Machine Learning and Data Mining

K-MEANS, MLP

Table of Contents

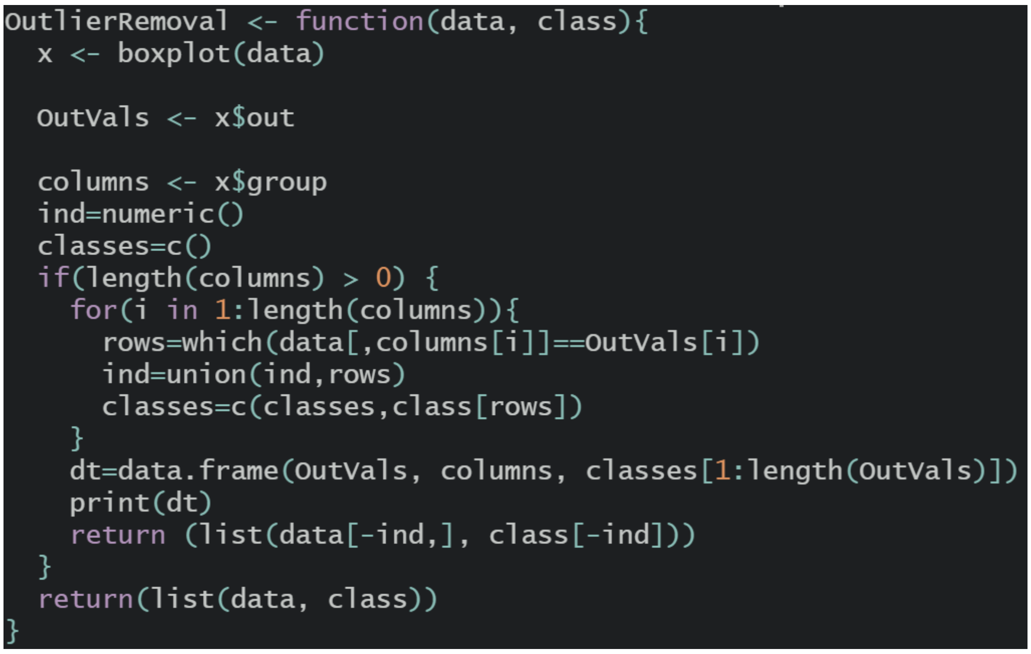
Dimensionality reduction

Data in today’s day and age can be quite easy to collect and can begin to accumulate at an unprecedented speed. While the accumulation of data can lead to more accurate predictions, after a certain point the performance and accuracy of the algorithms will begin to fall as the dimensionality of the data continues to grow. This phenomenon is commonly referred to as the “Curse of Dimensionality”. A set of problems begin to arise when working with high dimensionality data. For example, Distance concentration. This a problem that can arise as a result of high dimensionality data (facet of Curse of Dimensionality). Distance concentration refers to the issue where the pairwise distances between samples begin to merge to the same value as the dimensionality of our data continues to increase. This is because the proximity of the observations (result from distance metric) begins to become less important and relevant in high dimensional data. This can cause significant problems when performing analysis using various machine learning techniques such as clustering, using machine learning models such as k-means or k-Nearest Neighbours because these two methods both use distance metric calculations such as Euclidean, Manhattan etc to cluster/group samples. Data reduction techniques aim to provide solutions to such problems. Data reduction techniques are applied to obtain a reduced representation of the data which would be smaller in size, while also maintaining the quality of the data. This will enable the algorithm to maintain good time complexity and the ability to produce accurate results. An example of a dimensionality reduction technique is the Principal Component Analysis (PCA). PCA is by far the most popular dimensionality reduction algorithm. PCA aims to take a high dimensionality data set and find a low dimensionality representation that contains the largest amount of variance without losing much of the information. Each observation within our database is in dimensional space, however, is it quickly evident that not all of these dimensions within our data are equally important. PCA aims to have a small number of dimensions that are as varied as possible by creating Principal Components (PC) which represent the variance within the data with less dimensions. Whether a PC is considered “important” or “interesting” is dependent on the amount that the observation varies along each dimension. The first principal component in the resulting PCs will contain the data with the highest amount of variance, with the following Principal Component containing the second most variance in the data. As the Principal Components (PC) increase, the variation in the data decreases. For example, PC1 contains 61% variability, PC2 contains 27% and PC3 contains 5%. The combination of PC1 & PC2 explains 88% of the variability. This reduced representation of the data set can dramatically improve performance and maintain high accuracy.

Pre-Processing

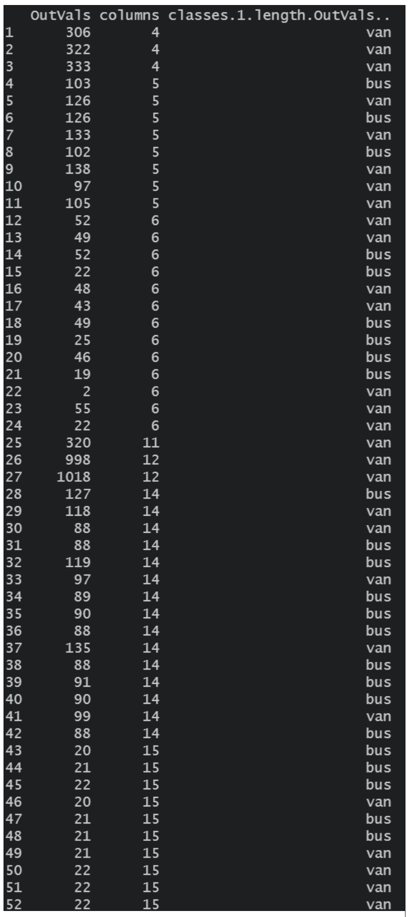
Data pre-processing is an important part of effective machine learning and data mining that involves transforming raw data into an understandable format. This will directly affect the ability of our machine learning model to learn.

The vehicles dataset required two main pre-processing tasks to be carried out to ensure we allowed our machine learning algorithm to correctly perform and produce reliable results. These two tasks included normalization in the form of scaling and outlier removal. The vehicles dataset contained many outliers which would produce an inaccurate and unreliable result, especially since the K-means algorithm is incredibly sensitive to outliers. Outliers in the dataset can affect the accuracy and reliability of the results. This is because an object that is larger in value to other points can distort the distribution of the data, which will produce inaccurate groupings. This is because the outlier can be randomly selected as the initial seed point. This will then be used to calculate distances to other data points to group/cluster which will result in poor accuracy and make the results unreliable. Furthermore, even if the initial centroid is not an outlier, when the centroids are recalculated and distances are calculated again against the data points, the outlier will eventually be clustered based on its distance and will be clustered incorrectly. This will result in an unreliable and inaccurate result, ultimately making the use of the K-means technique on the data ineffective.



```r
OutlierRemoval <- function(data, class){
  x <- boxplot(data)

  OutVals <- x$out

  columns <- x$group
  ind=numeric()
  classes=c()
  if(length(columns) > 0) {
    for(i in 1:length(columns)){
      rows=which(data[,columns[i]]==OutVals[i])
      ind=union(ind,rows)
      classes=c(classes,class[rows])
    }
    dt=data.frame(OutVals, columns, classes[1:length(OutVals)])
    print(dt)
    return (list(data[-ind,], class[-ind]))
  }
  return(list(data, class))
}
```

Figure 1: Outlier removal function in R



```
   OutVals columns classes.1.length.OutVals..
1      306       4                        van
2      322       4                        van
3      333       4                        van
4      103       5                        bus
5      126       5                        van
6      126       5                        bus
7      133       5                        van
8      102       5                        bus
9      138       5                        van
10      97       5                        van
11     105       5                        van
12      52       6                        van
13      49       6                        van
14      52       6                        bus
15      22       6                        bus
16      48       6                        van
17      43       6                        van
18      49       6                        bus
19      25       6                        bus
20      46       6                        bus
21      19       6                        bus
22       2       6                        van
23      55       6                        van
24      22       6                        van
25     320      11                        van
26     998      12                        van
27    1018      12                        van
28     127      14                        bus
29     118      14                        van
30      88      14                        van
31      88      14                        bus
32     119      14                        bus
33      97      14                        van
34      89      14                        bus
35      90      14                        bus
36      88      14                        bus
37     135      14                        van
38      88      14                        bus
39      91      14                        bus
40      90      14                        bus
41      99      14                        van
42      88      14                        bus
43      20      15                        bus
44      21      15                        bus
45      22      15                        bus
46      20      15                        van
47      21      15                        bus
48      21      15                        bus
49      21      15                        van
50      22      15                        van
51      22      15                        van
52      22      15                        van
```

Figure 1.1: Outliers found and from which class

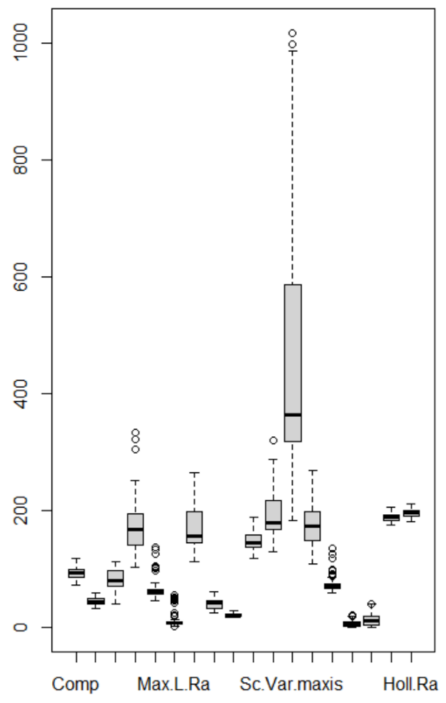


Figure 1.2: Before Outlier removal

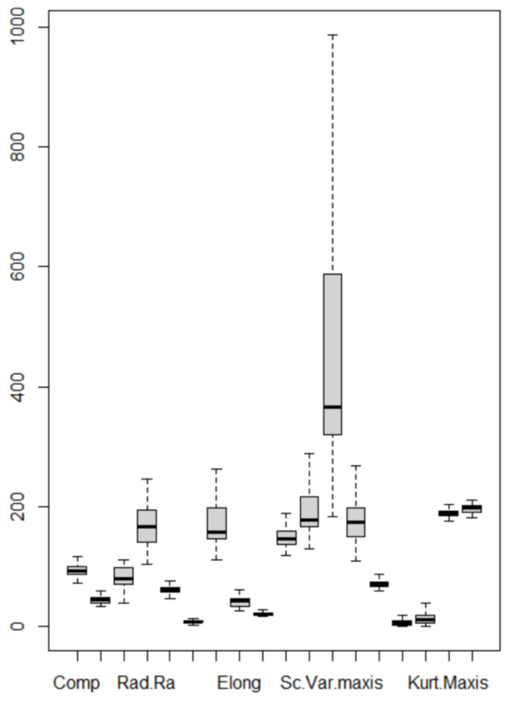
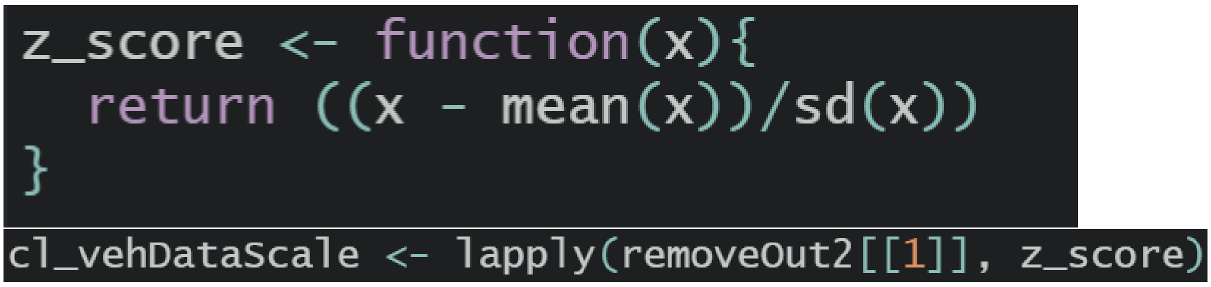


Figure 1.3: Outliers removed

To handle and remove the outliers from the dataset, I have created an OutlierRemoval function which uses the boxplot function, from which we can extract the outliers through the "$Out" component which will return the outlier values contained inside the dataset (data points which lie beyond the extremes of the whiskers). In order to remove the outliers from our dataset, we simply store the outliers index retrieved from the which() function and pass this information to our "Ind" variable which we will then use to return the data without those indexes by doing "data[-ind,]" which will remove those outlier values from our dataset. The OutlierRemoval function also returns a data frame which displays the outliers, from which column they belonged to and the classes they belonged to as evident in **Figure 1.1.** This is achieved by also passing the vehicle classes (stored inside "VehClass" variable) to the function as well, retrieve the index of the outlier through the which() function when the following condition is met: "which(data[,columns[i]]==OutVals[i]) from which we can then identify the class by indexing the vehicle classes using the rows variable "class[rows]". We concatenate these classes into the "classes variable" using the c() function and display them through a loop in the data frame. **Figure 1.2** displays the vehicle data before the removal of the outliers and **Figure 1.3** displays the vehicle data after the outliers have been removed.

The vehicles dataset also required normalization in the form of scaling. The goal of normalization is to ensure every data point has the same scale, so each feature is equally important. The data in the vehicle's dataset were on different scales. This will result in inaccuracy because when the algorithm begins to compare the data points, the features that are on a larger scale will completely dominate the other features, producing inaccurate and unreliable results.



```
z_score <- function(x){
  return ((x - mean(x))/sd(x))
}
cl_vehDataScale <- lapply(removeOut2[[1]], z_score)
```

Figure 1.4: Scaling function and code execute

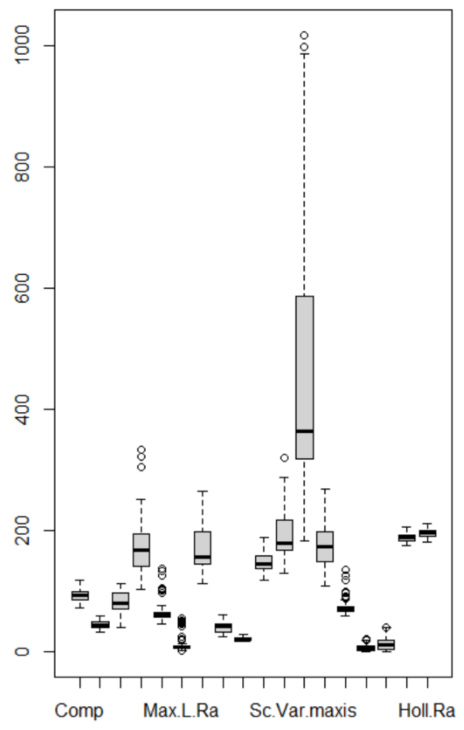


Figure 1.5: Before Scaling

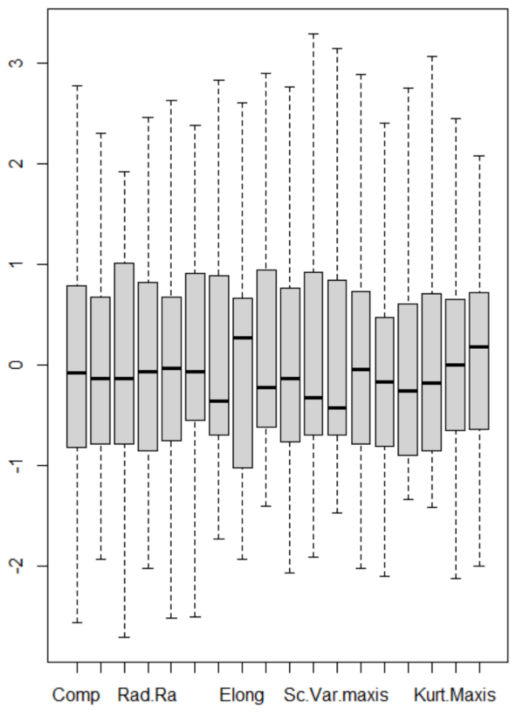


Figure 1.6: After Scaling

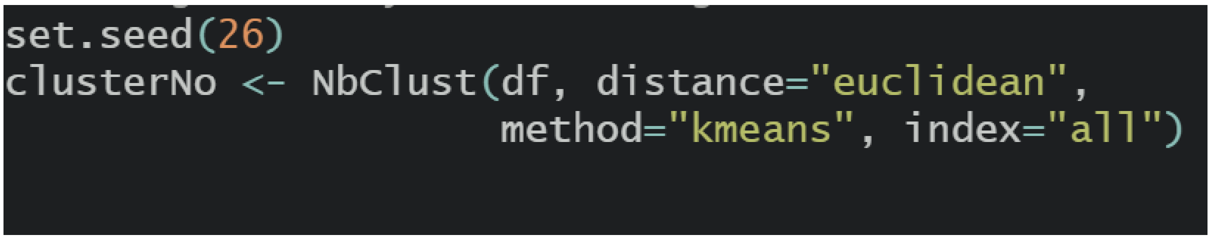
In order to scale the values, we could create our own scaling function (z-score) or use the build in R function scale() which would scale the values in the same way. I have made my own scaling function which performs z-score scaling. Z-score scaling measures exactly how many standard deviations above or below the mean a data point is. This will transform our data in such a way that the resulting distribution would have a mean of 0 and a standard deviation of 1. This will scale the features in our data and prevent features with a larger scale from dominating other features.

K-means number of cluster centres exploration

NbClust

One of the downsides of the K-means algorithm is that it requires prior knowledge of the cluster numbers beforehand i.e., defining the K for the k-means algorithm. This can be difficult to know especially since K-means is an unsupervised machine learning technique (no labels), meaning it is difficult to know how many K (clusters) we would need to cluster the data. Luckily for us there are some methods and tools such as NbClust and the Elbow method (automated tools) which we can use within R that can help and assist in the process. These can be used alongside our own manual experimentation to find the best possible K for our dataset.

NbClust cluster identification (Euclidean)



```
set.seed(26)
clusterNo <- NbClust(df, distance="euclidean",
                     method="kmeans", index="all")
```

Figure 1.7: NbClust R code

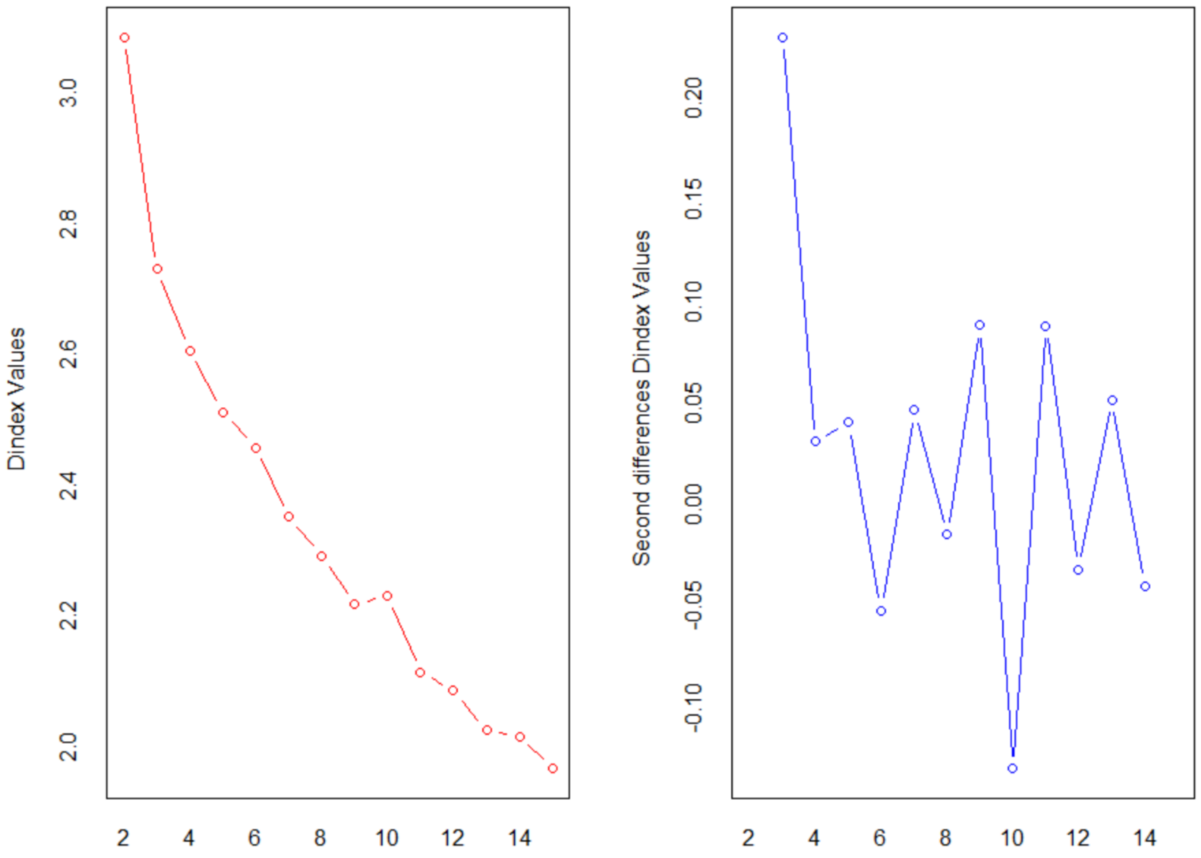
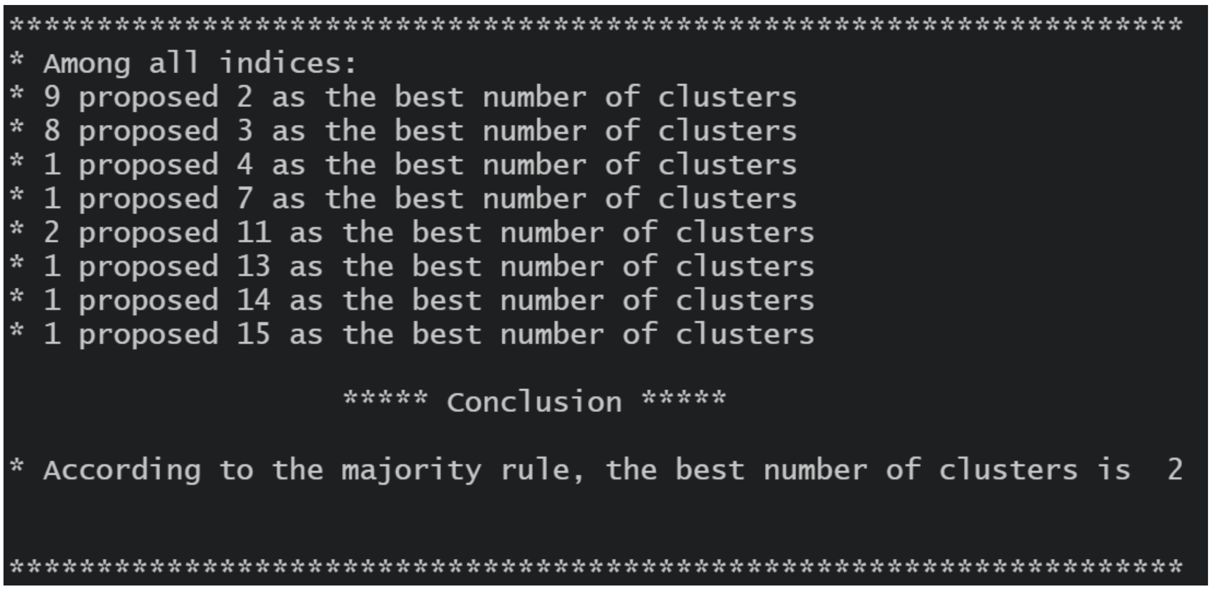


Figure 1.8: NbClust graph result



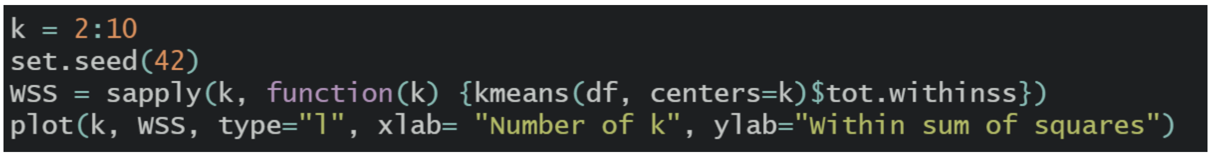
```
*******************************************************************
* Among all indices:
* 9 proposed 2 as the best number of clusters
* 8 proposed 3 as the best number of clusters
* 1 proposed 4 as the best number of clusters
* 1 proposed 7 as the best number of clusters
* 2 proposed 11 as the best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

                   ***** Conclusion *****

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


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

Figure 1.9: Suggested cluster number from NbClust

The NbClust function can be used to determine the best number of clusters. We provide our data and the distance metric we would like the clustering algorithm to use and the method which refers to the algorithm being used. In our case we are using Euclidean distance and the k-means algorithm. Finally, we specify to use “all” indexes to test against all 30 indices provided by the NbCust package. The function will return a result displaying the best number of clusters from top to bottom where the first result is the best number of clusters to use, proposed the most among the indices. From this result, we can see the best number of clusters to use is 2, proposed by 9 indices, closely followed by 3 clusters proposed by 8 indices. This is also evident in **Figure 1.8** where the bend of the begins between and 2 and 3.

Elbow method

The elbow method is another technique that can be used to identify the best number of clusters/K to use for our K-means clustering algorithm. The elbow method calculates the within sum of squares. This is a measurement of the variability of the data within each cluster. If the cluster has a small sum of squares, it is more closely packed than one that has a higher sum of squares. A cluster which contains a higher value would have greater variability of the data within the cluster. A higher variability within a cluster is not what we would want since we are looking to cluster objects, which means to group based on similarity of the data. After calculating the sum of squares, we can then plot the within sum of squares with the different number of K to find the best number of K by identifying a bend or a knee in the plot which is considered the indicator of an appropriate number of clusters.



```r
k = 2:10
set.seed(42)
WSS = sapply(k, function(k) {kmeans(df, centers=k)$tot.withinss})
plot(k, WSS, type="l", xlab= "Number of k", ylab="Within sum of squares")
```

Figure 2.0: Elbow method R code

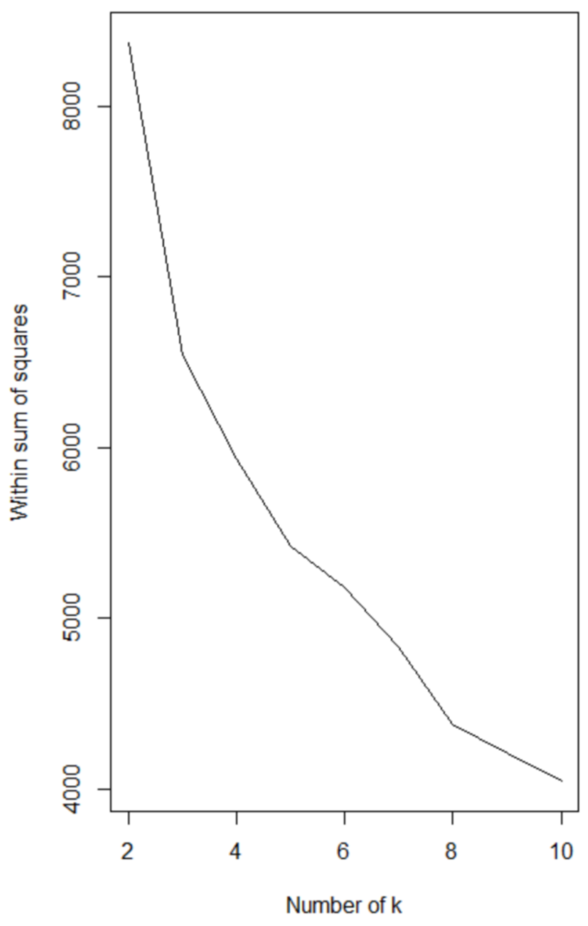


Figure 2.1: WSS Plot

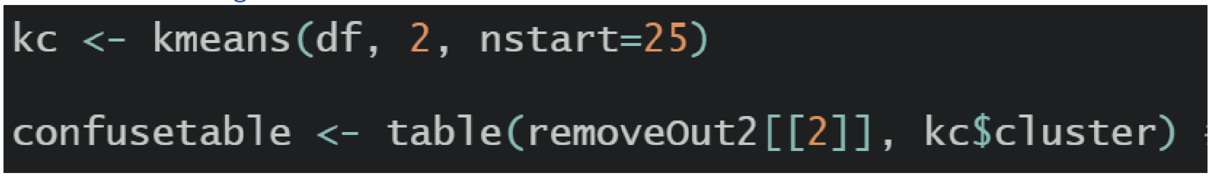
In order to apply the Elbow method, we must define the number of K range we would like to test. In this instance we have opted to test between k = 2:10. We then pass this as well as our data to the k-means function and use the “$tot.withinss” component to run the within sum of squares calculations. Once the calculation has completed, we can plot this result using the “plot” function which will give us a plot shown in **Figure 2.1**. From the plot we now need to identify the bend/knee in the plot. We can see that the bend begins at around 3 and just slightly passed 2. This seems almost similar our NbClust result, where 2 was proposed as the best cluster, closely followed by 3. This similar result is evident here. From this plot we can assume our best cluster is likely 3 and we should also be testing 2 clusters since the bend is not directly over 2 or 3.

K-means analysis

K-means is an unsupervised clustering algorithm that partitions a data set into K clusters. The algorithm aims to group the data based on similarity so that a group containing similar objects is dissimilar to objects in another group. The similarity/dissimilarity of objects is determined using distance metrics such as Euclidean and Manhattan. The algorithm consists of three main steps: initialisation by setting seeding points or initial centroids with a given K. Then it partitions all data points into K clusters based on number of clusters passed to the algorithm. Finally, the centroids are updated, and the process is repeated. The algorithm completes when all points have been grouped to a cluster.

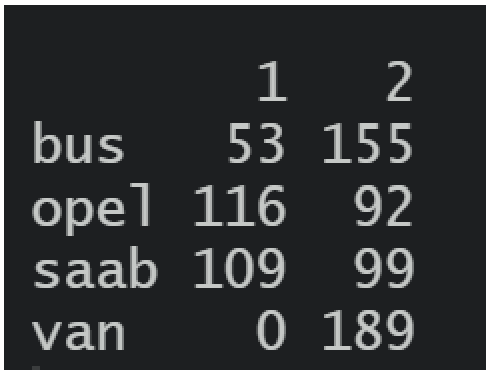
To begin our K-means analysis on our data, we will first define the number of K we would like to use. From our NbClust result and Elbow method, it is evident that we should begin by exploring with k = 2.

K-means clustering with k = 2



```
kc <- kmeans(df, 2, nstart=25)

confusetable <- table(removeOut2[[2]], kc$cluster)
```

Figure 2.2: k-means with k = 2 R code



```
        1   2
bus    53 155
opel  116  92
saab  109  99
van     0 189
```

Figure 2.3: K = 2 confuse table result

<u>Metrics</u>

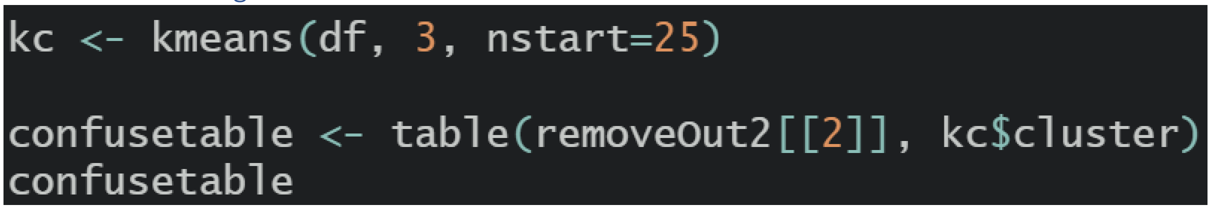
Recall or Sensitivity per class:
Bus: 155/208 = 0.745 -> 75%
Opel: 116/208 = 0.557 -> 56%
Saab: 109/208 = 0.524 = 52%
Van: 189/189 = 1 -> 100%

Precision per cluster:
Cluster1: 116 + 109/278 = 0.809 -> 81%
Cluster2: 155 + 189/535 = 0.642 -> 64%

Accuracy of clustering:
Accuracy = 155 + 116 + 109 + 189/813 = 0.699 -> 70%

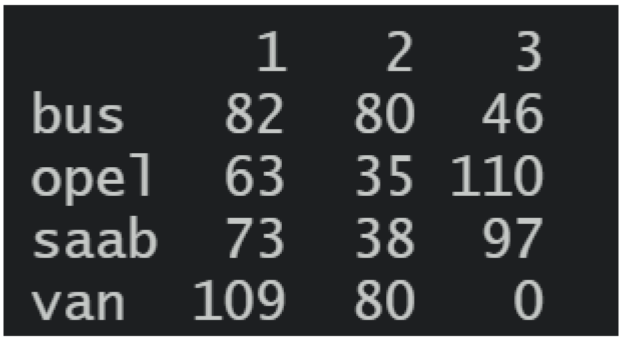
After performing K-means with 2 clusters, we can see that the algorithm has performed good and grouped based on large vehicles and smaller vehicles. There are some mistakes being made such as data points being clustered to cluster 1 although cluster 2 holds the majority, evident in the Bus class where 53 are grouped to cluster 1 while, cluster 2 holds the majority of 155. While this is the case with the other classes as well, the Van class is an excellent example of the algorithm performing well to correctly cluster the Van data points into a single cluster, where all data points belong to a single cluster (cluster2). From the recall and sensitivity results we can see that we have some good results, especially for the Van class with 100% accuracy and the Bus accuracy of 75%, suggesting the K-means analysis has performed well. However, when looking at the Opel and Saab recall and sensitivity, we see lower accuracy of 56% and 52% respectively. Also the precision of each cluster with cluster1 at 81% and cluster2 with 64%, cluster2 is predicting less accurately, suggesting that although, we do have some good results for some classes, there is still areas for improvement to get better results. We can also see that the clustering is 70% accurate. This seems like a good starting point to begin exploring different number of clusters to see if a greater number of clusters can improve our result. We will now attempt to perform K-means analysis with k = 3 clusters.

K-means clustering with k = 3



```
kc <- kmeans(df, 3, nstart=25)

confusetable <- table(removeOut2[[2]], kc$cluster)
confusetable
```

Figure 2.4: K-means with k = 3 R code



```
         1   2   3
bus     82  80  46
opel    63  35 110
saab    73  38  97
van    109  80   0
```

Figure 2.5: K = 3 confuse table result

Metrics

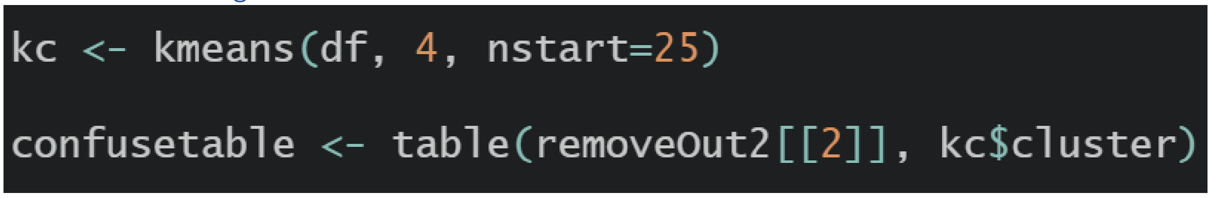
***Recall or Sensitivity per class**:*
Bus: 82/208 = 0.394 -> 39%
Opel: 110/208 = 0.528 = 53%
Saab: 97/208 = 0.466 = 47%
Van: 109/189 = 0.576 -> 58%

Precision per cluster:
Cluster1: 109/327 = 0.3 -> 30%
Cluster2: 80/233 = 0.686 -> 34%
Cluster3: 110 + 97/253 = 0.81 -> 81%

Accuracy of clustering:
Accuracy = 82 + 110 + 97 + 109/813 = 0.489 -> 49%

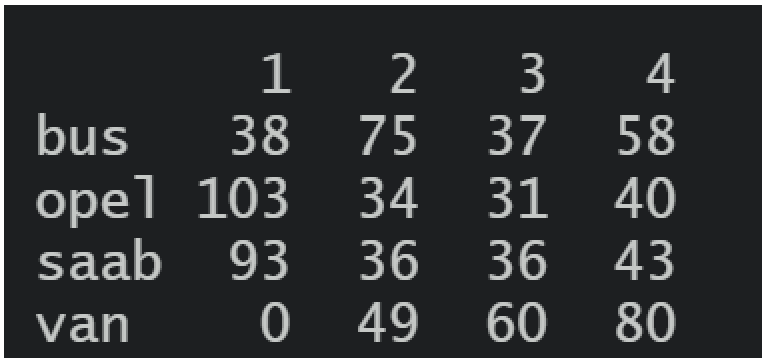
After performing our K-means algorithm with 3 clusters, we can see that the increase in the number of K has not improved the results and instead, has made the results a lot less accurate. The accuracy of clustering has significantly dropped from 70% with 2 clusters to 49% with 3 clusters. In addition to the reduction of accuracy of clustering, the accuracy per class has also decreased significantly. This is evident when comparing the results of recall and sensitivity of each class from the 2-cluster analysis to the 3-cluster result. Every class has a reduced accuracy in the 3-cluster analysis. This is particularly evident, when we compare the recall and sensitivity of the Van class when using 2-clusters with the result of the Van class using 3-clusters. In the 2 clusters analysis, the Van class had 100% recall and sensitivity, the algorithm managed to correctly group this data. However, with the 3 clusters, we have dropped down to 58% recall and sensitivity, a huge decrease. Furthermore, the precision per cluster, for the first 2 clusters has also seen a huge decrease. Cluster 1 and Cluster 2 maintained a higher precision with k = 2, with Cluster 1 being 81% and Cluster 2 being 64% respectively, compared to 30% and 34% for Cluster 1 and 2, with k = 3. From the results of k = 3 and k = 2, the use of 2 clusters is currently the most preferable number of clusters to use. We have now tested both the cluster numbers suggested by the NbClust and Elbow automatic methods. We can now begin to manually experiment with the number of K to see if we can improve the accuracy of the algorithm by testing our own K values. We will now attempt to perform the K-means analysis on the data using k = 4 clusters.

K-means clustering with k = 4



```
kc <- kmeans(df, 4, nstart=25)

confusetable <- table(removeOut2[[2]], kc$cluster)
```

Figure 2.6: K-means with k = 4 R code



```
        1   2   3   4
bus    38  75  37  58
opel  103  34  31  40
saab   93  36  36  43
van     0  49  60  80
```

Figure 2.7: K = 4 confuse table result

Metrics

Recall or Sensitivity per class:
Bus: 75/208 = 0.360 -> 36%
Opel: 103/208 = 0.495 - > 50%
Saab: 36/208 = 0.173 -> 17%
Van: 80/189 = 0.423 -> 42%

Precision per cluster:
Cluster1: 103/234 = 0.440 -> 44%
Cluster2: 75/194 = 0.386 -> 39%
Cluster3: 36/164 = 0.219 -> 22%
Cluster4: 80/221 = 0.361 -> 36%

Accuracy of clustering:
Accuracy = 75 + 103 + 36 + 80/813 = 0.361 -> 36%

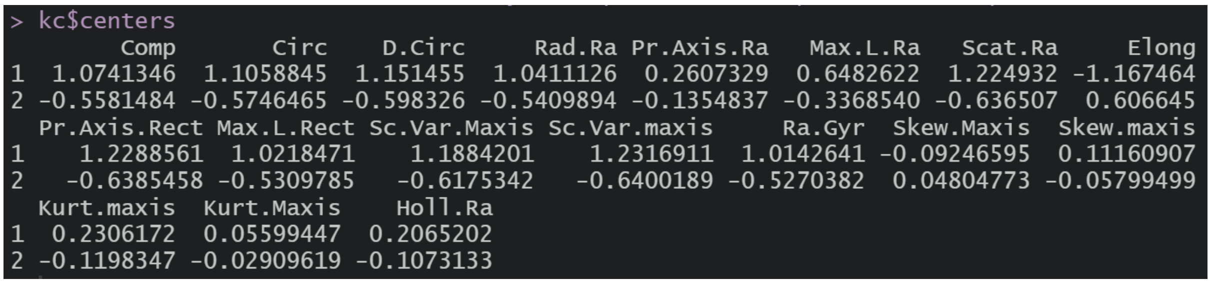
After performing our K-means algorithm with 4 clusters, we can see that increasing the number of clusters to 4 has not improved the accuracy of the K-means algorithm and has instead made the results less accurate. The accuracy has dropped further to 36% from the already low 45% from 3-clusters result. Furthermore, the recall and sensitivity has continued to decrease with an increase in the number of K. This is particularly evident in the Saab class which had a recall and sensitivity of

47% with k = 3 to 17% with k = 4. This is a huge decrease in accuracy for the Saab class. In addition to this, the cluster accuracy for cluster 3 has decreased with the increase in the number of K. The precision per cluster 3 from k = 3 was 81% which is significantly lower compared to the recall and precision of cluster 3 in k = 4 with only 22% accuracy. The reduction in accuracy of the clustering as well as the huge reduction in accuracy for the classes, clearly suggests that the 4-clusters is not the ideal number of K to use for clustering this specific data.

'Winner'

The K-means algorithm has now been performed using the suggested cluster numbers from automatic methods (NbClust and Elbow method) and using our own manual experimentation. This resulted in us trying the K-means analysis with k = 2,3 and 4. From these the K-means algorithm has performed best with 2 clusters, making the winner case k = 2. Performing K-means with 2 clusters resulted in the highest accuracy of clustering, resulting in 70% compared to the accuracy of 3 clusters of 49% and with 4 clusters with an accuracy of 36%. Furthermore, the recall and sensitivity (measurement of the algorithms ability to correctly predict the class) of each class when performing K-means with 2 clusters is significantly greater than the recall and sensitivity of the classes when performing K-mean with 3 and 4 clusters. This is particularly evident with the Saab class. For example, the recall and sensitivity of the Saab class with k = 2 is 52%, in k = 3 it was 47% and with k = 4 it was 17%. The accuracy continued to reduce as the number of clusters increased, with k = 2 sustaining the highest amount of accuracy. The accuracy of 2-clusters not only prevails in the recall and sensitivity results but also in precision per cluster. For example, with k = 2, cluster 1 and 2 have 81% and 64% precision, respectively. Both clusters are predicting with higher accuracy percentage compared to the precision of the clusters in k = 3 and k = 4, where k = 3 cluster precision was 30%, 34% and 81% for cluster 1, 2 and 3, respectively. The first 2 cluster precision results in k = 3 is significantly lower compared to the 2 clusters in k = 2. Furthermore, the first 2 clusters in k = 3 are predicting significantly more inaccurately compared to the third cluster, making the results very unreliable. Finally, for k = 4 the cluster precision was 44%, 39%, 22% and 36% for clusters 1,2,3 and 4 respectively. These cluster precisions are again, significantly lower than the cluster precision of k = 2. In addition to this, these cluster precision results from k = 4 are low in general, suggesting inaccurate and unreliable results being produced. From the results comparison, between k = 2, 3 and 4. It is clear that k = 2 is the winner, the best and most preferable number of clusters to use for our K-means algorithm for this particular data set.

Winner case centres



```
> kc$centers
        Comp       Circ    D.Circ     Rad.Ra Pr.Axis.Ra   Max.L.Ra   Scat.Ra     Elong
1  1.0741346  1.1058845  1.151455  1.0411126  0.2607329  0.6482622  1.224932 -1.167464
2 -0.5581484 -0.5746465 -0.598326 -0.5409894 -0.1354837 -0.3368540 -0.636507  0.606645
  Pr.Axis.Rect Max.L.Rect Sc.Var.Maxis Sc.Var.maxis     Ra.Gyr  Skew.Maxis  Skew.maxis
1    1.2288561  1.0218471    1.1884201    1.2316911  1.0142641 -0.09246595  0.11160907
2   -0.6385458 -0.5309785   -0.6175342   -0.6400189 -0.5270382  0.04804773 -0.05799499
  Kurt.maxis  Kurt.Maxis    Holl.Ra
1  0.2306172  0.05599447  0.2065202
2 -0.1198347 -0.02909619 -0.1073133
```

Figure 2.8: Winner K-means K centres

MLP

Time Series Input Methods

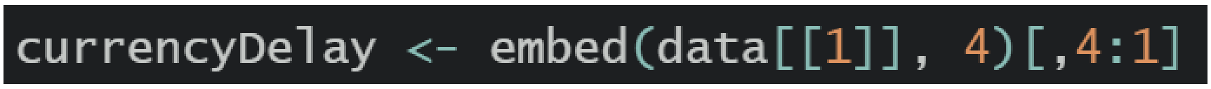
There are several methods that can be used in time-series problems to define the input vector. One of the methods that can be used in order to define the input vector for a time-series problem is an Autoregressive approach. The Autoregressive approach will forecast a variable based on a linear combination of the previous values. This is because the Autoregressive approach assumes that the future value will be like the values that have come before (Fern and o, 2020). To apply the Autoregressive approach, we can have a dataset and format the input vector in a way where the vector represents a linear combination of past values and have the model predict based on these previous values. Another method which can be used in time-series problems to define the input vector is the use of Decomposition techniques. Decomposition techniques aim to model the data in a way that takes into consideration time series patterns for example trend, seasonality and cycles (Hyndman, Athanasopoulos, 2013), commonly referred to as components. The combination of the components can be used in order to produce the outcome of interest. One of aims of decomposition techniques is to present values adjusted from seasons. Seasonally adjust values would then remove the effects a season can have on the values which will then allow us to see the trends more clearly (PennState Elberyl College of Science, 2021). There are two main approaches to decomposition brought about through classical decomposition. These approaches are Additive decomposition and Multiplicative decomposition. Additive decomposition is best used when the magnitude of the seasonal change or the variation in the trend cycle does not vary with the level of the time series. An example of an Addition decomposition structure is: Trend + Seasonal + Random. When the change in the seasonal or the trend cycle does change with the level of the time series then we would use Multiplicative decomposition. The Multiplicative structure would look like the following: Trend * Seasonal * Random, displaying exponential growth over time.These two methods can be used to define the input vector for a time series problem. ARIMA (Auto Regressive Integrated Moving Average) is another method that can be used in time-series problems to define the input vector. The ARIMA method for time series contains some similar principles from the Autoregressive approach such as predicting future values based on the values that have come before. This is because the Autoregressive approach and ARIMA come from the same class of models. The differences lie in the following: ARIMA considers its own lag as well the lagged errors whereas Autoregressive considers only its own lag. In addition to this, when using the ARIMA approach, we transform the time series data into a stationary one. This means to transform it into a dataset that does not contain trends or seasonality, achieved through differencing by subtracting the previous value from the current value. (Sangarshanan, 2019).

Normalization of Time Series

Normalization is an essential pre-processing technique performed on data before it is fed to a Neural Network. This is because significant issues regarding the accuracy and the performance arise when we do not perform normalization techniques to our data. These issues arise as features within our dataset can be on different scales. The feature with the larger scale will completely dominate the other features, producing inaccurate results from the Neural Network. The goal of normalization is to make every data point have the same scale and so each feature becomes equally important instead of one feature completely dominating the others because of a larger scale. One of the most common ways to normalize data is Min-Max normalization. Min-Max normalization takes every feature, and the minimum value of that feature is transformed into a 0, the maximum value is transformed into a 1 and all other values that fall between the range of 0 and 1 get transformed into values between 0 and 1 such as 0.5 or 0.3.

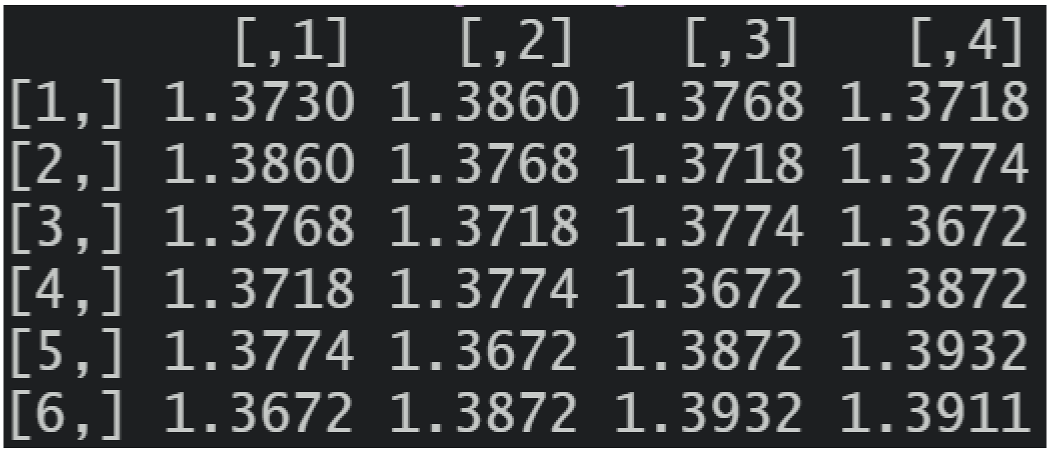
Single Hidden Layer

3 input vector input/output



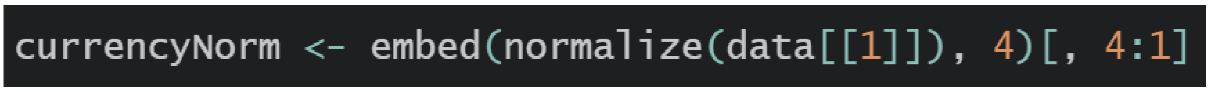
```r
currencyDelay <- embed(data[[1]], 4)[,4:1]
```

Figure 2.9: Input vector



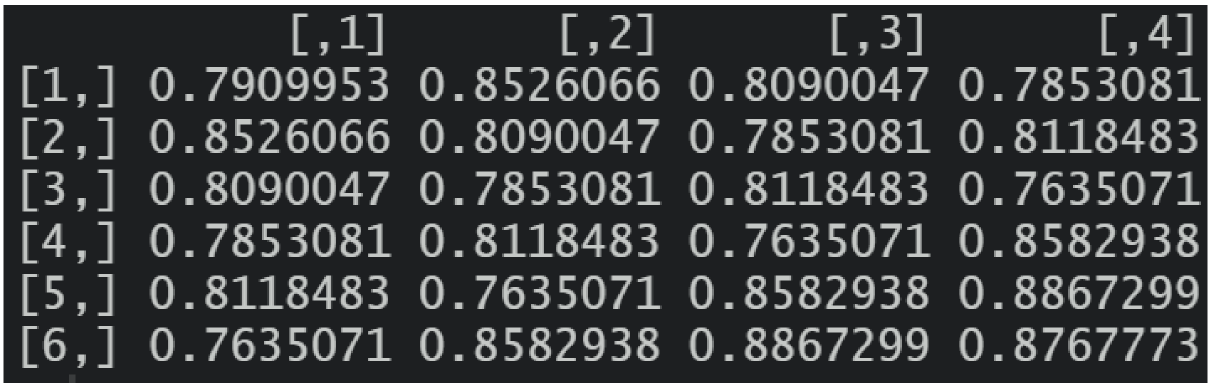
```
       [,1]   [,2]   [,3]   [,4]
[1,] 1.3730 1.3860 1.3768 1.3718
[2,] 1.3860 1.3768 1.3718 1.3774
[3,] 1.3768 1.3718 1.3774 1.3672
[4,] 1.3718 1.3774 1.3672 1.3872
[5,] 1.3774 1.3672 1.3872 1.3932
[6,] 1.3672 1.3872 1.3932 1.3911
```

Figure 3: Output



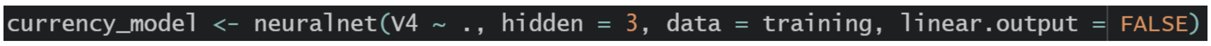
```r
currencyNorm <- embed(normalize(data[[1]]), 4)[, 4:1]
```

Figure 3.1: Normalization R code



```
          [,1]      [,2]      [,3]      [,4]
[1,] 0.7909953 0.8526066 0.8090047 0.7853081
[2,] 0.8526066 0.8090047 0.7853081 0.8118483
[3,] 0.8090047 0.7853081 0.8118483 0.7635071
[4,] 0.7853081 0.8118483 0.7635071 0.8582938
[5,] 0.8118483 0.7635071 0.8582938 0.8867299
[6,] 0.7635071 0.8582938 0.8867299 0.8767773
```

Figure 3.2: Normalized values



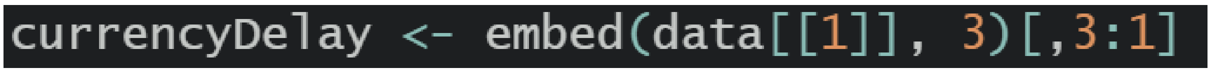
```r
currency_model <- neuralnet(V4 ~ ., hidden = 3, data = training, linear.output = FALSE)
```

Figure 3.3: Neural network model structure

Table 1.

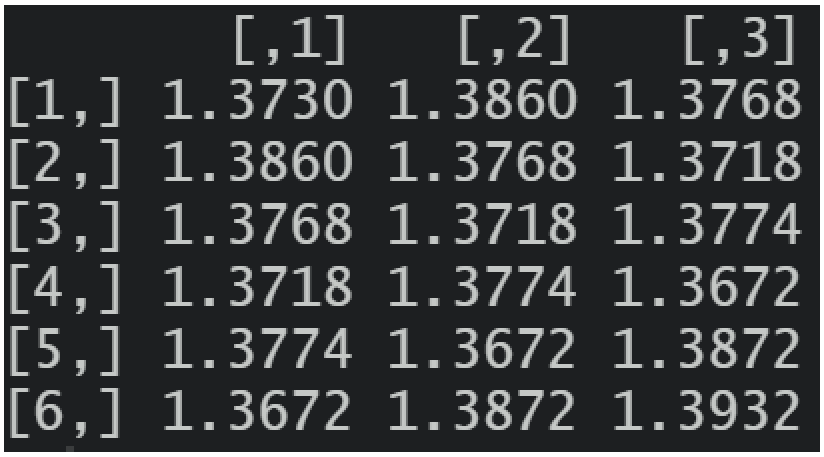
RMSE	MAE	MAPE
0.03526182	0.03058825	0.02364035

2 input vector input/output



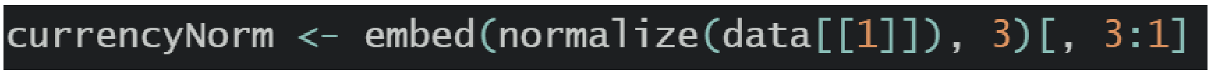
```
currencyDelay <- embed(data[[1]], 3)[,3:1]
```

Figure 3.4: Input vector



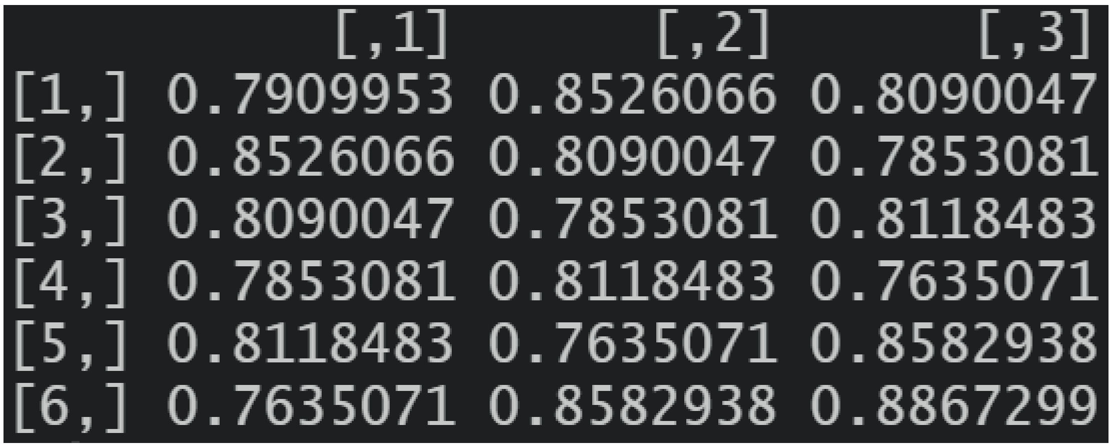
```
         [,1]   [,2]   [,3]
[1,] 1.3730 1.3860 1.3768
[2,] 1.3860 1.3768 1.3718
[3,] 1.3768 1.3718 1.3774
[4,] 1.3718 1.3774 1.3672
[5,] 1.3774 1.3672 1.3872
[6,] 1.3672 1.3872 1.3932
```

Figure 3.5: Output



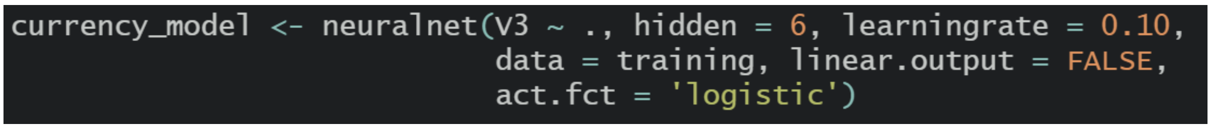
```
currencyNorm <- embed(normalize(data[[1]]), 3)[, 3:1]
```

Figure 3.6: Normalization



```
          [,1]      [,2]      [,3]
[1,] 0.7909953 0.8526066 0.8090047
[2,] 0.8526066 0.8090047 0.7853081
[3,] 0.8090047 0.7853081 0.8118483
[4,] 0.7853081 0.8118483 0.7635071
[5,] 0.8118483 0.7635071 0.8582938
[6,] 0.7635071 0.8582938 0.8867299
```

Figure 3.7: Normalized values



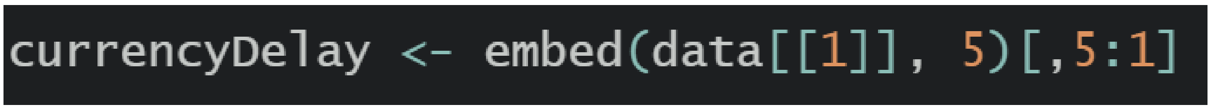
```
currency_model <- neuralnet(V3 ~ ., hidden = 6, learningrate = 0.10,
                            data = training, linear.output = FALSE,
                            act.fct = 'logistic')
```

Figure 3.8: Neural network model structure

Table 2.

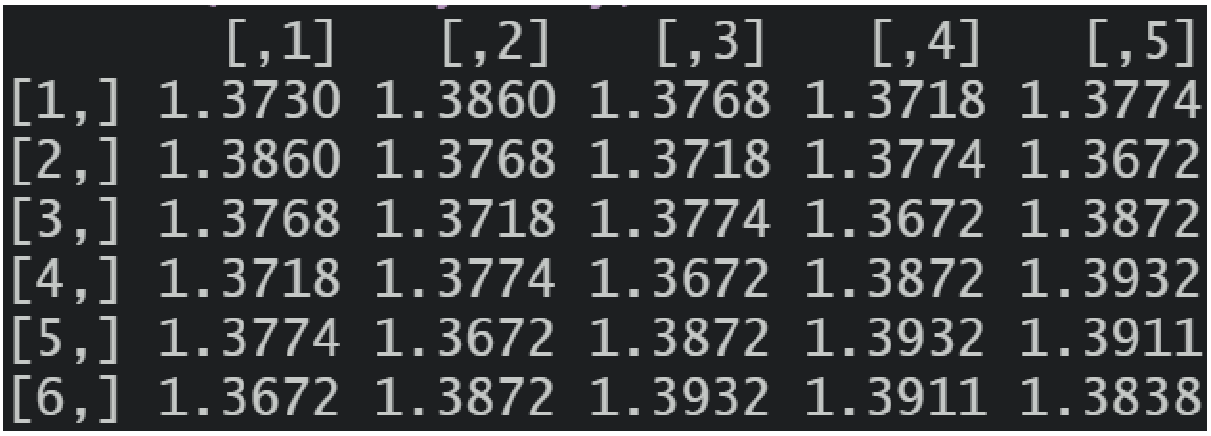
RMSE	MAE	MAPE
0.04134617	0.03663803	0.02847885

4 input vector input/output



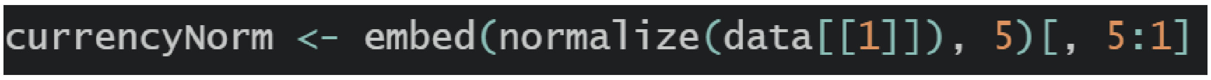
```r
currencyDelay <- embed(data[[1]], 5)[,5:1]
```

Figure 3.9: Input vector



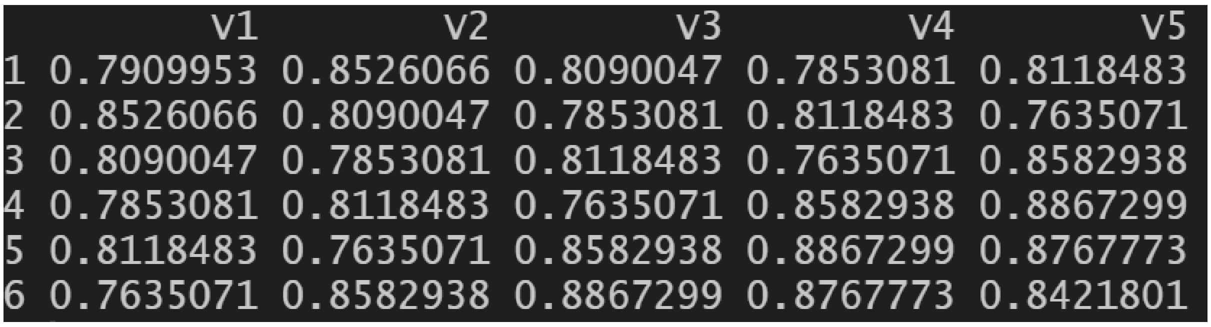
```
        [,1]   [,2]   [,3]   [,4]   [,5]
[1,] 1.3730 1.3860 1.3768 1.3718 1.3774
[2,] 1.3860 1.3768 1.3718 1.3774 1.3672
[3,] 1.3768 1.3718 1.3774 1.3672 1.3872
[4,] 1.3718 1.3774 1.3672 1.3872 1.3932
[5,] 1.3774 1.3672 1.3872 1.3932 1.3911
[6,] 1.3672 1.3872 1.3932 1.3911 1.3838
```

Figure 4: Output



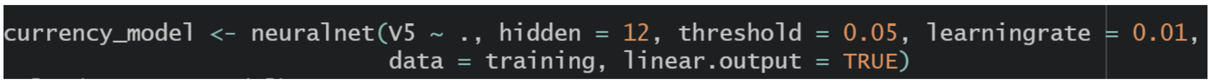
```r
currencyNorm <- embed(normalize(data[[1]]), 5)[, 5:1]
```

Figure 4.1: Normalization



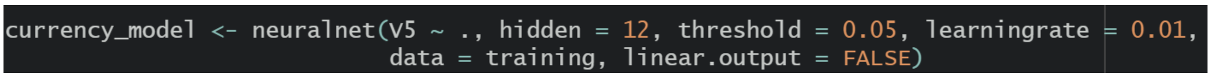
```
         V1        V2        V3        V4        V5
1 0.7909953 0.8526066 0.8090047 0.7853081 0.8118483
2 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071
3 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938
4 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299
5 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773
6 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801
```

Figure 4.2: Normalized values



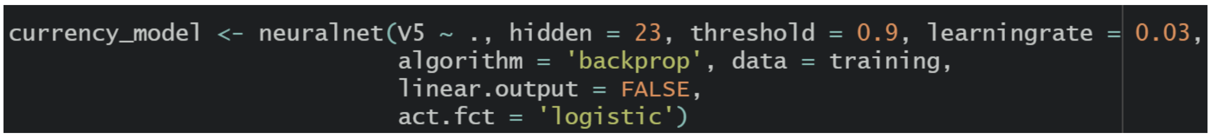
```r
currency_model <- neuralnet(V5 ~ ., hidden = 12, threshold = 0.05, learningrate = 0.01,
                            data = training, linear.output = TRUE)
```

*Figure 4.3: Neural Network model structure (Linear output **TRUE**)*



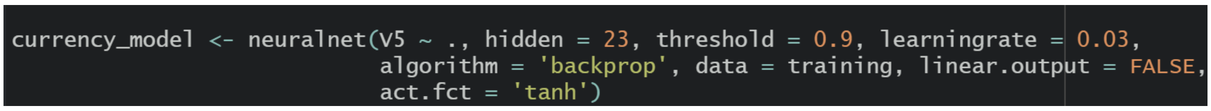
```r
currency_model <- neuralnet(V5 ~ ., hidden = 12, threshold = 0.05, learningrate = 0.01,
                            data = training, linear.output = FALSE)
```

*Figure 4.4: Neural Network model structure (Linear output **FALSE**)*



```r
currency_model <- neuralnet(V5 ~ ., hidden = 23, threshold = 0.9, learningrate = 0.03,
                            algorithm = 'backprop', data = training,
                            linear.output = FALSE,
                            act.fct = 'logistic')
```

Figure 4.5: Neural Network model structure: Increased neurons & learning rate. Also, backprop use and activation function



```r
currency_model <- neuralnet(V5 ~ ., hidden = 23, threshold = 0.9, learningrate = 0.03,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'tanh')
```

Figure 4.6: Neural Network model structure tanh activation function

Table 3 - Linear output **TRUE**

RMSE	MAE	MAPE
0.03643546	0.03209212	0.0248391

Table 4 - Linear output **FALSE**

RMSE	MAE	MAPE
0.03549543	0.03149274	0.02437519

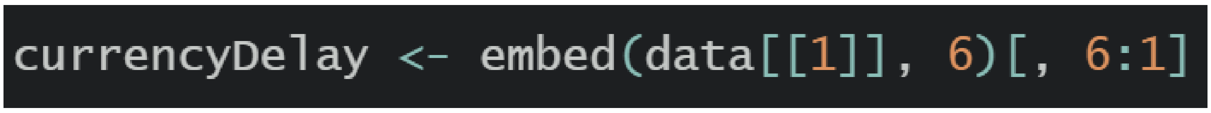
Table 5 – Results for ***increased neurons & learning rate. Also, backprop use and act func***

RMSE	MAE	MAPE
0.03966914	0.03521834	0.02725878

Table 6 – Results using **tanh function**

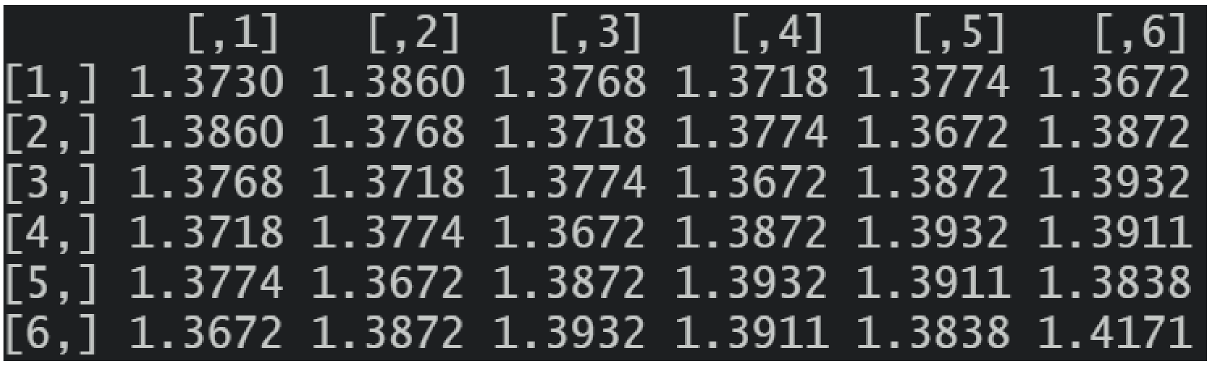
RMSE	MAE	MAPE
0.1251	0.1251	0.09682663

5 input vector input/output



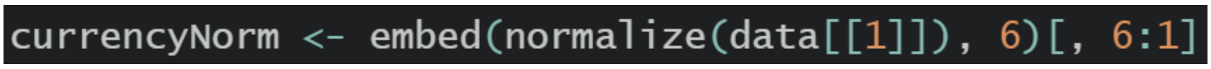
```
currencyDelay <- embed(data[[1]], 6)[, 6:1]
```

Figure 4.7: Input vector



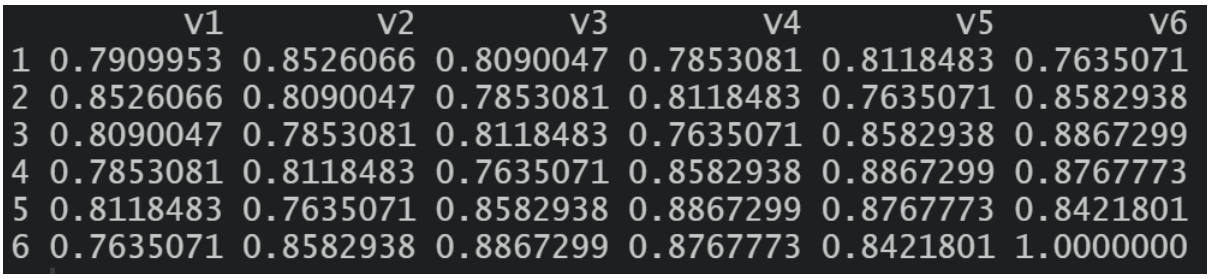
```
         [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
[1,] 1.3730 1.3860 1.3768 1.3718 1.3774 1.3672
[2,] 1.3860 1.3768 1.3718 1.3774 1.3672 1.3872
[3,] 1.3768 1.3718 1.3774 1.3672 1.3872 1.3932
[4,] 1.3718 1.3774 1.3672 1.3872 1.3932 1.3911
[5,] 1.3774 1.3672 1.3872 1.3932 1.3911 1.3838
[6,] 1.3672 1.3872 1.3932 1.3911 1.3838 1.4171
```

Figure 4.8: Output



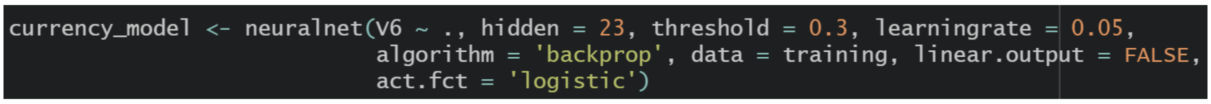
```
currencyNorm <- embed(normalize(data[[1]]), 6)[, 6:1]
```

Figure 4.9: Normalization



```
         V1        V2        V3        V4        V5        V6
1 0.7909953 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071
2 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938
3 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299
4 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773
5 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801
6 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801 1.0000000
```

Figure 5.0: Normalized values



```
currency_model <- neuralnet(V6 ~ ., hidden = 23, threshold = 0.3, learningrate = 0.05,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'logistic')
```

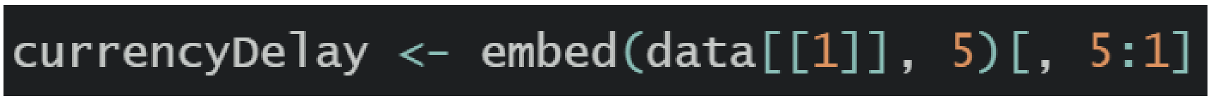
Figure 5.1: Neural Network model structure

Table 7 – Results using **23 Neurons**

RMSE	MAE	MAPE
0.03428424	0.02985867	0.02308184

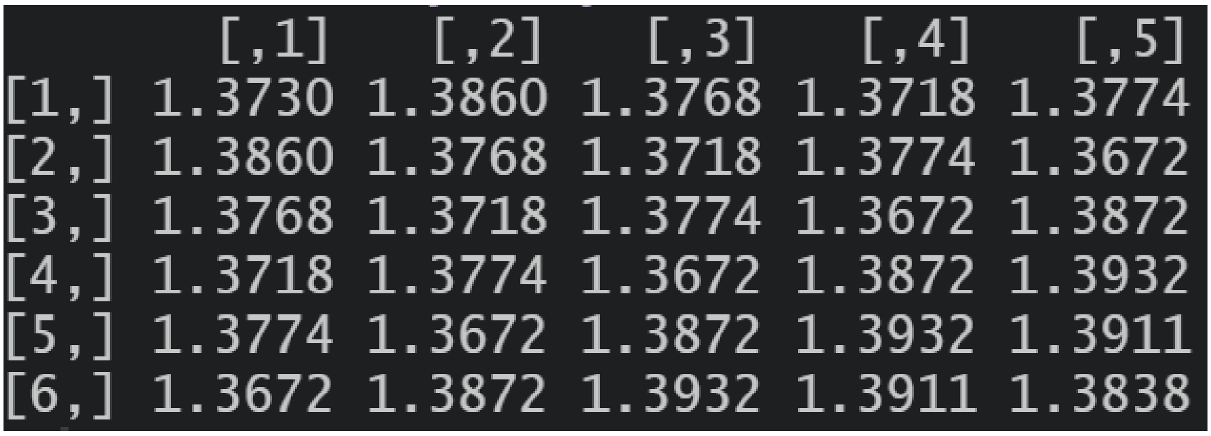
2-Hidden Layers

4 input vector input/output



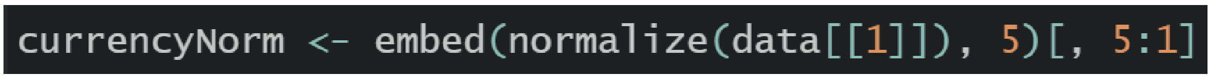
```
currencyDelay <- embed(data[[1]], 5)[, 5:1]
```

Figure 5.2: Input vector



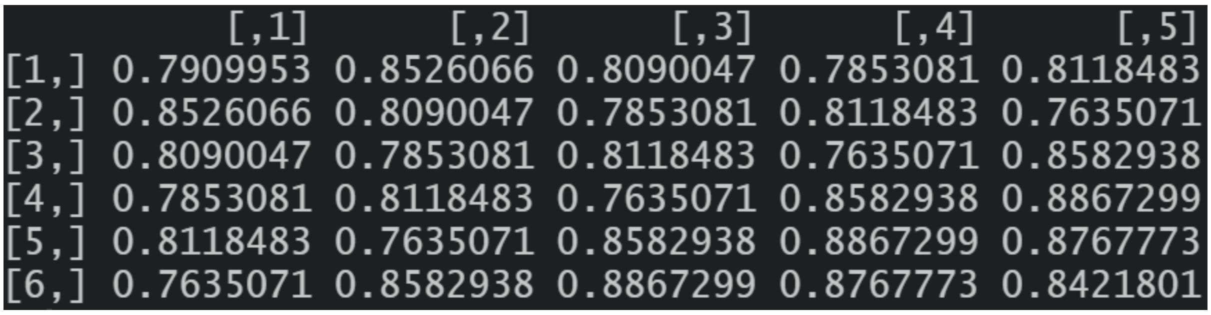
```
       [,1]   [,2]   [,3]   [,4]   [,5]
[1,] 1.3730 1.3860 1.3768 1.3718 1.3774
[2,] 1.3860 1.3768 1.3718 1.3774 1.3672
[3,] 1.3768 1.3718 1.3774 1.3672 1.3872
[4,] 1.3718 1.3774 1.3672 1.3872 1.3932
[5,] 1.3774 1.3672 1.3872 1.3932 1.3911
[6,] 1.3672 1.3872 1.3932 1.3911 1.3838
```

Figure 5.3: Output



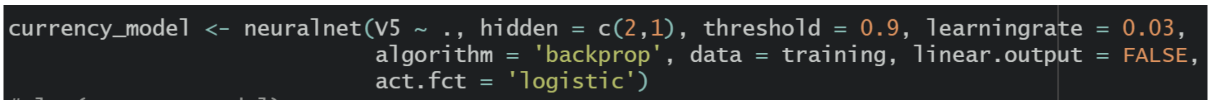
```
currencyNorm <- embed(normalize(data[[1]]), 5)[, 5:1]
```

Figure 5.4: Normalization



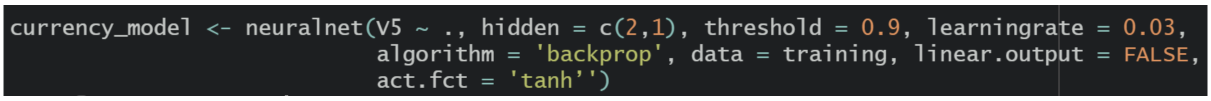
```
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.7909953 0.8526066 0.8090047 0.7853081 0.8118483
[2,] 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071
[3,] 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938
[4,] 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299
[5,] 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773
[6,] 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801
```

Figure 5.5: Normalized values



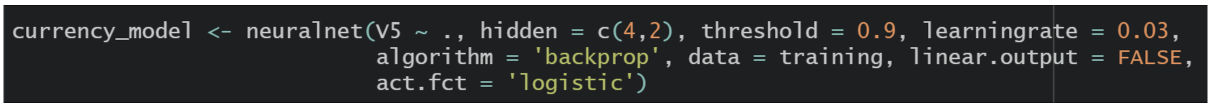
```
currency_model <- neuralnet(V5 ~ ., hidden = c(2,1), threshold = 0.9, learningrate = 0.03,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'logistic')
```

Figure 5.6: Neural Network model structure with logistic activation function



```
currency_model <- neuralnet(V5 ~ ., hidden = c(2,1), threshold = 0.9, learningrate = 0.03,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'tanh'')
```

Figure 5.7: Neural Network model structure with tanh activation function



```
currency_model <- neuralnet(V5 ~ ., hidden = c(4,2), threshold = 0.9, learningrate = 0.03,
                          algorithm = 'backprop', data = training, linear.output = FALSE,
                          act.fct = 'logistic')
```

Figure 5.8: Neural Network model structure with (4,2)

Table 8 – Result with ***Logistic*** activation function

RMSE	MAE	MAPE
0.009739075	0.009730179	0.007531098

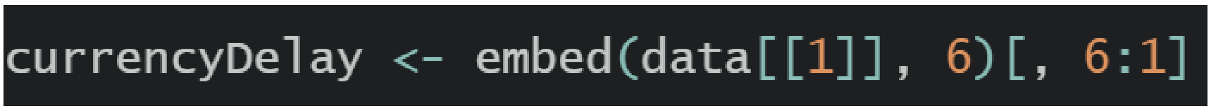
Table 9 – Result with ***tanh*** activation function

RMSE	MAE	MAPE
0.1251	0.1251	0.09682663

Table 10 – Result with ***4 neurons in first layer and 2 neurons in second layer (4,2)***

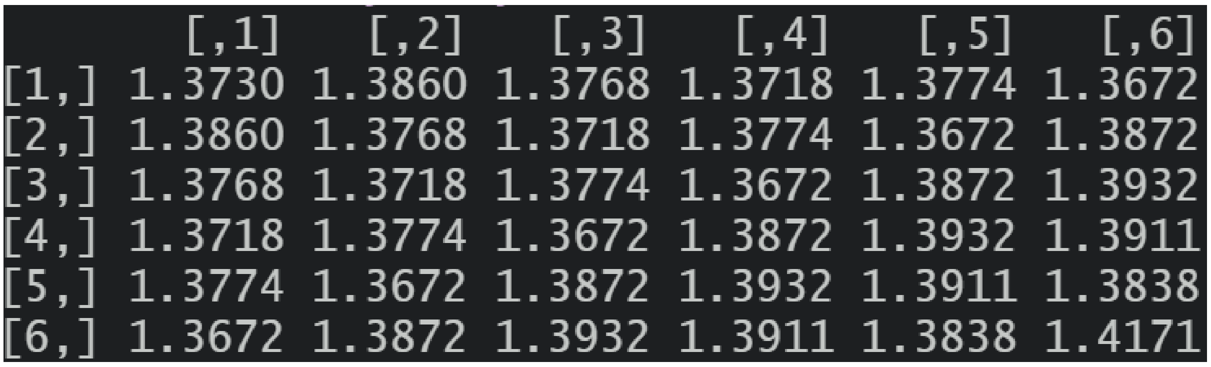
RMSE	MAE	MAPE
0.008317214	0.008309663	0.006431627

5 input vector input/output



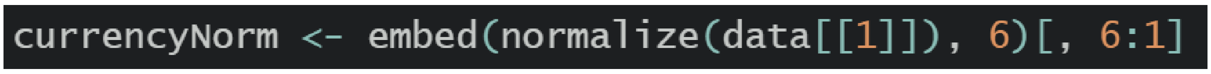
```
currencyDelay <- embed(data[[1]], 6)[, 6:1]
```

Figure 5.9: Input vector



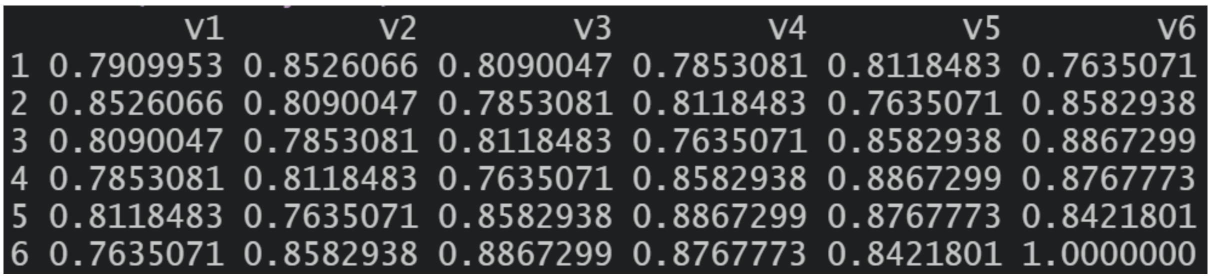
```
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
[1,] 1.3730 1.3860 1.3768 1.3718 1.3774 1.3672
[2,] 1.3860 1.3768 1.3718 1.3774 1.3672 1.3872
[3,] 1.3768 1.3718 1.3774 1.3672 1.3872 1.3932
[4,] 1.3718 1.3774 1.3672 1.3872 1.3932 1.3911
[5,] 1.3774 1.3672 1.3872 1.3932 1.3911 1.3838
[6,] 1.3672 1.3872 1.3932 1.3911 1.3838 1.4171
```

Figure 6.0: Output



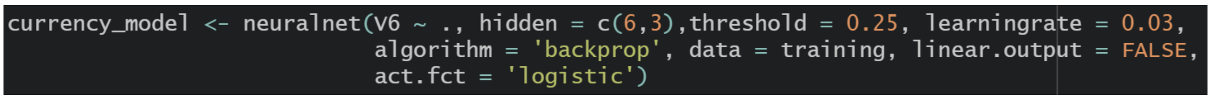
```
currencyNorm <- embed(normalize(data[[1]]), 6)[, 6:1]
```

Figure 6.1: Normalization



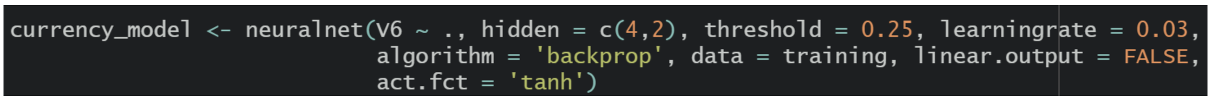
```
         V1        V2        V3        V4        V5        V6
1 0.7909953 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071
2 0.8526066 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938
3 0.8090047 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299
4 0.7853081 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773
5 0.8118483 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801
6 0.7635071 0.8582938 0.8867299 0.8767773 0.8421801 1.0000000
```

Figure 6.2: Normalized values



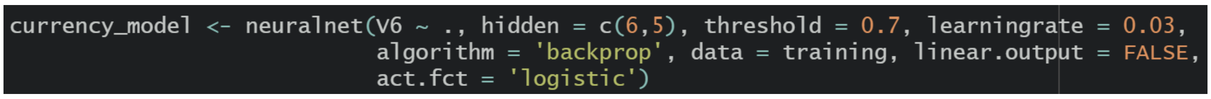
```
currency_model <- neuralnet(V6 ~ ., hidden = c(6,3),threshold = 0.25, learningrate = 0.03,
                          algorithm = 'backprop', data = training, linear.output = FALSE,
                          act.fct = 'logistic')
```

Figure 6.3: Neural Network model structure with logistic activation function



```
currency_model <- neuralnet(V6 ~ ., hidden = c(4,2), threshold = 0.25, learningrate = 0.03,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'tanh')
```

Figure 6.4: Neural Network model structure with tanha activation function



```
currency_model <- neuralnet(V6 ~ ., hidden = c(6,5), threshold = 0.7, learningrate = 0.03,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'logistic')
```

Figure 6.5: Neural Network model structure with logistic activation function and 6 neurons in first layer and 5 in second layer

Table 11 - Results using ***logistic***

RMSE	MAE	MAPE
0.03161166	0.02797625	0.02162666

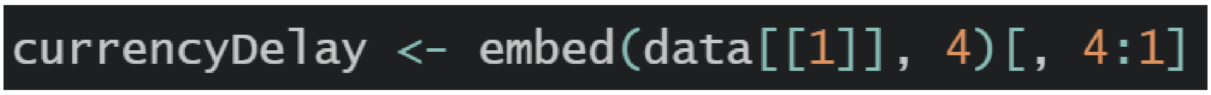
Table 12 - Results using ***tanh***

RMSE	MAE	MAPE
0.1235	0.1235	0.09547001

Table 13 - Results using ***6 neurons in first layer and 5 neurons in second layer (6,5)***

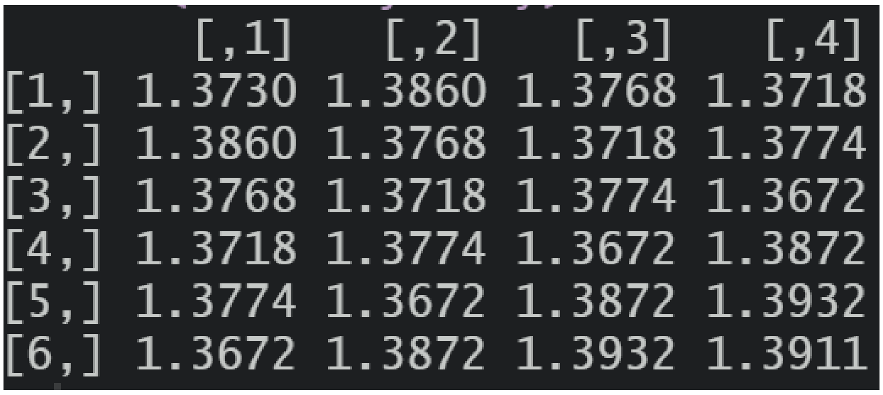
RMSE	MAE	MAPE
0.0241139	0.02165833	0.01674268

3 input vector input/output



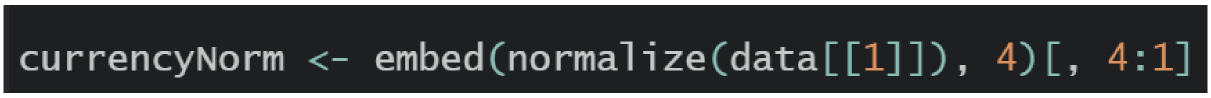
```
currencyDelay <- embed(data[[1]], 4)[, 4:1]
```

Figure 6.6: Input vector



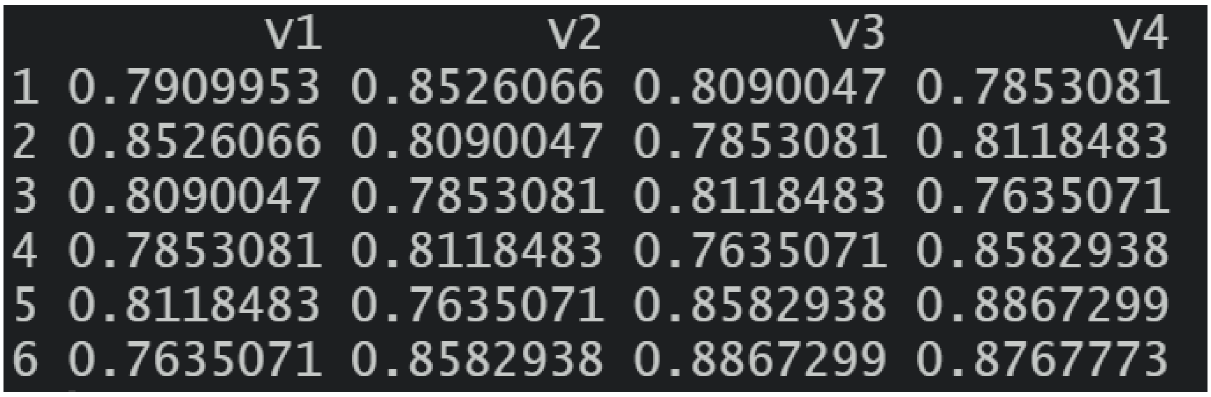
```
       [,1]   [,2]   [,3]   [,4]
[1,] 1.3730 1.3860 1.3768 1.3718
[2,] 1.3860 1.3768 1.3718 1.3774
[3,] 1.3768 1.3718 1.3774 1.3672
[4,] 1.3718 1.3774 1.3672 1.3872
[5,] 1.3774 1.3672 1.3872 1.3932
[6,] 1.3672 1.3872 1.3932 1.3911
```

Figure 6.7: Output



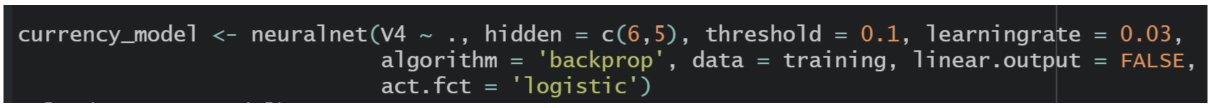
```
currencyNorm <- embed(normalize(data[[1]]), 4)[, 4:1]
```

Figure 6.8: Normalization



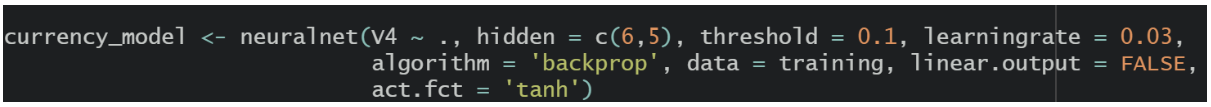
```
         V1        V2        V3        V4
1 0.7909953 0.8526066 0.8090047 0.7853081
2 0.8526066 0.8090047 0.7853081 0.8118483
3 0.8090047 0.7853081 0.8118483 0.7635071
4 0.7853081 0.8118483 0.7635071 0.8582938
5 0.8118483 0.7635071 0.8582938 0.8867299
6 0.7635071 0.8582938 0.8867299 0.8767773
```

Figure 6.9: Normalization values



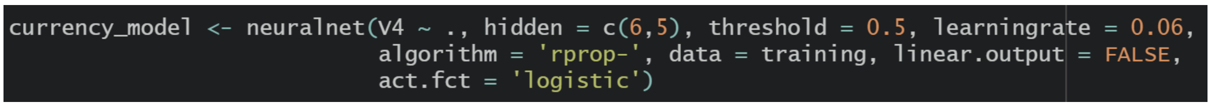
```
currency_model <- neuralnet(V4 ~ ., hidden = c(6,5), threshold = 0.1, learningrate = 0.03,
                           algorithm = 'backprop', data = training, linear.output = FALSE,
                           act.fct = 'logistic')
```

Figure 7: Neural Network model structure with logistic activation function



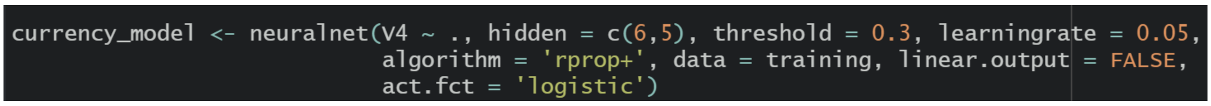
```
currency_model <- neuralnet(V4 ~ ., hidden = c(6,5), threshold = 0.1, learningrate = 0.03,
                           algorithm = 'backprop', data = training, linear.output = FALSE,
                           act.fct = 'tanh')
```

Figure 7.1: Neural Network model structure with tanh activation function



```
currency_model <- neuralnet(V4 ~ ., hidden = c(6,5), threshold = 0.5, learningrate = 0.06,
                           algorithm = 'rprop-', data = training, linear.output = FALSE,
                           act.fct = 'logistic')
```

Figure 7.2: Neural Network model structure with increased threshold and learning rate and rprop-



```
currency_model <- neuralnet(V4 ~ ., hidden = c(6,5), threshold = 0.3, learningrate = 0.05,
                           algorithm = 'rprop+', data = training, linear.output = FALSE,
                           act.fct = 'logistic')
```

Figure 7.3: Neural Network model structure with rprop+ algorithm

Table 14 - Results using ***logistic***

RMSE	MAE	MAPE
0.03346293	0.02931607	0.02265714

Table 15 - Results using ***tanh***

RMSE	MAE	MAPE
0.1232	0.1232	0.09521601

Table 16 - Results from adjusting ***threshold and learning rate***

RMSE	MAE	MAPE
0.02951173	0.02582732	0.01996083

Table 17 – Results from testing the rprop+ algorithm instead of rprop-

RMSE	MAE	MAPE
0.03190318	0.02818371	0.02818371

Evaluating the performance

In order to evaluate the performance of the neural network and assess the forecast accuracy, we use statistical indices such as RMSE (Root Mean Square Error), MAE (Mean Absolute Error) and MAPE (Mean Absolute Percent Error) in order to better understand the performance of the Neural Network in terms of how close the Neural Network output was against the desired output, which is something we would want to know in order to improve the Neural Network. The Mean Absolute Percent Error (MAPE) is a measurement of the average of the absolute percentage of errors. The error is calculated by taking the actual value, subtracting the forecasted value and taking the absolute value (no regard for the sign/negative values). MAPE will measure the size of the error in terms of a percentage, giving us a good idea how well the model is performing from an easy-to-understand percentage, the smaller the MAPE the better the forecasting results. Root Mean Square Error (RMSE) is another way to measure the error of our Neural Network to assess the accuracy. The error for RMSE is calculated by taking the actual values and subtracting the predicted values, giving us the error. We then square root the average of squared errors in order to get the RMSE result. The lower the RMSE result, the better our model is performing. Mean Absolute Error (MAE) is another way to measure the error of the Neural Network when performing forecasting for us to access the performance. MAE will find the error, achieved by taking the actual values and subtracting the predicted values. However, since this is Absolute Error, if the result is a negative value, the negative is ignored. MAE will take the average of the errors in the dataset and give an output. Since this is a performance metric based of errors, lower values would be more desirable, suggesting our model is performing well.

Best Single Hidden

After experimenting with the neural network using only one hidden layer and manipulating the number of neurons, activation function, threshold, learning rate and algorithm used. It is clear that from the results produced, the neural network that performed best with a single hidden layer was the neural network structure seen in **Figure 5.1.** This neural structure used 6 inputs as the input vector, 23 neurons in the single hidden layer, threshold of 0.3, learning rate of 0.05, backpropagation algorithm and the ‘logistic’ activation function. This neural network structure produced the results 0.03428424, 0.02985867 and 0.02308184 evaluated using the statistical indices RMSE, MAE and MAPE, respectively. This neural network structure was considered the best from the single hidden layer neural network structures because it was the neural network structure that had a significantly lower error result, suggested through the performance indices compared to the other neural network structures. For example, a neural network using 3 inputs, 6 hidden, learning rate of 0.10 and the logistic activation function (**Figure 3.8**) produced the results 0.04134617, 0.03663803 and 0.02847885 measured using RMSE, MAE and MAPE, respectively. The results produced through the neural network structure seen in **Figure 4.6** are significantly better, since the RMSE, MAE and MAPE are significantly lower, suggesting the model is performing a lot better since the errors are lower.

Best Single Hidden (Actual vs Predicted)

Best Single Hidden Layer Results

RMSE	MAE	MAPE
0.03428424	0.02985867	0.02308184

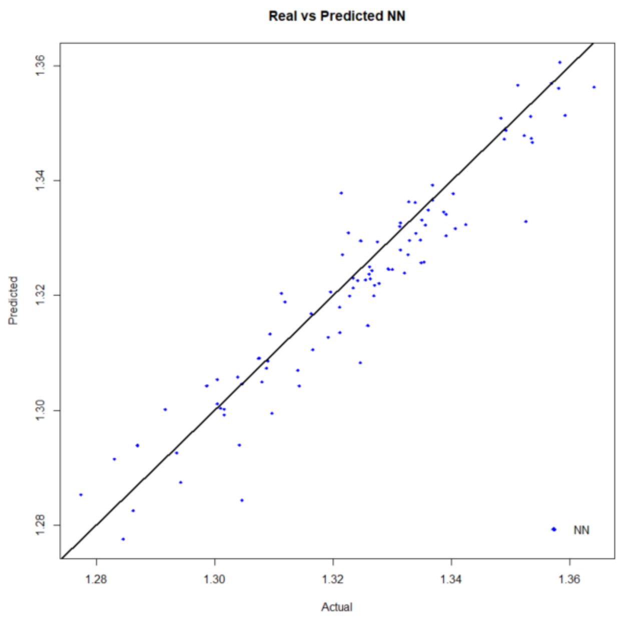


Figure 7.4: Plot of Actual vs Predicted

Best 2-Hidden

After experimenting with the neural network using two hidden layers and adjusting and trying out a different number of neurons, activation functions, threshold amount, different learning rate and also the algorithm used. From the results, it is clear that the best performing 2-Hidden network was the neural network structure seen in **Figure 6.5**. This neural network structure used 6 input variables in the input vector and has 6 neurons for the first hidden layer, 5 neurons for the second hidden layer, a threshold of 0.7, learning rate of 0.03, utilised the backpropagation algorithm and used the 'logistic' activation function. The neural network structure produced the results 0.0241139, 0.02165833 and 0.01674268, evaluated using the statistical indices RMSE, MAE and MAPE, respectively. This neural network structure is considered the best from all the neural network structures experimented with two hidden layers because the results produced, were significantly lower than other neural network structures, suggesting a significantly lower error rate which implies the neural network is performing better. This is clearly evident when comparing one of the other neural network structures such as the network structure seen in **Figure 6.4,** producing the results 0.1235, 0.1235 and 0.09547001, evaluated using the RMSE, MAE and MAPE statical performance indices, respectively. This neural network structure used 6 inputs, 4 neurons in the first layer, 2 neurons in the second layer, threshold of 0.25, learning rate of 0.03, backpropagation algorithm and the 'tanh activation function. Comparing this to the significantly lower results produced by the 'best' two hidden neural network structure seen in **Figure 6.5** which, as mentioned before produced the results 0.0241139, 0.02165833 and 0.01674268 (RMSE, MAE and MAPE) it is clearly evident that **Figure 6.5** neural network structure is the best. The lower results produced by the neural network structure seen in **Figure 6.5** suggests the neural network is significantly more accurate and reliable and therefore, **Figure 6.5** is the best two-hidden neural network structure.

Best 2-hidden (Actual vs Predicted)

Best 2-Hidden layer Results

RMSE	MAE	MAPE
0.0241139	0.02165833	0.01674268

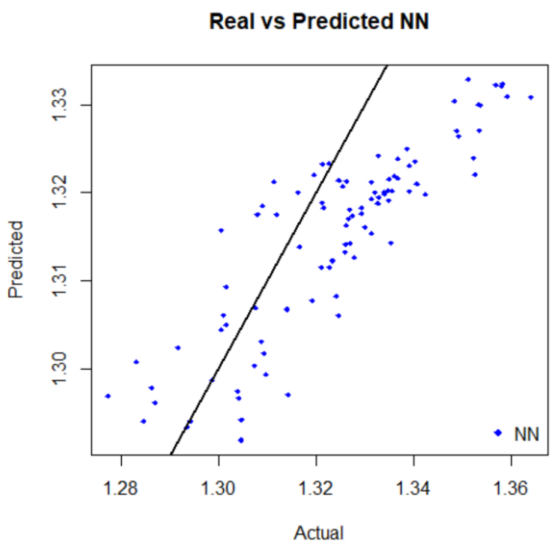


Figure 7.5: Plot of Actual vs Predicted

Preferable

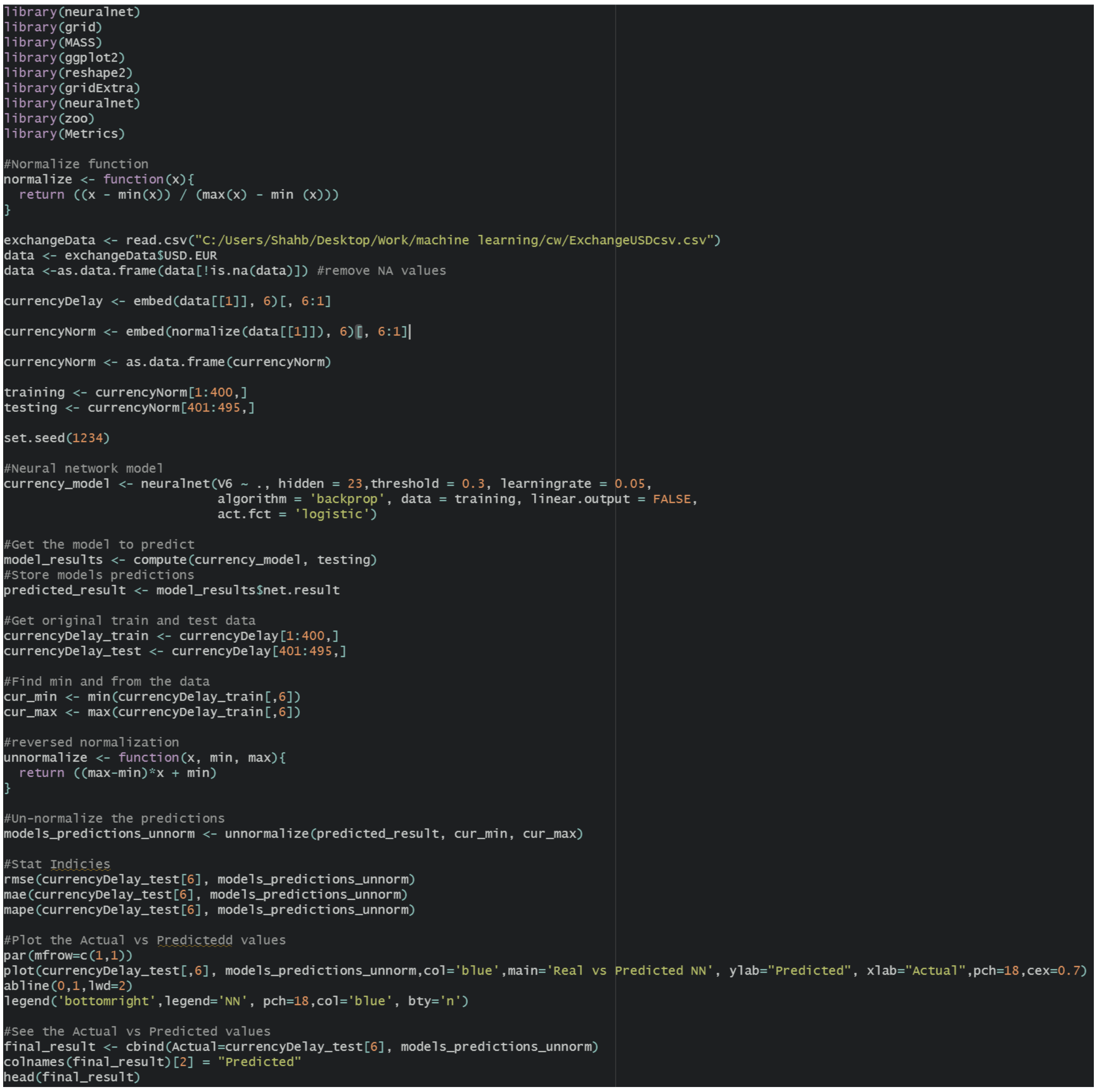
The neural network that produced the best results using a single hidden layer was **Figure 5.1**, producing the results 0.03428424, 0.02985867 and 0.02308184 and the neural network structure that produced the best results using two hidden layers was **Figure 6.5**, producing the results 0.0241139, 0.02165833and 0.01674268. These results represent the statical indices used to measure the performance of these neural networks (RMSE, MAE and MAPE). There is a noticeable difference in the two statistical performance indices of the neural network structures. The two hidden layer network results are significantly lower than those of the single hidden, suggesting that the two hidden layer network has less errors and therefore, performing a lot better. However, when looking at **Figure 7.4,** it seems that the single hidden layer is closer to predicting the actual values as the dots are closer to the line compared to those in the two-hidden layer (**Figure 7.5**). While, this might be the case, I would argue that the two-hidden layer neural network structure is preferable in this instance. This is because, the error rate is lower, evident in the statistical indices, making the neural network more reliable. Furthermore, the single hidden layer with 23 neurons is computationally heavy and therefore, less efficient compared to the two-hidden layer structure with 6 neurons in the first layer and 5 in the second layer. The single hidden layer would require adjustments of a total of 162 weights (including bias) while the two-hidden layer neural network structure has only 82 total weights (including bias). The better time complexity of the two hidden neural network structure as well as its lower error rate makes the two hidden layer neural network structure the preferable structure out of the two neural network structures.

Appendices

K-means R code

```
library(reshape2)
library(NbClust)
library(flexclust)
library(factoextra)
library(ggstatsplot)
library(dplyr)

#Outlier removal function (printing class labels)
OutlierRemoval <- function(data, class){
  x <- boxplot(data)

  OutVals <- x$out

  columns <- x$group
  ind=numeric()
  classes=c()

  if(length(columns) > 0) {
    for(i in 1:length(columns)){
      rows=which(data[,columns[i]]==OutVals[i])
      ind=union(ind,rows)
      classes=c(classes,class[rows])
    }
    dt=data.frame(OutVals, columns, classes[1:length(OutVals)])
    print(dt)
    return (list(data[-ind,], class[-ind]))
  }
  return(list(data, class))
}

# Scale data -> same as scale() function
z_score <- function(x){
  return ((x - mean(x))/sd(x))
}

vehicles <- read.csv("/Users/Shahb/Desktop/more/vehiclescsv.csv")
vehData <- vehicles[, -c(1,20)] #Remove sample and class col.
vehClass <- vehicles$Class #Keep vehicleClass

#Begin removing outliers
removeOut <- OutlierRemoval(vehData, vehClass)

#To make sure all outliers are completely removed by re-running the function
removeOut2 <- OutlierRemoval(removeOut[[1]], removeOut[[2]])

#Scale the data
cl_vehDataScale <- lapply(removeOut2[[1]], z_score)
boxplot(cl_vehDataScale) #view the scaled data

df <- as.data.frame(do.call(cbind, cl_vehDataScale))

#Finding how many clusters might be needed
set.seed(26)
clusterNo <- NbClust(df, distance="euclidean",
                     method="kmeans", index="all")
#Elbow method
k = 2:10
set.seed(42)
WSS = sapply(k, function(k) {kmeans(df, centers=k)$tot.withinss})
plot(k, WSS, type="l", xlab= "Number of k", ylab="Within sum of squares")

#Perform k means
kc <- kmeans(df, 2, nstart=25) #maybe keep nstart=25 improves the output, more clear
kc$centers

#removeOut2[[2]] refers to that particular datasets associated classes
confusetable <- table(removeOut2[[2]], kc$cluster) #Show result in confustion matrix table
confusetable
```

MLP - Neural Network R code



```r
library(neuralnet)
library(grid)
library(MASS)
library(ggplot2)
library(reshape2)
library(gridExtra)
library(neuralnet)
library(zoo)
library(Metrics)

#Normalize function
normalize <- function(x){
  return ((x - min(x)) / (max(x) - min (x)))
}

exchangeData <- read.csv("C:/Users/Shahb/Desktop/Work/machine learning/cw/ExchangeUSDcsv.csv")
data <- exchangeData$USD.EUR
data <-as.data.frame(data[!is.na(data)]) #remove NA values

currencyDelay <- embed(data[[1]], 6)[, 6:1]

currencyNorm <- embed(normalize(data[[1]]), 6)[, 6:1]

currencyNorm <- as.data.frame(currencyNorm)

training <- currencyNorm[1:400,]
testing <- currencyNorm[401:495,]

set.seed(1234)

#Neural network model
currency_model <- neuralnet(V6 ~ ., hidden = 23,threshold = 0.3, learningrate = 0.05,
                            algorithm = 'backprop', data = training, linear.output = FALSE,
                            act.fct = 'logistic')

#Get the model to predict
model_results <- compute(currency_model, testing)
#Store models predictions
predicted_result <- model_results$net.result

#Get original train and test data
currencyDelay_train <- currencyDelay[1:400,]
currencyDelay_test <- currencyDelay[401:495,]

#Find min and from the data
cur_min <- min(currencyDelay_train[,6])
cur_max <- max(currencyDelay_train[,6])

#reversed normalization
unnormalize <- function(x, min, max){
  return ((max-min)*x + min)
}

#Un-normalize the predictions
models_predictions_unnorm <- unnormalize(predicted_result, cur_min, cur_max)

#Stat Indicies
rmse(currencyDelay_test[6], models_predictions_unnorm)
mae(currencyDelay_test[6], models_predictions_unnorm)
mape(currencyDelay_test[6], models_predictions_unnorm)

#Plot the Actual vs Predictedd values
par(mfrow=c(1,1))
plot(currencyDelay_test[,6], models_predictions_unnorm,col='blue',main='Real vs Predicted NN', ylab="Predicted", xlab="Actual",pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN', pch=18,col='blue', bty='n')

#See the Actual vs Predicted values
final_result <- cbind(Actual=currencyDelay_test[6], models_predictions_unnorm)
colnames(final_result)[2] = "Predicted"
head(final_result)
```

References

Fern, J. and o (2020). *What Does Autoregressive Mean?* [online] Investopedia. Available at: https://www.investopedia.com/terms/a/autoregressive.asp#:~:text=Autoregressive%20models%20predict%20future%20values.

Hyndman, Athanasopoulos, R., George (2013). *Chapter 6 Time Series Decomposition | Forecasting: Principles and Practice (2nd ed)*. [online] *otexts.com*. OTexts: Melbourne, Australia. Available at: https://otexts.com/fpp2/decomposition.html [Accessed 17 Apr. 2021].

Sangarshanan (2019). *Time Series Forecasting — ARIMA Models*. [online] Medium. Available at: https://towardsdatascience.com/time-series-forecasting-arima-models-7f221e9eee06.

PennState Elberyl College of Science (2021). *5.1 Decomposition Models | STAT 510*. [online] PennState: Statistics Online Courses. Available at: https://online.stat.psu.edu/stat510/lesson/5/5.1 [Accessed 17 Apr. 2021].