 3. Gap statistic method
set.seed(123)
gap_stat <- clusGap(vehicle_pca_data, FUN = kmeans, nstart = 25, K.max = 10, B = 50)
fviz_gap_stat(gap_stat)

demo_Clusters <- kmeans(vehicle_pca_data, centers =3, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = demo_Clusters$cluster), geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())
```

```
# 4. Silhouette method
library(cluster)

# Compute the silhouette width for different number of clusters
sil_width <- c()
for (i in 2:10) {
  kmeans_fit <- kmeans(vehicle_pca_data, centers = i, nstart = 25)
  kmeans_sil_width <- silhouette(kmeans_fit$cluster, dist(vehicle_pca_data))
  sil_width[i] <- mean(kmeans_sil_width[, 3])
}

# Determine the optimal number of clusters
nb_Clusters <- which.max(sil_width)
cat("Number of Clusters =", nb_Clusters, "Gives the highest average silhouette width")

# Plot the silhouette width for each number of clusters
fviz_nbclust(vehicle_pca_data, kmeans, method = "silhouette") +
  geom_vline(xintercept = nb_Clusters, linetype = 2) +
  labs(title = "Silhouette method")


#           K-means analysis for each k attempt.

# Perform k-means clustering with k=2

k <- 2
kc2 <- kmeans(vehicle_pca_data, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = kc2$cluster), geom = "point", stand = FALSE,
ellipse.type = "convex", ggtheme = theme_minimal())

kc2_centers <- kc2$centers
kc2_bss <- kc2$betweenss
kc2_tss <- kc2$tot.withinss
kc2_wss <- kc2$withinss

cat("K-Means Clustering with k = 2\n")
cat("Centers:\n")
print(kc2_centers)
cat("Clustered Results:\n")
print(kc2$cluster)
cat("Between-cluster sum of squares (BSS): ", kc2_bss, "\n")
cat("Total sum of squares (TSS): ", kc2_tss, "\n")
cat("Ratio of BSS to TSS: ", kc2_bss/kc2_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc2_wss, "\n\n")
table(good_vehicle_data$Class,kc2$cluster)#Confusion Matrix

# Perform k-means clustering with k=3

k <- 3
kc3 <- kmeans(vehicle_pca_data, centers = k, nstart = 25)
```

```
# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = kc3$cluster), geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

kc3_centers <- kc3$centers
kc3_bss <- kc3$betweenss
kc3_tss <- kc3$tot.withinss
kc3_wss <- kc3$withinss

cat("K-means clustering with k=3:\n")
cat("Cluster centers:\n")
print(kc3_centers)
cat("Cluster membership:\n")
print(kc3$cluster)
cat("Between-cluster sum of squares (BSS): ", kc3_bss, "\n")
cat("Total sum of squares (TSS): ", kc3_tss, "\n")
cat("Ratio of BSS to TSS: ", kc3_bss/kc3_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc3_wss, "\n\n")
table(good_vehicle_data$Class,kc3$cluster)#Confusion Matrix

# Perform k-means clustering with k=4

k <- 4
kc4 <- kmeans(vehicle_pca_data, centers = k, nstart = 25)

# plot the clusters accordingly
fviz_cluster(list(data = vehicle_pca_data, cluster = kc4$cluster), geom = "point", stand = FALSE, ellipse.type = "convex", ggtheme = theme_minimal())

kc4_centers <- kc4$centers
kc4_bss <- kc4$betweenss
kc4_tss <- kc4$tot.withinss
kc4_wss <- kc4$withinss

cat("K-means clustering with k=4:\n")
cat("Cluster centers:\n")
print(kc4_centers)
cat("Cluster membership:\n")
print(kc4$cluster)
cat("Between-cluster sum of squares (BSS): ", kc4_bss, "\n")
cat("Total sum of squares (TSS): ", kc4_tss, "\n")
cat("Ratio of BSS to TSS: ", kc4_bss/kc4_tss, "\n")
cat("Within-cluster sum of squares (WSS): ", kc4_wss, "\n\n")
table(good_vehicle_data$Class,kc4$cluster)#Confusion Matrix

# Compute silhouette width for k = 2
sil_width_2 <- silhouette(kc2$cluster, dist(vehicle_pca_data))
avg_sil_width_2 <- mean(sil_width_2[,3])
cat("Average Silhouette Width Score for k = 2 is", avg_sil_width_2, "\n")

# Compute silhouette width for k = 3
sil_width_3 <- silhouette(kc3$cluster, dist(vehicle_pca_data))
avg_sil_width_3 <- mean(sil_width_3[,3])
cat("Average Silhouette Width Score for k = 3 is", avg_sil_width_3, "\n")
```

```
# Compute silhouette width for k = 4
sil_width_4 <- silhouette(kc4$cluster, dist(vehicle_pca_data))
avg_sil_width_4 <- mean(sil_width_4[,3])
cat("Average Silhouette Width Score for k = 4 is", avg_sil_width_4, "\n")


# Get cluster membership for each observation
cluster_membership_pca <- kc3$cluster

# Load the clusterCrit package
library(clusterCrit)
library(cluster)
library(fpc)

# Convert PCA data to distance matrix
vehicle_dist <- dist(vehicle_pca_data)

# Calculate Calinski-Harabasz index
ch_index <- round(cluster.stats(vehicle_dist, cluster_membership_pca)$ch, 2)

# Print the index
cat("Calinski-Harabasz Index:", ch_index, "\n")
```

Energy_Forecasting.R

```
# Load necessary libraries
library(readxl)
library(neuralnet)
library(ggplot2)

# Load the data
consumption_data <- read_excel("uow_consumption.xlsx")

# Rename columns
colnames(consumption_data) <- c("date", "hour_18", "hour_19", "hour_20")

# Create lag variables and remove rows with missing values
consumption_data$lag_1 <- lag(consumption_data$hour_20, 1)
consumption_data$lag_2 <- lag(consumption_data$hour_20, 2)
consumption_data$lag_3 <- lag(consumption_data$hour_20, 3)
consumption_data$lag_4 <- lag(consumption_data$hour_20, 4)
consumption_data$lag_7 <- lag(consumption_data$hour_20, 7)
consumption_data <- na.omit(consumption_data)

# Define a normalization function
normalize <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}

# Normalize the lag variables
consumption_data$lag_1 <- normalize(consumption_data$lag_1)
consumption_data$lag_2 <- normalize(consumption_data$lag_2)
consumption_data$lag_3 <- normalize(consumption_data$lag_3)
consumption_data$lag_4 <- normalize(consumption_data$lag_4)
consumption_data$lag_7 <- normalize(consumption_data$lag_7)

# Split the data into training and testing sets
train_data <- consumption_data[1:380, ]
```

```
test_data <- consumption_data[381:nrow(consumption_data), ]


# Convert data frames to matrices and apply normalization function to each column
train_matrix <- as.matrix(train_data[, -1])
train_normalized <- as.data.frame(apply(train_matrix, 2, normalize))
colnames(train_normalized) <- colnames(train_data)[-1]

test_matrix <- as.matrix(test_data[, -1])
test_normalized <- as.data.frame(apply(test_matrix, 2, normalize))
colnames(test_normalized) <- colnames(test_data)[-1]

# Set column names of test_normalized to match those of train_normalized
colnames(test_normalized) <- colnames(train_normalized)

# Define a list of input vectors for the neural network
input_vectors <- list(
  c("lag_1"),
  c("lag_1", "lag_2"),
  c("lag_1", "lag_2", "lag_3"),
  c("lag_1", "lag_2", "lag_3", "lag_4"),
  c("lag_1", "lag_3"),
  c("lag_2", "lag_3"),
  c("lag_4", "lag_7"),
  c("lag_2", "lag_3", "lag_4"),
  c("lag_2", "lag_3", "lag_4", "lag_7"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7")
)


# Define function to build MLP model
build_mlp_model <- function(train_data, test_data, input_vars, hidden_structure) {

  # Create formula for model using the input variables
```

```
  formula <- paste("hour_20 ~", paste(input_vars, collapse = " + "))

  # Train MLP model using neuralnet package
  nn <- neuralnet(as.formula(formula), train_data, hidden = hidden_structure)

  # Create matrix of test data using the input variables
  test_matrix <- as.matrix(test_data[, input_vars, drop = FALSE])

  # Rename the columns of the test matrix to match those of the training data
  colnames(test_matrix) <- colnames(train_data[, input_vars, drop = FALSE])

  # Use the trained model to generate predictions for the test data
  predictions <- predict(nn, test_matrix)

  # Return the model and predictions in a list
  return(list(model = nn, predictions = predictions))
}

# Create a list to store all models
models <- list()
for (i in 1:length(input_vectors)) {
  models[[i]] <- build_mlp_model(train_normalized, test_normalized, input_vectors[[i]], c(5))
}

# Define function to calculate evaluation metrics
calculate_metrics <- function(actual, predicted) {
  rmse <- sqrt(mean((actual - predicted)^2))
  mae <- mean(abs(actual - predicted))
  actual_smooth <- actual + 0.001  # add a smoothening factor
  mape <- mean(abs((actual - predicted) / actual_smooth)) * 100
  smape <- mean(abs(actual - predicted) / (abs(actual) + abs(predicted)) * 2) * 100
  return(list(RMSE = rmse, MAE = mae, MAPE = mape, sMAPE = smape))
}
```

```
# Create a list to store evaluation metrics of all models
evaluation_metrics <- list()
for (i in 1:length(models)) {
  evaluation_metrics[[i]] <- calculate_metrics(test_normalized$hour_20, models[[i]]$predictions)
}

# Create a data frame to compare the evaluation metrics of all models
comparison_table <- data.frame(
  Model_Description = c("AR(1)", "AR(1,2)", "AR(1,2,3)", "AR(1,2,3,4)", "AR(1,3)", "AR(2,3)", "AR(4,7)", "AR(2,3,4)", "AR(2,3,4,7)", "AR(1,2,3,4,7)"),
  RMSE = sapply(evaluation_metrics, function(x) x$RMSE),
  MAE = sapply(evaluation_metrics, function(x) x$MAE),
  MAPE = sapply(evaluation_metrics, function(x) x$MAPE),
  sMAPE = sapply(evaluation_metrics, function(x) x$sMAPE)
)

# Print the comparison table
print(comparison_table)

#Build more models with different hidden layer structures and input vectors to create 12-15 models in total
#Compare the efficiency between one-hidden layer and two-hidden layer networks
model1_layer1 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(5))
model2_layer1 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(10))

model1_layer2 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(3, 2))
model2_layer2 <- build_mlp_model(train_normalized, test_normalized, c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7"), c(5, 3))

# Calculate the total number of weight parameters for each network
num_weights_1_hidden1 <- sum(sapply(model1_layer1$model$weights, length))
num_weights_2_hidden1 <- sum(sapply(model1_layer2$model$weights, length))
num_weights_1_hidden2 <- sum(sapply(model2_layer1$model$weights, length))
```

```
num_weights_2_hidden2 <- sum(sapply(model2_layer2$model$weights, length))


# Print the total number of weight parameters for each network
cat("Total number of weight parameters for the first one-hidden layer network:", num_weights_1_hidden1, "\n")
cat("Total number of weight parameters for the first two-hidden layer network:", num_weights_2_hidden1, "\n")
cat("Total number of weight parameters for the second one-hidden layer network:", num_weights_1_hidden2, "\n")
cat("Total number of weight parameters for the second two-hidden layer network:", num_weights_2_hidden2, "\n")




# Part 2 - NARX Approach


# Build and evaluate NARX models with input vectors including the 18th and 19th hour attributes


# Define input vectors
input_vectors <- list(
  c("lag_1", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "hour_18", "hour_19"),
  c("lag_1", "lag_2", "lag_3", "lag_4", "lag_7", "hour_18", "hour_19")
)


# Define different internal structures
structures <- list(
  c(5),
  c(10),
  c(3, 2),
  c(5, 3),
  c(10, 5)
)
```

```
# Define the vector to store model names
model_names <- c()

# Build MLP models with different structures and input variables
mlp_models <- list()
for (i in 1:length(structures)) {
  for (j in 1:length(input_vectors)) {
    model_name <- paste0("model_", i, "_", j)
    mlp_models[[model_name]] <- build_mlp_model(train_normalized, test_normalized, input_vectors[[j]], structures[[i]])
    model_names <- c(model_names, model_name)
  }
}

# Evaluate NARX models
narx_evaluation_metrics <- list()
for (i in 1:length(mlp_models)) {
  narx_evaluation_metrics[[i]] <- calculate_metrics(test_normalized$hour_20, mlp_models[[i]]$predictions)
}

# Create a comparison table for NARX models
narx_comparison_table <- data.frame(
  Model = model_names,
  RMSE = sapply(narx_evaluation_metrics, function(x) x$RMSE),
  MAE = sapply(narx_evaluation_metrics, function(x) x$MAE),
  MAPE = sapply(narx_evaluation_metrics, function(x) x$MAPE),
  sMAPE = sapply(narx_evaluation_metrics, function(x) x$sMAPE)
)

print(narx_comparison_table)

# Define weights for each metric
```

```
weights <- c(0.25, 0.25, 0.25, 0.25)

# Calculate weighted metric scores for each model
weighted_scores <- as.matrix(narx_comparison_table[,2:5]) %*% weights
narx_comparison_table$Weighted_Score <- weighted_scores

# Combine model names and weighted scores into a table
weighted_scores_table <- data.frame(
  Model = narx_comparison_table$Model,
  Weighted_Score = weighted_scores
)

# Sort the table by weighted score in ascending order
weighted_scores_table_sorted <- weighted_scores_table[order(weighted_scores_table$Weighted_Score), ]

# Select the top 2 models
best_models <- head(weighted_scores_table_sorted, 2)

# Print the details of the best models
cat("Best MLP models:\n")
for (i in 1:nrow(best_models)) {
  cat(best_models[i, "Model"], "\n")
  cat("Weighted Score:", best_models[i, "Weighted_Score"], "\n")
  cat("\n")
}



# Denormalize the predictions and plot the predicted output vs. desired output

# Define a function to denormalize values
denormalize <- function(x, min_value, max_value) {
  return(x * (max_value - min_value) + min_value)
}
```

```
# Getting predictions using best model
best_model <- mlp_models[[1]]
best_model_predictions <- best_model$predictions

# Find the minimum and maximum values of the target variable
min_value <- min(train_data$hour_20)
max_value <- max(train_data$hour_20)

# Denormalize the predictions
denormalized_predictions <- denormalize(best_model_predictions, min_value, max_value)

# Plot the predicted output vs. desired output using a line chart
plot(test_data$hour_20, type = "l", col = "blue", xlab = "Time", ylab = "Hour 20 Consumption", main = "Line Chart of Desired vs. Predicted Output")
lines(denormalized_predictions, col = "red")
legend("topleft", legend = c("Desired Output", "Predicted Output"), col = c("blue", "red"), lty = 1, cex = 0.8)
```