# MACHINE LEARNING & DATA MINING

# SAMPLE PROJECT REPORT

*DINITH PERERA*

# TABLE OF CONTENT

# TABLE OF FIGURES

# Partitioning Clustering Part

## 1st Subtask Objectives:

### A.

First of all the necessary packages are installed and loaded. The packages **readxl** and **dplyr** are now installed and loaded. **Readxl** is used to read data stored in Excel files. Statistical functions and distributions are offered by the **stats** package. Many of the essential R functions and data structures are provided by the **base** package. **dplyr** is used for data manipulation.

```
install.packages(c("readxl", "stats", "base", "dplyr"))
library(readxl)
library(stats)
library(base)
library(dplyr)
```

Then a data frame named **vehicles_df** is created to store the Excel data.

```
vehicles_df <- read_excel("vehicles.xlsx")
View(head(vehicles_df))
```

Next, the **Samples** column is removed because it is an index column.

| | Samples | Comp | Circ | D.Circ | Rad.Ra |
|---|---|---|---|---|---|
| 1 | 1 | 95 | 48 | 83 | 178 |
| 2 | 2 | 91 | 41 | 84 | 141 |
| 3 | 3 | 104 | 50 | 106 | 209 |
| 4 | 4 | 93 | 41 | 82 | 159 |
| 5 | 5 | 85 | 44 | 70 | 205 |
| 6 | 6 | 107 | 57 | 106 | 172 |

*Figure 1: Before the "Samples" column removed*

| | Comp | Circ | D.Circ | Rad.Ra | Pr.Axis.Ra |
|---|---|---|---|---|---|
| 1 | 95 | 48 | 83 | 178 | 72 |
| 2 | 91 | 41 | 84 | 141 | 57 |
| 3 | 104 | 50 | 106 | 209 | 66 |
| 4 | 93 | 41 | 82 | 159 | 63 |
| 5 | 85 | 44 | 70 | 205 | 103 |
| 6 | 107 | 57 | 106 | 172 | 50 |

*Figure 2: After the "Samples" column removed*

```
vehicles_df <- vehicles_df[, -which(names(vehicles_df) == "Samples")]
View(vehicles_df)
```

Now, the columns with numeric data are chosen for the preprocessing tasks.

```
numeric_cols <- sapply(vehicles_df, is.numeric)
```

```
vehicles_df_numeric <- vehicles_df[, numeric_cols]
```

Then the outliers are removed by using the **"z-score"** method.

```
z_scores <- as.matrix(scale(vehicles_df[, numeric_cols]))
outliers <- apply(z_scores, 1, function(x) any(abs(x) > 3))
clean_df <- vehicles_df[!outliers,]
View(clean_df)
```

Here the data frame's numerical variables are all assigned standardized z-scores in the first line of code. By removing the mean and dividing by the standard deviation, the **scale()** function standardizes the variables. A matrix of z-scores is produced as a consequence, one for each numerical variable in the data frame.

The z-score matrix z_scores is given the **apply()** function in the second line of code. Each row or column of a matrix can have a function applied to it using the **apply()** method. In this instance, each row of the z-score matrix is subjected to **any()** function, which determines whether any z-scores in that row are larger than 3 or less than -3. The any() method returns TRUE, indicating that an outlier is present in the row, if any z-scores are larger than 3 or less than -3. The function returns FALSE in all other cases.

The resulting logical vector, the outliers variable, shows which rows in the **vehicles_df** data frame have outliers.

**Clean_df** is a new data frame that excludes the rows with outliers in the last line of code. By negating the outliers vector, the **"!"** operator only chooses the rows that are free of outliers. The **View()** function is then used to view this cleaned data frame.

Once the outliers are removed, the **scale** function is used to standardize the dataset.

```
num_cols <- sapply(clean_df, is.numeric)
clean_df[, num_cols] <- scale(clean_df[, num_cols])
View (head(clean_df[, num_cols]))
```

To determine which columns in clean_df contain numerical data, the **sapply()** function is first used to generate a logical vector called num_cols.

The selected numeric columns in **clean_df** are then standardized using the **scale()** function by deducting the mean and dividing by the standard deviation. The original data frame is then allocated the resulting matrix of standardized z-scores, replacing the original numeric columns with the standardized versions.

Finally, just the standardized numeric columns chosen using **clean_df[, num_cols]** are displayed in the first few rows of the standardized data frame using the **View()** function in a new window.

| | Comp | Circ | D.Circ | Rad.Ra | Pr.Axis.Ra | Max.L.Ra |
|---|---|---|---|---|---|---|
| 1 | 0.17940319 | 0.5268943 | 0.068720924 | 0.3226507 | 1.9112974 | 0.8276342 |
| 2 | -0.31500560 | -0.6169245 | 0.132345689 | -0.8360035 | -0.7475249 | 0.3771913 |
| 3 | 1.29182298 | 0.8536997 | 1.532090527 | 1.2934151 | 0.8477685 | 0.8276342 |
| 4 | -0.06780121 | -0.6169245 | 0.005096158 | -0.2723339 | 0.3160040 | 0.3771913 |
| 5 | 1.66262957 | 1.9975185 | 1.532090527 | 0.1347608 | -1.9883086 | -0.9741375 |
| 6 | 0.42660759 | -0.2901191 | -0.567526730 | 0.1660758 | 0.6705137 | -0.9741375 |

*Figure 3: clean_df*

B.

Before determining the number of clusters, the column **Class** should be removed because this is unsupervised learning.

```
clean_df <- clean_df[, -which(names(clean_df) == "Class")]
View(head(clean_df))

First, let's try the NbClust method.
The relevant package is installed and loaded.

install.packages("NbClust")
library(NbClust)
```

R has a function called **set.seed(123)** that can be used to seed the random number generator. This makes sure that every time the random number generator is used in the R code following this command, the same set of random numbers will be created. This is helpful for ensuring results can be replicated in experiments or simulations that use randomization. As long as the seed value is consistent across numerous runs, the choice of seed value is irrelevant.

```
set.seed(123)

nb_clusters <- NbClust(clean_df, distance = "euclidean", min.nc=2,
max.nc=10, method="kmeans")
table(nb_clusters$Best.nc[1,])
```

As shown above, the best number of clusters for a particular dataset is determined by the code using the **NbClust()** function. The list returned by **NbClust()** comprises the optimal number of clusters as determined by one or more internal validation indices for clustering. The **table()** function is then used by the code to construct a frequency table of the best number of clusters discovered by **NbClust()**. The frequency of the optimal number of clusters identified by **NbClust()** overall validating indices are displayed in the ensuing table. A consensus on the ideal number of clusters can be reached using this.

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

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

*******************************************************************
* Among all indices:
* 6 proposed 2 as the best number of clusters
* 13 proposed 3 as the best number of clusters
* 1 proposed 5 as the best number of clusters
* 1 proposed 6 as the best number of clusters
* 1 proposed 8 as the best number of clusters
* 2 proposed 10 as the best number of clusters

                   ***** Conclusion *****

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


*******************************************************************
> table(nb_clusters$Best.nc[1,])

 0  2  3  5  6  8 10
 2  6 13  1  1  1  2
> table(nb_clusters$Best.nc[1,])

 0  2  3  5  6  8 10
 2  6 13  1  1  1  2
```

*Figure 4: NbClust result*

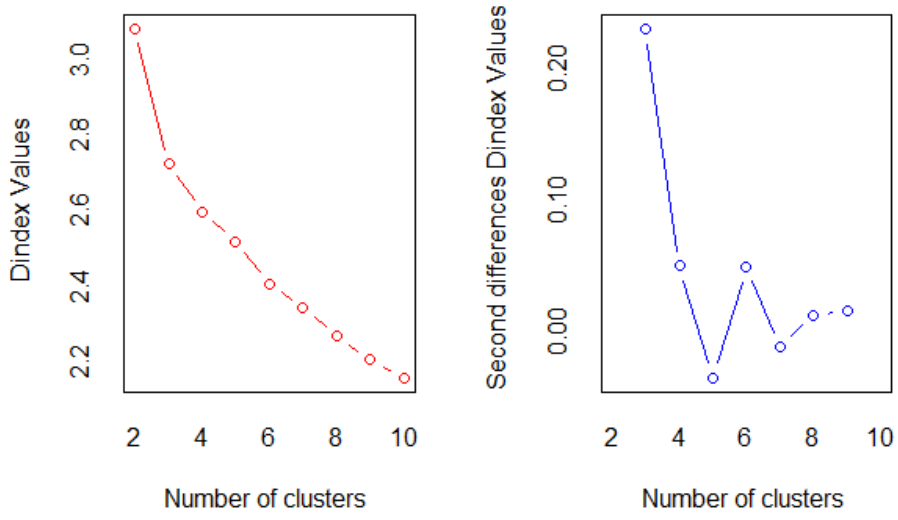


*Figure 5: NbClust graph*

According to the **NbClust** method, the best number of clusters is 3.
Before experimenting with the **Elbow method**, the following packages are installed and then loaded.

```
library(tidyverse)
library(cluster)
library(ggplot2)
```

**tidyverse** is a collection of packages for data manipulation and visualization. **cluster** provides clustering algorithms for data analysis. **ggplot2** is a powerful package for data visualization.
Together, these packages provide a set of tools for data analysis, manipulation, clustering, and visualization in R.

Like before the seed is assigned first.

```
set.seed(123)
wss <- sapply(1:10,
              function(k) {
                kmeans(clean_df, k, nstart = 10, iter.max =
50)$tot.withinss
              })
```

For k values between 1 and 10, the **sapply()** function uses the clean_df data to apply the **kmeans()** clustering algorithm. K-means clustering, an unsupervised machine learning approach used to divide data into k different clusters, is carried out using the **kmeans()** function in R. The function accepts a dataset as input and, depending on how close an observation is to the cluster centroids, places it in one of the k clusters. Here, the algorithm is executed ten times for each value of **k** to determine the total within-cluster sum of squares (WSS). The **WSS** values for each **k** value are contained in the resulting **wss** vector.

```
elbow_plot <- tibble(k = 1:10, wss = wss) %>%
  ggplot(aes(x = k, y = wss)) +
  geom_line() +
  geom_point() +
  scale_x_continuous(breaks = 1:10) +
  labs(x = "Number of Clusters (k)", y = "Within-Cluster Sum of
Squares (WSS)") +
  theme_minimal()
elbow_plot
```

The **tibble()** function first makes a table with two columns: **k** , and **wss**. Then, a plot is made using the **ggplot()** function with **k** on the x-axis and **wss** on the y-axis.
Plotting the relationship between wss and various values of **k** results in a line and points. The x- and y-axis labels are set using the **labs()** method. The **theme_minimal()** function, which sets the plot theme to a simple layout, is the last step. According to the below plot the best number of cluster is again 3.

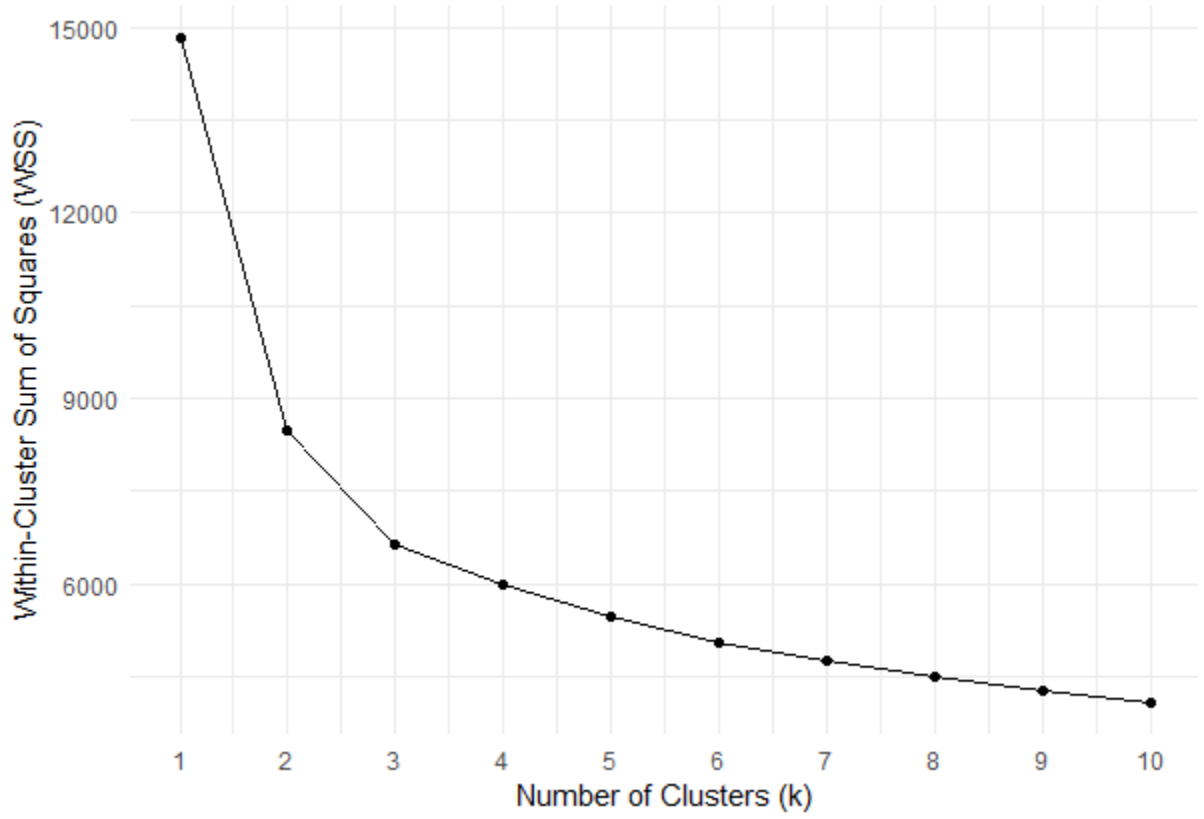


*Figure 6: Elbow curve*

For the **Gap Statistics** the following package is installed and loaded. **Factoextra** is a data visualization package

```
install.packages("factoextra")
library(factoextra)

set.seed(123)
gap_stat <- clusGap(clean_df, FUN = kmeans, nstart = 25, K.max = 10, B
= 50)

# Plot the gap statistic
fviz_gap_stat(gap_stat) + labs(title = "Gap Statistic Plot")

# Display the results
gap_stat
```

The gap statistic is calculated for a variety of cluster numbers using k-means clustering with the given parameters using the **clusGap()** function from the cluster package. The **gap_stat** object, which contains a

list of the gap statistic, its standard error, and the anticipated ideal number of clusters, stores the output. The results of the gap statistic are then plotted using the **factoextra** package's **fviz_gap_stat()** function. The findings are then displayed by printing the **gap_stat** object to the terminal. The best number of clusters for a particular dataset can be found using the gap statistic.

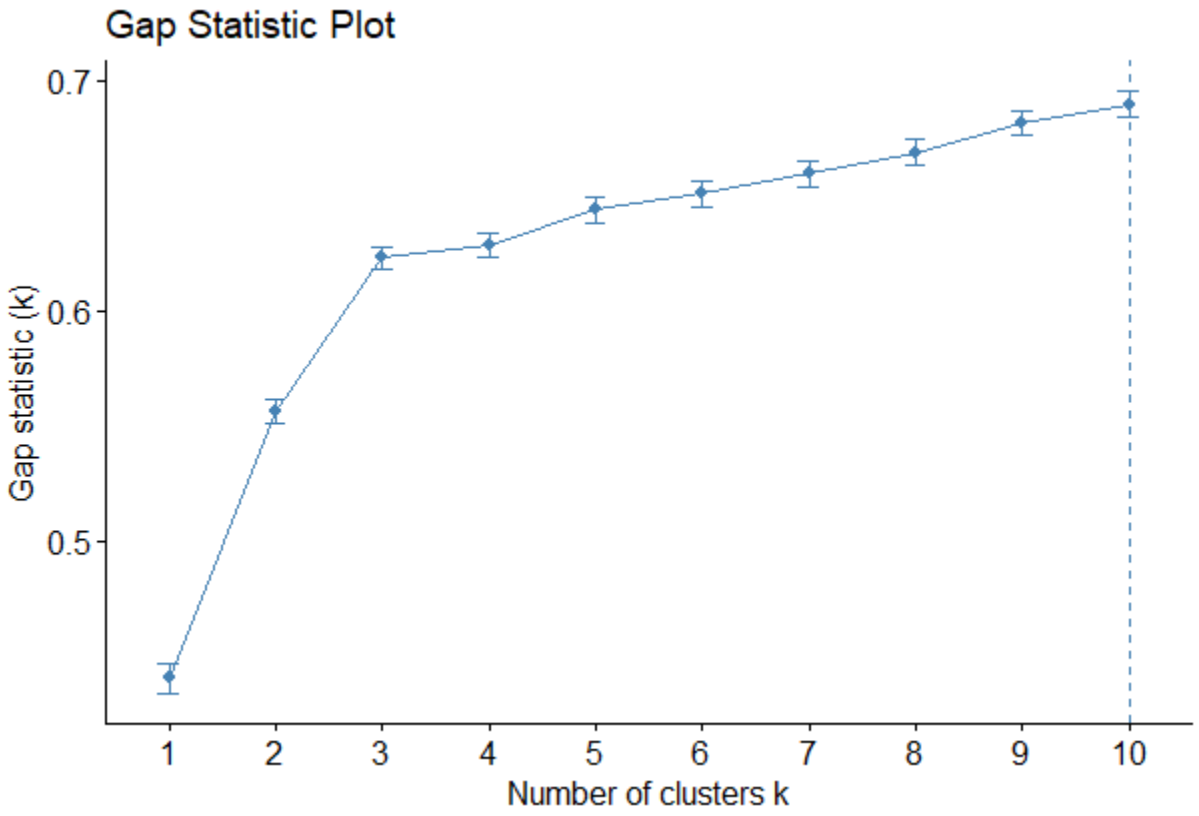


*Figure 7: Gap Static plot*

```
Clustering Gap statistic ["clusGap"] from call:
clusGap(x = clean_df, FUNcluster = kmeans, K.max = 10, B = 50, nstart = 25)
B=50 simulated reference sets, k = 1..10; spaceH0="scaledPCA"
 --> Number of clusters (method 'firstSEmax', SE.factor=1): 10
          logW   E.logW       gap      SE.sim
 [1,] 7.051790 7.492089 0.4402995 0.006579732
 [2,] 6.789396 7.345868 0.5564723 0.005262194
 [3,] 6.669110 7.292125 0.6230148 0.004766509
 [4,] 6.618782 7.247537 0.6287550 0.005309541
 [5,] 6.572141 7.216018 0.6438771 0.005535669
 [6,] 6.537970 7.188784 0.6508145 0.005683779
 [7,] 6.506523 7.166079 0.6595554 0.005380829
 [8,] 6.477435 7.146291 0.6688558 0.005503016
 [9,] 6.447444 7.129239 0.6817944 0.005508077
[10,] 6.424240 7.114142 0.6899017 0.005307107
```

*Figure 8: Gap Static result*

According to the Gap Static plot the best number of clusters is 3.
To determine the best number of clusters using the **Silhouette method**, the following code can be used.

```
set.seed(123)
silhouettemethod <- fviz_nbclust(clean_df, FUNcluster = kmeans, method
= "silhouette", nstart = 25, k.max = 10, verbose = FALSE)
plot(silhouettemethod)
```

The **fviz_nbclust()** function from the **factoextra** package is used in the code to get the average silhouette width for various clustering densities. The generated **silhouettemethod** object includes the widths of each silhouette as well as additional data for each **k**. The silhouette widths are then shown against the number of clusters using the **plot()** function, showing both the average silhouette width for each **k** and the silhouette widths for each individual observation in each cluster.

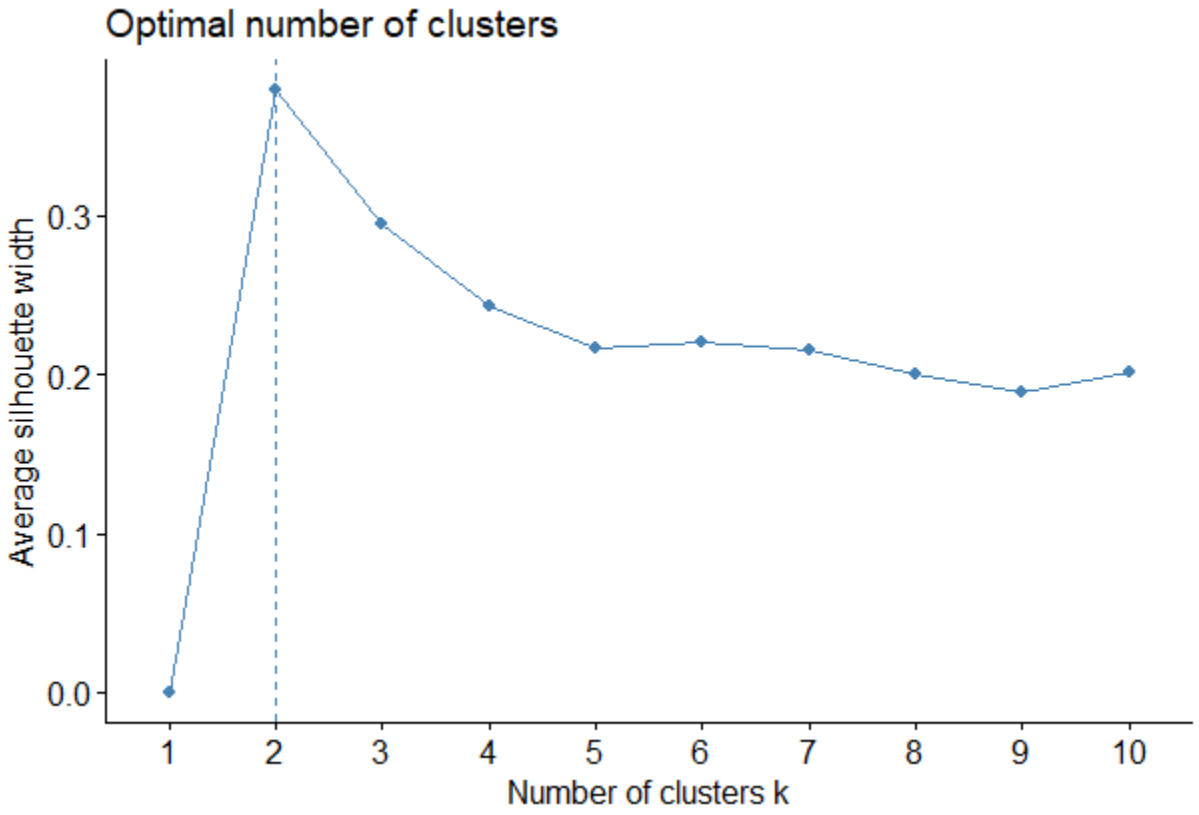


*Figure 9: Silhouette plot*

According to the above plot, the best number of clusters is 2.

In conclusion, according to all the above-automated tools, the best number of clusters can be identified as 3.

## C.

Following is the Kmeans analysis for K = 3:

```
set.seed(123)
kmeans_model <- kmeans(clean_df, centers = 3, nstart = 25, iter.max =
50)
kmeans_model$cluster
```

The number of times the algorithm to be run with various initial seeds is specified by the value of **nstart**, which is 25. The algorithm will be executed 25 times in this scenario. **iter.max** = 50: This specifies the maximum number of iterations that should be performed by the algorithm in order to locate the best clusters. The algorithm in this situation will run for a maximum of 50 iterations.
The **kmeans_model** variable holds the outcome.

```
  [1] 3 3 2 3 2 3 3 3 3 3 3 3 3 3 2 1 3 2 2 1 1 3 3 2 3 1 2 2 1 3 3 3 2 3 3 1 2 1 2 1 1 3 1
 [44] 1 1 1 3 1 3 2 3 2 3 3 1 2 1 2 1 1 1 3 1 1 2 3 2 2 2 3 1 3 2 3 1 2 1 1 2 3 1 3 3 1 3 1
 [87] 2 3 2 3 1 2 1 1 2 1 3 3 1 2 2 2 1 1 3 3 3 1 1 1 3 2 2 1 3 1 1 3 3 3 1 3 3 2 2 3 1 2 1
[130] 3 1 3 3 1 2 1 3 2 3 3 3 3 2 3 3 2 3 2 3 1 3 3 1 2 3 3 2 2 3 2 1 1 2 2 3 2 3 3 3 3 3 1
[173] 2 1 3 1 2 3 3 3 2 3 2 3 3 2 3 1 2 1 1 1 3 3 2 2 3 3 3 1 1 2 3 3 3 2 1 3 1 2 1 3 2 1 2
[216] 1 1 3 2 3 2 1 1 1 1 2 3 1 3 1 2 1 3 3 1 2 1 1 3 3 2 1 1 2 1 3 3 2 3 3 2 2 1 3 3 3 2 1
[259] 1 3 3 1 1 3 3 3 2 3 1 1 2 3 3 1 1 2 1 3 3 1 2 1 1 3 3 2 3 2 1 3 3 2 3 3 3 1 3 2 2 2 2
[302] 2 3 3 2 1 1 1 3 1 2 2 1 2 3 1 2 1 3 3 3 2 2 1 2 2 1 2 3 3 3 1 1 2 2 2 2 3 3 3 3 2 1 3 1
[345] 2 3 3 2 3 2 2 2 3 3 1 2 3 1 1 3 3 3 3 3 1 2 2 1 1 2 1 2 1 2 3 3 3 3 2 1 3 3 3 3 3 3 3
[388] 2 3 2 3 2 3 1 1 3 3 3 1 1 3 1 2 3 3 1 3 1 2 3 1 3 3 2 3 2 3 2 2 1 1 2 3 1 1 3 2 2 1 1
[431] 2 2 1 2 2 2 3 3 3 3 3 2 1 1 3 2 3 3 2 3 1 2 1 1 2 2 3 3 2 2 2 1 2 2 3 3 1 2 2 3 3 1 1
[474] 2 3 1 2 2 3 1 2 2 3 2 1 1 2 2 2 1 1 2 2 2 3 3 2 1 3 2 1 1 2 1 3 3 1 3 2 3 2 2 3 1 3 2
[517] 2 1 1 3 2 3 2 2 3 3 3 3 3 1 1 3 3 2 1 1 3 1 2 3 2 1 1 2 2 3 2 3 3 3 2 3 1 3 2 3 3 1 2
[560] 2 2 2 3 1 1 1 2 2 2 3 2 1 3 2 1 1 1 3 1 3 3 3 3 3 3 3 2 3 3 2 3 3 3 1 2 1 1 3 1 3 3 1
[603] 1 2 2 1 3 1 2 3 3 2 3 1 2 1 2 1 1 3 1 3 2 2 3 2 3 3 1 3 1 2 3 2 1 3 3 3 1 3 1 3 2 3 2
[646] 1 3 3 3 3 2 3 1 2 3 2 3 3 2 1 2 1 3 3 3 1 2 3 1 3 1 2 3 3 2 1 3 1 3 3 1 3 2 2 3 3 2 2
[689] 3 1 3 2 2 2 2 3 2 3 3 2 2 3 2 3 2 3 1 2 3 1 2 2 2 3 1 1 2 2 2 3 2 3 3 2 3 1 3 1 3 2 3
[732] 1 3 3 3 1 2 1 1 1 2 1 2 2 1 3 3 2 3 1 2 2 1 3 3 2 2 2 1 2 3 2 2 1 1 2 1 2 3 1 3 2 2 3
[775] 1 3 2 2 3 3 1 3 3 2 1 3 2 1 1 2 1 3 1 1 1 3 2 2 3 1 2 3 2 2 1 3 2 1 1 3 3 2 1 1 1 3 3
[818] 3 3 3 3 2 3 1
```

*Figure 10: Cluster assignments*

The final command delivers the cluster assignments for each data point in the original dataset by gaining access to the cluster component of the **kmeans_model** object.

```
# Set the seed for reproducibility
set.seed(123)
# Calculate the within-cluster sum of squares (WSS)
wss <- sum(kmeans_model$withinss)
wss

# Calculate the between-cluster sum of squares (BSS)
bss <- sum(kmeans_model$betweenss)
bss
```

```
# Print the ratio of between_cluster_sums_of_squares (BSS) over
total_sum_of_Squares (TSS)
cat("\n\nBSS/TSS ratio:\n")
cat(kmeans_model$betweenss / kmeans_model$totss)
```

This code calculates the within-cluster sum of squares (WSS) and the between-cluster sum of squares (BSS) for the k-means clustering model stored in kmeans_model. It then prints the ratio of the BSS to the total sum of squares (TSS)

```
> wss <- sum(kmeans_model$withinss)
> wss
[1] 6624.09
> bss <- sum(kmeans_model$betweenss)
> bss
[1] 8189.91
> cat("\n\nBSS/TSS ratio:\n")


BSS/TSS ratio:
> cat(kmeans_model$betweenss / kmeans_model$totss)
0.5528493
```

*Figure 11: Cluster analysis.*

The amount of data variability that can be explained by the clustering model is indicated by the BSS/TSS ratio. The clusters are more identifiable and the clustering model is a better fit for the data when the BSS/TSS ratio is larger.
Here, the BSS/TSS ratio of 0.5528493 shows that the clustering model accounts for nearly 55% of the total variability in the data.

For the K-means clustering analysis, a plot is created using the below code to represent each cluster.

```
library(factoextra)
fviz_cluster(kmeans_model, geom = "point", data = clean_df, stand =
FALSE, palette = "jco", ggtheme = theme_minimal(), main = "Kmeans
Clustering Results")
```

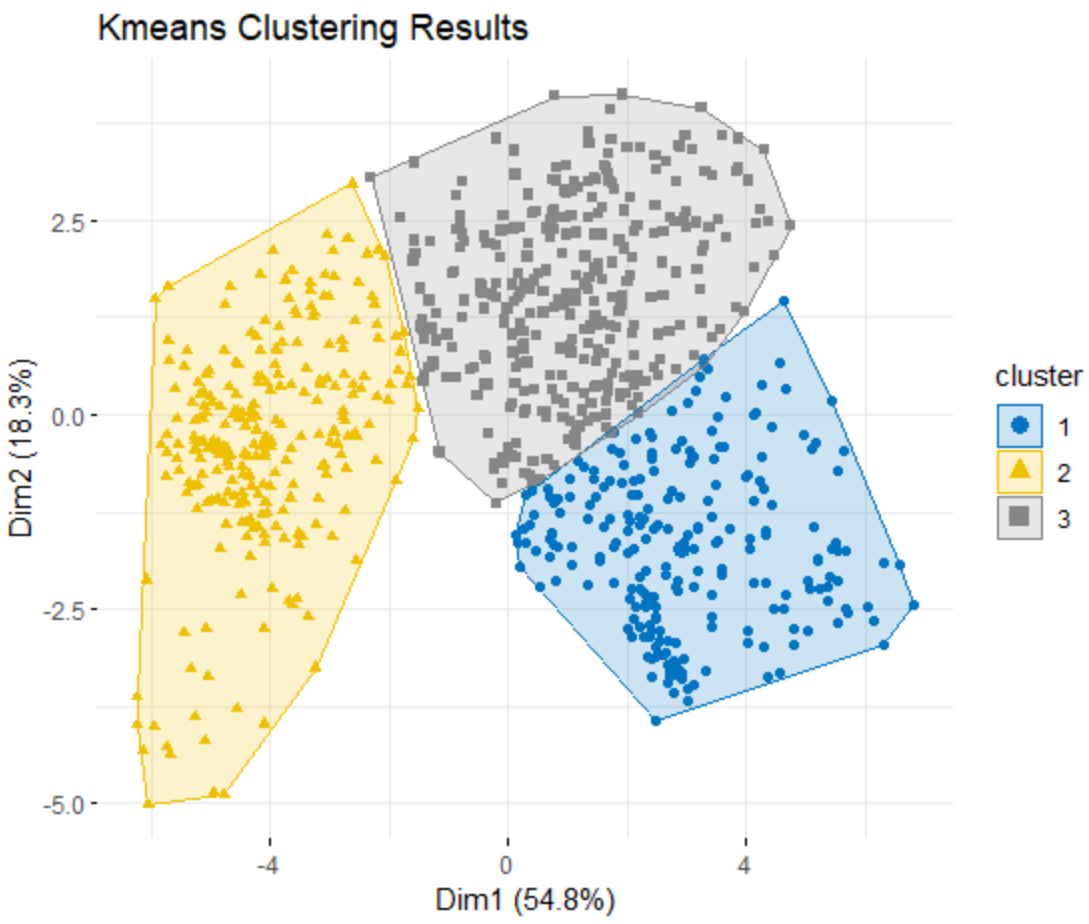


*Figure 12: K-means clustering results*

D.

```
# Calculate silhouette widths
silwidths <- silhouette(kmeans_model$cluster, dist(clean_df))
head(silwidths)
```

```
     cluster neighbor  sil_width
[1,]       3        2 0.21352716
[2,]       3        1 0.09877913
[3,]       2        3 0.45956538
[4,]       3        1 0.39695831
[5,]       2        1 0.38253937
[6,]       3        1 0.35259643
```

*Figure 13: silhouette widths*

```
# Plot the silhouette widths
plot(silwidths, main = "Silhouette Plot for Kmeans Clustering", border
= NA, col = 1:3)
```

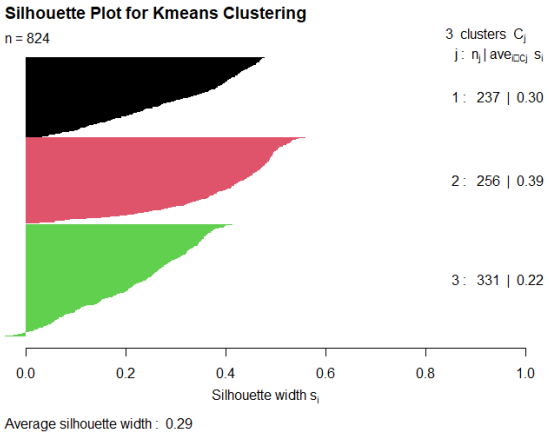


*Figure 14: Silhouette plot for kmeans clustering*

```
# Average silhouette width
mean(silwidths[,3])
```

```
[1] 0.2941369
```

The discovered clusters' quality can be characterized as moderate based on the average silhouette width of 0.2941369. A data point's silhouette width indicates how well it fits within its assigned cluster in relation to other clusters. A point may be allocated to the incorrect cluster if its silhouette width is low,

while a bigger silhouette width signals that a point is well-matched to its own cluster and poorly suited to other clusters. An average silhouette width of 0.29 indicates that the clustering solution is reasonable, but there may be some overlap between clusters or points that are not well-matched to any cluster just like the small overlap in clusters 2 and 3.

## 2nd Subtask Objectives:

## E.

```
PCAVehicles <- prcomp(clean_df, center = TRUE, scale = FALSE)
eigenvalue <- PCAVehicles$sdev^2
eigenvalue
eigenvector <- PCAVehicles$rotation
head(eigenvector)
```

The **clean_df** is subjected to a Principal Component Analysis (PCA). With **scale = FALSE** and **centre = TRUE** (which centers the data), the **prcomp()** function performs the PCA. The **PCAVehicles** object contains the PCA's results.

The eigenvalues and eigenvectors are then calculated by the programme. The weights that define each principal component as a linear combination of the original variables are represented by the eigenvectors, while the eigenvalues indicate the variance explained by each principal component.

```
 [1] 9.8655415144 3.3026315672 1.2050866140 1.1255984677 0.8773731809 0.6636174794 0.3374343341
 [8] 0.2274918343 0.1176165632 0.0871789864 0.0607683889 0.0450646831 0.0292070985 0.0213994038
[15] 0.0150961795 0.0123913361 0.0061400710 0.0003622977
```

*Figure 15: Eigen values*

```
                PC1         PC2         PC3         PC4         PC5          PC6         PC7
Comp      -0.2709955  0.08819711 -0.03979285 -0.14247427 -0.15979926  0.219704493  0.25075003
Circ      -0.2853800 -0.14799378 -0.19761320  0.02334808  0.12602923 -0.019390179 -0.38184560
D.Circ    -0.3007838  0.04064437  0.07450874 -0.10451348  0.07338676  0.000941066  0.10924250
Rad.Ra    -0.2759548  0.19284625  0.04085638  0.24408001 -0.12620414 -0.153234232  0.13812347
Pr.Axis.Ra -0.1079011  0.24598582 -0.10092681  0.61190884 -0.05646656 -0.599471567  0.06368508
Max.L.Ra  -0.1869378  0.06836380 -0.10600156 -0.25524165  0.70801896 -0.255529947  0.40902849
                PC8         PC9         PC10        PC11         PC12         PC13         PC14
Comp      -0.76291750  0.33672726 -0.17080380  0.06059915  0.01623622 -0.15538799 -0.084941797
Circ      -0.08499684  0.04816196  0.14521912 -0.06103582 -0.10800251 -0.02379761  0.200359434
D.Circ     0.30756035  0.36929755  0.09330027  0.74865950  0.02723692  0.23107314 -0.032038645
Rad.Ra     0.06236231  0.15903921 -0.02487175 -0.17932432 -0.14827879  0.02028449  0.782962301
Pr.Axis.Ra -0.14661865  0.03307520  0.08677308  0.04900671  0.06151173  0.03176897 -0.360686574
Max.L.Ra  -0.03264265 -0.22773912 -0.25103768 -0.10840290 -0.10328486  0.09369549 -0.004005999
                PC15         PC16         PC17          PC18
Comp      -0.009893937  0.01473145  0.002287114 -0.0001888306
Circ      -0.411699600  0.63319765  0.193560642  0.0189798203
D.Circ    -0.128176485 -0.03294137 -0.033808260 -0.0095717960
Rad.Ra    -0.002680653 -0.26218516  0.006068730 -0.0275176970
Pr.Axis.Ra  0.022171363  0.09120709 -0.009140544  0.0177603416
Max.L.Ra  -0.047699349  0.02392462 -0.004840491 -0.0083581152
```

*Figure 16: Head of Eigan vectors*

```
cum_score <- cumsum(PCAVehicles$sdev^2 / sum(PCAVehicles$sdev^2))
print(cum_score)
```

the above code determines the cumulative proportion of variance explained by each principal component. To achieve this, use **PCAVehicles$sdev2** to divide the square of each component's standard deviation by the total of the squares of all standard deviations, and then use the **cumsum** function to calculate the cumulative sum of these values. The number of primary components that satisfy a particular threshold of variance explained can be determined using the resulting cumulative percentage of variance explained.

```
 [1] 0.5480856 0.7315652 0.7985144 0.8610477 0.9097906 0.9466583 0.9654046 0.9780431 0.9845773
[10] 0.9894206 0.9927966 0.9953002 0.9969228 0.9981117 0.9989503 0.9996388 0.9999799 1.0000000
```

*Figure 17: cum_score values*

Next the below code will select the number of PCs that reaches 92.

```
num_PCAs <- which(cum_score >= 0.92)[1]
num_PCAs
```

And the number of PCs = **6.**

Finally, a new data frame called “transformed” is created to store the above 6 PCs.

```
transformed <- as.data.frame(PCAVehicles$x[, 1:num_PCAs])
view(head(transformed))
```

F.

**NbClust method**

```
set.seed(123)
PCA_nbClust <- NbClust(transformed, distance = "euclidean", min.nc = 2, max.nc = 15, method = "kmeans", index = "all")
```

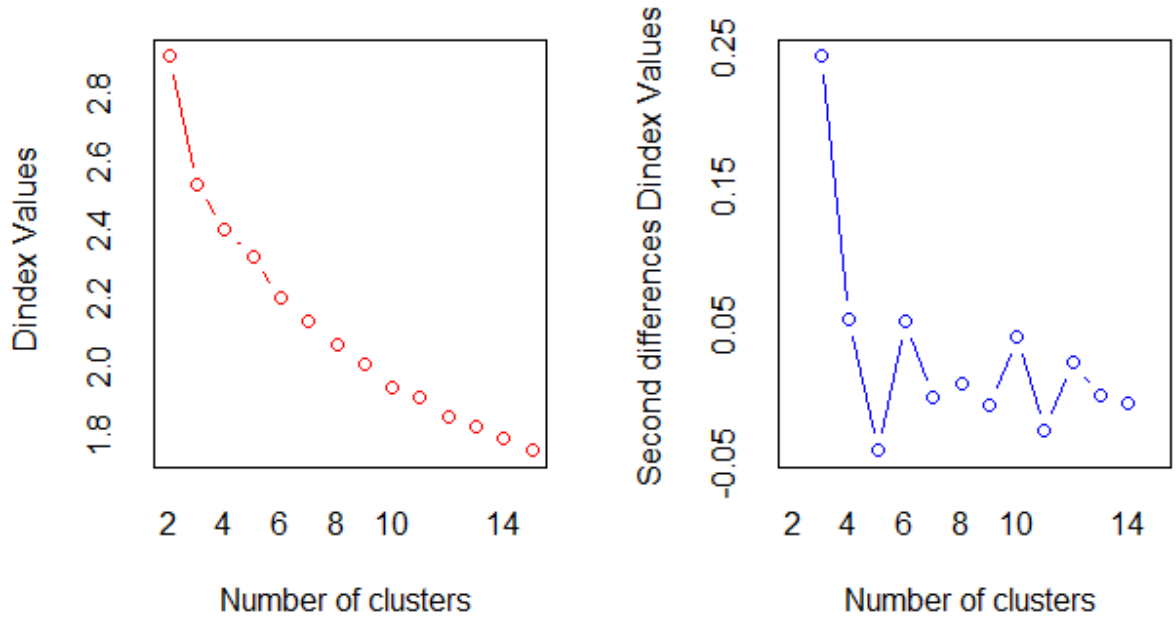


*Figure 18: NbClust graph II*

*Figure 19: NbClust result 2*

According to the graph the best number of clusters is 3.

**Elbow method**

```
fviz_nbclust(transformed, kmeans, method= "wss") + labs(subtitle = "Elbow method")
```

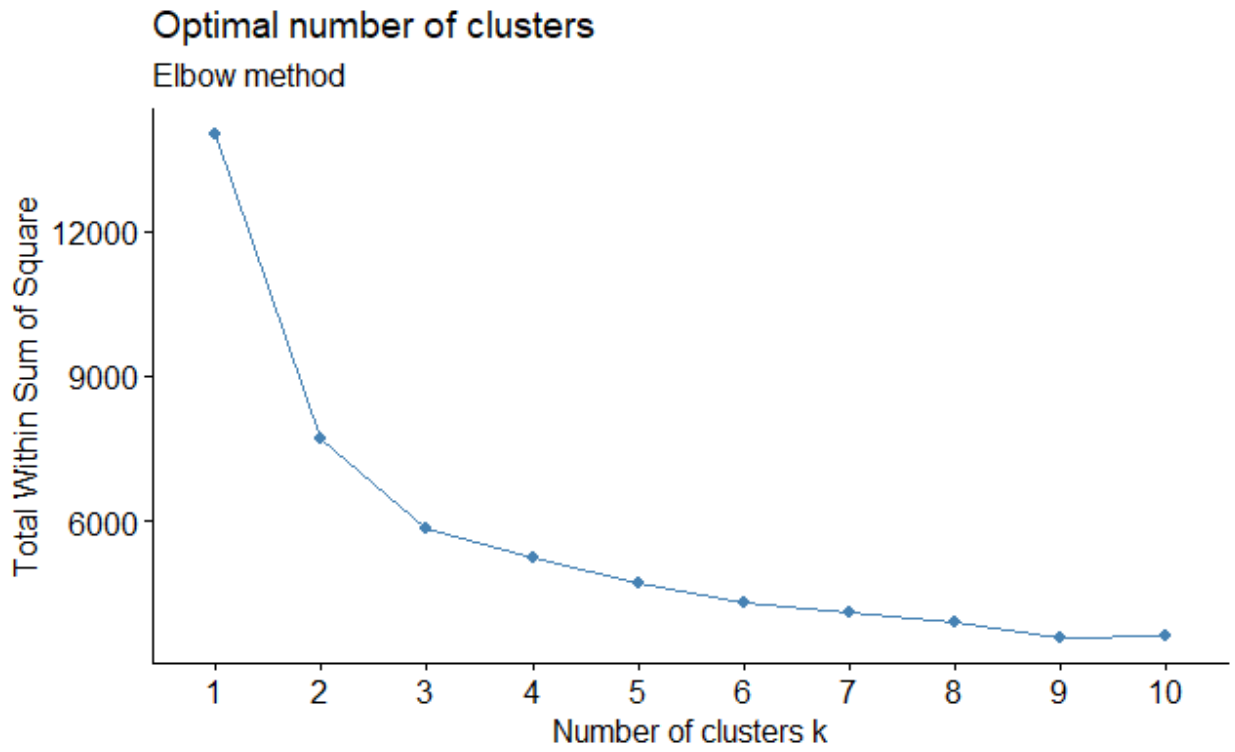


*Figure 20: Elbow curve II*

According to the graph the best number of clusters is 3.

**Gap Statistic method**

**fviz_nbclust(transformed, kmeans, nstart = 25,  method = "gap_stat", nboot = 50)+labs(subtitle = "Gap statistic method")**

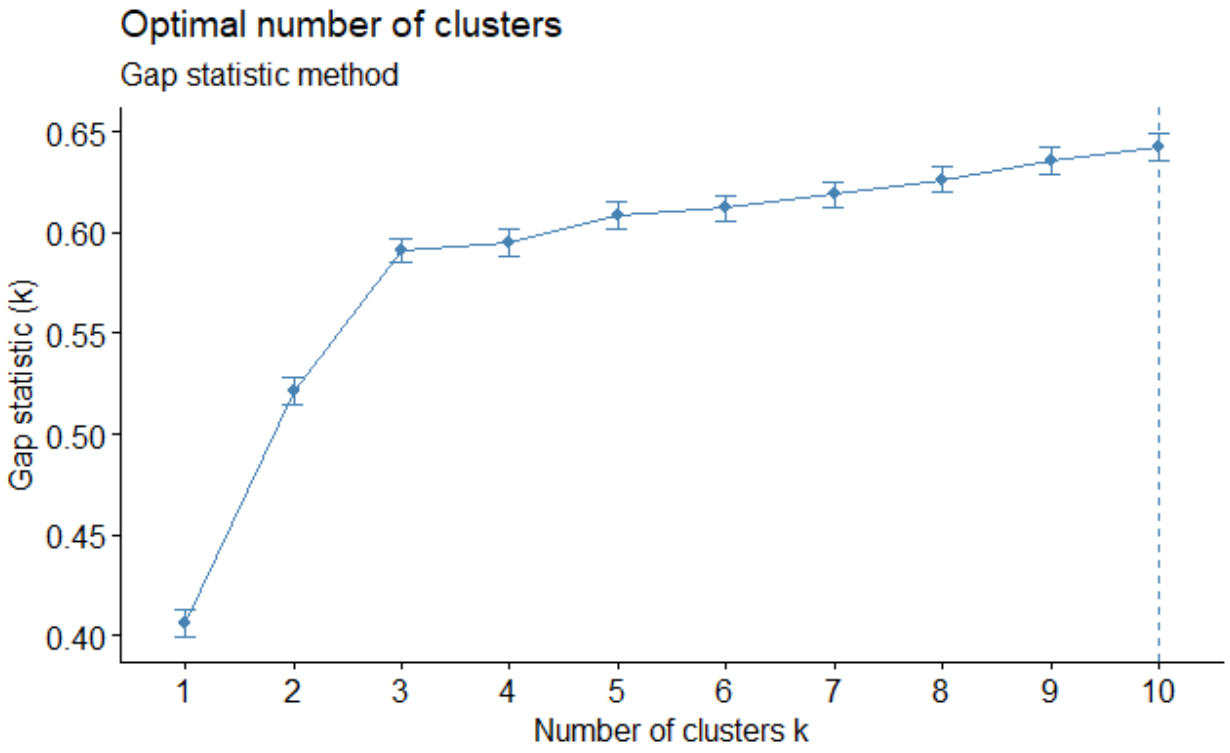


*Figure 21: Gap Statistic plot II*

According to the graph the best number of clusters is 3.

**Silhouette method**

```
fviz_nbclust(transformed, kmeans, method = "silhouette") + labs(subtitle = "Silhouette method")
```

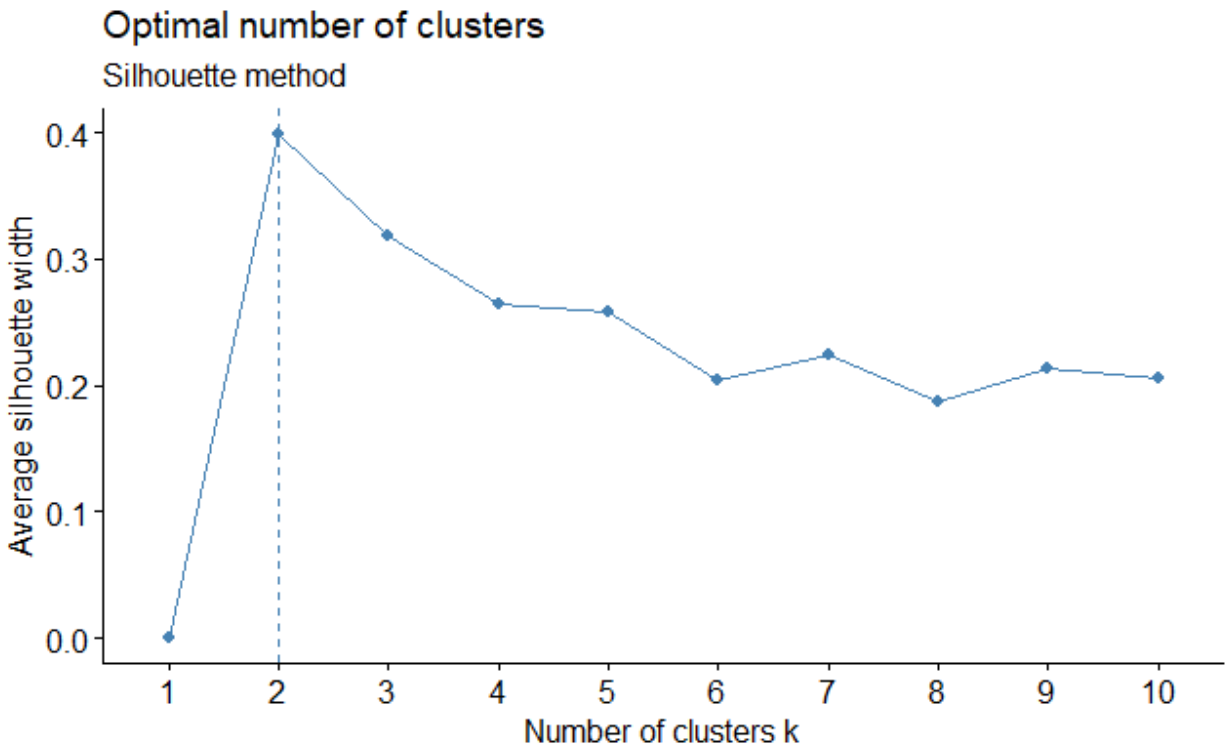


*Figure 22: Silhouette plot II*

According to the graph the best number of clusters is 2.

Therefore after plotting the 4 graphs, we can conclude that the optimal number of clusters is 3.

## G.

This part is done exactly as the above k- means analysis in subtask 1.c.

Following is the Kmeans analysis for K = 3:

```
set.seed(123)
PCA_kmeans_model <- kmeans(transformed, centers = 3, nstart = 25,
iter.max = 50)
PCA_kmeans_model$cluster
```

```
  [1] 3 3 2 3 2 3 3 3 3 3 3 3 3 3 2 1 3 2 2 1 1 3 3 2 3 1 2 2 1 3 3 3 2 3 3 1 2 1 2 1 1 3 1
 [44] 1 1 1 3 1 3 2 3 2 3 3 1 2 1 2 1 1 1 3 1 1 2 3 2 2 2 3 1 3 2 3 1 2 1 1 2 3 1 3 3 1 3 1
 [87] 2 3 2 3 1 2 1 1 2 1 3 3 1 2 2 2 1 1 3 3 3 1 1 1 3 2 2 1 3 1 1 3 3 3 1 3 3 2 2 3 1 2 1
[130] 3 1 3 3 1 2 1 3 2 3 3 3 3 2 3 3 2 3 2 3 1 3 3 1 2 3 3 2 2 3 2 1 1 2 2 3 2 3 3 3 3 3 1
[173] 2 1 3 1 2 3 3 3 2 3 2 3 3 2 3 1 2 1 1 1 3 3 2 2 3 3 3 1 1 2 3 3 3 2 1 3 1 2 1 3 2 1 2
[216] 1 1 3 2 3 2 1 1 1 1 2 3 1 3 1 2 1 3 3 1 2 1 1 3 3 2 1 1 2 1 3 3 2 3 3 2 2 1 3 3 3 2 1
[259] 1 3 3 1 1 3 3 3 2 3 1 1 2 3 3 1 1 2 1 3 3 1 2 1 1 3 3 2 3 2 1 3 3 2 3 3 3 1 3 2 2 2 2
[302] 2 3 3 2 1 1 1 3 1 2 2 1 2 3 1 2 1 3 3 3 2 2 1 2 2 1 2 3 3 3 1 1 2 2 2 2 3 3 3 2 1 3 1
[345] 2 3 3 2 3 2 2 2 3 3 1 2 3 1 1 3 3 3 3 3 1 2 2 1 1 2 1 2 1 2 3 3 3 3 2 1 3 3 3 3 3 3 3
[388] 2 3 2 3 2 3 1 1 3 3 3 1 1 3 1 2 3 3 1 3 1 2 3 1 3 3 2 3 2 3 2 2 1 1 2 3 1 1 3 2 2 1 1
[431] 2 2 1 2 2 2 3 3 3 3 3 2 1 1 3 2 3 3 2 3 1 2 1 1 2 2 3 3 2 2 2 1 2 2 3 3 1 2 2 3 3 1 1
[474] 2 3 1 2 2 3 1 2 2 3 2 1 1 2 2 2 1 1 2 2 2 3 3 2 1 3 2 1 1 2 1 3 3 1 3 2 3 2 2 3 1 3 2
[517] 2 1 1 3 2 3 2 2 3 3 3 3 3 1 1 3 3 2 1 1 3 1 2 3 2 1 1 2 2 3 2 3 3 3 2 3 1 3 2 3 3 1 2
[560] 2 2 2 3 1 1 1 2 2 2 3 2 1 3 2 1 1 1 3 1 3 3 3 3 3 3 3 2 3 3 2 3 3 3 1 2 1 1 3 1 3 3 1
[603] 1 2 2 1 3 1 2 3 3 2 3 1 2 1 2 1 1 3 1 3 2 2 3 2 3 3 1 3 1 2 3 2 1 3 3 3 1 3 1 3 2 3 2
[646] 1 3 3 3 3 2 3 1 2 3 2 3 3 2 1 2 1 3 3 3 1 2 3 1 3 1 2 3 3 2 1 3 1 3 3 1 3 2 2 3 3 2 2
[689] 3 1 3 2 2 2 2 3 2 3 3 2 2 3 2 3 2 3 1 2 3 1 2 2 2 3 1 1 2 2 2 3 2 3 3 2 3 1 3 1 3 2 3
[732] 1 3 3 3 1 2 1 1 1 2 1 2 2 1 3 3 2 3 1 2 2 1 3 3 2 2 2 1 2 3 2 2 1 1 2 1 2 3 1 3 2 2 3
[775] 1 3 2 2 3 3 1 3 3 2 1 3 2 1 1 2 1 3 1 1 1 3 2 2 3 1 2 3 2 2 1 3 2 1 1 3 3 2 1 1 1 3 3
[818] 3 3 3 3 2 3 1
```

*Figure 23: cluster assignments II*

```
set.seed(123)
# Calculate the within-cluster sum of squares (WSS)
PCA_wss <- sum(PCA_kmeans_model$withinss)
PCA_wss

# Calculate the between-cluster sum of squares (BSS)
PCA_bss <- sum(PCA_kmeans_model$betweenss)
PCA_bss

# Print the ratio of between_cluster_sums_of_squares (BSS) over
total_sum_of_Squares (TSS)
cat("\n\nBSS/TSS ratio:\n")
cat(PCA_kmeans_model$betweenss / PCA_kmeans_model$totss)
```

```
[1] 6624.09
> # Calculate the between-cluster sum of squares (BSS)
> PCA_bss <- sum(PCA_kmeans_model$betweenss)
> PCA_bss
[1] 8189.91
> # Print the ratio of between_cluster_sums_of_squares (BSS) over total_sum_of_Squares (TSS)
> cat("\n\nBSS/TSS ratio:\n")


BSS/TSS ratio:
> cat(PCA_kmeans_model$betweenss / PCA_kmeans_model$totss)
0.5528493
```

*Figure 24: k-means analysis II*

## H.

This part is done same as the subtask 1. D.

```
# Calculate silhouette widths
PCA_silwidths <- silhouette(PCA_kmeans_model$cluster, dist(transformed))
head(PCA_silwidths)
```

```
     cluster neighbor sil_width
[1,]       3        2 0.2342583
[2,]       3        1 0.1097372
[3,]       2        3 0.4954922
[4,]       3        1 0.4287002
[5,]       2        1 0.3904031
[6,]       3        1 0.3697722
```

*Figure 25: Silhouette widths II*

```
# Plot the silhouette widths
plot(PCA_silwidths, main = "Silhouette Plot for Kmeans Clustering", border = NA, col = 1:3)
```

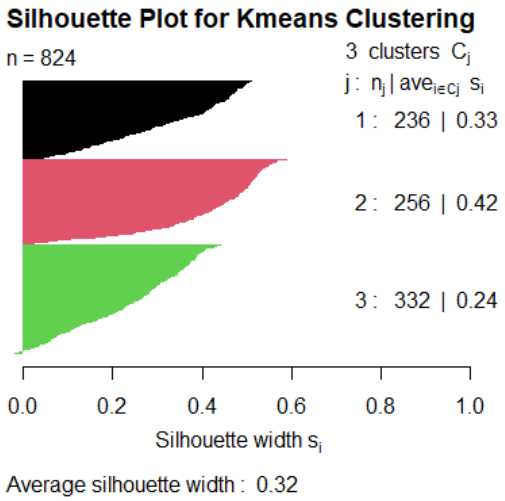


*Figure 26: Silhouette plot for K-means clustering II*

I.

First, a function called **kmeans_func** created and then uses it to calculate and visualize the **Calinski-Harabasz** index using the **clusGap()** and **fviz_gap_stat()** functions.

```
kmeans_func <- function(x, k) {
  kmeans(x, centers = k, nstart = 25)
}

# Visualization of Calinski-Harabasz Index
calinski_harabasz <- clusGap(transformed, kmeans_func, K.max =
length(PCA_kmeans_model$size))
fviz_gap_stat(calinski_harabasz) + labs(subtitle = "Calinski-Harabasz
Index")
```

Here the maximum number of clusters is set to the length of **PCA_kmeans_model$size**, which suggests that the number of clusters is determined based on a previous analysis stored in **PCA_kmeans_model.**

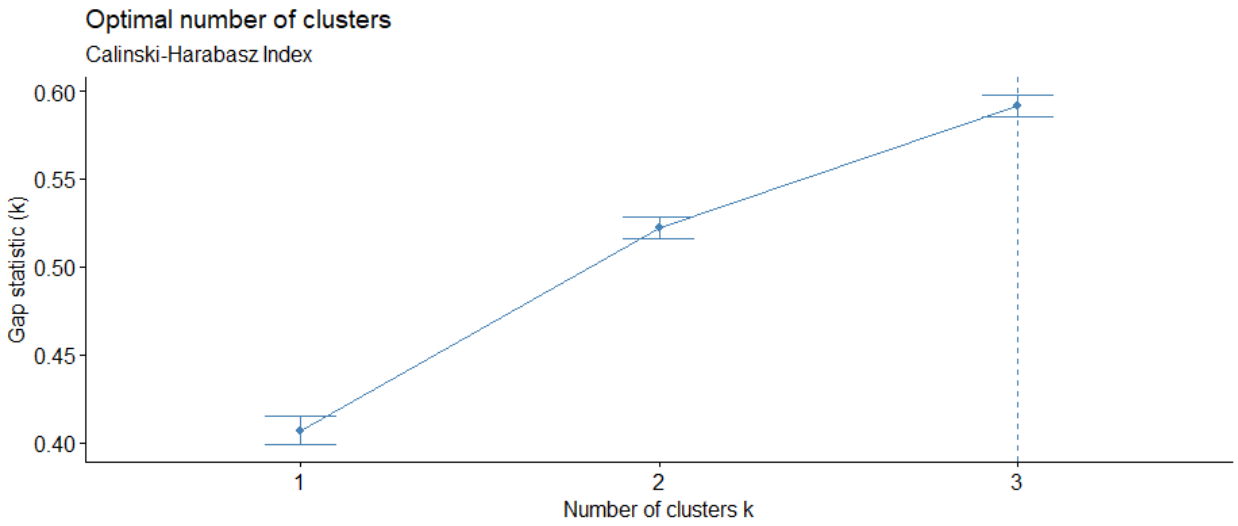


*Figure 26: Calinski_Harabasz index*

According to the Calinski-Harabarsz Index the optimal number of clusters is 3.

# Energy Forecasting Part

## 2nd Subtask Objectives:

### a)

First of all the necessary libraries are loaded.

```
library(neuralnet)
library(dplyr)
library(readxl)
library(MLmetrics)
library(Metrics)
```

The **neuralnet** library offers tools for training and testing neural networks. Using the backpropagation technique enables the development and training of feedforward neural networks with numerous hidden layers. The **dplyr** offers a number of tools for transforming and manipulating data and the **readxl** library makes it possible to read data from Excel files. Machine learning model comparison and evaluation functions are available in the R **MLmetrics** and **Metrics** libraries. These libraries include a range of metrics and techniques for evaluating the efficacy and accuracy of models.

Next, the uow_consumption excel file is loaded into the R-studio using the **read_excel** function.

electricity_consumption<- read_excel("electricity_consumption.xlsx")
View(head(electricity_consumption))

| | date | 0.75 | 0.79166666666666663 | 0.83333333333333337 |
|---|---|---|---|---|
| 1 | 2018-01-01 | 38.9 | 38.9 | 38.9 |
| 2 | 2018-01-02 | 42.3 | 41.9 | 41.9 |
| 3 | 2018-01-03 | 40.8 | 40.5 | 40.7 |
| 4 | 2018-01-04 | 42.3 | 41.9 | 41.9 |
| 5 | 2018-01-05 | 44.0 | 44.1 | 44.0 |
| 6 | 2018-01-06 | 45.6 | 44.5 | 44.3 |

*Figure 27: uow_consumption data frame*

Next, the column names are renamed.

```
colnames(electricity_consumption) <- c("Date", "18", "19", "20")
View(head(electricity_consumption))
```

| | Date | 18 | 19 | 20 |
|---|---|---|---|---|
| 1 | 2018-01-01 | 38.9 | 38.9 | 38.9 |
| 2 | 2018-01-02 | 42.3 | 41.9 | 41.9 |
| 3 | 2018-01-03 | 40.8 | 40.5 | 40.7 |
| 4 | 2018-01-04 | 42.3 | 41.9 | 41.9 |
| 5 | 2018-01-05 | 44.0 | 44.1 | 44.0 |
| 6 | 2018-01-06 | 45.6 | 44.5 | 44.3 |

*Figure 28: Renamed uow_consumption data frame*

Then, with the help of the lag() function from the **dplyr** library, time-delayed variables can be produced. The values of the **uow_consumption$'20'** variable are shifted by the amount of time periods supplied using the **lag()** function. The variable is then created with lagged versions. Then, a new data frame named **time_delayed** is created by horizontally binding these lag variables together using the **bind_cols()** function from the **dplyr** package. The lag period is followed by a name that begins with "T" for each lagged variable. T7, for instance, stands for a variable that is time-delayed by 7 periods. The **lag()** function is also used to construct the **outputprediction** variable, which reflects the original variable instantly and has a lag of 0.

```
time_delayed <- bind_cols(
  T7 = lag(electricity_consumption$'20', 7),
  T4 = lag(electricity_consumption$'20', 4),
  T3 = lag(electricity_consumption$'20', 3),
  T2 = lag(electricity_consumption$'20', 2),
  T1 = lag(electricity_consumption$'20', 1),
  outputprediction = lag(electricity_consumption$'20', 0)
)
```

Now the rows with missing values are removed from the data frame.

```
time_delayed <- na.omit(time_delayed)
```

Based on the **time_delayed** data frame, several time-delayed input vectors and their matching input/output matrices are built.

```
T_1 <- cbind(time_delayed$T1, time_delayed$outputprediction)
colnames(T_1)<- c("Input", "Output")

T_2 <- cbind(time_delayed$T1, time_delayed$T2,
time_delayed$outputprediction)
colnames(T_2)<- c("Input_1", "Input_2", "Output")

T_3 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3,
time_delayed$outputprediction)
colnames(T_3)<- c("Input_1", "Input_2", "Input_3", "Output")

T_4 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3,
time_delayed$T4, time_delayed$outputprediction)
colnames(T_4)<- c("Input_1", "Input_2", "Input_3", "Input_4",
"Output")

T_5 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3,
time_delayed$T4,time_delayed$T7, time_delayed$outputprediction)
colnames(T_5)<- c("Input_1", "Input_2", "Input_3", "Input_4",
"Input_5", "Output")
```

Different combinations of lag variables are captured by each vector of time-delayed inputs. The **cbind()** function is used to combine the variables to create each matrix. For instance, the **time_delayed$T1** variable (lag 1) and the **time_delayed$outputprediction** variable (lag 0) are combined using the **cbind()** method in T_1. These matrices store the matching output variable as well as various combinations of lagged variables as inputs. These matrices are most likely being created with the intention of using them as training data for neural networks.

b)

There are various benefits to normalizing data before using it in an MLP (Multi-Layer Perceptron) or other neural networks. First, normalization enhances convergence during training by bringing input features to a similar scale. This guarantees that the optimization process is effective and efficient by preventing some features from dominating others. Second, normalization aids in avoiding deep neural network problems with vanishing or ballooning gradients. Normalization helps to reduce issues with numerical instability and promotes more seamless training by maintaining the input values within a tolerable range. Additionally, normalization can be thought of as a type of regularisation because it limits input values to a certain range, preventing overfitting and improving the generalizability of the model.

A function is created for normalization and applied to the I/O matrices

```
#Defining the normalize function (Min-Max Normalization)
norm <- function(x) {
  return ((x - min(x)) / (max(x)-min(x)))
}

#Normalizing the I/O Matrices
normT_1 <- norm(T_1)
normT_2 <- norm(T_2)
normT_3 <- norm(T_3)
normT_4 <- norm(T_4)
normT_5 <- norm(T_5)
```

c)

The training and testing sets are defined for each input/output (I/O) matrix. The training set is used to train the neural network model, while the testing set is used to evaluate the model's performance on unknown data. The first 380 rows are part of the training set, and the remaining rows, beginning with row 381, are part of the testing set. You can train the neural network on the training set and evaluate its performance on the independent testing set by dividing the data into training and testing sets. This allows you to evaluate how well the model generalizes to unseen data and assess its overall predictive capabilities.

```
train_T_1 <- normT_1[1:380,]
test_T_1 <- normT_1[381: nrow(normT_1),]

train_T_2 <- normT_2[1:380,]
test_T_2 <- normT_2[381: nrow(normT_2),]

train_T_3 <- normT_3[1:380,]
test_T_3 <- normT_3[381: nrow(normT_3),]

train_T_4 <- normT_4[1:380,]
test_T_4 <- normT_4[381: nrow(normT_4),]

train_T_5 <- normT_5[1:380,]
test_T_5 <- normT_5[381: nrow(normT_5),]
```

For each I/O matrix, different iterations of the MLP (Multi-Layer Perceptron) architecture are used to train neural networks in the given below code.

Version 1 of one hidden layer neural networks:

```
T_1_NN1 <- neuralnet(Output ~ Input, data = train_T_1, hidden = 4,
linear.output = TRUE)
plot(T_1_NN1)
```

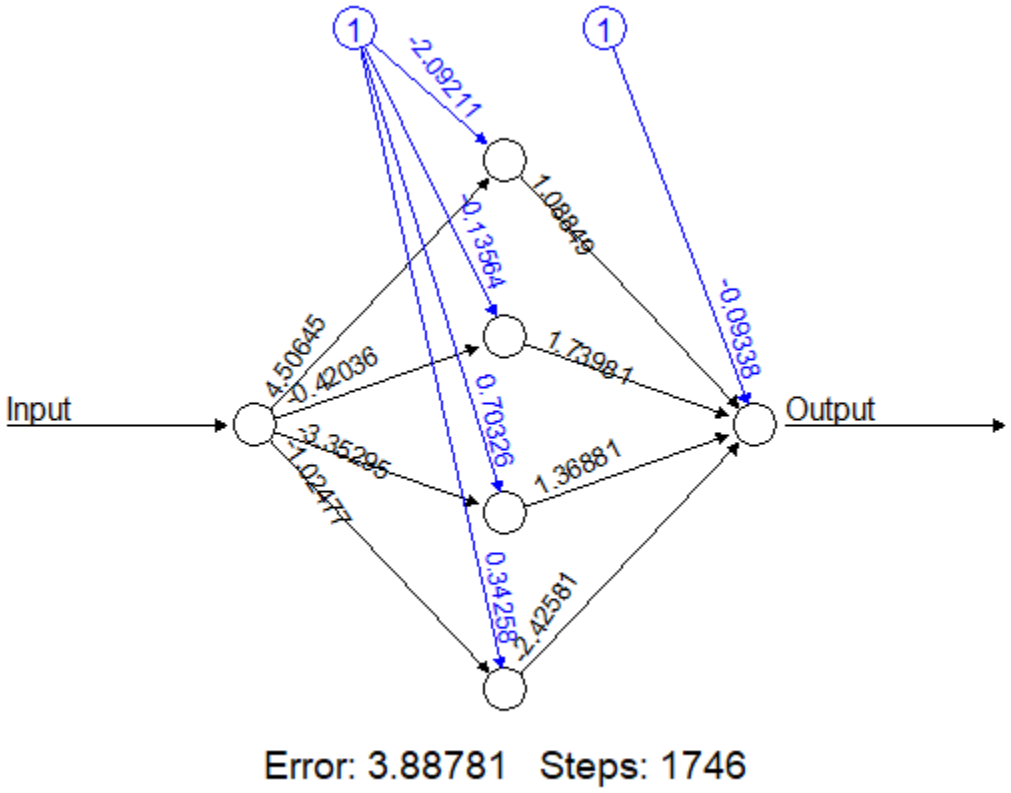


*Figure 29: T_1_NN1 plot*

```
T_2_NN1 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2,
hidden = 4, linear.output = TRUE)
plot(T_2_NN1)
```

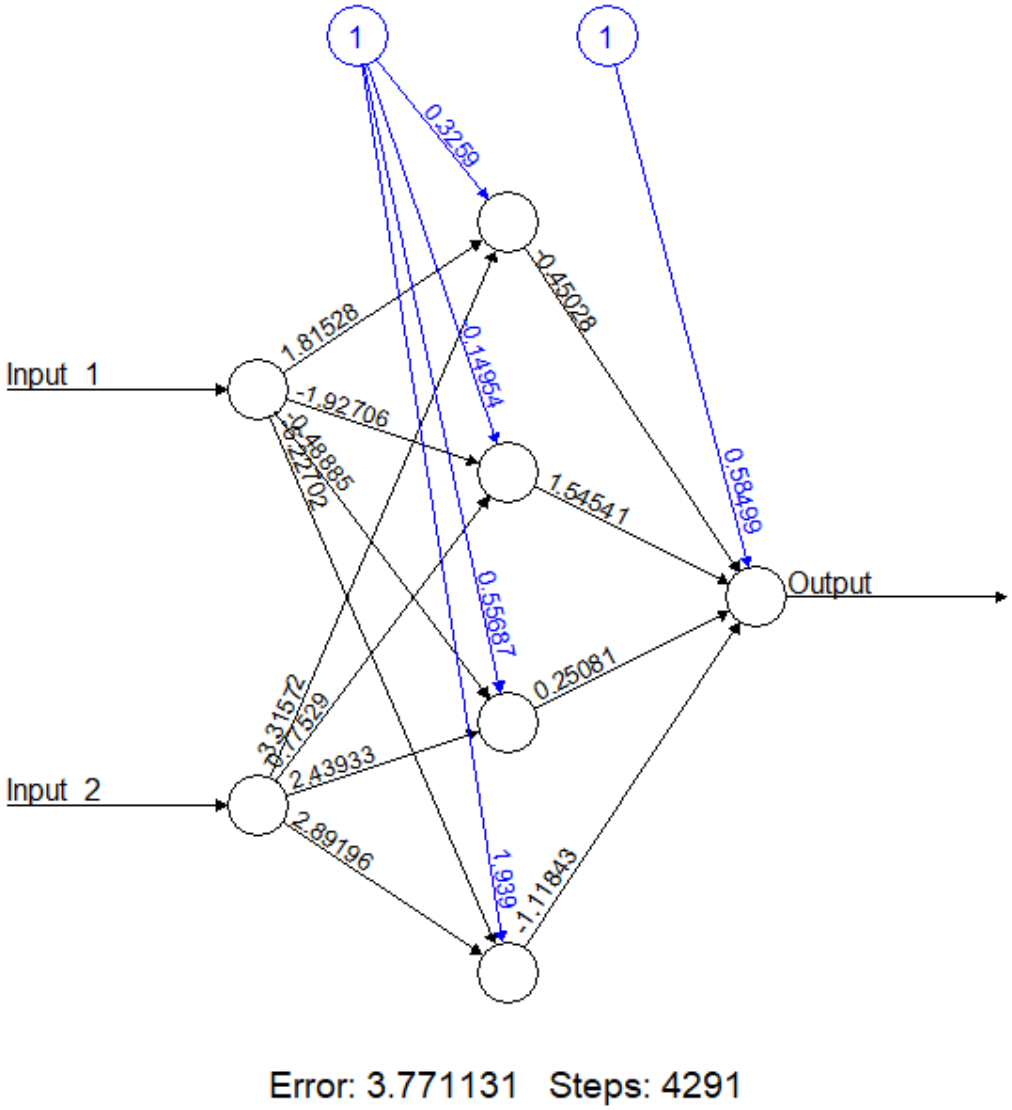


*Figure 30: T_2_NN1 plot*

```
T_3_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data =
train_T_3, hidden = 4, linear.output = TRUE)
plot(T_3_NN1)
```

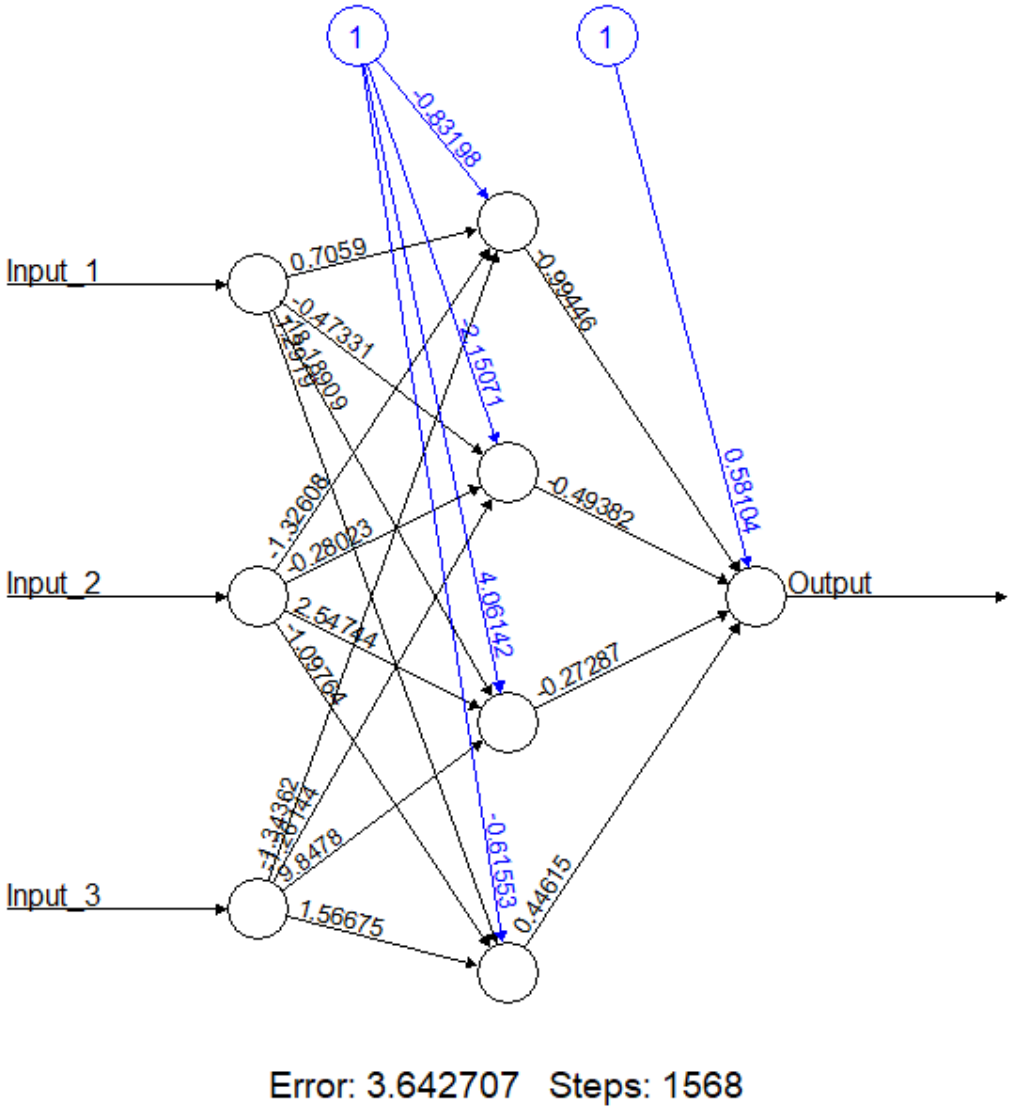


*Figure 31: T_3_NN1 plot*

```
T_4_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4,
data = train_T_4, hidden = 4, linear.output = TRUE)
plot(T_4_NN1)
```

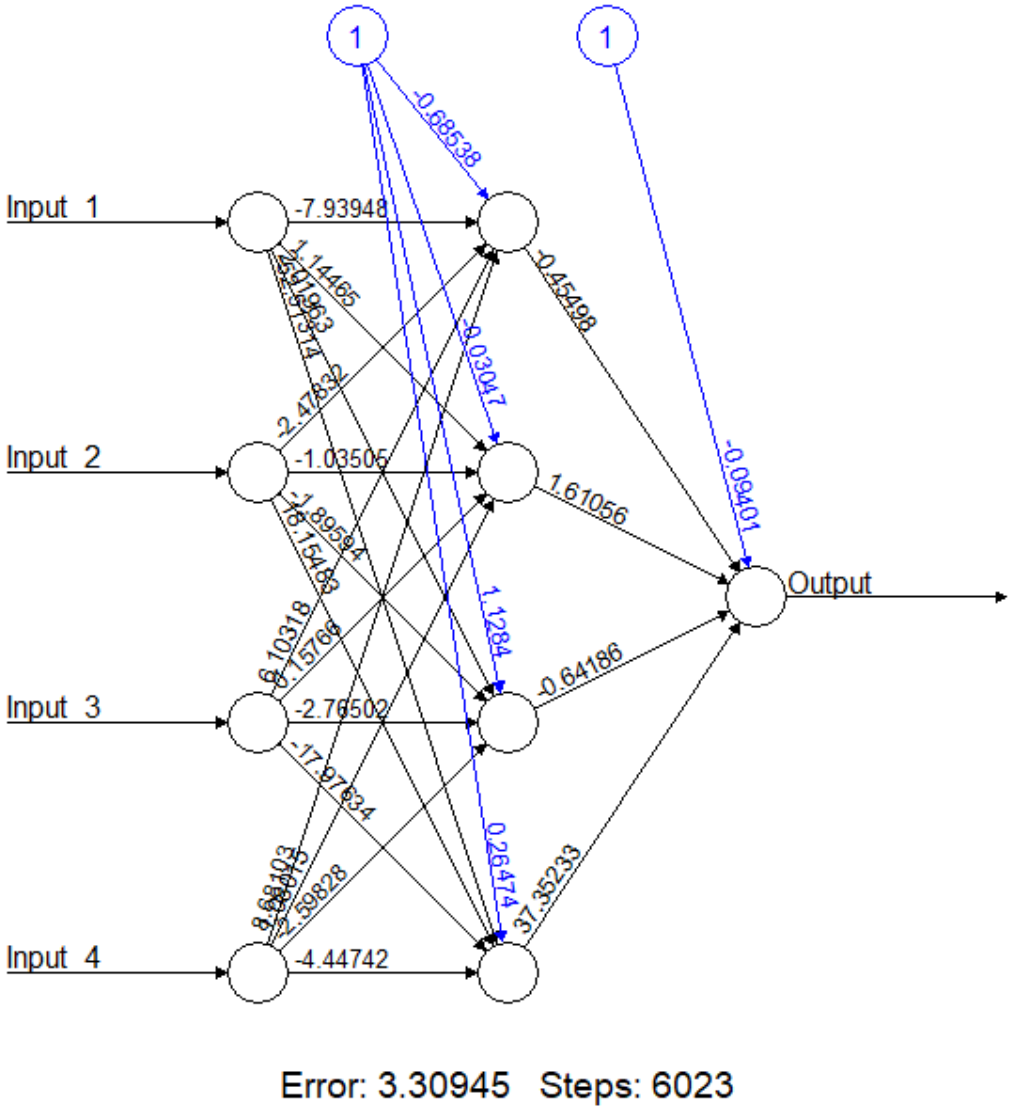


*Figure 32: T_4_NN1 plot*

```
T_5_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 +
Input_5, data = train_T_5, hidden = 4, linear.output = TRUE)
plot(T_5_NN1)
```

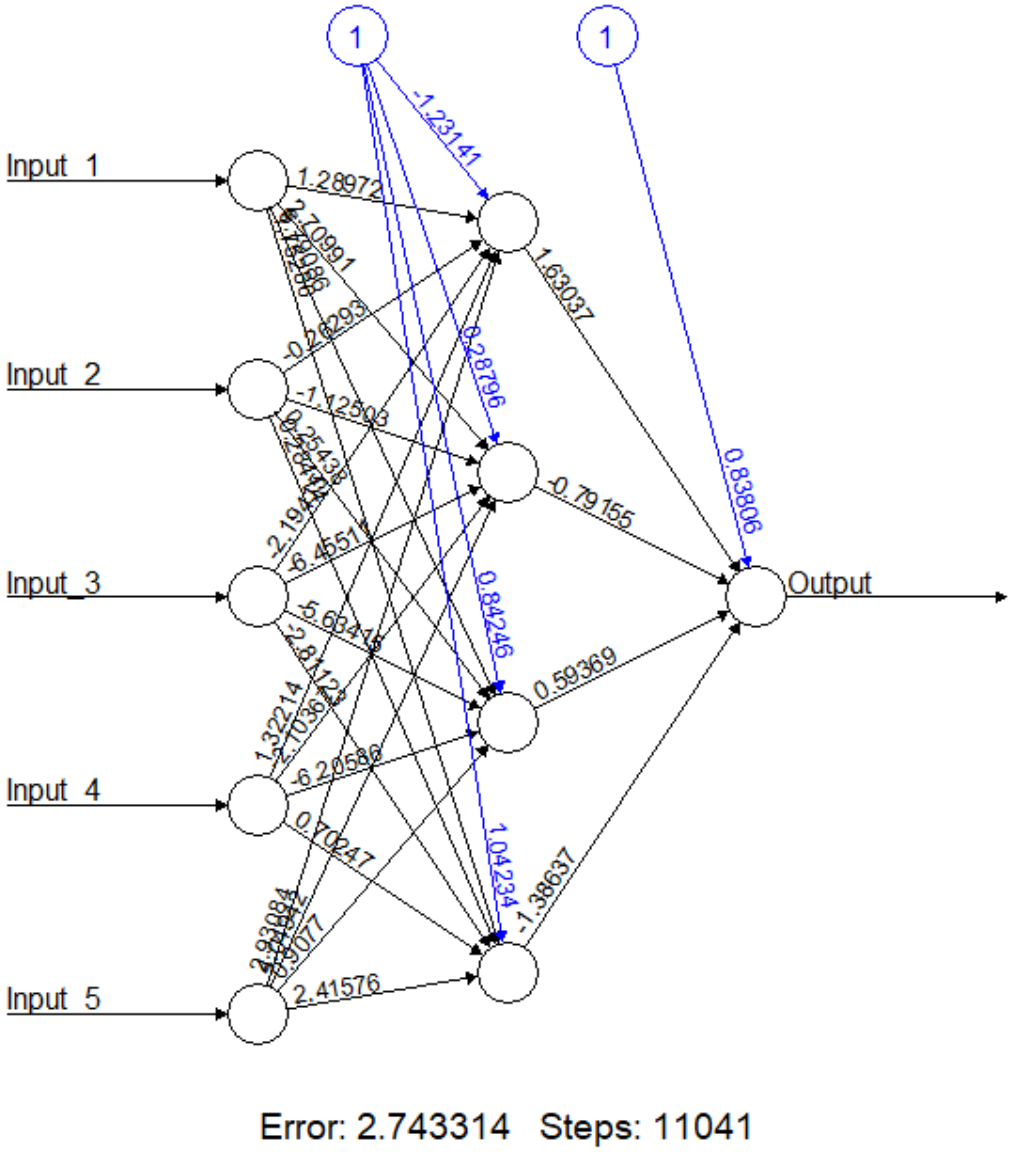


*Figure 33: T_5_NN1 plot*

The neuralnet function is used to build a neural network model for each I/O matrix (T_1, T_2, T_3, T_4, T_5). The train_T_X dataset, where X stands for the matching I/O matrix number, is used to train the model. The neural network comprises one hidden layer and five neurons, as shown by the hidden parameter, which is set to 4.

The network's output is linear because the linear.output option is set to TRUE.

It is possible to see the trained neural network model using the plot function.

Two neural networks with hidden layers:

```
T_1_NN2 <- neuralnet(Output ~ Input, data = train_T_1, hidden =
c(4,4), linear.output = TRUE)
plot(T_1_NN2)
```

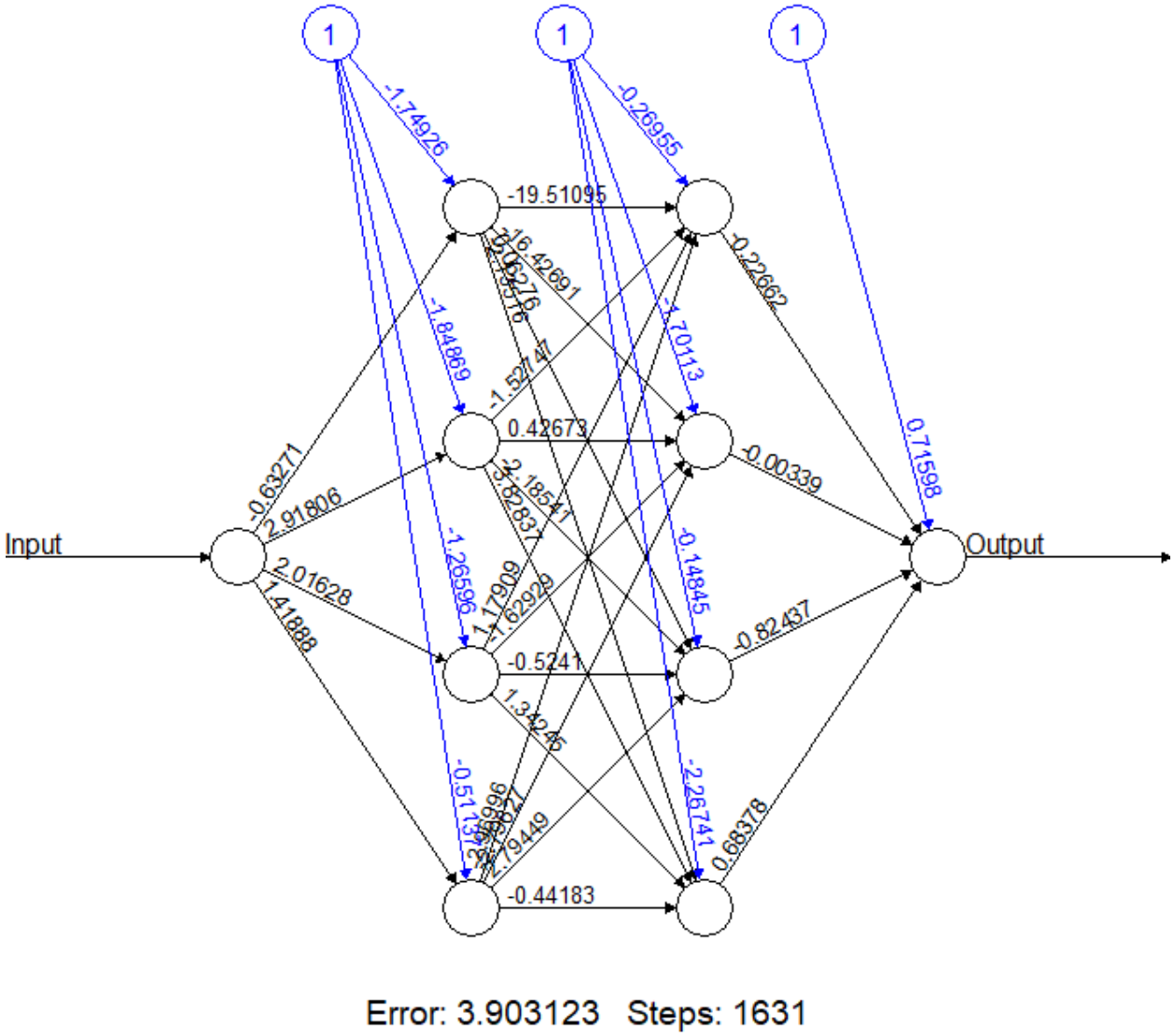


*Figure 34: T_1_NN2 plot*

```
T_2_NN2 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2,
hidden =c (4,4), linear.output=TRUE)
plot(T_2_NN2)
```

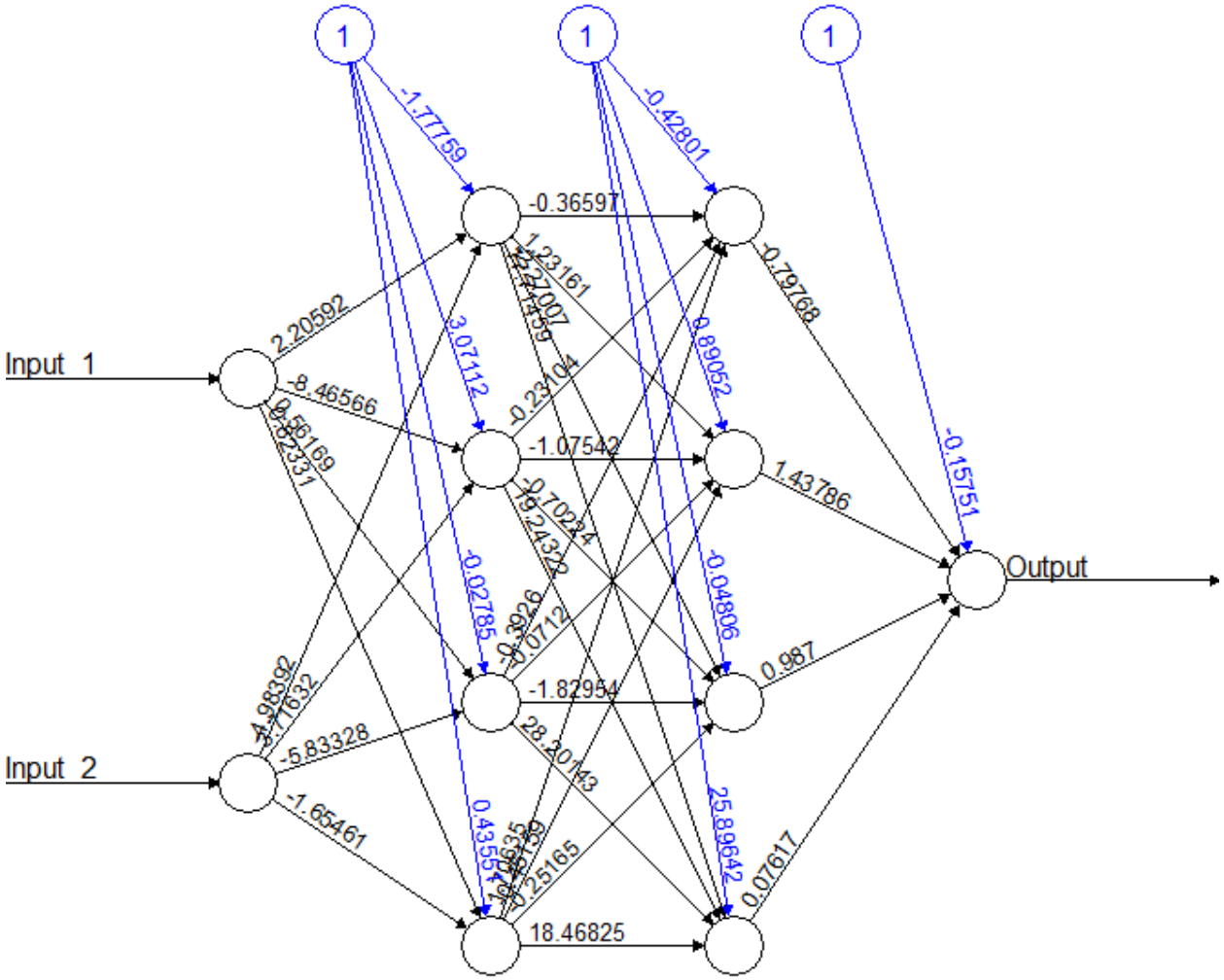


*Figure 35: T_2_NN2 plot*

```
T_3_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data =
train_T_3, hidden = c(4,4), linear.output = TRUE)
plot(T_3_NN2)
```

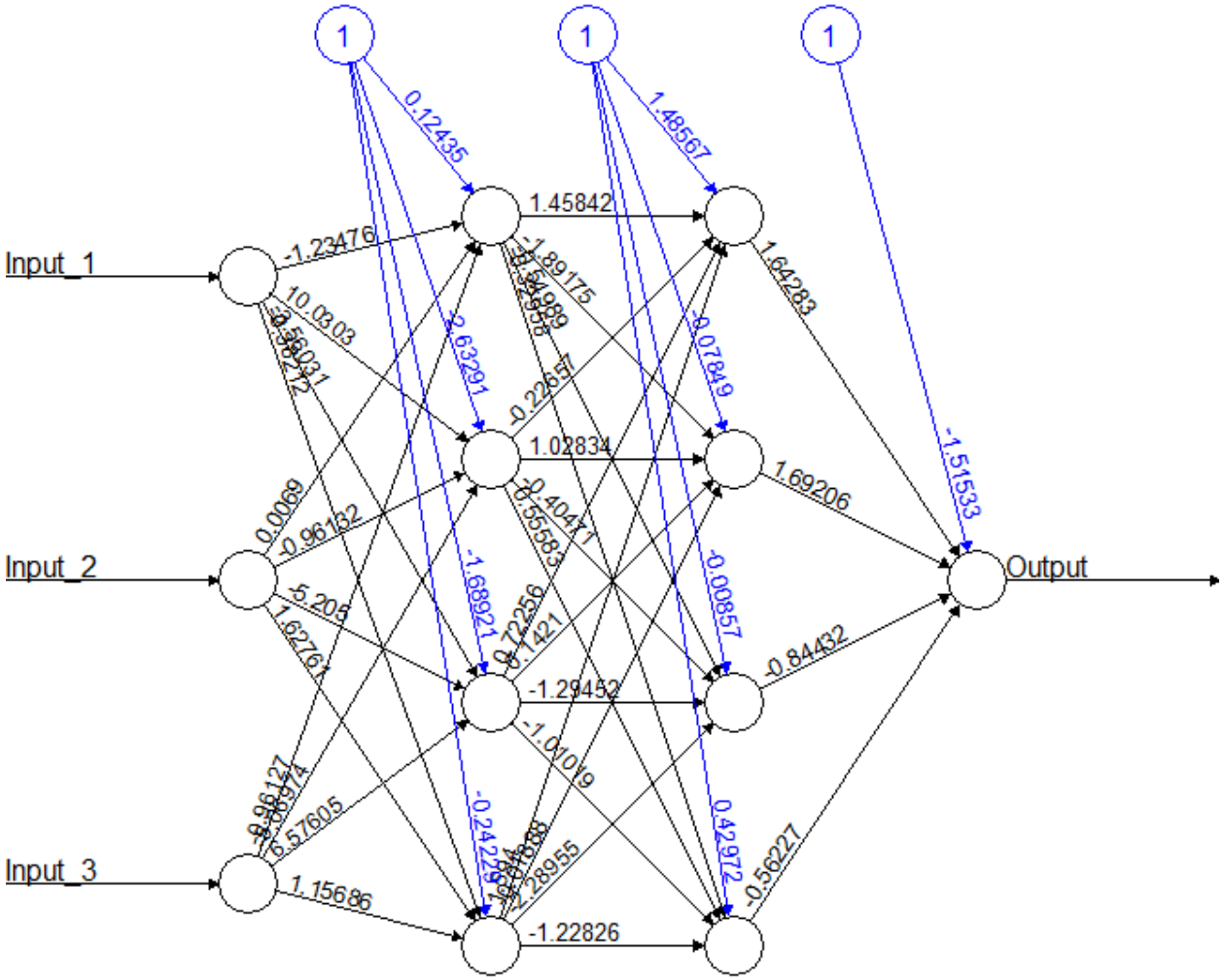


*Figure 36: T_3_NN2 plot*

```
T_4_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4,
data = train_T_4, hidden = c(4,4), linear.output = TRUE)
plot(T_4_NN2)
```

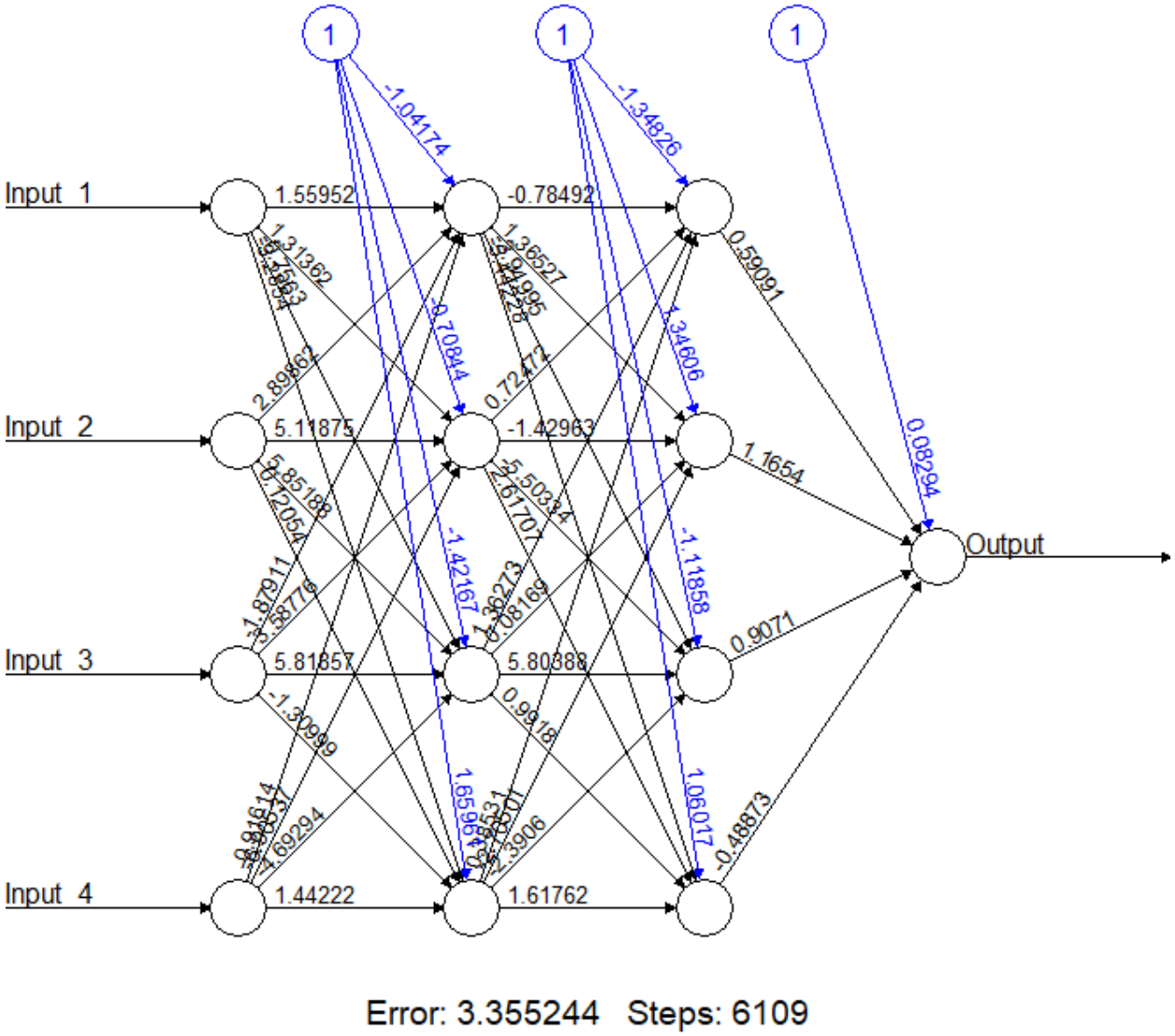


*Figure 37: T_4_NN2 plot*

```
T_5_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 +
Input_5, data = train_T_5, hidden = c(4,4), linear.output = TRUE)
plot(T_5_NN2)
```

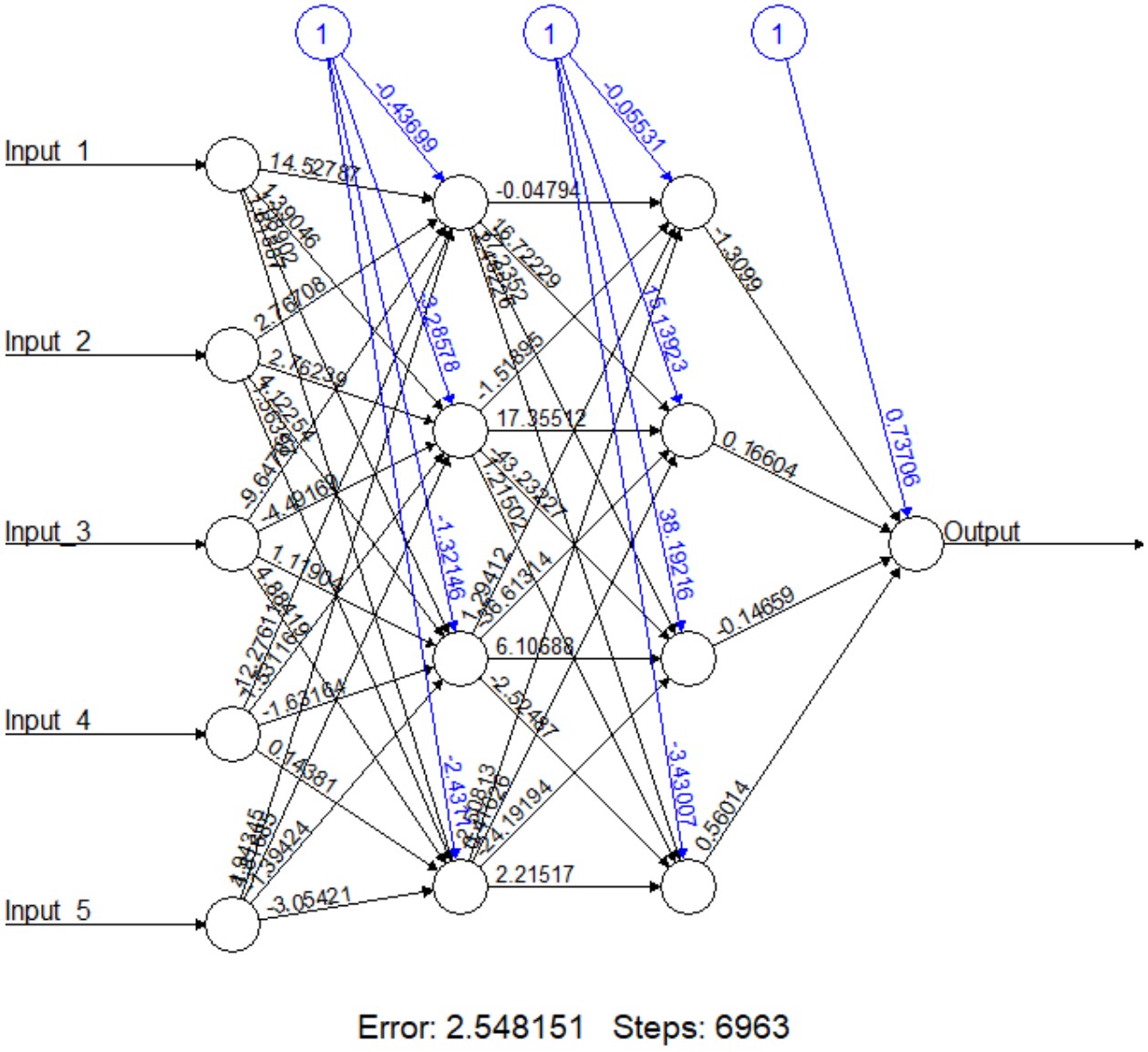


*Figure 38: T_5_NN2 plot*

Neural network models are built and trained for each I/O matrix, similar to the previous part. The hidden parameter is different since it is set to c(4,4), which denotes that the neural network has two hidden layers and a total of five neurons in each.

Version 2 of one hidden layer neural network:

```
T_1_NN3 <- neuralnet(Output ~ Input, data = train_T_1, hidden = 7,
linear.output = TRUE)
plot(T_1_NN3)
```

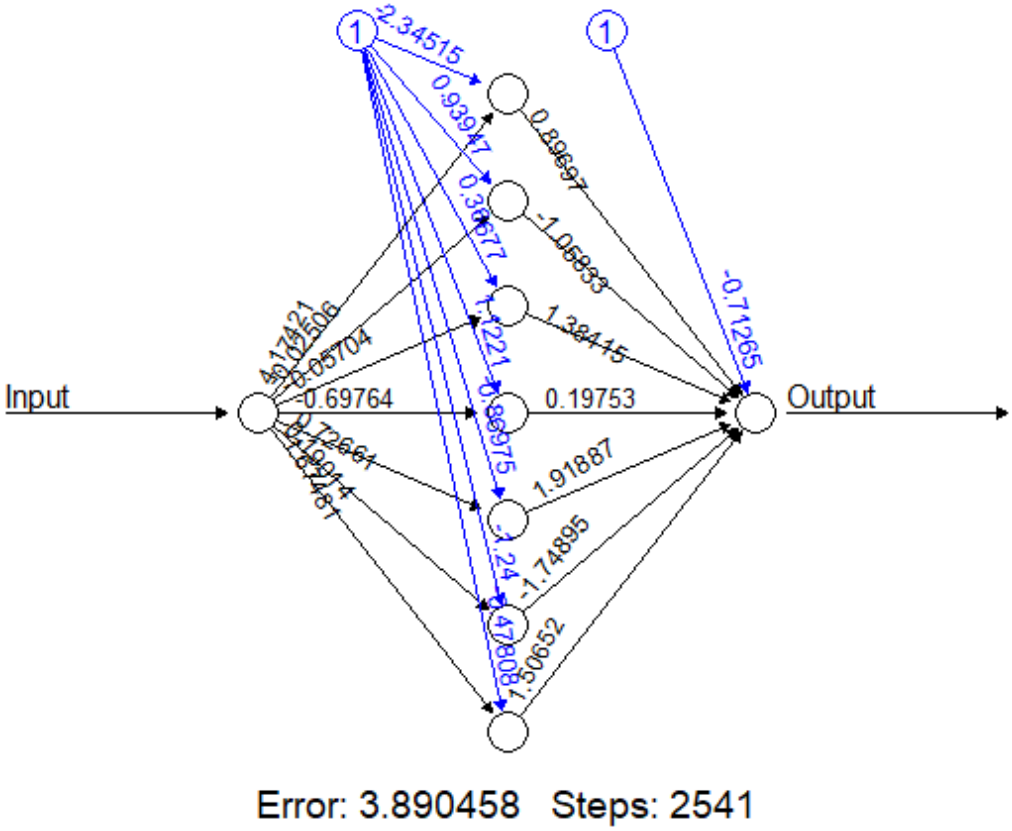


*Figure 39: T_1_NN3 plot*

```
T_2_NN3 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2,
hidden = 7, linear.output = TRUE)
plot(T_2_NN3)
```

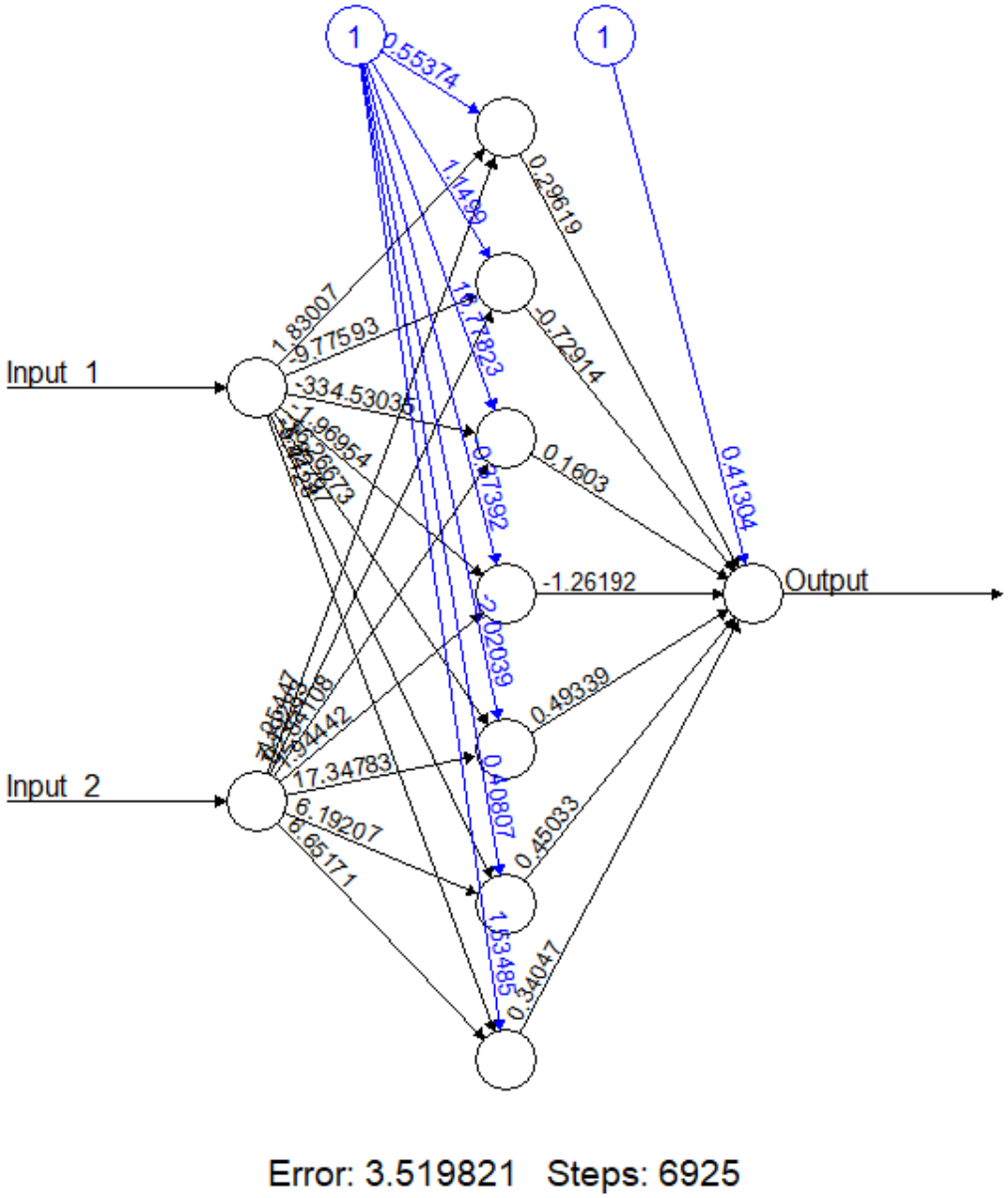


*Figure 40: T_2_NN3 plot*

```
T_3_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data =
train_T_3, hidden = 7, linear.output = TRUE)
plot(T_3_NN3)
```

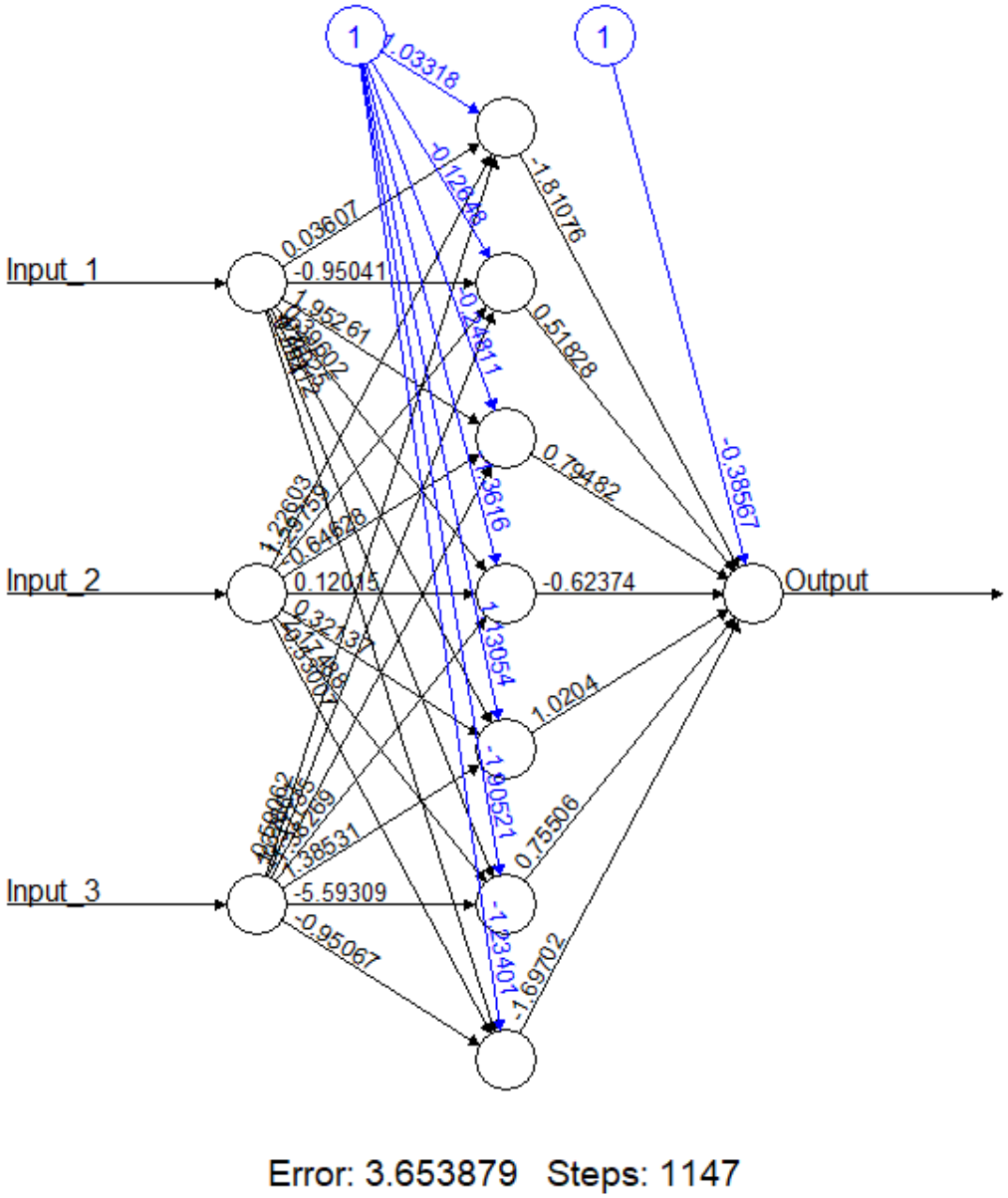


*Figure 41: T_3_NN3 plot*

```
T_4_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4,
data = train_T_4, hidden = 7, linear.output = TRUE)
plot(T_4_NN3)
```

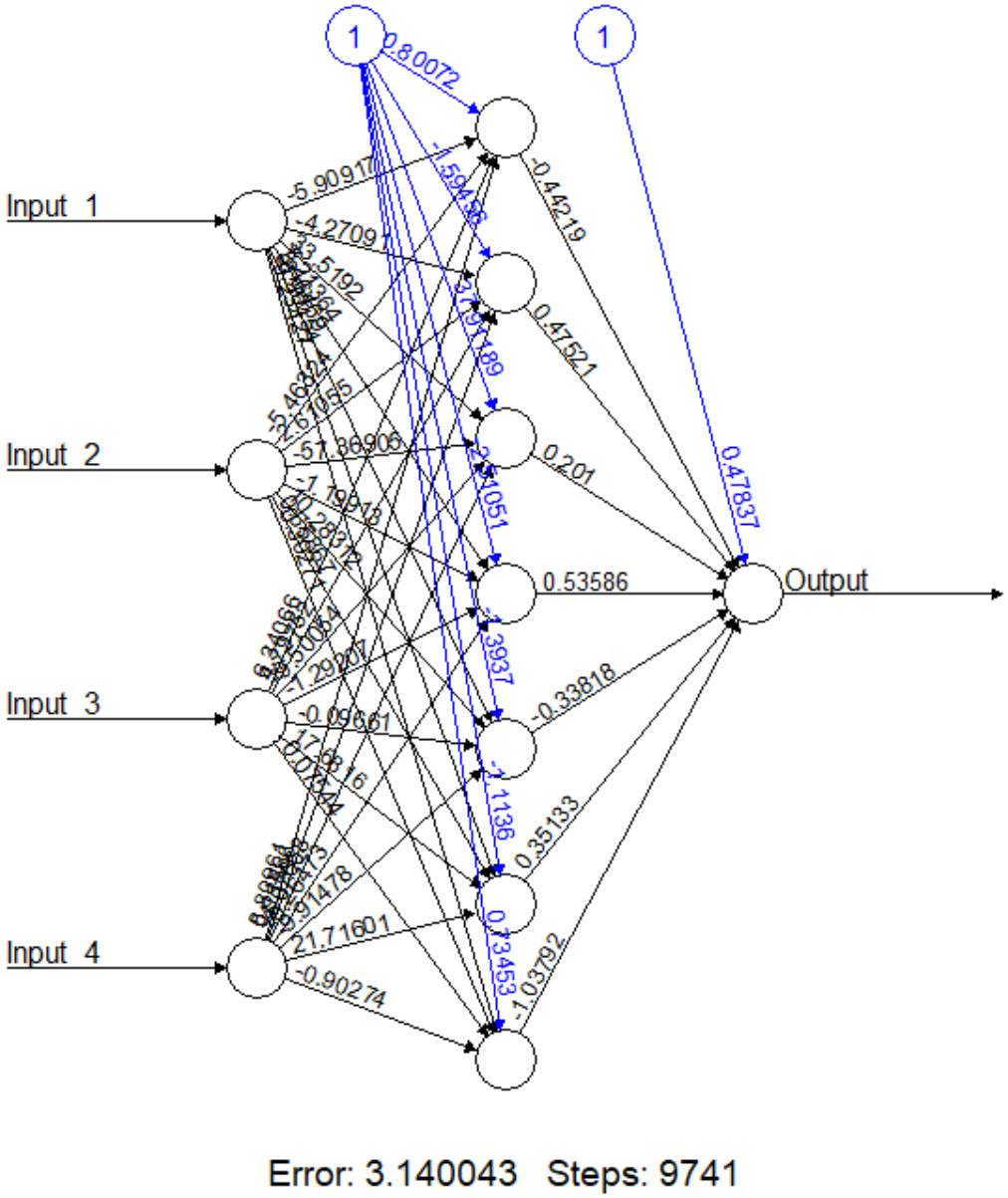


*Figure 42: T_4_NN3 plot*

```
T_5_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 +
Input_5, data = train_T_5, hidden = 7, linear.output = TRUE)
plot(T_5_NN3)
```

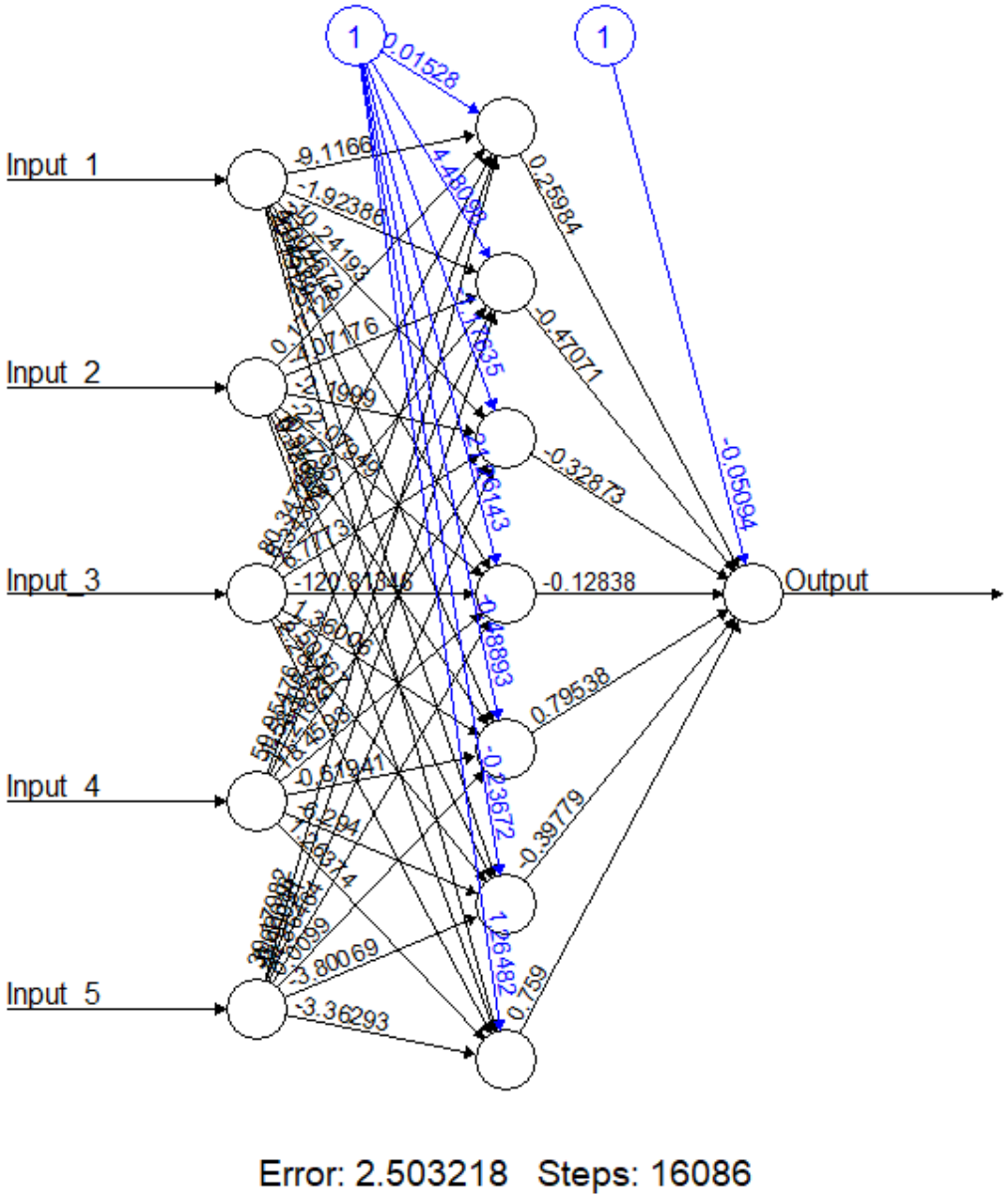


*Figure 43: T_5_NN3 plot*

For each I/O matrix, new neural network models are built and trained.
The neural network comprises one hidden layer and seven neurons, as shown by the hidden parameter, which is set to 7.

The programme investigates several architectures to find the best model for the given dataset by training different variants of MLPs with various numbers of hidden layers and neurons.
The network topology may be seen and the learned weights and connections can be understood using the plots produced for each trained model.

The next step is to extract the actual output values for each I/O matrix from the testing data. To compare the expected results from the neural network models with the actual values, this step is important.

```
T_1_act_output <- test_T_1[, "Output"]
T_2_act_output <- test_T_2[, "Output"]
T_3_act_output <- test_T_3[, "Output"]
T_4_act_output <- test_T_4[, "Output"]
T_5_act_output <- test_T_5[, "Output"]
```

The anticipated outputs from each neural network model for the corresponding testing data sets are computed in the next block of code. The trained neural network models are used together with the **predict** function to produce predictions.

```
T_1_pred_output1 <- predict(object = T_1_NN1, test_T_1)
T_1_pred_output2 <- predict(object = T_1_NN2, test_T_1)
T_1_pred_output3 <- predict(object = T_1_NN3, test_T_1)

T_2_pred_output1 <- predict(object = T_2_NN1, test_T_2)
T_2_pred_output2 <- predict(object = T_2_NN2, test_T_2)
T_2_pred_output3 <- predict(object = T_2_NN3, test_T_2)

T_3_pred_output1 <- predict(object = T_3_NN1, test_T_3)
T_3_pred_output2 <- predict(object = T_3_NN2, test_T_3)
T_3_pred_output3 <- predict(object = T_3_NN3, test_T_3)

T_4_pred_output1 <- predict(object = T_4_NN1, test_T_4)
T_4_pred_output2 <- predict(object = T_4_NN2, test_T_4)
T_4_pred_output3 <- predict(object = T_4_NN3, test_T_4)

T_5_pred_output1 <- predict(object = T_5_NN1, test_T_5)
T_5_pred_output2 <- predict(object = T_5_NN2, test_T_5)
T_5_pred_output3 <- predict(object = T_5_NN3, test_T_5)
```

In order to calculate the RMSE, MAPE, MAE & SMAPE, First the actual outputs and the predicted outputs must be denormalized.

```
T_1_pred_output1_unnorm <- unnormalize(T_1_pred_output1,
min(train_T_1[, "Output"]), max(train_T_1[, "Output"]))
T_1_pred_output2_unnorm <- unnormalize(T_1_pred_output2,
min(train_T_1[, "Output"]), max(train_T_1[, "Output"]))
T_1_pred_output3_unnorm <- unnormalize(T_1_pred_output3,
min(train_T_1[, "Output"]), max(train_T_1[, "Output"]))
```

```
T_1_act_output_unnorm <- unnormalize(T_1_act_output, min(train_T_1[,
"Output"]), max(train_T_1[, "Output"]))

T_2_pred_output1_unnorm <- unnormalize(T_2_pred_output1,
min(train_T_2[, "Output"]), max(train_T_2[, "Output"]))
T_2_pred_output2_unnorm <- unnormalize(T_2_pred_output2,
min(train_T_2[, "Output"]), max(train_T_2[, "Output"]))
T_2_pred_output3_unnorm <- unnormalize(T_2_pred_output3,
min(train_T_2[, "Output"]), max(train_T_2[, "Output"]))

T_2_act_output_unnorm <- unnormalize(T_2_act_output, min(train_T_2[,
"Output"]), max(train_T_2[, "Output"]))

T_3_pred_output1_unnorm <- unnormalize(T_3_pred_output1,
min(train_T_3[, "Output"]), max(train_T_3[, "Output"]))
T_3_pred_output2_unnorm <- unnormalize(T_3_pred_output2,
min(train_T_3[, "Output"]), max(train_T_3[, "Output"]))
T_3_pred_output3_unnorm <- unnormalize(T_3_pred_output3,
min(train_T_3[, "Output"]), max(train_T_3[, "Output"]))

T_3_act_output_unnorm <- unnormalize(T_3_act_output, min(train_T_3[,
"Output"]), max(train_T_3[, "Output"]))

T_4_pred_output1_unnorm <- unnormalize(T_4_pred_output1,
min(train_T_4[, "Output"]), max(train_T_4[, "Output"]))
T_4_pred_output2_unnorm <- unnormalize(T_4_pred_output2,
min(train_T_4[, "Output"]), max(train_T_4[, "Output"]))
T_4_pred_output3_unnorm <- unnormalize(T_4_pred_output3,
min(train_T_4[, "Output"]), max(train_T_4[, "Output"]))

T_4_act_output_unnorm <- unnormalize(T_4_act_output, min(train_T_4[,
"Output"]), max(train_T_4[, "Output"]))

T_5_pred_output1_unnorm <- unnormalize(T_5_pred_output1,
min(train_T_5[, "Output"]), max(train_T_5[, "Output"]))
T_5_pred_output2_unnorm <- unnormalize(T_5_pred_output2,
min(train_T_5[, "Output"]), max(train_T_5[, "Output"]))
T_5_pred_output3_unnorm <- unnormalize(T_5_pred_output3,
min(train_T_5[, "Output"]), max(train_T_5[, "Output"]))

T_5_act_output_unnorm <- unnormalize(T_5_act_output, min(train_T_5[,
"Output"]), max(train_T_5[, "Output"]))
```

After denormalization a function is introduced to find out the performance metrics.

```
perform_metrics <- function(act_output, pred_output){
  return(list(RMSE = rmse(act_output, pred_output),
              MAE = mae(act_output, pred_output),
              MAPE = mape(act_output, pred_output),
              SMAPE = smape(act_output, pred_output)))
}
```

```
> T_1_NN1_performance
$RMSE
[1] 0.1143494

$MAE
[1] 0.09200909

$MAPE
[1] 0.1819552

$SMAPE
[1] 0.1790814

> T_1_NN2_performance
$RMSE
[1] 0.1156922

$MAE
[1] 0.09289811

$MAPE
[1] 0.1839568

$SMAPE
[1] 0.1813157

> T_1_NN3_performance
$RMSE
[1] 0.1154984

$MAE
[1] 0.0925191

$MAPE
[1] 0.1829175

$SMAPE
[1] 0.1806113
```

*Figure 44: Performance metrics*

## d)

RMSE

The statistical measurement known as RMSE (Root Mean Square Error) assesses the average size of differences between expected and observed values. The difference between the predicted and actual numbers is squared, and this difference is used to produce the indicator. Lower values indicate greater fit, and the RMSE provides a measurement of how well the model's predictions match the observed data.

MAE

The statistical measurement called MAE (Mean Absolute Error) tracks the average size of differences between expected and observed data. It is derived by averaging the absolute disparities between the values that were anticipated and those that actually occurred. Without taking into account the direction of the mistakes, MAE gives a measure of the typical forecast error magnitude.

MAPE

MAPE (Mean Absolute Percentage Error) calculates the percentage-based relative prediction accuracy. It is derived by averaging the absolute percentage differences between the values that were anticipated and those that actually occurred. The average percentage prediction error is measured by MAPE, which is helpful for evaluating performance across various datasets and sizes. It can, however, be sensitive to actual values that are 0 or nearly zero, leading to undefined or endless values.

SMAPE

A symmetric version of MAPE that solves some of its drawbacks is called sMAPE (Symmetric Mean Absolute Percentage Error). By averaging the predicted and actual values in the denominator, it is determined by averaging the absolute percentage discrepancies between the predicted and actual values. The average percentage prediction error is measured by sMAPE, which symmetrically takes into account overestimations and underestimations. It is especially helpful when both are present.

e)

| | Description | RMSE | MAE | MAPE | SMAPE |
|---|---|---|---|---|---|
| T_1_NN1_performance | 1 input<br>1 hidden layer<br>4 nodes | 0.1143494 | 0.09200909 | 0.1819552 | 0.1790814 |
| T_1_NN2_performance | 1 input<br>2 hidden layers<br>4 nodes each | 0.1156922 | 0.09289811 | 0.1839568 | 0.1813157 |
| T_1_NN3_performance | 1 input<br>1 hidden layer<br>7 nodes | 0.1154984 | 0.0925191 | 0.1829175 | 0.1806113 |
| T_2_NN1_performance | 2 inputs<br>1 hidden layer<br>4 nodes | 0.1164267 | 0.09337747 | 0.1847721 | 0.1817711 |
| T_2_NN2_performance | 2 inputs<br>2 hidden layers<br>4 nodes each | 0.1188537 | 0.09514281 | 0.1886646 | 0.1862348 |
| T_2_NN3_performance | 2 inputs<br>1 hidden layer<br>7 nodes | 0.1155624 | 0.09310833 | 0.1849944 | 0.1813996 |
| T_3_NN1_performance | 3 inputs<br>1 hidden layer<br>4 nodes | 0.1148035 | 0.09158253 | 0.1791321 | 0.1784869 |
| T_3_NN2_performance | 3 inputs<br>2 hidden layers<br>4 nodes each | 0.1141183 | 0.09032088 | 0.1764453 | 0.1760485 |
| T_3_NN3_performance | 3 inputs<br>1 hidden layer<br>7 nodes | 0.1148824 | 0.09146325 | 0.1801737 | 0.1777883 |
| T_4_NN1_performance | 4 inputs<br>1 hidden layer<br>4 nodes | 0.1181858 | 0.09536171 | 0.1876931 | 0.1850859 |
| T_4_NN2_performance | 4 inputs<br>2 hidden layers<br>4 nodes each | 0.1189114 | 0.09688359 | 0.1907169 | 0.1878767 |

| | | | | | |
|---|---|---|---|---|---|
| T_4_NN3_performance | 4 inputs<br>1 hidden layer<br>7 nodes | 0.1127517 | 0.0899943 | 0.1768538 | 0.1743513 |
| T_5_NN1_performance | 5 inputs<br>1 hidden layer<br>4 nodes | 0.09491112 | 0.07569171 | 0.1508038 | 0.1442668 |
| T_5_NN2_performance | 5 inputs<br>2 hidden layers<br>4 nodes each | 0.09295335 | 0.07353304 | 0.1454874 | 0.140523 |
| T_5_NN3_performance | 5 inputs<br>1 hidden layer<br>7 nodes | 0.09197394 | 0.07305019 | 0.1427465 | 0.1386944 |

The best one layer neural network with least errors is T_5_NN3 with 5 inputs and 7 nodes.
The best two-layer neural network with the least errors is T_5_NN2 with 5 inputs and 4 nodes each.

f)

As mentioned in the previous question the best one-layer neural network with the least errors is **T_5_NN3** with 5 inputs and 7 nodes. The best two-layer neural network with the least errors is **T_5_NN2** with 5 inputs and 4 nodes each.

```
> # Calculate total number of weight parameters for T_5_NN3
> T_5_NN3_weights <- sum(sapply(T_5_NN3$weights, function(x) length(x)))
> T_5_NN3_weights
[1] 2
>
> # Calculate total number of weight parameters for T_5_NN2
> T_5_NN2_weights <- sum(sapply(T_5_NN2$weights, function(x) length(x)))
> T_5_NN2_weights
[1] 3
```

*Figure 45: Total weights of the selected NNs.*

The neural network models **T_5_NN3** and **T_5_NN2's** weights attribute are iterated over in this code using the **sapply()** function. For each layer, it calculates the length of the weight vector and returns a vector of lengths. The **sum()** function is then used to add together all the lengths to determine the total number of weight parameters.

The total number of weight parameters in a neural network can provide insights into its complexity and potential efficiency. Although it is not the only component to take into account, the quantity of weight parameters can have a significant impact on a neural network's effectiveness. The number of weight parameters reflects the complexity of the neural network model. Generally, a larger number of weight parameters can indicate a more complex model that is capable of capturing complex patterns and relationships in the data. Here the more complex neural network is **T_5_NN2**.
The number of weight parameters affects the computational requirements during both training and inference. More computational resources are typically needed for models with more parameters. Although using more parameters may improve performance on training data, this does not guarantee that it will also improve performance on untrained data. By avoiding overfitting and lowering the danger of large variance, a simpler model with fewer parameters may occasionally generalize better and demonstrate superior efficiency.

# Appendix

## Partitioning clustering

```
# 1st subtask.....

#a.....

# Install and load necessary packages
install.packages(c("readxl", "stats", "base", "dplyr"))
library(readxl)
library(stats)
library(base)
library(dplyr)

# Read in the data
vehicles_df <- read_excel("vehicles.xlsx")
View(head(vehicles_df))

# Remove Samples Column
vehicles_df <- vehicles_df[, -which(names(vehicles_df) == "Samples")]
View(head(vehicles_df))

# Select only numeric columns
numeric_cols <- sapply(vehicles_df, is.numeric)
vehicles_df_numeric <- vehicles_df[, numeric_cols]

# Identify and remove outliers using z-score method
z_scores <- as.matrix(scale(vehicles_df[, numeric_cols]))
outliers <- apply(z_scores, 1, function(x) any(abs(x) > 3))
clean_df <- vehicles_df[!outliers,]
View(head(clean_df))

# scale the numeric columns
num_cols <- sapply(clean_df, is.numeric)
clean_df[, num_cols] <- scale(clean_df[, num_cols])
View (head(clean_df[, num_cols]))

# Remove the column class
clean_df <- clean_df[, -which(names(clean_df) == "Class")]
```

```
View(head(clean_df))

#b.....

install.packages("NbClust")
library(NbClust)

# Determine the number of clusters using NbClust
set.seed(123)
nb_clusters <- NbClust(clean_df, distance = "euclidean", min.nc=2, max.nc=10, method="kmeans")
table(nb_clusters$Best.nc[1,])

library(tidyverse)
library(cluster)
library(ggplot2)

# Elbow method to determine the optimal number of clusters
set.seed(123)
wss <- sapply(1:10,
          function(k) {
            kmeans(clean_df, k, nstart = 10, iter.max = 50)$tot.withinss
          })

# Plot the elbow curve
elbow_plot <- tibble(k = 1:10, wss = wss) %>%
 ggplot(aes(x = k, y = wss)) +
 geom_line() +
 geom_point() +
 scale_x_continuous(breaks = 1:10) +
 labs(x = "Number of Clusters (k)", y = "Within-Cluster Sum of Squares (WSS)") +
 theme_minimal()

# Display the plot
elbow_plot

install.packages("factoextra")
library(factoextra)

# Gap statistic to determine the optimal number of clusters
set.seed(123)
gap_stat <- clusGap(clean_df, FUN = kmeans, nstart = 25, K.max = 10, B = 50)
```

```
# Plot the gap statistic
fviz_gap_stat(gap_stat) + labs(title = "Gap Statistic Plot")

# Display the results
gap_stat

# Determine the optimal number of cluster centers using silhouette method
set.seed(123)  # for reproducibility
silhouettemethod <- fviz_nbclust(clean_df, FUNcluster = kmeans, method = "silhouette", nstart = 25,
k.max = 10, verbose = FALSE)

# Plot the silhouette plot
plot(silhouettemethod)


#c.....

set.seed(123)
kmeans_model <- kmeans(clean_df, centers = 3, nstart = 25, iter.max = 50)

# Print the cluster assignments
kmeans_model$cluster


# Set the seed for reproducibility
set.seed(123)

# Calculate the within-cluster sum of squares (WSS)
wss <- sum(kmeans_model$withinss)
wss

# Calculate the between-cluster sum of squares (BSS)
bss <- sum(kmeans_model$betweenss)
bss

# Print the ratio of between_cluster_sums_of_squares (BSS) over total_sum_of_Squares (TSS)
cat("\n\nBSS/TSS ratio:\n")
cat(kmeans_model$betweenss / kmeans_model$totss)

library(factoextra)
# Visualize the clusters using the first two principal components
```

```
fviz_cluster(kmeans_model, geom = "point", data = clean_df, stand = FALSE, palette = "jco", ggtheme =
theme_minimal(), main = "Kmeans Clustering Results")

#d.....

library(cluster)

# Calculate silhouette widths
silwidths <- silhouette(kmeans_model$cluster, dist(clean_df))
head(silwidths)

# Plot the silhouette widths
plot(silwidths, main = "Silhouette Plot for Kmeans Clustering", border = NA, col = 1:3)

# Average silhouette width
mean(silwidths[,3])



#2nd subtask.....

# e.....

# Performing PCA on clean_df
PCAVehicles <- prcomp(clean_df, center = TRUE, scale = FALSE)
eigenvalue <- PCAVehicles$sdev^2
eigenvalue
eigenvector <- PCAVehicles$rotation
head(eigenvector)

# Identify the PCAs which satisfies the threshold
cum_score <- cumsum(PCAVehicles$sdev^2 / sum(PCAVehicles$sdev^2))
print(cum_score)

# number of PCAs that reaches the cumulative score of 92
num_PCAs <- which(cum_score >= 0.92)[1]
num_PCAs

# Create the newdataset to store the PCAs
transformed <- as.data.frame(PCAVehicles$x[, 1:num_PCAs])
view(head(transformed))
```

```
#e.....

# NbCLust method
set.seed(123)
PCA_nbClust <- NbClust(transformed, distance = "euclidean", min.nc = 2, max.nc = 15, method =
"kmeans", index = "all")

# Elbow method
fviz_nbclust(transformed, kmeans, method= "wss") + labs(subtitle = "Elbow method")

# Gap statistics method
fviz_nbclust(transformed, kmeans, nstart = 25,  method = "gap_stat", nboot = 50)+labs(subtitle = "Gap
statistic method")

# Silhoutte method
fviz_nbclust(transformed, kmeans, method = "silhouette") + labs(subtitle = "Silhouette method")

#f.....

# kmeans analysis
set.seed(123)
PCA_kmeans_model <- kmeans(transformed, centers = 3, nstart = 25, iter.max = 50)
PCA_kmeans_model$cluster

# Set the seed for reproducibility
set.seed(123)
# Calculate the within-cluster sum of squares (WSS)
PCA_wss <- sum(PCA_kmeans_model$withinss)
PCA_wss

# Calculate the between-cluster sum of squares (BSS)
PCA_bss <- sum(PCA_kmeans_model$betweenss)
PCA_bss

# Print the ratio of between_cluster_sums_of_squares (BSS) over total_sum_of_Squares (TSS)
cat("\n\nBSS/TSS ratio:\n")
cat(PCA_kmeans_model$betweenss / PCA_kmeans_model$totss)


#g.....

# Calculate silhouette widths
```

```
PCA_silwidths <- silhouette(PCA_kmeans_model$cluster, dist(transformed))
head(PCA_silwidths)

# Plot the silhouette widths
plot(PCA_silwidths, main = "Silhouette Plot for Kmeans Clustering", border = NA, col = 1:3)


#h.....

kmeans_func <- function(x, k) {
  kmeans(x, centers = k, nstart = 25)
}

# Visualization of Calinski-Harabasz Index
calinski_harabasz <- clusGap(transformed, kmeans_func, K.max = length(PCA_kmeans_model$size))
fviz_gap_stat(calinski_harabasz) + labs(subtitle = "Calinski-Harabasz Index")
```

# Energy Forecasting

```
# 2nd Subtask Objectives.....

library(neuralnet)
library(dplyr)
library(readxl)
library(MLmetrics)
library(Metrics)


# b.....


# Load the data set
uow_consumption <- read_excel("uow_consumption.xlsx")
View(head(uow_consumption))

# Replace the column names with new relevant column names
colnames(uow_consumption) <- c("Date", "18", "19", "20")
View(head(uow_consumption))

# Create time delayed input variables
time_delayed <- bind_cols(
  T7 = lag(uow_consumption$'20', 7),
  T4 = lag(uow_consumption$'20', 4),
  T3 = lag(uow_consumption$'20', 3),
  T2 = lag(uow_consumption$'20', 2),
  T1 = lag(uow_consumption$'20', 1),
  outputprediction = lag(uow_consumption$'20', 0)
)

# Remove rows with missing values
time_delayed <- na.omit(time_delayed)

# Construction of different time-delayed input vectors and related i/o matrices.
T_1 <- cbind(time_delayed$T1, time_delayed$outputprediction)
colnames(T_1)<- c("Input", "Output")

T_2 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$outputprediction)
colnames(T_2)<- c("Input_1", "Input_2", "Output")
```

```
T_3 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3, time_delayed$outputprediction)
colnames(T_3)<- c("Input_1", "Input_2", "Input_3", "Output")

T_4 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3, time_delayed$T4,
time_delayed$outputprediction)
colnames(T_4)<- c("Input_1", "Input_2", "Input_3", "Input_4", "Output")

T_5 <- cbind(time_delayed$T1, time_delayed$T2, time_delayed$T3, time_delayed$T4,time_delayed$T7,
time_delayed$outputprediction)
colnames(T_5)<- c("Input_1", "Input_2", "Input_3", "Input_4", "Input_5", "Output")


# c.....

#Defining the normalize function (Min-Max Normalization)
norm <- function(x) {
  return ((x - min(x)) / (max(x)-min(x)))
}

#Normalizing the I/O Matrices
normT_1 <- norm(T_1)
normT_2 <- norm(T_2)
normT_3 <- norm(T_3)
normT_4 <- norm(T_4)
normT_5 <- norm(T_5)

# d.....

#define the training sets and testing sets for each I/O matrix
train_T_1 <- normT_1[1:380,]
test_T_1 <- normT_1[381: nrow(normT_1),]

train_T_2 <- normT_2[1:380,]
test_T_2 <- normT_2[381: nrow(normT_2),]

train_T_3 <- normT_3[1:380,]
test_T_3 <- normT_3[381: nrow(normT_3),]

train_T_4 <- normT_4[1:380,]
test_T_4 <- normT_4[381: nrow(normT_4),]
```

```
train_T_5 <- normT_5[1:380,]
test_T_5 <- normT_5[381: nrow(normT_5),]

# Training the Neural network for normT_1

# One hidden layer neural networks version #1

T_1_NN1 <- neuralnet(Output ~ Input, data = train_T_1, hidden = 4, linear.output = TRUE)
plot(T_1_NN1)

T_2_NN1 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2, hidden = 4, linear.output = TRUE)
plot(T_2_NN1)

T_3_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data = train_T_3, hidden = 4, linear.output
= TRUE)
plot(T_3_NN1)

T_4_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4, data = train_T_4, hidden = 4,
linear.output = TRUE)
plot(T_4_NN1)

T_5_NN1 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 + Input_5, data = train_T_5,
hidden = 4, linear.output = TRUE)
plot(T_5_NN1)


# Two hidden layer Neural networks

T_1_NN2 <- neuralnet(Output ~ Input, data = train_T_1, hidden = c(4,4), linear.output = TRUE)
plot(T_1_NN2)

T_2_NN2 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2, hidden =c (4,4),
linear.output=TRUE)
plot(T_2_NN2)

T_3_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data = train_T_3, hidden = c(4,4),
linear.output = TRUE)
plot(T_3_NN2)

T_4_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4, data = train_T_4, hidden = c(4,4),
linear.output = TRUE)
plot(T_4_NN2)
```

```
T_5_NN2 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 + Input_5, data = train_T_5,
hidden = c(4,4), linear.output = TRUE)
plot(T_5_NN2)


# One hidden layer neural networks version #2

T_1_NN3 <- neuralnet(Output ~ Input, data = train_T_1, hidden = 7, linear.output = TRUE)
plot(T_1_NN3)

T_2_NN3 <- neuralnet(Output ~ Input_1 + Input_2, data = train_T_2, hidden = 7, linear.output = TRUE)
plot(T_2_NN3)

T_3_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3, data = train_T_3, hidden = 7, linear.output
= TRUE)
plot(T_3_NN3)

T_4_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4, data = train_T_4, hidden = 7,
linear.output = TRUE)
plot(T_4_NN3)

T_5_NN3 <- neuralnet(Output ~ Input_1 + Input_2 + Input_3 + Input_4 + Input_5, data = train_T_5,
hidden = 7, linear.output = TRUE)
plot(T_5_NN3)


# Using performance indicators to determine the optimal NN topologies

# Calculation of the actual output of each I/O matrix's testing data

T_1_act_output <- test_T_1[, "Output"]
T_2_act_output <- test_T_2[, "Output"]
T_3_act_output <- test_T_3[, "Output"]
T_4_act_output <- test_T_4[, "Output"]
T_5_act_output <- test_T_5[, "Output"]

#Then the predicted output from each model is calculated

T_1_pred_output1 <- predict(object = T_1_NN1, test_T_1)
T_1_pred_output2 <- predict(object = T_1_NN2, test_T_1)
T_1_pred_output3 <- predict(object = T_1_NN3, test_T_1)
```

```
T_2_pred_output1 <- predict(object = T_2_NN1, test_T_2)
T_2_pred_output2 <- predict(object = T_2_NN2, test_T_2)
T_2_pred_output3 <- predict(object = T_2_NN3, test_T_2)

T_3_pred_output1 <- predict(object = T_3_NN1, test_T_3)
T_3_pred_output2 <- predict(object = T_3_NN2, test_T_3)
T_3_pred_output3 <- predict(object = T_3_NN3, test_T_3)

T_4_pred_output1 <- predict(object = T_4_NN1, test_T_4)
T_4_pred_output2 <- predict(object = T_4_NN2, test_T_4)
T_4_pred_output3 <- predict(object = T_4_NN3, test_T_4)

T_5_pred_output1 <- predict(object = T_5_NN1, test_T_5)
T_5_pred_output2 <- predict(object = T_5_NN2, test_T_5)
T_5_pred_output3 <- predict(object = T_5_NN3, test_T_5)

# Define the unnormalize function
unnormalize <- function(x, min_val, max_val) {
  return (x * (max_val - min_val) + min_val)
}

# Unnormalize the predicted outputs and actual outputs for each model

T_1_pred_output1_unnorm <- unnormalize(T_1_pred_output1, min(train_T_1[, "Output"]),
max(train_T_1[, "Output"]))
T_1_pred_output2_unnorm <- unnormalize(T_1_pred_output2, min(train_T_1[, "Output"]),
max(train_T_1[, "Output"]))
T_1_pred_output3_unnorm <- unnormalize(T_1_pred_output3, min(train_T_1[, "Output"]),
max(train_T_1[, "Output"]))

T_1_act_output_unnorm <- unnormalize(T_1_act_output, min(train_T_1[, "Output"]), max(train_T_1[,
"Output"]))

T_2_pred_output1_unnorm <- unnormalize(T_2_pred_output1, min(train_T_2[, "Output"]),
max(train_T_2[, "Output"]))
T_2_pred_output2_unnorm <- unnormalize(T_2_pred_output2, min(train_T_2[, "Output"]),
max(train_T_2[, "Output"]))
T_2_pred_output3_unnorm <- unnormalize(T_2_pred_output3, min(train_T_2[, "Output"]),
max(train_T_2[, "Output"]))
```

```
T_2_act_output_unnorm <- unnormalize(T_2_act_output, min(train_T_2[, "Output"]), max(train_T_2[,
"Output"]))

T_3_pred_output1_unnorm <- unnormalize(T_3_pred_output1, min(train_T_3[, "Output"]),
max(train_T_3[, "Output"]))
T_3_pred_output2_unnorm <- unnormalize(T_3_pred_output2, min(train_T_3[, "Output"]),
max(train_T_3[, "Output"]))
T_3_pred_output3_unnorm <- unnormalize(T_3_pred_output3, min(train_T_3[, "Output"]),
max(train_T_3[, "Output"]))

T_3_act_output_unnorm <- unnormalize(T_3_act_output, min(train_T_3[, "Output"]), max(train_T_3[,
"Output"]))

T_4_pred_output1_unnorm <- unnormalize(T_4_pred_output1, min(train_T_4[, "Output"]),
max(train_T_4[, "Output"]))
T_4_pred_output2_unnorm <- unnormalize(T_4_pred_output2, min(train_T_4[, "Output"]),
max(train_T_4[, "Output"]))
T_4_pred_output3_unnorm <- unnormalize(T_4_pred_output3, min(train_T_4[, "Output"]),
max(train_T_4[, "Output"]))

T_4_act_output_unnorm <- unnormalize(T_4_act_output, min(train_T_4[, "Output"]), max(train_T_4[,
"Output"]))

T_5_pred_output1_unnorm <- unnormalize(T_5_pred_output1, min(train_T_5[, "Output"]),
max(train_T_5[, "Output"]))
T_5_pred_output2_unnorm <- unnormalize(T_5_pred_output2, min(train_T_5[, "Output"]),
max(train_T_5[, "Output"]))
T_5_pred_output3_unnorm <- unnormalize(T_5_pred_output3, min(train_T_5[, "Output"]),
max(train_T_5[, "Output"]))

T_5_act_output_unnorm <- unnormalize(T_5_act_output, min(train_T_5[, "Output"]), max(train_T_5[,
"Output"]))

#Introduce a function to find performance metrics

perform_metrics <- function(act_output, pred_output){
  return(list(RMSE = rmse(act_output, pred_output),
              MAE = mae(act_output, pred_output),
              MAPE = mape(act_output, pred_output),
              SMAPE = smape(act_output, pred_output)))
}
```

```
#  calculion of performance metrics for each model

T_1_NN1_performance <- perform_metrics(T_1_act_output, T_1_pred_output1)
T_1_NN2_performance <- perform_metrics(T_1_act_output, T_1_pred_output2)
T_1_NN3_performance <- perform_metrics(T_1_act_output, T_1_pred_output3)

T_2_NN1_performance <- perform_metrics(T_2_act_output, T_2_pred_output1)
T_2_NN2_performance <- perform_metrics(T_2_act_output, T_2_pred_output2)
T_2_NN3_performance <- perform_metrics(T_2_act_output, T_2_pred_output3)

T_3_NN1_performance <- perform_metrics(T_3_act_output, T_3_pred_output1)
T_3_NN2_performance <- perform_metrics(T_3_act_output, T_3_pred_output2)
T_3_NN3_performance <- perform_metrics(T_3_act_output, T_3_pred_output3)

T_4_NN1_performance <- perform_metrics(T_4_act_output, T_4_pred_output1)
T_4_NN2_performance <- perform_metrics(T_4_act_output, T_4_pred_output2)
T_4_NN3_performance <- perform_metrics(T_4_act_output, T_4_pred_output3)

T_5_NN1_performance <- perform_metrics(T_5_act_output, T_5_pred_output1)
T_5_NN2_performance <- perform_metrics(T_5_act_output, T_5_pred_output2)
T_5_NN3_performance <- perform_metrics(T_5_act_output, T_5_pred_output3)

T_1_NN1_performance
T_1_NN2_performance
T_1_NN3_performance

T_2_NN1_performance
T_2_NN2_performance
T_2_NN3_performance

T_3_NN1_performance
T_3_NN2_performance
T_3_NN3_performance

T_4_NN1_performance
T_4_NN2_performance
T_4_NN3_performance

T_5_NN1_performance
T_5_NN2_performance
T_5_NN3_performance
```

```
# g.....

# Calculate total number of weight parameters for T_5_NN3
T_5_NN3_weights <- sum(sapply(T_5_NN3$weights, function(x) length(x)))
T_5_NN3_weights

# Calculate total number of weight parameters for T_5_NN2
T_5_NN2_weights <- sum(sapply(T_5_NN2$weights, function(x) length(x)))
T_5_NN2_weights
```