UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

BSc (Hons) Computer Science

Module Name - Machine Learning and Data Mining
Module Code - 5DATA001C.2

Individual Coursework

Student Details

Student Name	M. A. Dilana Sasanka Mathugama
Student UOW ID	w1901980
Student IIT ID	20212147

Table of Contents

List of Figures

List of Tables

Partitioning Clustering

1.1 Data Preprocessing - scaling and removing outliers

In the exploration of the white wine dataset, ensuring data quality and preparing it for analysis are pivotal steps. The dataset, comprising 2700 varieties of white wine from a specific region in Portugal, holds insights into various chemical and sensory attributes. However, before conducting any analysis, it's essential to preprocess the data to ensure its quality and suitability for analysis. In this section, the steps taken to scale the data and handle outliers in the white wine dataset are described.

Upon loading the dataset, the first step in the analysis was to separate the features (attributes) from the dataset. This ensured that only relevant variables were included in subsequent analyses.

```
# Load the dataset
data <- read_excel("C:/Users/HP 15 - CS1032TX/Desktop/Whitewine_v6.xlsx")

# Separate features (attributes)
data <- data[, 1:11]
```

Figure 1: Load dataset Separate features wine data

Scaling the data was a crucial preprocessing step to ensure that all variables were on the same scale. This is particularly important for distance-based algorithms like k-means clustering, where the magnitude of variables can influence clustering results.

```
# Scale data
scaled_data <- scale(data)
```

Figure 2: Scale wine data

For outlier detection, the Interquartile Range (IQR) method was employed due to its robustness in handling skewed distributions and resistance to extreme values. The use of 2.5 times the IQR as the threshold for identifying outliers was justified to capture potentially extreme values while minimizing the risk of excluding valid data points.

```r
# Detect Outliers

# Calculate the quartiles for each column
q1 <- apply(scaled_data, 2, quantile, probs = c(0.25))
q3 <- apply(scaled_data, 2, quantile, probs = c(0.75))
iqr <- q3 - q1

# Calculate the upper and lower limits
upper <- q3 + 2.5 * iqr
lower <- q1 - 2.5 * iqr

# Identify the outliers
outliers <- apply(scaled_data, 2, function(x) x < lower | x > upper)
```

Figure 3: Detect Outliers wine data

After identifying outliers using the IQR method, they are removed from the dataset.

```r
# Remove the outliers
data_no_outliers <- scaled_data[!apply(outliers, 1, any),]
```

Figure 4: Remove Outliers wine data

The plots below illustrate the data distribution before and after outlier detection. It is evident that the outliers have been significantly reduced after the removal process.

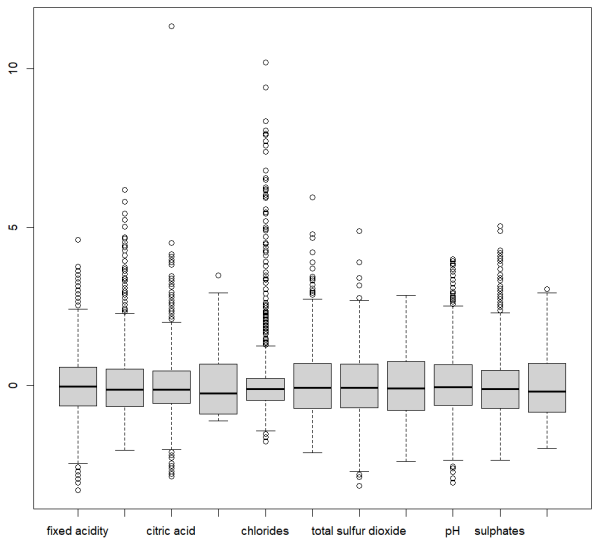

Figure 5: Before removing Outliers

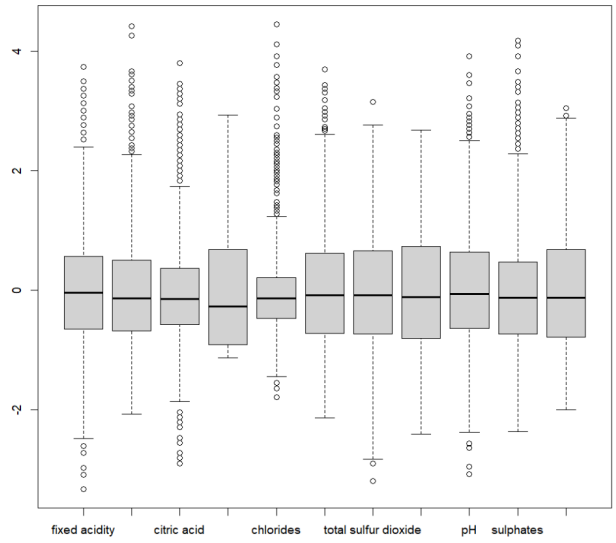

Figure 6: After removing Outliers

Out of the 2700 data points in the dataset, a total of 167 outliers were detected and removed, ensuring the quality and reliability of the data for further analysis.

1.2 Determining the Number of Clusters

In k-means clustering, specifying the number of clusters is crucial, as it directly impacts the quality of the clustering outcome. Determining the optimal number of clusters for a given dataset is a critical step in ensuring effective clustering results. Various methods exist to calculate this optimal number, each aiming to strike a balance between model complexity and clustering performance.

1.2.1 Identifying optimal number of clusters with Nbclust method

In determining the optimal number of clusters for our white wine dataset, we employed the NbClust method, a widely used technique for cluster analysis. NbClust provides a comprehensive framework for evaluating the number of clusters in a dataset by integrating multiple indices and algorithms. By leveraging various clustering criteria, NbClust assists in selecting the most suitable number of clusters based on the characteristics of the data.

To utilize the NbClust method, we utilized the NbClust library in R. This library offers a convenient interface for conducting cluster analysis and accessing a range of clustering indices. By specifying parameters such as distance metric, minimum and maximum number of clusters, and clustering algorithm, NbClust facilitates a systematic exploration of clustering solutions.

```
# NBclust
nb_clusters <- NbClust(data_no_outliers, distance = "euclidean", min.nc = 2,
                       max.nc = 10, method = "kmeans", index = "all")
```

Figure 7: NbClust Method

```
*******************************************************************
* Among all indices:
* 13 proposed 2 as the best number of clusters
* 4 proposed 3 as the best number of clusters
* 5 proposed 4 as the best number of clusters
* 1 proposed 6 as the best number of clusters
* 1 proposed 10 as the best number of clusters

                   ***** Conclusion *****

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


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

Figure 8: NbClust Method Output

According to NbClust, the best number of clusters is determined to be 2.

1.2.2 Identifying optimal number of clusters with the Elbow method

The Elbow method serves as an automated technique for identifying the suitable number of clusters within a dataset. It operates by computing the Within Cluster Sum of Squares (WSS) across varying cluster numbers, usually spanning from 2 to a predetermined maximum. As the number of clusters rises, the WSS value typically declines. The pivotal point occurs when this decrease starts to level off, forming an identifiable "elbow" shape in the WSS versus cluster number plot. This inflection point signifies the optimal cluster count for the dataset.

```
# Elbow method
fviz_nbclust(data_no_outliers, kmeans, method = "wss")
```

Figure 9: ElbowMethod

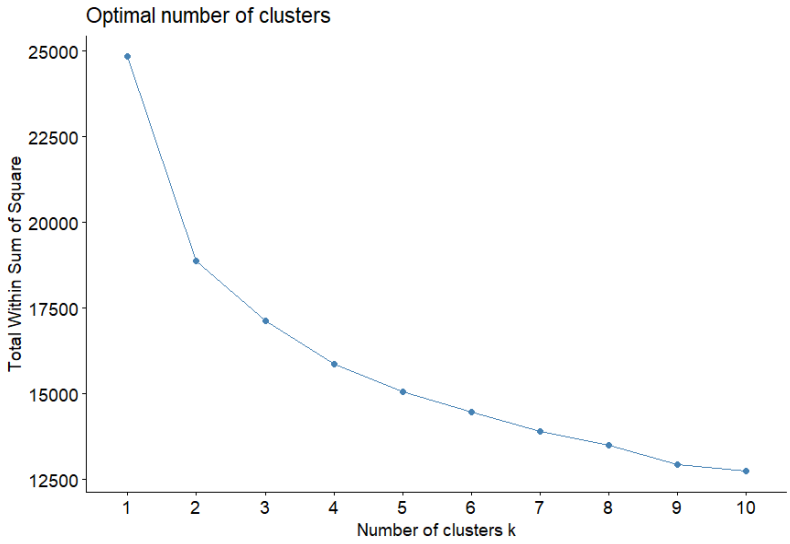

Figure 10: Elbow Method Plot

In the plot generated by the Elbow method, the optimal number of clusters (k) is identified at the point where the reduction in WSS slows down, forming an elbow shape. This elbow point represents the optimal k value according to the plot which is 2.

1.2.3 Identifying optimal number of clusters with the Gap Stat method

The Gap statistic method is another valuable approach for determining the appropriate number of clusters in a dataset. Unlike other methods, the Gap statistic evaluates the difference between the observed within-cluster dispersion and its expected value under a null reference distribution. This method compares the clustering structure of the actual data with that of randomly generated data, providing insights into the optimal number of clusters.

```
# Gap static method
fviz_nbclust(data_no_outliers, kmeans, method = "gap_stat")
```

Figure 11: Gap Stat Method

The plot generated by the Gap statistic method is presented below.

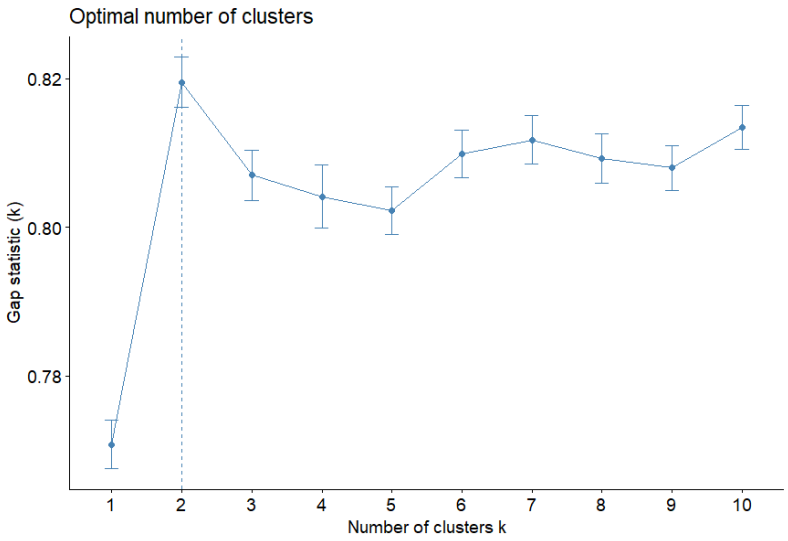

Figure 12: Gap Stat Method Plot

According to the Gap Static method, the best number of clusters is determined to be 2.

1.2.4 Identifying optimal number of clusters with the silhouette method

The Silhouette method offers another avenue for determining the optimal number of clusters within a dataset. This method assesses the quality of clustering by measuring how similar an observation is to its own cluster compared to other clusters. A silhouette score close to +1 indicates that the observation is well-clustered, while a score close to -1 suggests that it may have been assigned to the wrong cluster. The overall silhouette score is calculated as the average silhouette score across all observations.

```
# Silhouette method
fviz_nbclust(data_no_outliers, kmeans, method = "silhouette")
```

Figure 13: Silhouette Method

The plot generated by the Silhouette method is shown below.

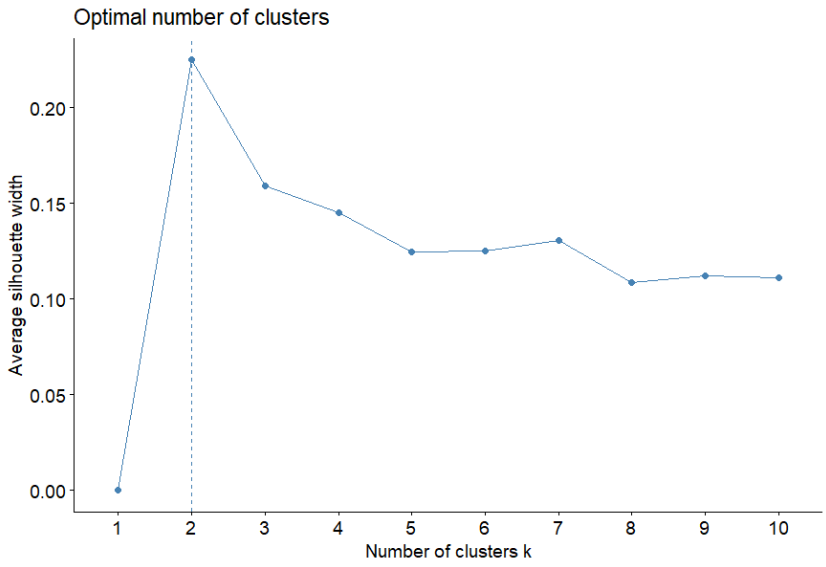

Figure 14: Silhouette Method Plot

According to the silhouette method, the best number of clusters is determined to be 2.

After analyzing the results from NBclust, Elbow, Gap statistics, and silhouette methods collectively, it is concluded that the dataset is best represented by 2 clusters.

1.3 Implementing K-means Clustering

For the clustering analysis using the K-means algorithm, we utilized the optimal number of clusters determined through our previous evaluations. With the optimal K value established, we proceeded to perform the K-means clustering on the preprocessed dataset.

```
# Clustering using kmeans algorithm
kmeans_model <- kmeans(data_no_outliers, centers = 2)
```

Figure 15: K-mean Method

Following the execution of the K-means algorithm, we visualized the resulting clusters using a scatter plot. This plot provides a visual representation of how the data points are grouped into distinct clusters.

```
fviz_cluster(kmeans_model, data = data_no_outliers, palette = c("#2E9FDF", "#DB4035",
             geom = "point",
             ellipse.type = "convex",
             ggtheme = theme_bw())
```

Figure 16: K-mean Method Plot Method

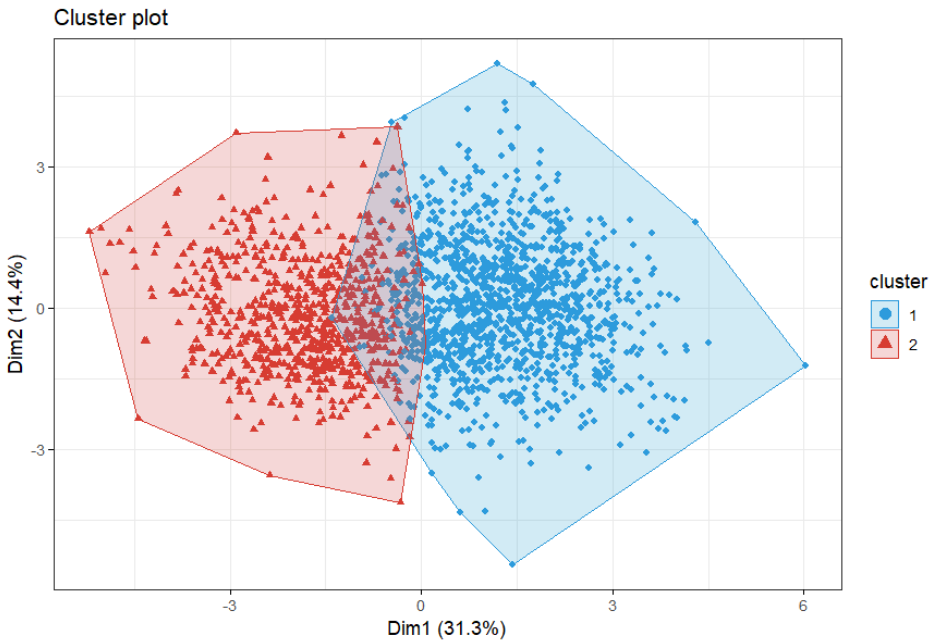

Figure 17: K-mean Method Plot

The related R-based kmeans output, including information regarding the cluster centers and the assignment of data points to clusters, is provided below.

```
Cluster means:
  fixed acidity volatile acidity citric acid residual sugar  chlorides free sulfur dioxide
1    -0.1432787      -0.09008752  -0.1510136     -0.5969408 -0.2823827          -0.4088851
2     0.2055177       0.05210709   0.1404777      0.9245099  0.1700048           0.5624547
  total sulfur dioxide    density         pH   sulphates    alcohol
1           -0.5262840 -0.6641832  0.1340157 -0.05549640  0.5317914
2            0.7645799  0.9829231 -0.2003060  0.04944134 -0.7599434

Clustering vector:
   [1] 2 1 1 2 2 1 1 2 2 1 2 1 2 1 1 2 2 1 2 2 1 1 1 1 1 1 1 1 1 1 2 2 2 2 1 1 1 2 2 1 1 2 1 2 1
  [47] 1 1 1 1 1 1 1 2 2 1 1 1 1 2 1 2 1 2 2 2 2 2 2 2 1 1 1 2 1 1 1 2 1 2 1 2 1 1 2 2 1 2 2 1 2 2 2 2
  [93] 2 2 2 2 2 1 1 1 1 1 2 1 1 2 2 1 2 2 2 2 2 1 2 2 1 2 1 2 1 1 1 2 1 2 1 1 2 1 2 1 2 2 2 2 2 1 1 1 2
 [139] 2 2 1 2 2 2 2 1 1 2 2 1 2 2 2 2 2 2 1 1 1 2 2 2 2 1 1 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2
 [185] 1 2 2 2 2 2 2 2 1 1 1 1 1 1 2 1 2 1 1 1 1 1 1 2 2 2 2 2 1 2 2 1 2 1 1 1 1 1 1 2 1 1 1 1 2 1 2
 [231] 2 2 2 2 2 1 2 1 2 2 1 1 1 2 2 2 1 2 1 2 2 2 2 2 2 1 1 2 2 1 2 2 2 2 2 2 2 2 2 1 1 1 2 1 2 2 2 1
 [277] 2 1 2 2 2 1 1 2 1 1 1 2 2 1 1 2 2 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2
 [323] 2 1 2 2 2 1 2 2 1 2 2 1 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 1 2 1
 [369] 1 2 2 2 2 1 1 2 2 2 2 1 2 2 1 2 2 2 2 1 2 2 2 1 2 1 2 2 1 2 2 1 1 1 2 1 2 2 2 1 2 1 2 2 2 2 2 2
 [415] 1 2 2 1 2 1 2 2 2 2 2 2 2 1 1 1 2 2 2 2 2 2 1 1 2 1 2 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 1 2 1 2 1 1 1 1
 [461] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 1 2 2 2 2 1 1 1 2 1 1 1 2 1 1 1 2 1 2 2 1 2 2 1 2 2 2 2 2 1 2
 [507] 1 1 2 1 2 1 1 1 1 1 1 2 1 1 1 2 2 1 1 2 1 2 1 2 1 1 1 2 2 2 2 1 1 1 2 1 2 1 2 1 1 1 1 2 1 2 1
 [553] 1 2 2 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 2 1 2 1 2 2 2 2 2 1 1 2 1 1 1 1 2 2 2 2 1 1 2
 [599] 2 2 2 1 2 2 2 1 1 2 2 1 1 2 2 1 2 2 1 2 2 2 2 2 2 1 1 2 1 1 2 2 2 2 1 2 1 2 1 1 1 2 2 1 2 1 2
 [645] 2 1 1 2 2 1 2 2 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 1 2 1 1 1 1 1 2 1 1
 [691] 1 1 2 2 1 2 1 1 1 2 1 2 1 2 1 2 2 2 2 2 2 2 1 2 1 1 1 1 1 1 2 2 1 1 1 1 2 2 2 2 1 1 2 1 2 2 1
 [737] 1 2 2 2 1 1 1 1 1 2 1 1 2 1 1 1 2 2 1 1 1 2 2 2 2 1 1 1 1 2 2 1 2 1 1 1 2 2 1 1 2 1 1 2 2 2 2
 [783] 2 1 2 2 2 2 2 2 2 1 1 1 2 2 1 1 1 2 2 1 2 2 2 2 2 2 2 2 1 2 2 2 2 1 1 1 1 2 2 2 2 2 1 2 2 1 2 2 1
 [829] 2 2 2 2 2 2 1 2 2 2 2 2 1 1 2 1 2 2 2 2 2 1 1 2 2 1 1 2 1 1 1 1 1 1 2 1 2 1 2 1 1 1 1 2 1 1 2 1 1 1
 [875] 2 2 1 1 2 2 1 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 2 2 2 2 1 1
 [921] 1 2 2 2 2 2 2 1 1 1 1 2 1 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 2 1 2 1 1 2 2 2 2 1 1 2 1 1 1 1 1 2
 [967] 2 2 1 1 2 1 1 1 1 2 2 1 2 1 1 1 2 1 2 1 1 2 2 1 2 2 2 2 1 1 2 1 2 1 1 1
 [ reached getOption("max.print") -- omitted 1533 entries ]

Within cluster sum of squares by cluster:
[1] 11549.166  7332.126
 (between_SS / total_SS =  24.0 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss" "betweenss"
[7] "size"         "iter"         "ifault"
```

Figure 18: K-mean R-outputs

The results of these internal evaluation metrics, namely the BSS/TSS ratio and the WSS indices, are detailed below, providing insights into the effectiveness of the clustering solution.

```
wss <- kmeans_model$tot.withinss
bss <- kmeans_model$betweenss
tss <- wss + bss
bss_ratio <- bss / tss
```

Figure 19: Evaluation Metrics

within sum of squares = 18881.29
between sum of squares = 5961.403
total sum squares = 22782.41
between sum of squares ratio = between sum of squares / total sum squares = 0.3055128

1.4 Calculating Average Silhouette Width for Clusters

In clustering analysis, the average silhouette width serves as a metric to assess the quality and coherence of the resulting clusters. It measures how similar each data point is to its assigned cluster compared to other clusters. A higher average silhouette width indicates better-defined clusters, where data points are more tightly clustered within their respective clusters and well-separated from other clusters.

The plot below illustrates the silhouette widths for each data point within the clusters generated by the K-means algorithm.

```
# Silhouette
sil <- silhouette(kmeans_model$cluster, dist(data_no_outliers))
fviz_silhouette(sil)
```

Figure 20: Silhouette average method

```
> fviz_silhouette(sil)
  cluster size ave.sil.width
1       1 1536          0.22
2       2  997          0.23
```

Figure 21: Silhouette average method console output

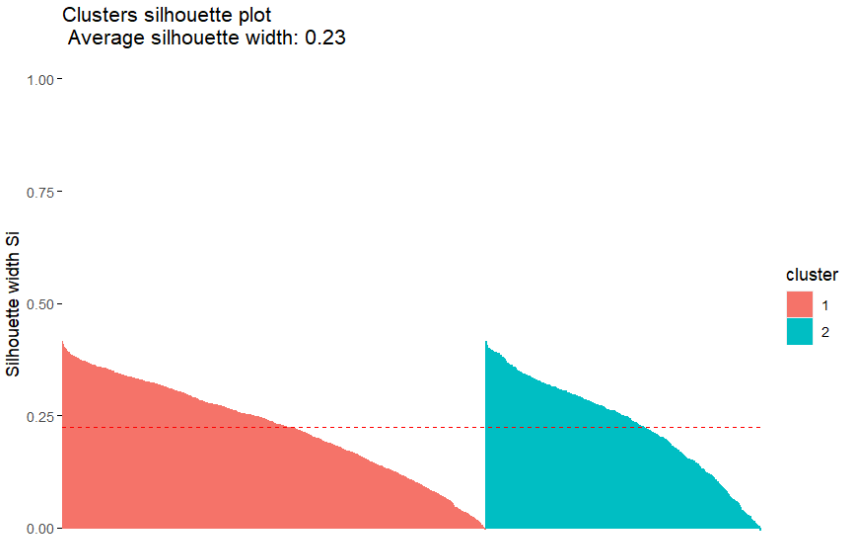

Figure 22: Silhouette average plot

Upon examining the silhouette plot, we observe that the average silhouette widths for clusters 1 and 2 are approximately 0.22 and 0.23, respectively. These values suggest that both clusters exhibit a moderate level of coherence, with data points reasonably well-clustered within their

respective groups and adequately separated from other clusters. Overall, the obtained clusters demonstrate a satisfactory level of quality, as indicated by the average silhouette widths.

1.5 Application of PCA on the wine dataset

As this is a typical multi-dimensional problem in terms of features, it is essential to apply the Principal Component Analysis (PCA) method to the wine dataset. PCA is a dimensionality reduction technique that identifies patterns in data by transforming it into a new coordinate system, thereby capturing the maximum variance in fewer dimensions.

```
# PCA Analysis
pca_result <- prcomp(data_no_outliers, center = TRUE, scale = TRUE)
summary(pca_result)
```

Figure 23: PCA Method

The PCA implementation showing all R-outputs related to PCA analysis is presented below.

```
Importance of components:
                          PC1    PC2    PC3    PC4     PC5     PC6     PC7     PC8     PC9    PC10
Standard deviation     1.856 1.2579 1.1066 1.0398 0.98484 0.87390 0.83030 0.74131 0.62507 0.54059
Proportion of Variance 0.313 0.1439 0.1113 0.0983 0.08817 0.06943 0.06267 0.04996 0.03552 0.02657
Cumulative Proportion  0.313 0.4569 0.5682 0.6665 0.75465 0.82408 0.88676 0.93671 0.97223 0.99880
                          PC11
Standard deviation     0.1149
Proportion of Variance 0.0012
Cumulative Proportion  1.0000
```

Figure 24: PCA Method Results

From the summary, it can be observed that the cumulative proportion exceeds 85% after PC7. Therefore, PC1 to PC7 will be selected as principal components, capturing approximately 89% of the cumulative proportion from the total dataset.

We then create a new transformed dataset using these selected principal components as new features.

```
# transformed dataset
data_transformed <- as.data.frame(-pca_result$x[,1:7])
head(data_transformed)
```

Figure 25: Transformation of dataset

The new dataset, composed of the selected principal components, is illustrated below.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
1	1.040142924	-1.44779851	1.53607721	0.57446896	0.39445790	0.700297056	-0.07230327
2	-0.800685350	-1.61613469	1.12783537	-0.81797620	-1.37942151	-0.620804296	1.67637766
3	-0.800685350	-1.61613469	1.12783537	-0.81797620	-1.37942151	-0.620804296	1.67637766
4	1.040142924	-1.44779851	1.53607721	0.57446896	0.39445790	0.700297056	-0.07230327
5	2.577249877	-2.10215613	-1.06082155	-1.36350756	0.42463769	0.148411490	2.31037200
6	-0.012219223	-1.33987980	0.39689315	-0.98840704	0.79855706	-0.871959246	-1.37589774
7	0.128671725	-0.25271567	-0.12415213	-1.38561852	2.35158409	-0.005469144	0.39925381
8	4.522642914	-1.21079706	0.34738568	-3.76420764	-3.29798520	0.191342411	2.22660326
9	1.255554777	-3.66356018	-2.75674808	0.35253887	-1.30842735	-0.427038151	-0.57680530
10	0.260274635	-4.02951148	-0.88118767	-0.40834982	-0.63226133	-0.735940646	-0.05180821

Figure 26: New dataset view

1.6 Determining the Number of Clusters for new data set

In order to effectively cluster the new transformed dataset, it is crucial to determine the optimal number of clusters.

1.6.1 Identifying optimal number of clusters with the Nbclust method

The NbClust method offers a comprehensive approach to determining the optimal number of clusters for the transformed dataset. By executing the below code, the NbClust method can be applied:

```
# NBclust
nb_clusters <- NbClust(data_transformed, distance = "euclidean", min.nc = 2,
                       max.nc = 10, method = "kmeans", index = "all")
```

Figure 27: NbClust Method

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


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

Figure 28: NbClust Method Output

According to the results obtained from the NbClust analysis, the optimal number of clusters (k) for the new transformed dataset is determined to be 2.

1.6.2 Identifying optimal number of clusters with the elbow method

The Elbow method serves as another valuable tool for identifying the optimal number of clusters in the transformed dataset. By executing the below code, the Elbow method can be applied:

```
# Elbow method
fviz_nbclust(data_transformed, kmeans, method = "wss")
```

Figure 29: Elbow Method

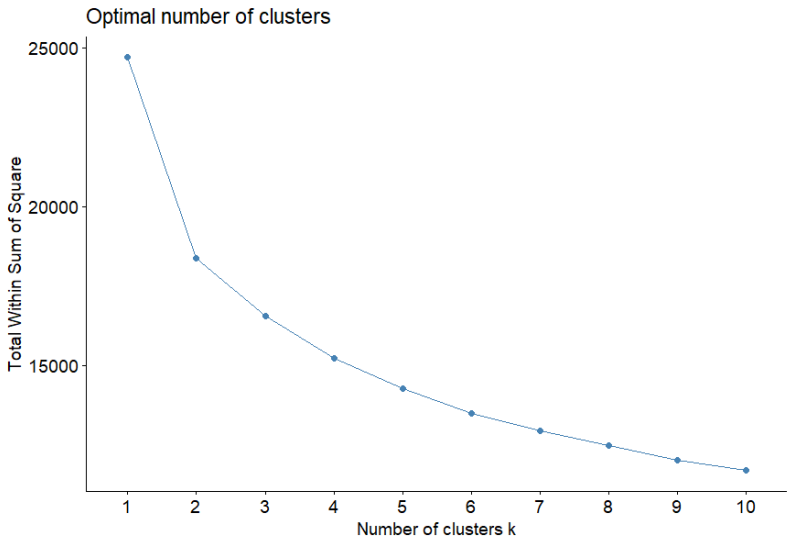

Figure 30: Elbow Method Plot

Upon analyzing the plot generated by the Elbow method, it is observed that the optimal number of clusters (k) for the new transformed dataset is 2.

1.6.3 Identifying optimal number of clusters with the silhouette method

The Silhouette method offers another perspective on determining the optimal number of clusters for the transformed dataset. By executing the below code, the Silhouette method can be applied:

```
# Silhouette method
fviz_nbclust(data_transformed, kmeans, method = "silhouette")
```

Figure 31: Silhouette Method

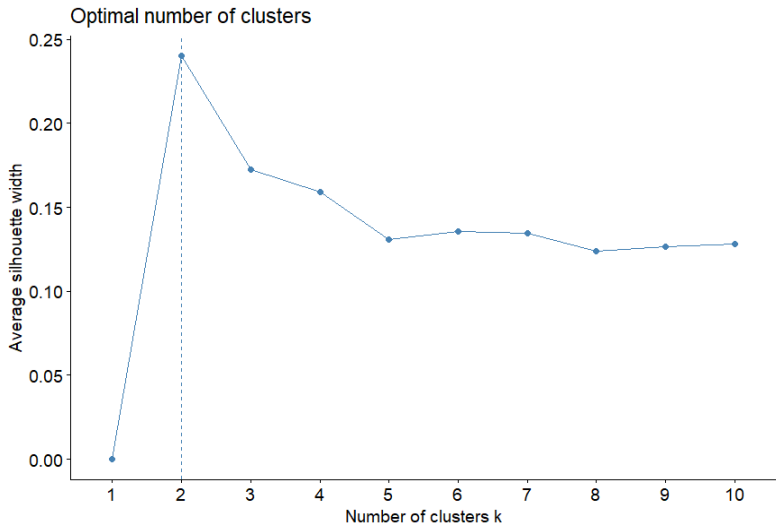

Figure 32: Silhouette Method Plot

Based on the plot generated by the Silhouette method, the optimal number of clusters (k) for the new transformed dataset is again identified as 2.

1.6.4 Identifying optimal number of clusters with the Gap Stat method

Similarly, the Gap statistics method provides insights into the optimal number of clusters for the transformed dataset. By executing the below code, the Gap statistics method can be applied:

```
# Gap static method
fviz_nbclust(data_transformed, kmeans, method = "gap_stat")
```

Figure 33: Gap Stat Method

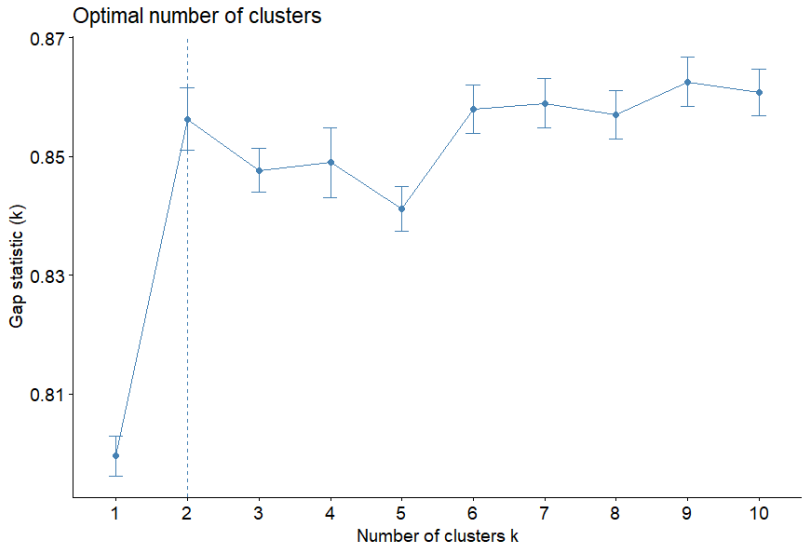

Figure 34: Gap Stat Method Plot

The plot generated by the Gap statistics method indicates that the optimal number of clusters (k) for the new transformed dataset is 2.

After analyzing the results from NBclust, Elbow, Gap statistics, and silhouette methods collectively, it is concluded that the transformed dataset is best represented by 2 clusters.

1.7 Implementing K-means Clustering for the new dataset

After determining the optimal number of clusters required to cluster our new dataset, the K-means clustering algorithm can be applied as follows:

```
# Clustering using kmeans algorithm
kmeans_model <- kmeans(data_transformed, centers = 2)

kmeans_model

fviz_cluster(kmeans_model, data = data_transformed, palette = c("#2E9FDF", "#E7B800"
             geom = "point",
             ellipse.type = "convex",
             ggtheme = theme_bw())
```

Figure 35: K-means clustering

The related R-based kmeans output, including information regarding the cluster centers and the assignment of data points to clusters, is provided below.

```
Cluster means:
        PC1         PC2         PC3         PC4           PC5         PC6         PC7
1 -1.302443 -0.08091794 -0.06794648  0.04167969 -0.0008771008 -0.03106011 -0.03245513
2  1.897449  0.11788433  0.09898702 -0.06072055  0.0012777938  0.04524955  0.04728186

Clustering vector:
   [1] 2 1 1 2 2 1 1 2 2 1 2 1 2 1 1 2 2 1 2 2 1 1 2 1 1 1 1 1 1 1 2 2 2 2 1 1 1 2 2 1 1 2 1 2 1
  [47] 1 1 1 1 1 1 2 2 1 1 1 1 2 1 2 1 2 2 2 2 2 2 1 1 1 2 2 2 1 2 1 2 1 2 1 1 2 2 1 2 2 1 2 2 2 2
  [93] 2 2 2 2 1 1 1 1 1 2 1 1 2 2 1 2 2 2 2 1 2 2 1 2 1 2 1 1 1 2 1 2 1 1 2 1 2 1 2 2 2 2 1 2 2 2
 [139] 2 2 1 2 2 2 2 1 1 2 2 1 2 2 2 2 2 2 1 1 1 2 2 2 2 1 1 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 1 1 1 2
 [185] 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 1 2 1 1 1 1 1 2 2 2 2 2 1 2 2 1 2 1 1 1 1 1 2 1 1 1 1 1 1 2
 [231] 2 2 2 2 1 2 1 1 2 2 1 1 1 2 2 2 1 2 1 2 2 2 2 2 1 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 2 1 2 2 2 1
 [277] 2 1 2 2 2 1 2 2 1 1 1 2 2 1 1 2 2 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2
 [323] 2 1 2 2 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 1 2 1
 [369] 1 2 2 2 2 1 1 2 2 2 2 1 2 2 2 1 2 2 2 2 1 2 2 2 1 2 1 2 2 1 2 2 2 1 2 1 2 2 2 2 2 2 2 2 2
 [415] 1 2 2 1 2 1 2 2 2 2 2 2 2 1 1 1 2 2 2 2 2 2 1 1 2 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 1 2 1 2 1 1 1 1
 [461] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 1 2 2 2 2 1 1 1 2 1 1 1 2 1 1 1 2 1 2 2 2 1 2 2 2 2 2 2 2 2 1 2
 [507] 1 1 2 1 2 1 1 1 1 1 1 2 1 1 1 2 2 1 2 2 2 1 2 1 1 1 2 2 2 2 1 1 1 2 1 1 2 1 2 1 1 1 1 2 1 2 1
 [553] 1 2 2 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 1 2 1 1 2 1 2 1 2 2 2 2 2 1 1 2 1 1 1 1 2 2 2 2 1 1 2
 [599] 2 2 2 1 2 2 2 1 1 2 2 1 1 2 2 1 2 2 1 2 2 2 2 2 1 1 2 1 1 2 2 2 2 1 2 1 2 1 1 1 2 2 1 2 1 2
 [645] 2 1 1 2 2 1 2 2 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 2 2 2 1 1 2 2 2 2 2 2 2 2 1 2 1 1 1 1 1 2 1 1
 [691] 1 1 2 2 1 2 1 2 2 2 1 2 1 2 1 2 2 2 2 2 2 2 1 2 1 1 1 1 1 1 2 2 1 1 1 1 2 2 2 2 1 1 2 1 2 2 1
 [737] 1 2 2 2 2 1 1 1 1 1 2 1 1 2 1 1 1 2 2 1 1 1 2 2 2 1 1 1 2 1 2 2 1 1 2 2 1 1 2 1 1 2 1 1 2 2 2
 [783] 2 1 2 2 2 2 2 2 2 1 1 1 2 2 1 2 1 2 2 1 2 2 2 2 2 2 2 2 1 2 2 2 2 2 1 1 1 2 2 2 2 2 1 2 2 1 2 2 1
 [829] 2 2 2 2 2 2 1 2 2 2 2 1 1 2 1 2 2 2 2 2 1 1 2 2 1 1 2 1 1 1 1 1 2 1 2 1 2 1 1 1 1 2 1 1 2 1 1 1
 [875] 2 2 1 1 2 2 2 1 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 2 2 2 2 2 1 1
 [921] 1 2 2 2 2 2 2 2 1 1 1 1 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 2 1 2 1 1 2 2 2 2 1 1 2 1 1 1 1 1 2
 [967] 2 2 1 2 2 1 1 1 1 2 1 2 1 1 1 2 1 2 1 1 2 2 1 2 2 2 2 1 1 2 1 2 1 1 1
 [ reached getOption("max.print") -- omitted 1533 entries ]

Within cluster sum of squares by cluster:
[1] 10799.730  7583.259
 (between_SS / total_SS =  25.6 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss" "betweenss"
[7] "size"         "iter"         "ifault"
```

Figure 36: K-means clustering R based outputs

The results of these internal evaluation metrics, namely the BSS/TSS ratio and the WSS indices, are detailed below, providing insights into the effectiveness of the clustering solution.

within sum of squares = 18382.99
between sum of squares = 6314.913
total sum squares = 24697.9
between sum of squares ratio = between sum of squares / total sum squares = 0.2556862

1.8 Assessing Clustering Accuracy with Average Silhouette Score

After performing the K-means clustering on the new transformed dataset, the silhouette analysis was conducted to assess the quality of the obtained clusters. The average silhouette width score, which measures how well each data point fits into its assigned cluster compared to other clusters, was computed.

The plot below illustrates the silhouette widths for each data point within the clusters generated by the K-means algorithm.

```
# Silhouette
sil <- silhouette(kmeans_model$cluster, dist(data_transformed))
fviz_silhouette(sil)
```

Figure 37: Silhouette average method

```
> fviz_silhouette(sil)
  cluster size ave.sil.width
1       1 1502          0.24
2       2 1031          0.24
```

Figure 38: Silhouette average method results

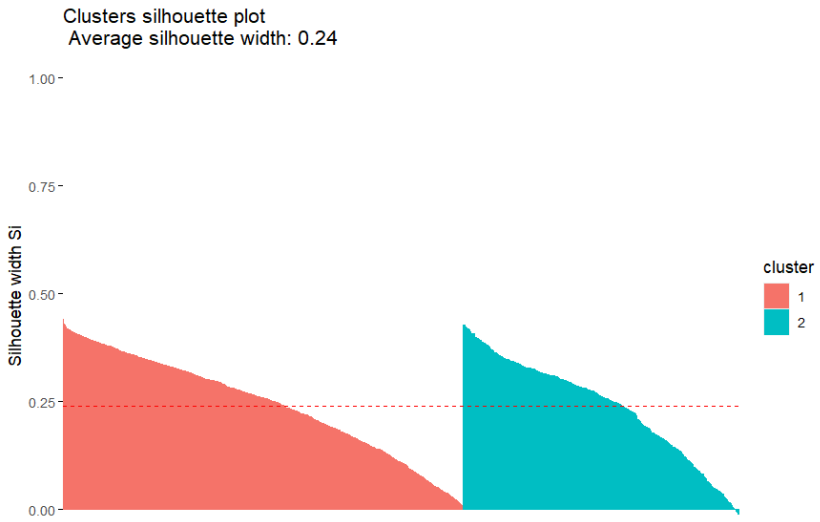

Figure 39: Silhouette average method plot

The average silhouette width score for cluster 1 was found to be 0.24, indicating that the majority of data points in this cluster are well-clustered and have a high similarity to other points within the same cluster. Similarly, the average silhouette width score for cluster 2 was also 0.24, suggesting that the clustering of data points in this cluster is also of good quality. Overall, the average silhouette width score of 0.24 for both clusters indicates a moderate to good level of clustering quality. The clusters are reasonably well-separated from each other, and most data points are appropriately assigned to their respective clusters.

1.9 Implementation of Calinski-Harabasz

Following the K-means analysis for the new "pca" dataset, the Calinski-Harabasz Index was implemented to evaluate the clustering performance. This index is a well-known internal evaluation metric used to assess the compactness and separation of clusters.

```r
install.packages("fpc")
library(fpc)

# Obtain the cluster assignments
clustering <- kmeans_model$cluster

# Compute the Calinski-Harabasz Index
calinski_harabasz <- calinhara(data_transformed, clustering)

# Print the Calinski-Harabasz Index
print(calinski_harabasz)
```

Figure 83: Calinski Harabasz Implementation

The Calinski-Harabasz Index value obtained for the K-means model with 2 clusters was found to be 869.4476. This value provides insight into the compactness and separation of clusters, with higher values indicating better-defined clusters.

To further analyze the impact of different numbers of clusters on the Calinski-Harabasz Index, the index was computed for varying numbers of clusters ranging from 2 to 10.

```r
# Compute Calinski-Harabasz Index for kmeans clustering with varying number of clusters
calinski_results <- numeric(10)
for (k in 2:10) {
  kmeans_model <- kmeans(data_transformed, centers = k)
  clustering <- kmeans_model$cluster
  calinski_results[k] <- calinhara(data_transformed, clustering)
}

# Visualize the index
plot(2:10, calinski_results[2:10], type = "b", xlab = "Number of Clusters", ylab = "Calinski-Harabasz Index")
```

Figure 84: Calinski Harabasz for different k

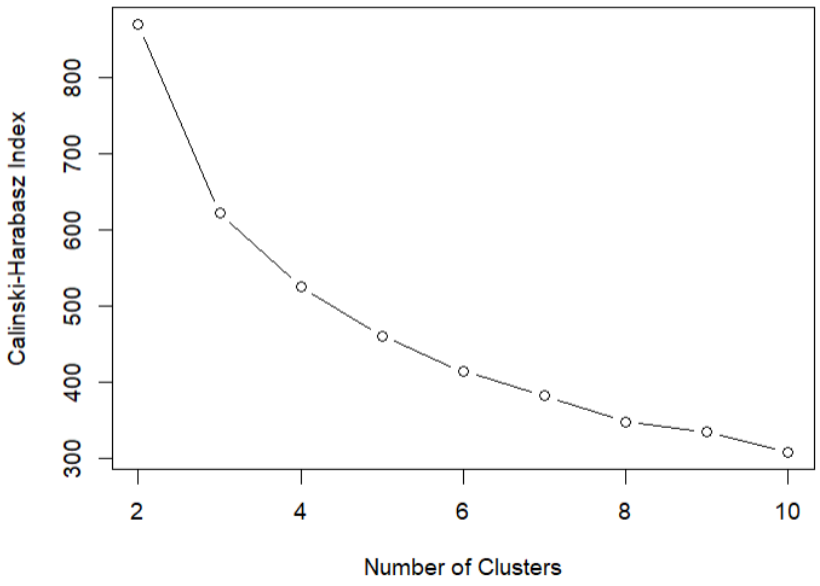

Figure 85: Calinski Harabasz plot for different k

The plot illustrates the Calinski-Harabasz Index values for different numbers of clusters. From the plot, it is evident that the index reaches its highest value when the number of clusters is set to 2, indicating that this configuration yields the most well-defined and separated clusters.

In summary, the Calinski-Harabasz Index analysis suggests that clustering the new transformed dataset with 2 clusters results in the most optimal clustering solution, as it maximizes the compactness within clusters while maximizing the separation between clusters.

Financial Forecasting

2.1 Brief discussion of Input Vector Construction Methods in Exchange Rate Forecasting

In exchange rate forecasting, input vectors for neural network (NN) models play a vital role in capturing relevant information and patterns. Different methods are used to define these input vectors, tailored to the specific characteristics of exchange rate time-series data.

Autoregressive (AR) Approach: This method involves using lagged values of the target variable (exchange rate) as input features. Including lagged exchange rates from previous time steps allows the model to capture temporal dependencies and historical trends.
Moving Average (MA) Approach: Here, moving averages of the exchange rate over specific time windows are incorporated as input features. This helps smooth out noise and fluctuations in the data, providing a clearer signal for the NN.
Technical Indicators: Various indicators derived from exchange rate data, such as moving averages, relative strength index (RSI), and Bollinger Bands, can be used as input features. These indicators offer insights into market trends and volatility.
Fundamental Factors: Input vectors may include fundamental economic factors like interest rates, inflation rates, and trade balances. Incorporating such factors improves the model's understanding of the economic forces influencing exchange rates (Dautel et al., 2020).
External Factors: Factors like macroeconomic indicators of other countries, global market sentiment, and geopolitical tensions can also be included. Integrating these external factors provides a broader perspective on exchange rate movements.
Hybrid Approaches: Combining multiple methods or incorporating domain-specific knowledge can further enhance predictive performance. These approaches are tailored to capture the unique dynamics of exchange rate data.

In summary, the choice of input vector construction method depends on objectives, data characteristics, and domain knowledge. Each method offers advantages in capturing different aspects of exchange rate dynamics, and selecting an appropriate approach is crucial for accurate forecasting (Pfahler, 2022).

2.2 Construction of Input/Output Matrices

In this section, the process of constructing input/output matrices for the one-step-ahead forecasting of the USD/EUR exchange rate using a multilayer neural network (MLP-NN) with an autoregressive (AR) approach is detailed. Utilizing time-delayed values of the 3rd column attribute from the dataset, input vectors are created, exploring various configurations up to a lag of (t-4) to assess their impact on the model's predictive performance.

Below is a visualization illustrating the logic behind the creation of various input vectors up to the (t-4) level. This visual aid provides clarity on how the temporal relationships between past exchange rate values are utilized to inform the model's predictions.

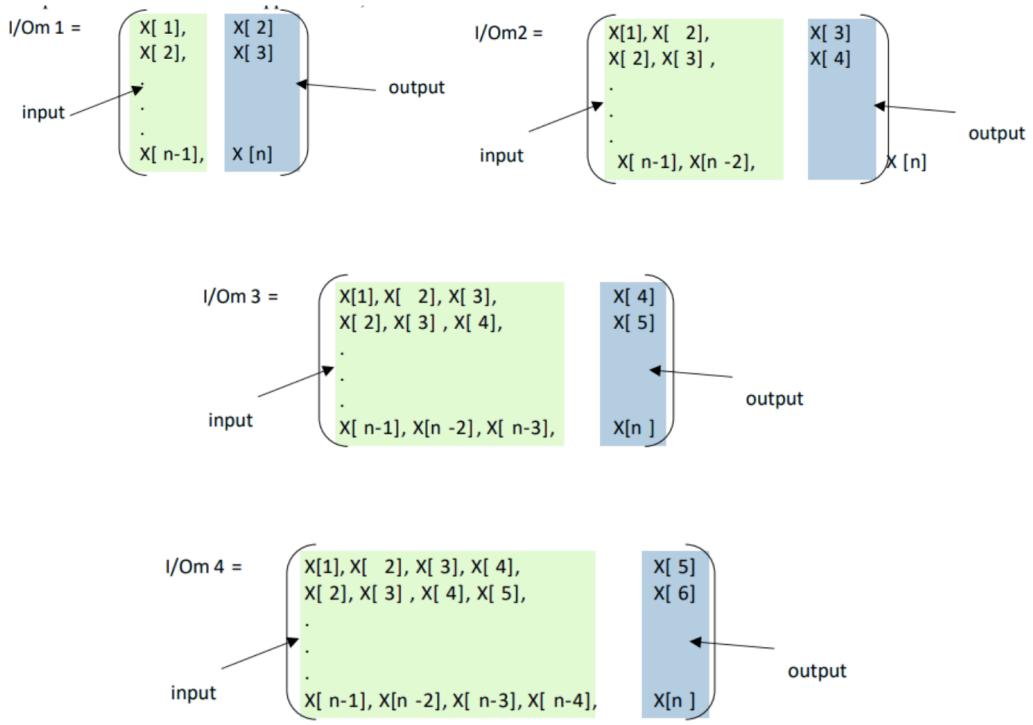

Figure 40: Various input vectors logic

Each step in the process, from loading the dataset to handling missing values, is explained below with code snippets. This approach ensures the creation of structured datasets suitable for training and testing the MLP-NN model.

The code snippet provided loads the dataset containing the USD/EUR exchange rates from an Excel file.

```
# Load data
exchange_data <- read_excel("D:/ML/ExchangeUSD.xlsx")
```

Figure 41: Load rates data

The third column from the dataset, corresponding to the USD/EUR exchange rates, is extracted and stored in a dataframe.

```
# Extract the third column
exchange_rates <- exchange_data$`USD/EUR`
exchange_rates_df <- data.frame(exchange_rates)
```

Figure 42: Extract 3rd column rates data

The dataset is divided into training and testing sets, with the first 400 data points used for training and the remaining data points for testing.

```
# Split data into training and testing sets
train_data <- exchange_rates_df[1:400,]
test_data <- exchange_rates_df[401:length(exchange_rates),]
```

Figure 43: Split data into training and testing

Lagged input-output matrices are constructed for both the training and testing datasets, employing an autoregressive approach with a lag of 4.

```
# Creating lagged input-output matrix for training data set t4
train_lagged_t4 <- bind_cols(G_prev3 = lag(train_data,4),
                             G_prev2 = lag(train_data,3),
                             G_prev = lag(train_data,2),
                             G_curr = lag(train_data,1),
                             G_pred = train_data)

# Creating lagged input-output matrix for testing data set t4
test_lagged_t4 <- bind_cols(G_prev3 = lag(test_data,4),
                            G_prev2 = lag(test_data,3),
                            G_prev = lag(test_data,2),
                            G_curr = lag(test_data,1),
                            G_pred = test_data)
```

Figure 44: I/O matrices creation

Upon executing the code, the resulting output is displayed as follows:

	G_prev3	G_prev2	G_prev	G_curr	G_pred
1	NA	NA	NA	NA	1.3730
2	NA	NA	NA	1.3730	1.3860
3	NA	NA	1.3730	1.3860	1.3768
4	NA	1.3730	1.3860	1.3768	1.3718
5	1.3730	1.3860	1.3768	1.3718	1.3774
6	1.3860	1.3768	1.3718	1.3774	1.3672
7	1.3768	1.3718	1.3774	1.3672	1.3872
8	1.3718	1.3774	1.3672	1.3872	1.3932
9	1.3774	1.3672	1.3872	1.3932	1.3911

	G_prev3	G_prev2	G_prev	G_curr	G_pred
1	NA	NA	NA	NA	1.2817
2	NA	NA	NA	1.2817	1.2910
3	NA	NA	1.2817	1.2910	1.2865
4	NA	1.2817	1.2910	1.2865	1.2939
5	1.2817	1.2910	1.2865	1.2939	1.2920
6	1.2910	1.2865	1.2939	1.2920	1.2936
7	1.2865	1.2939	1.2920	1.2936	1.2870
8	1.2939	1.2920	1.2936	1.2870	1.2943
9	1.2920	1.2936	1.2870	1.2943	1.3042

Figure 45: Training data matrix

Figure 46: Testing data matrix

Rows with missing values(N/A values) are removed from the lagged input-output matrices for both the training and testing datasets.

```
# Remove N/A values from t4
train_lagged_t4 <- train_lagged_t4[complete.cases(train_lagged_t4),]
test_lagged_t4 <- test_lagged_t4[complete.cases(test_lagged_t4),]
```

Figure 47: Removing N/A Values

Following the removal of N/A values, the resultant clean input/output matrix appears as shown below:

	G_prev3	G_prev2	G_prev	G_curr	G_pred
1	1.3730	1.3860	1.3768	1.3718	1.3774
2	1.3860	1.3768	1.3718	1.3774	1.3672
3	1.3768	1.3718	1.3774	1.3672	1.3872
4	1.3718	1.3774	1.3672	1.3872	1.3932
5	1.3774	1.3672	1.3872	1.3932	1.3911
6	1.3672	1.3872	1.3932	1.3911	1.3838
7	1.3872	1.3932	1.3911	1.3838	1.4171
8	1.3932	1.3911	1.3838	1.4171	1.4164
9	1.3911	1.3838	1.4171	1.4164	1.3947

Figure 48: Training data removed N/A

	G_prev3	G_prev2	G_prev	G_curr	G_pred
1	1.2817	1.2910	1.2865	1.2939	1.2920
2	1.2910	1.2865	1.2939	1.2920	1.2936
3	1.2865	1.2939	1.2920	1.2936	1.2870
4	1.2939	1.2920	1.2936	1.2870	1.2943
5	1.2920	1.2936	1.2870	1.2943	1.3042
6	1.2936	1.2870	1.2943	1.3042	1.2987
7	1.2870	1.2943	1.3042	1.2987	1.3097
8	1.2943	1.3042	1.2987	1.3097	1.3074
9	1.3042	1.2987	1.3097	1.3074	1.3088

Figure 49: Testing data removed N/A

It is acknowledged that the accuracy of the MLP-NN model may vary based on the chosen input vectors. To comprehensively investigate this aspect, input vectors up to lag levels t1, t2, t3, and t4 are created and evaluated. This comprehensive analysis allows us to understand the influence of different input vector configurations on the model's predictive performance, guiding us in selecting the most suitable approach for the forecasting task.

```
# Creating lagged input-output matrix for training data set t3
train_lagged_t3 <- bind_cols(G_prev2 = lag(train_data,3),
                             G_prev = lag(train_data,2),
                             G_curr = lag(train_data,1),
                             G_pred = train_data)

# Creating lagged input-output matrix for testing data set t3
test_lagged_t3 <- bind_cols(G_prev2 = lag(test_data,3),
                            G_prev = lag(test_data,2),
                            G_curr = lag(test_data,1),
                            G_pred = test_data)

# Creating lagged input-output matrix for training data set t2
train_lagged_t2 <- bind_cols(G_prev = lag(train_data,2),
                             G_curr = lag(train_data,1),
                             G_pred = train_data)

# Creating lagged input-output matrix for testing data set t2
test_lagged_t2 <- bind_cols(G_prev = lag(test_data,2),
                            G_curr = lag(test_data,1),
                            G_pred = test_data)

# Creating lagged input-output matrix for training data set t1
train_lagged_t1 <- bind_cols(G_curr = lag(train_data,1),
                             G_pred = train_data)

# Creating lagged input-output matrix for testing data set t1
test_lagged_t1 <- bind_cols(G_curr = lag(test_data,1),
                            G_pred = test_data)
```

Figure 50: Various Input vector creation

2.3 Normalization and Denormalization Techniques

Before training the neural network (NN) model, it is essential to normalize the data to ensure consistent and efficient learning. Additionally, after obtaining predictions from the model, it is crucial to denormalize the results to interpret them accurately.

To achieve this, we utilize a normalization function, normalize_data, which scales the data to a range between 0 and 1. This function subtracts the minimum value from each data point and divides it by the range of the dataset. This process ensures that all input variables are on a similar scale.

```
# Function to normalize data
normalize_data <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}

# Normalize data for t4
train_normalized_t4 <- as.data.frame(lapply(train_lagged_t4, normalize_data))
test_normalized_t4 <- as.data.frame(lapply(test_lagged_t4, normalize_data))
```

Figure 51: Normalizing data

The figures below illustrate the plots of the data both before and after normalization. Prior to normalization, the data exhibits a range from 1.20 to 1.40. However, post-normalization, all data points are confined within the range of 0 to 1.

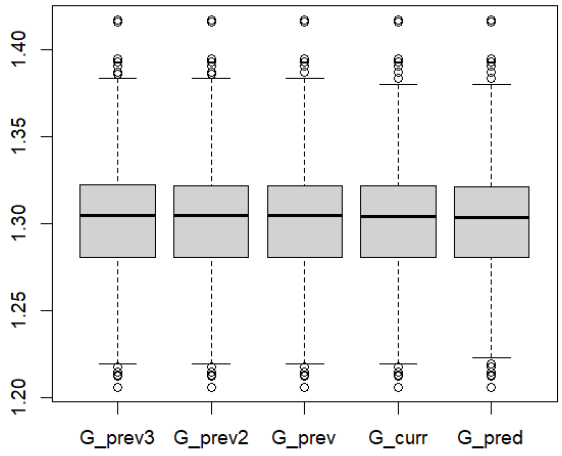

Figure 52: Before Normalizing data

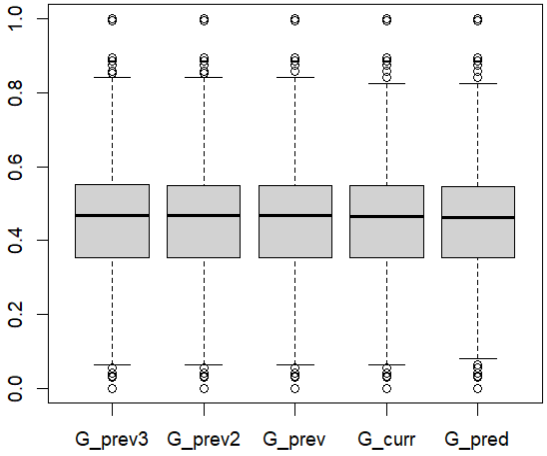

Figure 53: After Normalizing data

After normalization, the original data is stored for denormalization purposes. The denormalization function, denormalize_data, reverses the normalization process by multiplying the normalized results by the range of the original dataset and adding back the minimum value. This ensures that the predicted results are in the same scale as the original data, facilitating their interpretation.

```
# Get original data from denormalized data
train_original_pred <- train_lagged_t4["G_pred"]
test_original_pred <- test_lagged_t4["G_pred"]

# Find min and max value from dataset
min_val <- min(train_original_pred)
max_val <- max(train_original_pred)

# Create function to denormalize data
denormalize_data <- function(x) {
  min_val <- min(train_original_pred)
  max_val <- max(train_original_pred)
  return( (max_val - min_val) * x + min_val )
}
```

Figure 54: Denormalizing data

```
# unnormalize NN predicted data
pred_NN_Results <- denormalize_data(predicted_results_norm)
```

Figure 55: Denormalizing data function apply

	V1
1	0.19153711
2	0.16991195
3	0.19059062
4	0.11683964
5	0.19309153
6	0.29527828
7	0.25060279
8	0.36512215

Figure 56: Before Denormalizing

	V1
1	1.246514
2	1.241951
3	1.246315
4	1.230753
5	1.246842
6	1.268404
7	1.258977
8	1.283141

Figure 57: After Denormalizing

Normalization is essential for Multilayer Perceptron (MLP) structures because it standardizes the input data to a common scale, thereby improving the convergence and performance of the neural network. MLPs are sensitive to the scale of input features, and when features are on different scales, it can lead to slow convergence or difficulties in training. By normalizing the data to a uniform range, typically between 0 and 1 or -1 and 1, we ensure that all input features contribute equally to the training process. This uniformity enables the neural network to learn more efficiently, avoiding dominance by features with larger magnitudes and facilitating faster convergence towards optimal solutions. Additionally, normalization helps in stabilizing the training process by preventing numerical instabilities that may arise due to large input values. Overall, normalization is a crucial preprocessing step for MLP structures to ensure effective learning and improved model performance.

2.4 Implementation of Various MLP Models

To explore the effectiveness of different Multilayer Perceptron (MLP) structures in predicting the exchange rate, a function was developed to create, train, and evaluate neural network models. This function facilitates the testing of various MLP architectures with different input vectors, providing insights into their performance for the forecasting task.

```r
# train and test neural network function
train_test_neural_net <- function(training_data, testing_data, neurons_hidden_layer,
                                  linear_output, activation_function) {

  formula_names <- as.formula(paste("G_pred~", paste(names(training_data)[-ncol(training_data)],
                                                     collapse = "+")))
  #Training model
  rates_load_nn <- neuralnet(formula_names,
                             hidden=neurons_hidden_layer,
                             data=training_data,
                             act.fct = activation_function,
                             linear.output=linear_output)
  plot(rates_load_nn)

  # Extract column names for testing
  predict_label_removed <- setdiff(names(training_data), "G_pred")

  # Testing model

  # Removing Prediction variable
  temp_test <- subset(testing_data, select=predict_label_removed)
  head(temp_test)

  # Creating predicted variables
  model_results <- compute(rates_load_nn, temp_test)
  predicted_results_norm <- model_results$net.result

  # unnormalize NN predicted data
  pred_NN_Results <- denormalize_data(predicted_results_norm)

  # testing performance of RMSE
  rmse <- rmse(exp(pred_NN_Results),test_original_pred$G_pred)

  # testing performance of MAE
  mae <- mae(exp(pred_NN_Results),test_original_pred$G_pred)

  # testing performance of MAPE
  mape <- mean(abs((test_original_pred$G_pred - pred_NN_Results)/test_original_pred$G_pred))*100

  # testing performance of sMAPE
  smape <- mean(2 * abs(test_original_pred$G_pred - pred_NN_Results) / (abs(test_original_pred$G_pred) +
                                                                        abs(pred_NN_Results))) * 100

  return(list(RMSE = rmse, MAE = mae, MAPE = mape, sMAPE = smape))
}
```

Figure 58: Train and test NN function

The train_test_neural_net function oversees the entire process of building, training, and assessing MLP models. It first constructs the MLP structure based on the specified parameters, including the number of hidden neurons, activation function, and output type. Then, it trains the model using the training data and evaluates its performance on the testing dataset. The function returns metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and Symmetric Mean Absolute Percentage Error (sMAPE), which offer comprehensive insights into the model's predictive accuracy.

The train_test_neural_net function is used with different configurations, testing various input vectors and MLP structures. This helps compare model performance systematically, showing what affects prediction accuracy.

```
# Train and test neural network models with different input vectors and structures
results_1 <- train_test_neural_net(train_normalized_t1, test_normalized_t1, c(4), TRUE, "linear")
results_2 <- train_test_neural_net(train_normalized_t2, test_normalized_t2, c(4), TRUE, "linear")
results_3 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(4), TRUE, "linear")
results_4 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4), TRUE, "linear")
results_5 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(5), TRUE, "linear")
results_6 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(6), TRUE, "linear")
results_7 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(5), TRUE, "linear")
results_8 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(6), TRUE, "linear")
results_9 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4,3), TRUE, c("linear", "linear", "linear"))
results_10 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(5,4), TRUE, c("linear", "linear", "linear"))
results_11 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(6,5), TRUE, c("linear", "linear", "linear"))
results_12 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(4,3), TRUE, c("linear", "linear", "linear"))
results_13 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(5,4), TRUE, c("linear", "linear", "linear"))
results_14 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(6,5), TRUE, c("linear", "linear", "linear"))
results_15 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4), FALSE, "logistic")
```

Figure 59: Apply Train and test NN function

The following images showcase the results of training and testing the MLP models with various input vectors and structures. These plots provide visual representations of the performance metrics, aiding in the interpretation of the experimental outcomes.

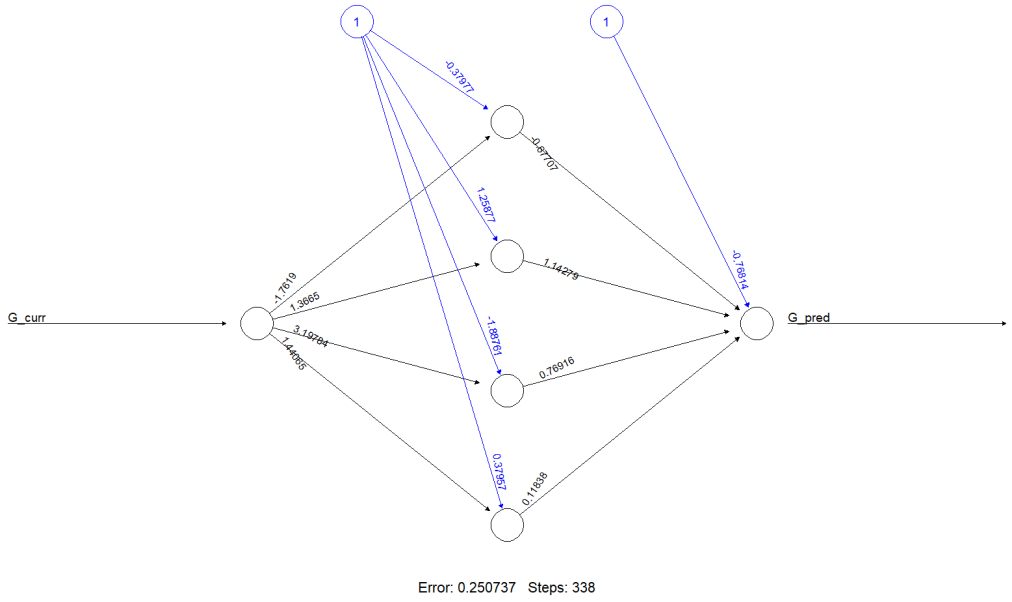

Figure 60: One input One hidden layer 4 neurons NN

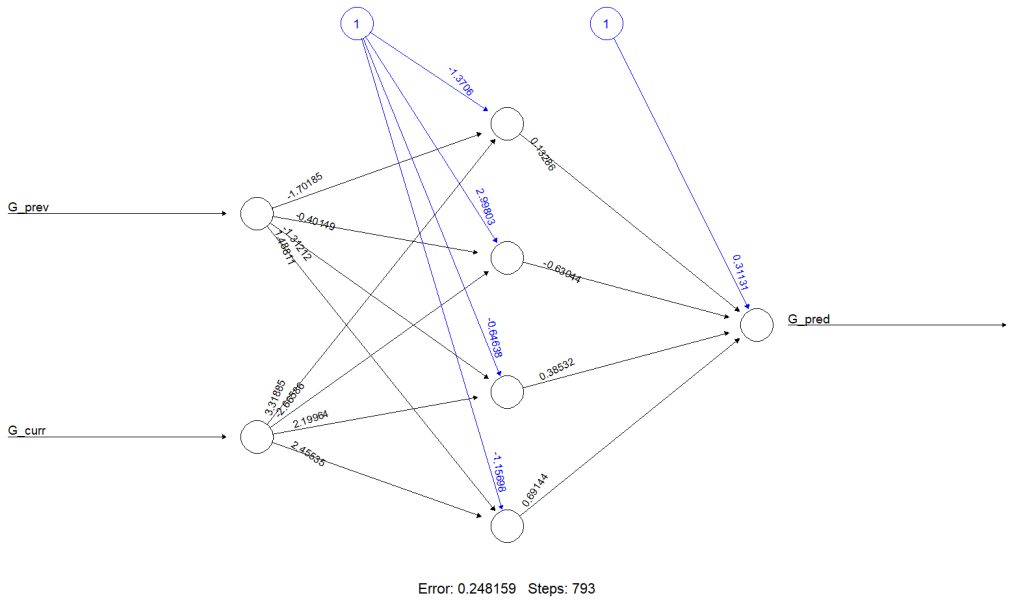

Figure 61: 2 input One hidden layer 4 neurons NN

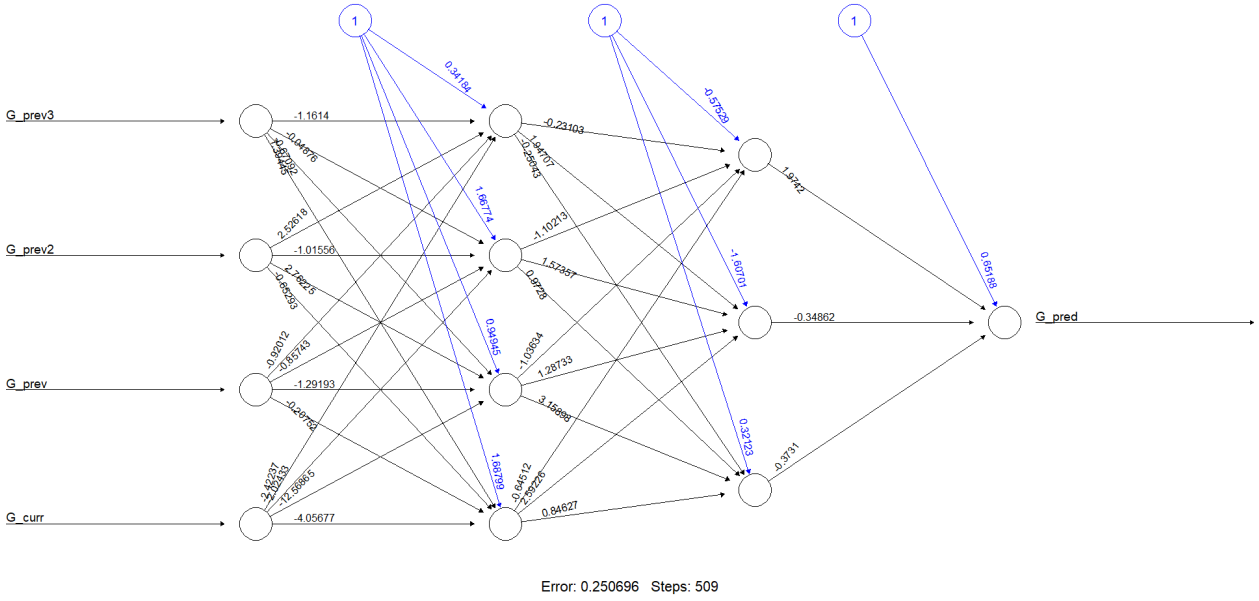

Figure 62: 4 input 2 hidden layer 4,3 neurons NN

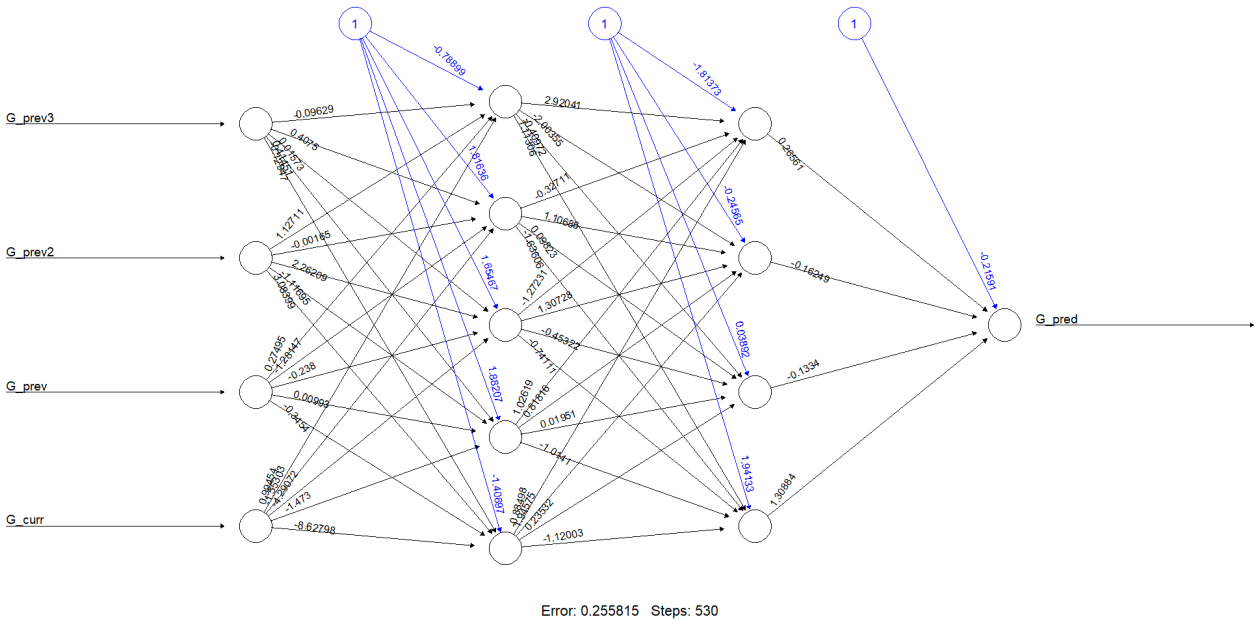

Figure 63: 4 input 2 hidden layer 5,4 neurons NN

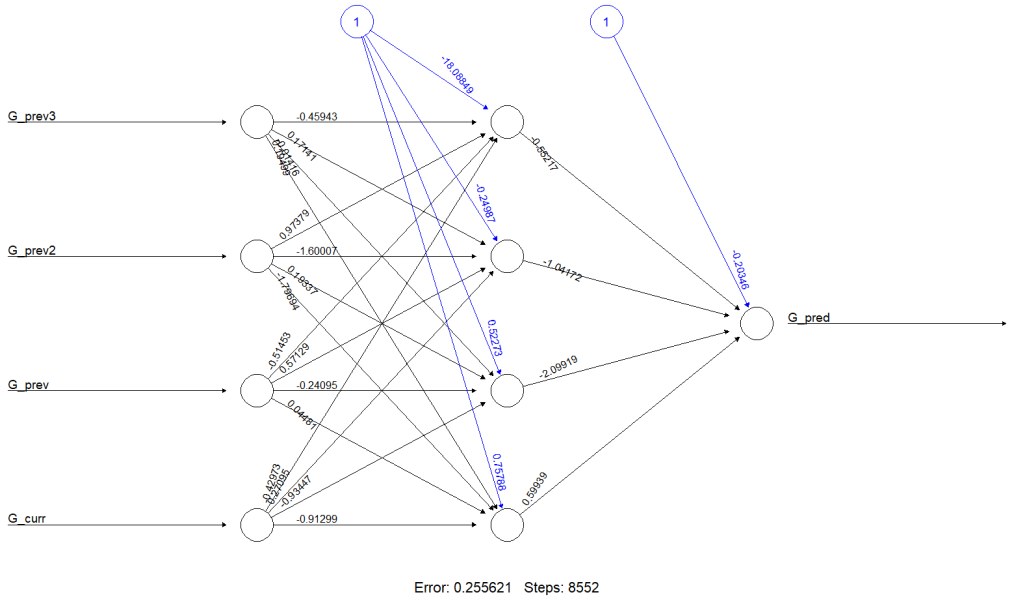

Figure 64: 4 input 1 hidden layer 4 neurons NN tanh

Remaining images of various MLP models tested can be found in Appendix B.2.

2.5. Analysis of RMSE, MAE, MAPE, and sMAPE

Root Mean Square Error (RMSE): RMSE is a measure of the average deviation between the predicted values and the actual values. It calculates the square root of the average of squared differences between predicted and actual values.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(predicted_i - actual_i)^2}$$

Figure 65: RMSE formula

Mean Absolute Error (MAE): MAE measures the average magnitude of errors between predicted and actual values without considering their direction. It provides an understanding of the average absolute deviation.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|predicted_i - actual_i|$$

Figure 66: MAE formula

Mean Absolute Percentage Error (MAPE): MAPE calculates the average percentage difference between predicted and actual values relative to the actual values. It's useful for understanding the accuracy of forecasts in terms of percentage.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{actual_i - predicted_i}{actual_i}\right| \times 100$$

Figure 67: MAPE formula

Symmetric Mean Absolute Percentage Error (sMAPE): sMAPE is a symmetric version of MAPE, which measures the percentage difference between predicted and actual values relative to their average. It's symmetric because it treats overestimation and underestimation equally.

$$\text{sMAPE} = \frac{1}{n}\sum_{i=1}^{n}\frac{|actual_i - predicted_i|}{|actual_i| + |predicted_i|} \times 100$$

Figure 68: sMAPE formula

Choosing the best neural network model is based on these four statistical indices. Lower values of RMSE, MAE, MAPE, and sMAPE indicate better model performance.

2.6 Comparison Table

Num of inputs	Num of hidden layers	Num of neurons	Description of NN structure	RMSE	MAE	MAPE	sMAPE
1	1	4	Utilizes a single hidden layer with 4 neurons, linear output, and logistic activation function with input vector t1	2.410763 30009686	2.405669 43593568	1.787475 37293032	1.802947 9156406 1
2	1	4	Utilizes a single hidden layer with 4 neurons, linear output, and logistic activation function with input vector t2	2.413731 98367413	2.408768 68504137	1.759653 26294425	1.773289 3054206 2
3	1	4	Utilizes a single hidden layer with 4 neurons, linear output, and logistic activation function with input vector t3	2.413700 49575528	2.408916 21724223	1.702983 08135791	1.717370 2790399 7
4	1	4	Utilizes a single hidden layer with 4 neurons, linear output, and logistic activation function with input vector t4	2.418064 42923172	2.413352 56991133	1.707515 06582013	1.720036 8858245
4	1	5	Utilizes a single hidden layer with 5 neurons, linear output, and logistic activation function with input vector t4	2.417084 58825687	2.412397 72989175	1.694154 35789788	1.707050 7861304 7
4	1	6	Utilizes a single hidden layer with 6 neurons, linear output, and logistic activation function with input vector t4	2.418081 45505477	2.413362 37797486	1.691523 90159221	1.703979 5390788 8
3	1	5	Utilizes a single hidden layer with 5 neurons,	2.415608 41323265	2.410747 45890346	1.734393 02663835	1.747855 9524145

			linear output, and logistic activation function with input vector t3				5
3	1	6	Utilizes a single hidden layer with 6 neurons, linear output, and logistic activation function with input vector t3	2.414873 02173552	2.409999 51024589	1.737013 27953228	1.750518 0676160 3
4	2	4	3	Utilizes two hidden layers with 4 and 3 neurons, linear output, and logistic activation function with input vector t4	2.419267 53357049	2.414557 17513488	1.717414 40313939	1.729513 8285392 8
4	2	5	4	Utilizes two hidden layers with 5 and 4 neurons, linear output, and logistic activation function with input vector t4	2.418632 66844199	2.413918 49568593	1.711156 03309535	1.723809 9165761 7
4	2	6	5	Utilizes two hidden layers with 6 and 5 neurons, linear output, and logistic activation function with input vector t4	2.419275 73571874	2.414444 41529708	1.727601 52726774	1.739346 607244
3	2	4	3	Utilizes two hidden layers with 4 and 3 neurons, linear output, and logistic activation function with input vector t3	2.415507 98123335	2.410688 49699973	1.730798 0520686	1.744376 0816521 9
3	2	5	4	Utilizes two hidden layers with 5 and 4 neurons, linear output, and logistic activation function with input vector t3	2.414243 79520913	2.409525 98224395	1.700140 69074406	1.714230 2770159 3
3	2	6	5	Utilizes two hidden layers with 6 and 5 neurons, linear output, and logistic activation	2.415071 72102798	2.410328 23788351	1.710274 72421165	1.723910 7747825 3

				function with input vector t3				
4	1	4		Utilizes a single hidden layer with 4 neurons, non-linear output, and tanh activation function with input vector t4	2.415120 21219716	2.410686 24485223	1.648548 78651747	1.662176 6361494 9

Table 1: Comparison Table

2.7 Efficiency of Best one-layer MLP and best two-layer MLP

To identify the best one-hidden layer and two-hidden layer networks, models with the lowest values across all performance metrics (RMSE, MAE, MAPE, sMAPE) were considered.

	RMSE	MAE	MAPE	sMAPE
results_1	2.41076330009686	2.40566943593568	1.78747537293032	1.80294791564061
results_2	2.41373198367413	2.40876868504137	1.75965326294425	1.77328930542062
results_3	2.41370049575528	2.40891621724223	1.70298308135791	1.71737027903997
results_4	2.41806442923172	2.41335256991133	1.70751506582013	1.7200368858245
results_5	2.41708458825687	2.41239772989175	1.69415435789788	1.70705078613047
results_6	2.41808145505477	2.41336237797486	1.69152390159221	1.70397953907888
results_7	2.41560841323265	2.41074745890346	1.73439302663835	1.74785595241455
results_8	2.41487302173552	2.40999951024589	1.73701327953228	1.75051806761603
results_9	2.41926753357049	2.41455717513488	1.71741440313939	1.72951382853928
results_10	2.41863266844199	2.41391849568593	1.71115603309535	1.72380991657617
results_11	2.41927573571874	2.41444441529708	1.72760152726774	1.739346607244
results_12	2.41550798123335	2.41068849699973	1.7307980520686	1.74437608165219
results_13	2.41424379520913	2.40952598224395	1.70014069074406	1.71423027701593
results_14	2.41507172102798	2.41032823788351	1.71027472421165	1.72391077478253
results_15	2.41512021219716	2.41068624485223	1.64854878651747	1.66217663614949

Figure 69: Performance metrics results for NNs

After analysis, the best one-hidden layer network was the NN Model with a tanh activation function and non-linear output using input vector t4. For the top-performing two-hidden layer network, NN Model with linear output and logistic activation function with input vector t3 stood out.

Considering efficiency, the complexity of the network structures, particularly in terms of the total number of weight parameters, was assessed. The one-hidden layer network proved more efficient, with a simpler architecture and fewer weight parameters compared to the two-hidden layer network. This simplicity can lead to computational efficiency gains and mitigate overfitting issues, making it a favorable choice, especially in scenarios with limited computing resources. Therefore, while both one-hidden layer and two-hidden layer networks have their strengths, the

simplicity and efficiency of the one-hidden layer NN Model with a tanh activation function and non-linear output using input vector t4 make it a more preferable option.

2.8 Results of the Best MLP Network

Using a scatter plot, the prediction output was graphically compared to the desired output. The plot illustrates the relationship between the desired output and the predicted output, with a red trendline indicating the linear regression fit.

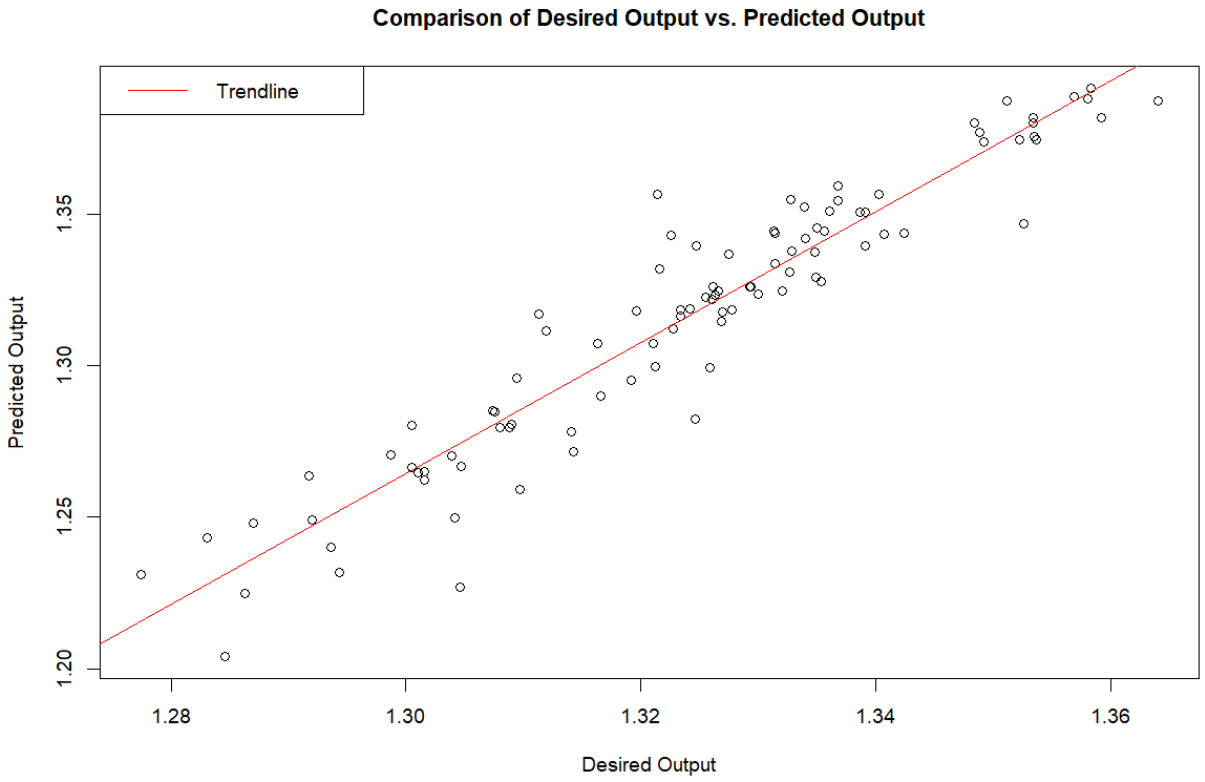

Figure 70: scatter plot

```
# Plotting the scatter plot
plot(test_original_pred$G_pred, pred_NN_Results,
     xlab = "Desired Output", ylab = "Predicted Output",
     main = "Comparison of Desired Output vs. Predicted Output")

# Adding a trendline
abline(lm(pred_NN_Results ~ test_original_pred$G_pred), col = "red")

# Adding a legend
legend("topleft", legend = "Trendline", col = "red", lty = 1)
```

Figure 71: scatter plot code

The performance indices for the best MLP network were evaluated. The results are as follows:

```
> cat("RMSE:", rmse_best, "\n")
RMSE: 2.41512
> cat("MAE:", mae_best, "\n")
MAE: 2.410686
> cat("MAPE:", mape_best, "%\n")
MAPE: 1.648549 %
> cat("sMAPE:", smape_best, "%\n")
sMAPE: 1.662177 %
```

Figure 72: Performance metrics

These statistical indices provide insights into the accuracy and performance of the best MLP network model.

References

1. Dautel, A. et al. (2020) *Forex exchange rate forecasting using deep recurrent neural networks*. Digit Finance. Available from https://doi.org/10.1007/s42521-020-00019-x [Accessed 22 March 2024]

2. Pfahler, J. (2022) Exchange Rate Forecasting with Advanced Machine Learning Methods. *Journal of Risk and Financial Management*. Available from https://doi.org/10.3390/jrfm15010002 [Accessed 25 March 2024]

Appendix A - Partition Clustering

Appendix A.1 - Partition Clustering Part Code

```
install.packages("readxl")
install.packages("NbClust")
install.packages("factoextra")
install.packages("cluster")

library(readxl)
library(NbClust)
library(factoextra)
library(cluster)

# SUBTASK 1

# Load the dataset
data <- read_excel("C:/Users/HP 15 - CS1032TX/Desktop/Whitewine_v6.xlsx")

# Separate features (attributes)
data <- data[, 1:11]

# Scale data
scaled_data <- scale(data)

# Visualize data before outlier removal
boxplot(scaled_data)

# Detect Outliers

# Calculate the quartiles for each column
q1 <- apply(scaled_data, 2, quantile, probs = c(0.25))
q3 <- apply(scaled_data, 2, quantile, probs = c(0.75))
iqr <- q3 - q1

# Calculate the upper and lower limits
upper <- q3 + 2.5 * iqr
lower <- q1 - 2.5 * iqr

# Identify the outliers
```

```
outliers <- apply(scaled_data, 2, function(x) x < lower | x > upper)

# Remove the outliers
data_no_outliers <- scaled_data[!apply(outliers, 1, any),]

# Visualize data after outlier removal
boxplot(data_no_outliers)

# Determine the Number of Cluster Centers

set.seed(42)

# NBclust
nb_clusters <- NbClust(data_no_outliers, distance = "euclidean", min.nc = 2,
                       max.nc = 10, method = "kmeans", index = "all")

# Elbow method
fviz_nbclust(data_no_outliers, kmeans, method = "wss")

# Silhouette method
fviz_nbclust(data_no_outliers, kmeans, method = "silhouette")

# Gap static method
fviz_nbclust(data_no_outliers, kmeans, method = "gap_stat")

# K Means

# Clustering using kmeans algorithm
kmeans_model <- kmeans(data_no_outliers, centers = 2)

kmeans_model

fviz_cluster(kmeans_model, data = data_no_outliers, palette = c("#2E9FDF", "#DB4035",
"#E7B800", "#b800e7"),
             geom = "point",
             ellipse.type = "convex",
             ggtheme = theme_bw())


wss <- kmeans_model$tot.withinss
```

```
bss <- kmeans_model$betweenss
tss <- wss + bss
bss_ratio <- bss / tss

wss
bss
tss
bss_ratio

# Silhouette
sil <- silhouette(kmeans_model$cluster, dist(data_no_outliers))
fviz_silhouette(sil)


# SUBTASK 2

# PCA Analysis
pca_result <- prcomp(data_no_outliers, center = TRUE, scale = TRUE)
summary(pca_result)

# The cumulative proportion exceeds 85% after PC7

# transformed dataset
data_transformed <- as.data.frame(-pca_result$x[,1:7])
head(data_transformed)

View(data_transformed)

# NBclust
nb_clusters <- NbClust(data_transformed, distance = "euclidean", min.nc = 2,
                       max.nc = 10, method = "kmeans", index = "all")

# Elbow method
fviz_nbclust(data_transformed, kmeans, method = "wss")

# Silhouette method
fviz_nbclust(data_transformed, kmeans, method = "silhouette")

# Gap static method
fviz_nbclust(data_transformed, kmeans, method = "gap_stat")
```

```
# Clustering using kmeans algorithm
kmeans_model <- kmeans(data_transformed, centers = 2)

kmeans_model

fviz_cluster(kmeans_model, data = data_transformed, palette = c("#2E9FDF", "#E7B800",
"#00AFBB",  "#b800e7"),
             geom = "point",
             ellipse.type = "convex",
             ggtheme = theme_bw())


wss <- kmeans_model$tot.withinss
bss <- kmeans_model$betweenss
tss <- wss + bss
bss_ratio <- bss / tss

wss
bss
tss
bss_ratio

# Silhouette
sil <- silhouette(kmeans_model$cluster, dist(data_transformed))
fviz_silhouette(sil)

install.packages("fpc")
library(fpc)

# Obtain the cluster assignments
clustering <- kmeans_model$cluster

# Compute the Calinski-Harabasz Index
calinski_harabasz <- calinhara(data_transformed, clustering)

# Print the Calinski-Harabasz Index
print(calinski_harabasz)
```

```
# Compute Calinski-Harabasz Index for kmeans clustering with varying number of clusters
calinski_results <- numeric(10)
for (k in 2:10) {
  kmeans_model <- kmeans(data_transformed, centers = k)
  clustering <- kmeans_model$cluster
  calinski_results[k] <- calinhara(data_transformed, clustering)
}

# Visualize the index
plot(2:10, calinski_results[2:10], type = "b", xlab = "Number of Clusters", ylab =
"Calinski-Harabasz Index")
```

Appendix B - Financial Forecasting

Appendix B.1 - Financial Forecasting Part Code

```
# Install required packages
install.packages("readxl")
install.packages("neuralnet")
install.packages("dplyr")
install.packages("Metrics")
install.packages("MLmetrics")

# Load necessary libraries
library(readxl)
library(neuralnet)
library(dplyr)
library(Metrics)
library(MLmetrics)
library(ggplot2)

# Load data
exchange_data <- read_excel("D:/ML/ExchangeUSD.xlsx")

# Extract the third column
exchange_rates <- exchange_data$`USD/EUR`
exchange_rates_df <- data.frame(exchange_rates)

# Split data into training and testing sets
train_data <- exchange_rates_df[1:400,]
test_data <- exchange_rates_df[401:length(exchange_rates),]

# Creating lagged input-output matrix for training data set t4
train_lagged_t4 <- bind_cols(G_prev3 = lag(train_data,4),
                        G_prev2 = lag(train_data,3),
                        G_prev = lag(train_data,2),
                        G_curr = lag(train_data,1),
                        G_pred = train_data)

# Creating lagged input-output matrix for testing data set t4
test_lagged_t4 <- bind_cols(G_prev3 = lag(test_data,4),
                       G_prev2 = lag(test_data,3),
```

```
                    G_prev = lag(test_data,2),
                    G_curr = lag(test_data,1),
                    G_pred = test_data)

# Creating lagged input-output matrix for training data set t3
train_lagged_t3 <- bind_cols(G_prev2 = lag(train_data,3),
                     G_prev = lag(train_data,2),
                     G_curr = lag(train_data,1),
                     G_pred = train_data)

# Creating lagged input-output matrix for testing data set t3
test_lagged_t3 <- bind_cols(G_prev2 = lag(test_data,3),
                    G_prev = lag(test_data,2),
                    G_curr = lag(test_data,1),
                    G_pred = test_data)

# Creating lagged input-output matrix for training data set t2
train_lagged_t2 <- bind_cols(G_prev = lag(train_data,2),
                     G_curr = lag(train_data,1),
                     G_pred = train_data)

# Creating lagged input-output matrix for testing data set t2
test_lagged_t2 <- bind_cols(G_prev = lag(test_data,2),
                    G_curr = lag(test_data,1),
                    G_pred = test_data)

# Creating lagged input-output matrix for training data set t1
train_lagged_t1 <- bind_cols(G_curr = lag(train_data,1),
                     G_pred = train_data)

# Creating lagged input-output matrix for testing data set t1
test_lagged_t1 <- bind_cols(G_curr = lag(test_data,1),
                    G_pred = test_data)

# Display the input-output matrices as tables
View(train_lagged_t4)
View(test_lagged_t4)

# Remove N/A values from t4
train_lagged_t4 <- train_lagged_t4[complete.cases(train_lagged_t4),]
```

```
test_lagged_t4 <- test_lagged_t4[complete.cases(test_lagged_t4),]

# Display the cleaned input-output matrices as tables
View(train_lagged_t4)
View(test_lagged_t4)

# Remove N/A values from t3
train_lagged_t3 <- train_lagged_t3[complete.cases(train_lagged_t3),]
test_lagged_t3 <- test_lagged_t3[complete.cases(test_lagged_t3),]

# Remove N/A values from t2
train_lagged_t2 <- train_lagged_t2[complete.cases(train_lagged_t2),]
test_lagged_t2 <- test_lagged_t2[complete.cases(test_lagged_t2),]

# Remove N/A values from t1
train_lagged_t1 <- train_lagged_t1[complete.cases(train_lagged_t1),]
test_lagged_t1 <- test_lagged_t1[complete.cases(test_lagged_t1),]

# Function to normalize data
normalize_data <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}

# Normalize data for t4
train_normalized_t4 <- as.data.frame(lapply(train_lagged_t4, normalize_data))
test_normalized_t4 <- as.data.frame(lapply(test_lagged_t4, normalize_data))

# Normalize data for t3
train_normalized_t3 <- as.data.frame(lapply(train_lagged_t3, normalize_data))
test_normalized_t3 <- as.data.frame(lapply(test_lagged_t3, normalize_data))

# Normalize data for t2
train_normalized_t2 <- as.data.frame(lapply(train_lagged_t2, normalize_data))
test_normalized_t2 <- as.data.frame(lapply(test_lagged_t2, normalize_data))

# Normalize data for t1
train_normalized_t1 <- as.data.frame(lapply(train_lagged_t1, normalize_data))
test_normalized_t1 <- as.data.frame(lapply(test_lagged_t1, normalize_data))

# Plot boxplot for normalized training data and original(t4)
```

```
boxplot(train_normalized_t4)
boxplot(train_lagged_t4)

# Get original data from denormalized data
train_original_pred <- train_lagged_t4["G_pred"]
test_original_pred_t1 <- test_lagged_t1["G_pred"]
test_original_pred_t2 <- test_lagged_t2["G_pred"]
test_original_pred_t3 <- test_lagged_t3["G_pred"]
test_original_pred_t4 <- test_lagged_t4["G_pred"]

View(test_original_pred)

# Find min and max value from dataset
min_val <- min(train_original_pred)
max_val <- max(train_original_pred)

# Create function to denormalize data
denormalize_data <- function(x) {
  min_val <- min(train_original_pred)
  max_val <- max(train_original_pred)
  return( (max_val - min_val) * x + min_val )
}

# train and test neural network function
train_test_neural_net <- function(training_data, testing_data, neurons_hidden_layer,
linear_output, activation_function, test_original_pred) {

  set.seed(123)

  formula_names <- as.formula(paste("G_pred~",
paste(names(training_data)[-ncol(training_data)], collapse = "+")))

  #Training model
  if (is.null(activation_function)) {
    rates_load_nn <- neuralnet(formula_names,
                               hidden=neurons_hidden_layer,
                               data=training_data,
                               linear.output=linear_output)
  } else {
    rates_load_nn <- neuralnet(formula_names,
```

```
                    hidden=neurons_hidden_layer,
                    data=training_data,
                    act.fct = activation_function,
                    linear.output=linear_output)
}
plot(rates_load_nn)

# Extract column names for testing
predict_label_removed <- setdiff(names(training_data), "G_pred")

# Testing model

# Removing Prediction variable
temp_test <- subset(testing_data, select=predict_label_removed)
head(temp_test)

# Creating predicted variables
model_results <- compute(rates_load_nn, temp_test)
predicted_results_norm <- model_results$net.result
#View(predicted_results_norm)

# unnormalize NN predicted data
pred_NN_Results <- denormalize_data(predicted_results_norm)
#View(pred_NN_Results)

par(mfrow = c(1, 2))  # Splitting the plotting area into 1 row and 2 columns
plot(rates_load_nn)
# Plotting the scatter plot
plot(test_original_pred$G_pred, pred_NN_Results,
    xlab = "Desired Output", ylab = "Predicted Output",
    main = "Comparison of Desired Output vs. Predicted Output")

# Adding a trendline
abline(lm(pred_NN_Results ~ test_original_pred$G_pred), col = "red")

# Adding a legend
legend("topleft", legend = "Trendline", col = "red", lty = 1)

# testing performance of RMSE
rmse <- rmse(exp(pred_NN_Results),test_original_pred$G_pred)
```

```
  # testing performance of MAE
  mae <- mae(exp(pred_NN_Results),test_original_pred$G_pred)

  # testing performance of MAPE
  mape <- mean(abs((test_original_pred$G_pred -
pred_NN_Results)/test_original_pred$G_pred))*100

  # testing performance of sMAPE
  smape <- mean(2 * abs(test_original_pred$G_pred - pred_NN_Results) /
(abs(test_original_pred$G_pred) + abs(pred_NN_Results))) * 100


  return(list(RMSE = rmse, MAE = mae, MAPE = mape, sMAPE = smape))
}

# Train and test neural network models with different input vectors and structures
results_1 <- train_test_neural_net(train_normalized_t1, test_normalized_t1, c(4), TRUE, NULL,
test_original_pred_t1)
results_2 <- train_test_neural_net(train_normalized_t2, test_normalized_t2, c(4), TRUE, NULL,
test_original_pred_t2)
results_3 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(4), TRUE, NULL,
test_original_pred_t3)
results_4 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4), TRUE, NULL,
test_original_pred_t4)
results_5 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(5), TRUE, NULL,
test_original_pred_t4)
results_6 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(6), TRUE, NULL,
test_original_pred_t4)
results_7 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(5), TRUE, NULL,
test_original_pred_t3)
results_8 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(6), TRUE, NULL,
test_original_pred_t3)
results_9 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4,3), TRUE,
NULL, test_original_pred_t4)
results_10 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(5,4), TRUE,
NULL, test_original_pred_t4)
results_11 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(6,5), TRUE,
NULL, test_original_pred_t4)
```

```
results_12 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(4,3), TRUE, NULL, test_original_pred_t3)
results_13 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(5,4), TRUE, NULL, test_original_pred_t3)
results_14 <- train_test_neural_net(train_normalized_t3, test_normalized_t3, c(6,5), TRUE, NULL, test_original_pred_t3)
results_15 <- train_test_neural_net(train_normalized_t4, test_normalized_t4, c(4), FALSE, "tanh", test_original_pred_t4)


# Combine metrics into a single data frame
all_results <- rbind(results_1, results_2, results_3, results_4, results_5, results_6, results_7, results_8, results_9, results_10, results_11, results_12, results_13, results_14, results_15)

# Display the table
View(all_results)

# Extracting the statistical indices for the best MLP network (results_15)
rmse_best <- results_15$RMSE
mae_best <- results_15$MAE
mape_best <- results_15$MAPE
smape_best <- results_15$sMAPE

# Printing out the statistical indices
cat("RMSE:", rmse_best, "\n")
cat("MAE:", mae_best, "\n")
cat("MAPE:", mape_best, "%\n")
cat("sMAPE:", smape_best, "%\n")
```

Appendix B.2 - Implementation of different MLPs

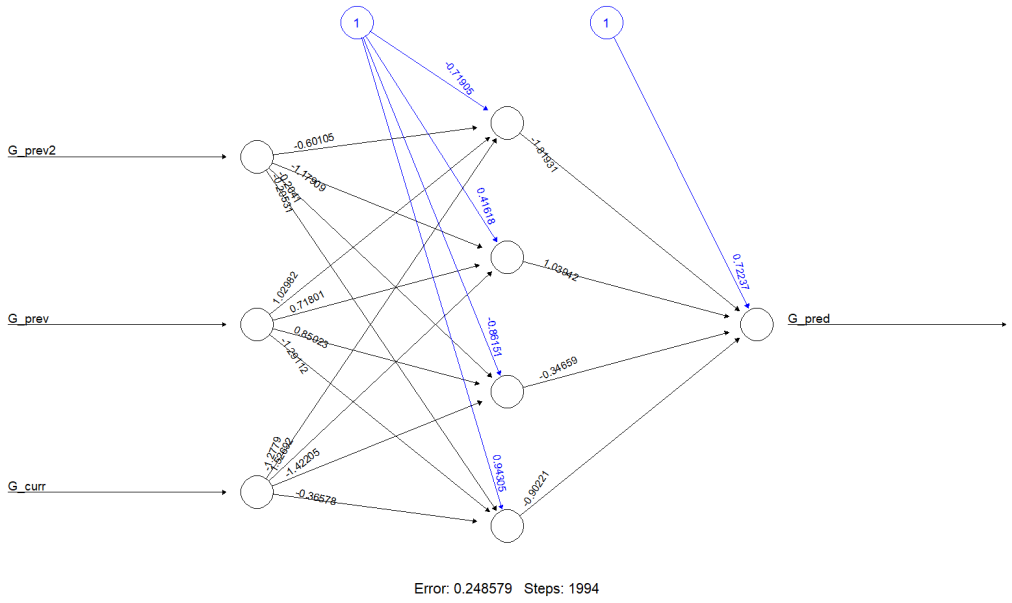

Figure 73: 3 input 1 hidden layer 4 neurons NN

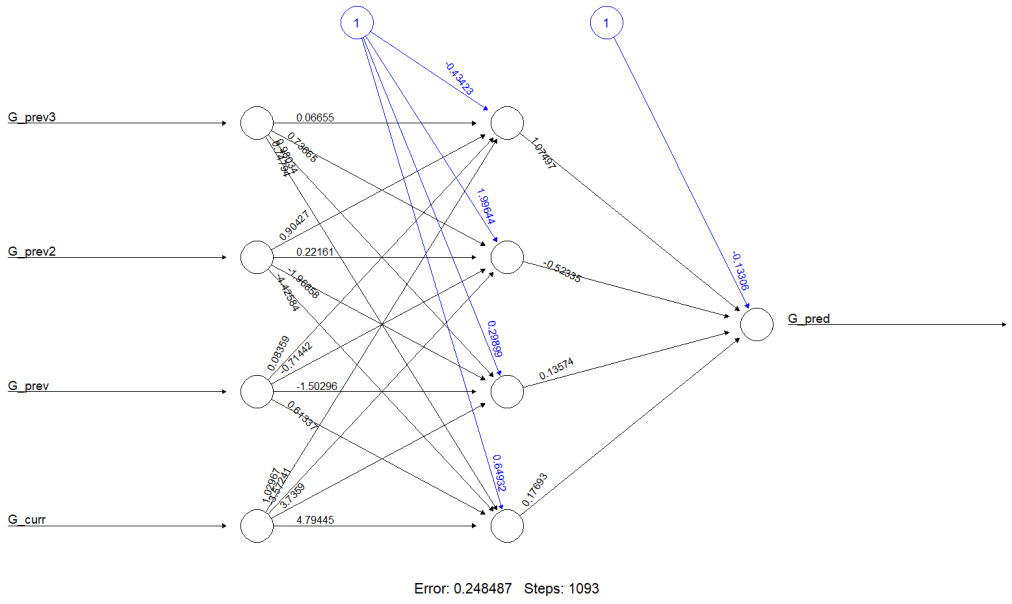

Figure 74: 4 input 1 hidden layer 4 neurons NN

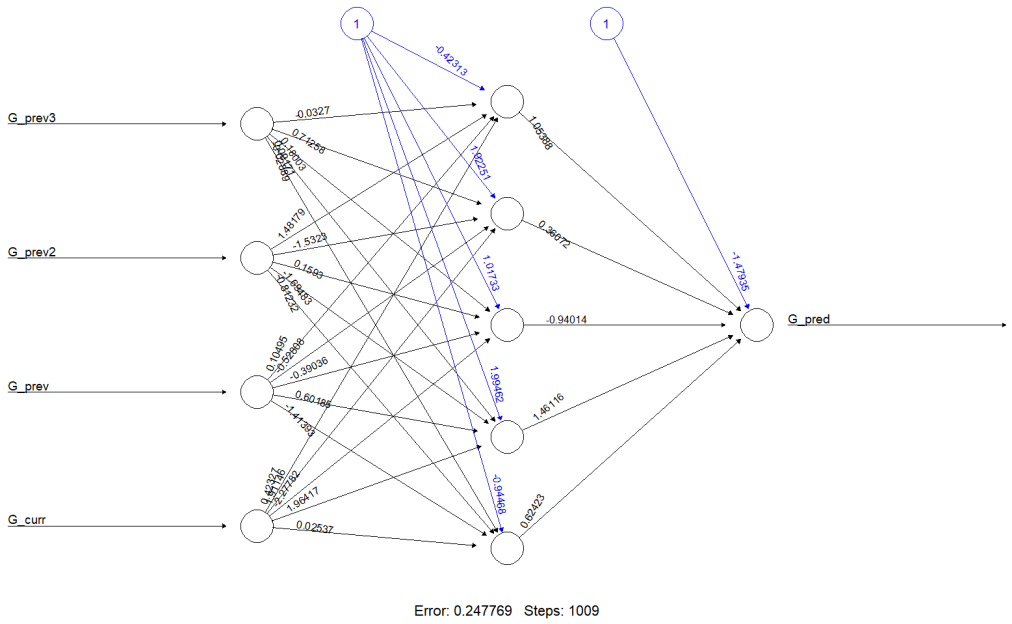

Figure 75: 4 input 1 hidden layer 5 neurons NN

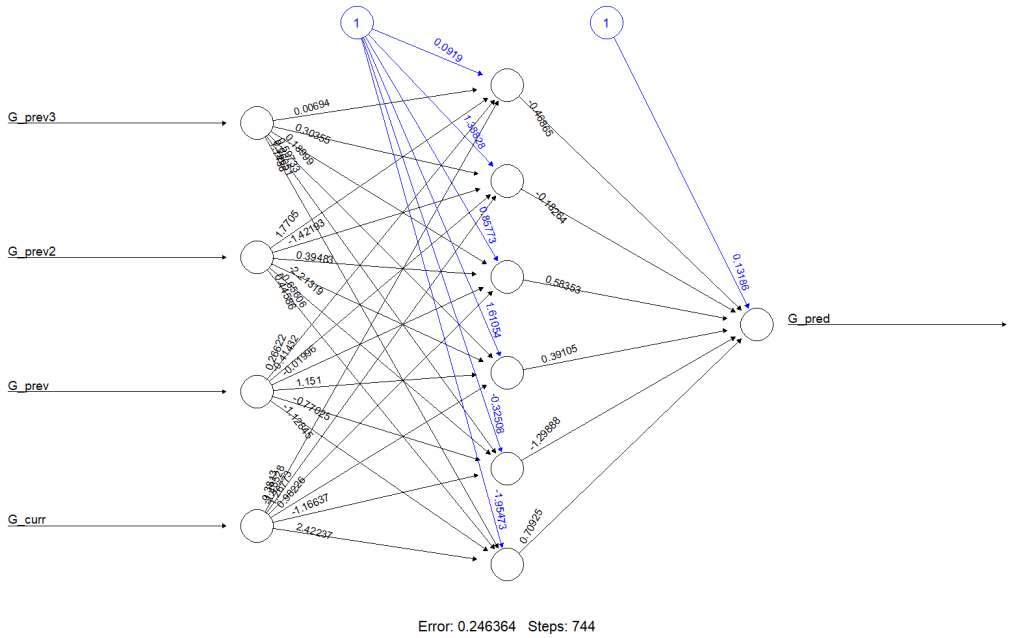

Figure 76: 4 input 1 hidden layer 6 neurons NN

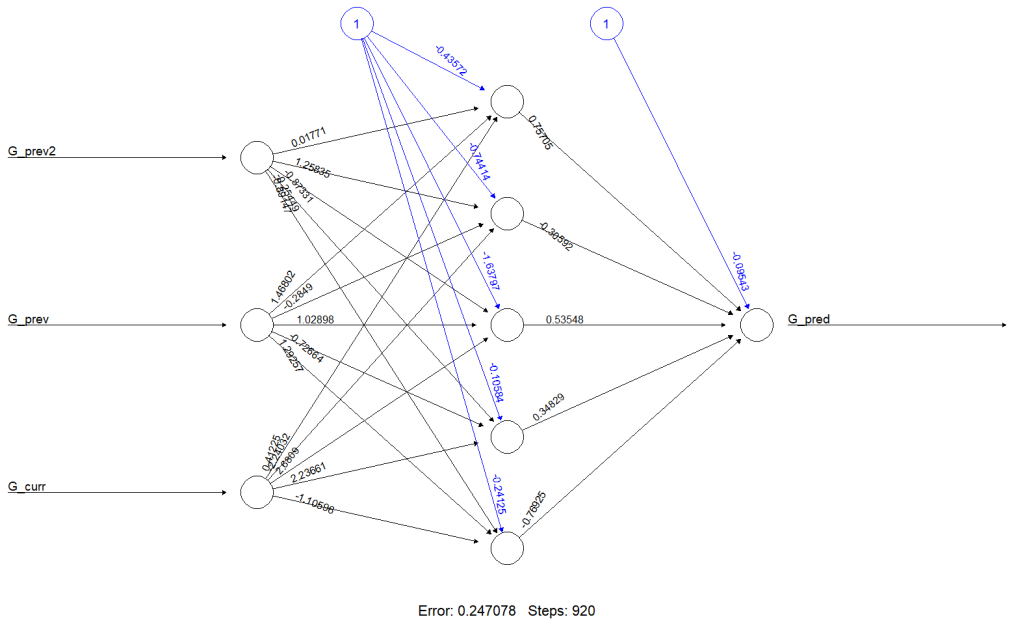

Figure 77: 3 input 1 hidden layer 5 neurons NN

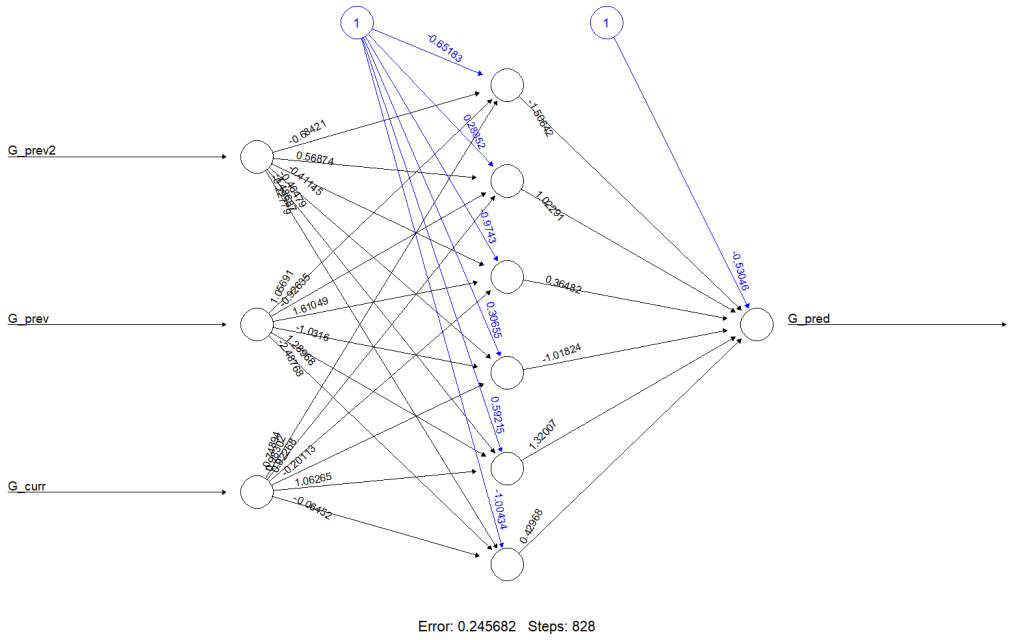

Figure 78: 3 input 1 hidden layer 6 neurons NN

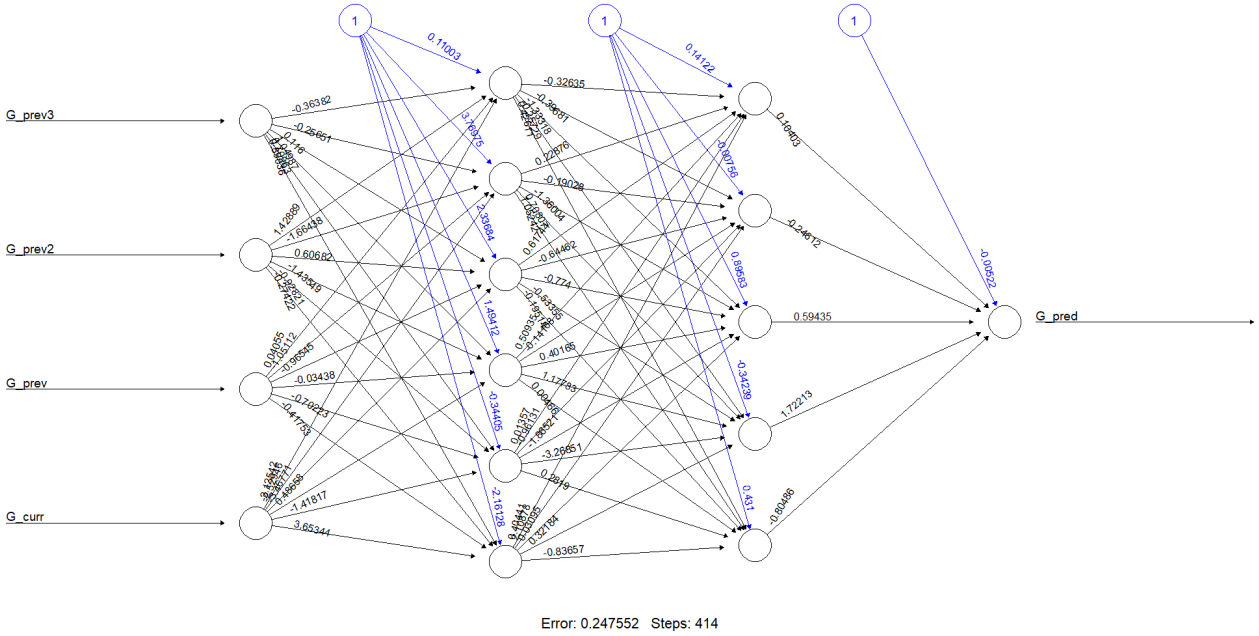

Figure 79: 4 input 2 hidden layer 6,5 neurons NN

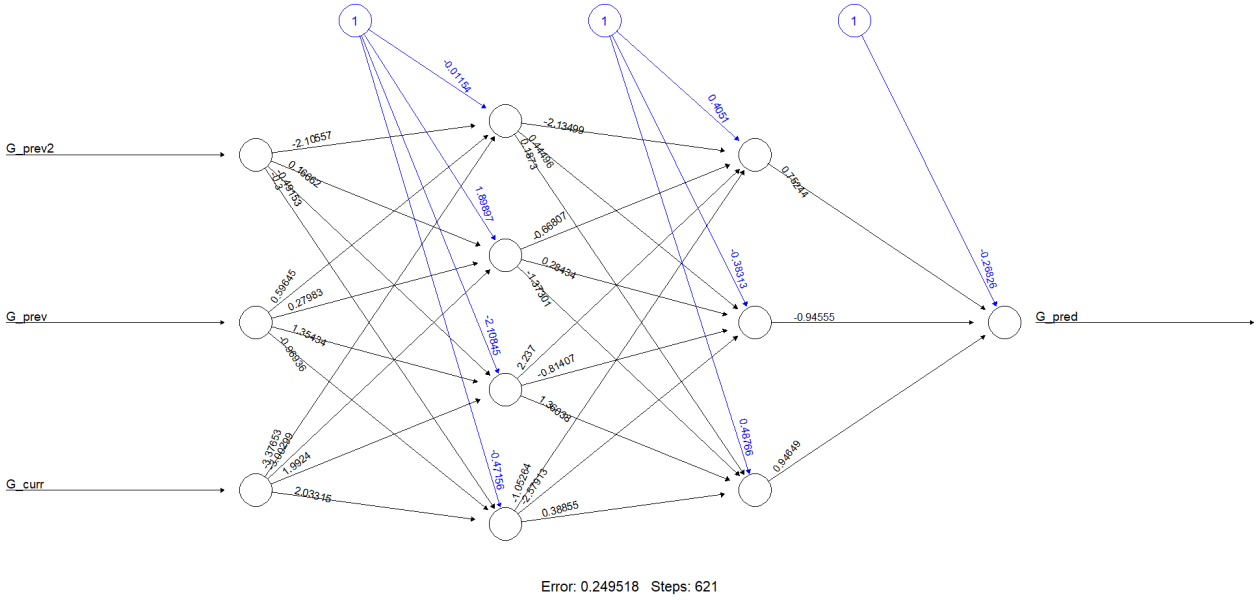

Figure 80: 3 input 2 hidden layer 4,3 neurons NN

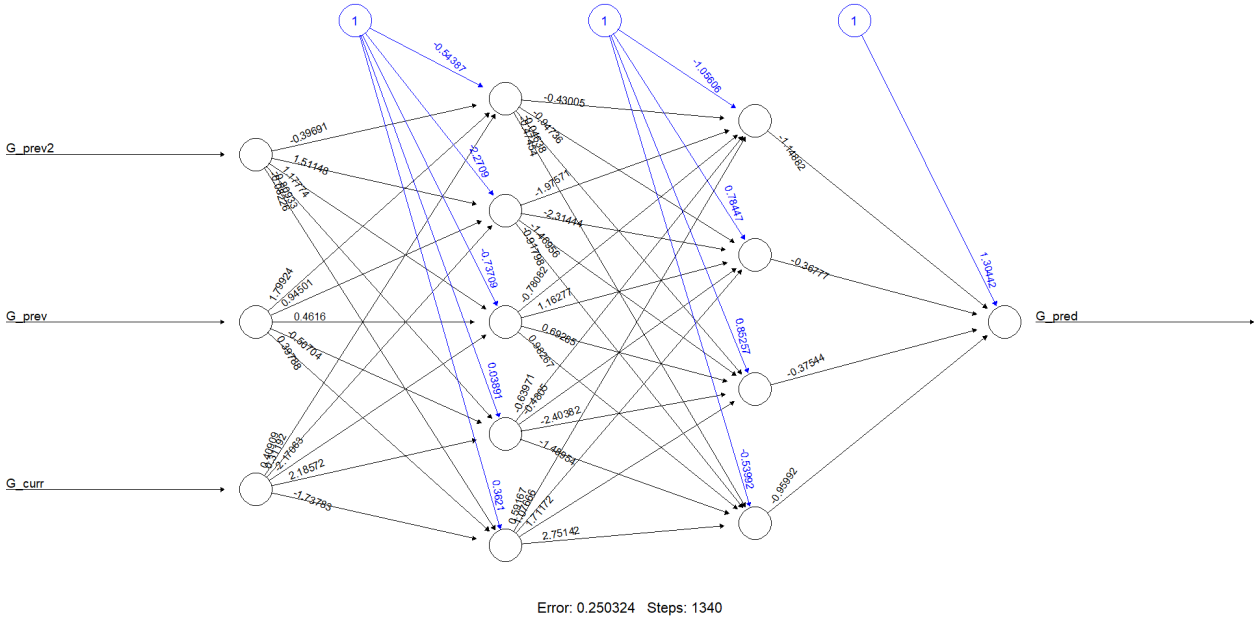

Figure 81: 3 input 2 hidden layer 5,4 neurons NN

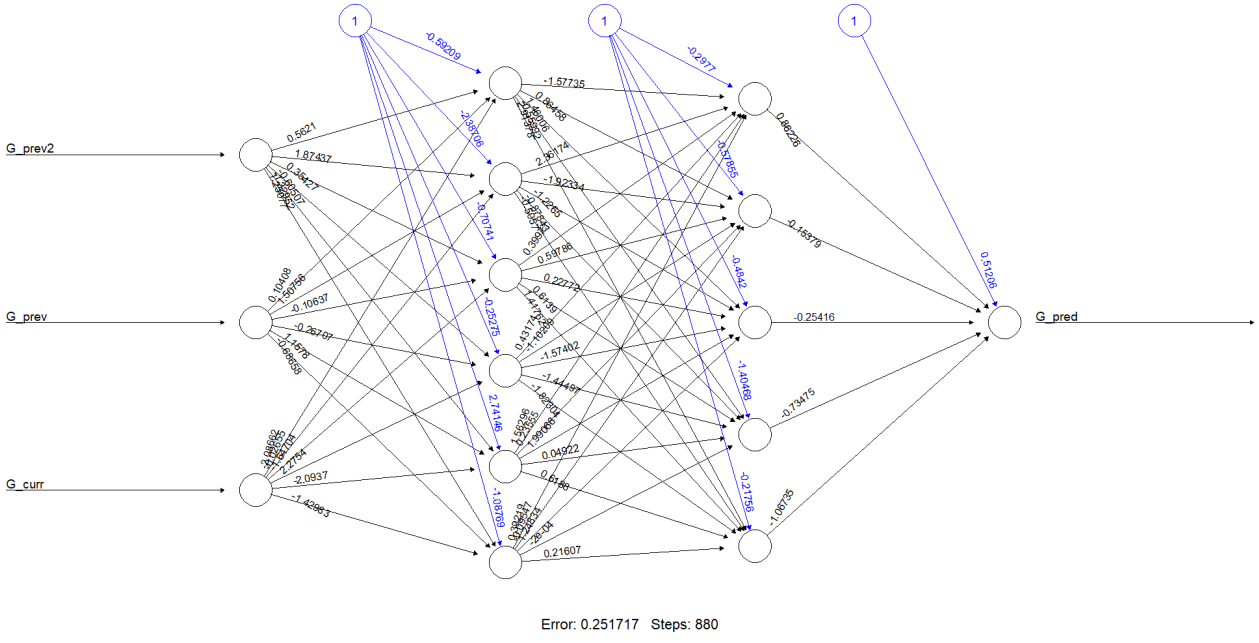

Figure 82: 3 input 2 hidden layer 6,5 neurons NN