**Financial Forecasting Part**

**Ans2**

Schemes/methods used to define this input vector in this financial domain.

**The Autoregressive (AR) Method**:

The exchange rate's time-delayed values serve as input variables for the AR approach. This indicates that the exchange rate's previous values at various time lags make up the input vector. A one-step-ahead prediction, for instance, would have the exchange rate values at t-1, t-2, t-3, and so on in the input vector.

**Moving Normal (Ma) Approach:**

The moving averages of the exchange rate over a predetermined number of time periods are included in the input vector of the MA method. Short-term trends can be better captured and data noise can be reduced thanks to this. The moving average of the exchange rate over the previous five days, ten days, or any other window size could be included in the input vector, for example.

**Technical Indicators:**

The use of a variety of technical indicators as input variables is one more strategy. Mathematical calculations based on historical price and volume data are known as technical indicators. Models incorporate moving midpoints, relative strength record (RSI), stochastic oscillator, and Bollinger Groups. Market trends, volatility, and conditions of overbought or oversold can all benefit from additional information from these indicators.

**External Components:**

Relevant external factors that could have an impact on exchange rates can also be included in the input vector. Macroeconomic indicators like interest rates, inflation rates, GDP growth, or geopolitical events could be these factors. The neural network is capable of capturing the influence of fundamental factors on exchange rate movements by including such variables.

**Sentiment Analysis:**

Market or news sentiment can be used as input variables with the help of techniques for sentiment analysis. This involves looking at data from social media, news articles, or other sources to figure out how people feel about the currency pair you're considering (positive, negative, or neutral). The dynamics of the market can be better understood with the help of this information.

References:  
Cheung, Y. W., & Chinn, M. D. (1998). Integration, cointegration, and the forecast consistency of structural exchange rate models. Journal of International Money and Finance, 17(5).

Exchange Rate Forecasting with An Artificial Neural Network Model: Can We Beat a Random Walk Model? By Y. SUN



# Import the data

library(readxl)

library(caret)

# Load the dataset

dataset <- read\_excel("D:/New folder/ExchangeUSD.xlsx")

exchange\_rates <- dataset$`USD/EUR`

# Define the maximum time delay for the input vector

max\_delay <- 4

# Initialize the input/output matrices

input\_matrix <- matrix(NA, nrow = length(exchange\_rates) - max\_delay, ncol = max\_delay)

output\_vector <- exchange\_rates[(max\_delay + 1):length(exchange\_rates)]

# Construct the input/output matrices

for (i in 1:max\_delay) {

input\_matrix[, i] <- exchange\_rates[(max\_delay - i + 1):(length(exchange\_rates) - i)]

}

# Split the data into training and testing sets

train\_samples <- 400

train\_input <- input\_matrix[1:train\_samples, ]

train\_output <- output\_vector[1:train\_samples]

test\_input <- input\_matrix[(train\_samples + 1):nrow(input\_matrix), ]

test\_output <- output\_vector[(train\_samples + 1):length(output\_vector)]

# Perform any necessary preprocessing or normalization of the input and output data

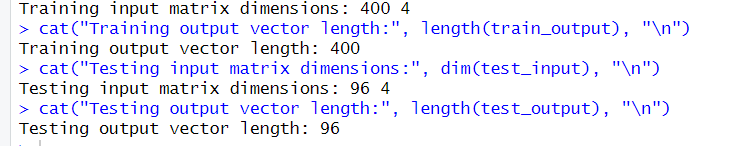
# Print the dimensions of the input/output matrices

cat("Training input matrix dimensions:", dim(train\_input), "\n")

cat("Training output vector length:", length(train\_output), "\n")

cat("Testing input matrix dimensions:", dim(test\_input), "\n")

cat("Testing output vector length:", length(test\_output), "\n")



These dimensions indicate that the training input matrix has 400 samples, each with 4 time-delayed input variables. The training output vector has 400 corresponding output values.

Similarly, the testing input matrix has 96 samples, each with 4 time-delayed input variables. The testing output vector has 96 corresponding output values.

1. #ans c

train\_input\_norm <- scale(train\_input)

test\_input\_norm <- scale(test\_input)

# Check the distribution of the normalized data

hist(train\_input\_norm)

# De-normalize the data

train\_input\_denorm <- train\_input\_norm \* sd(train\_input) + mean(train\_input)

test\_input\_denorm <- test\_input\_norm \* sd(test\_input) + mean(test\_input)

# Truncate the train\_input\_denorm vector to the same length as the train\_output vector

train\_input\_denorm <- train\_input\_denorm[1:length(train\_output)]

# Plot the de-normalized data

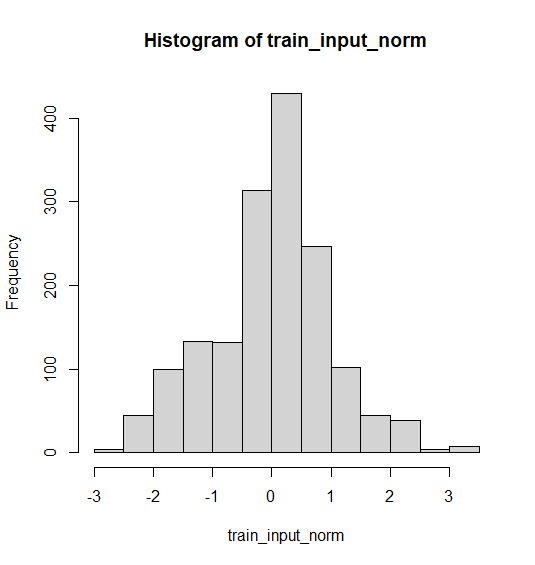
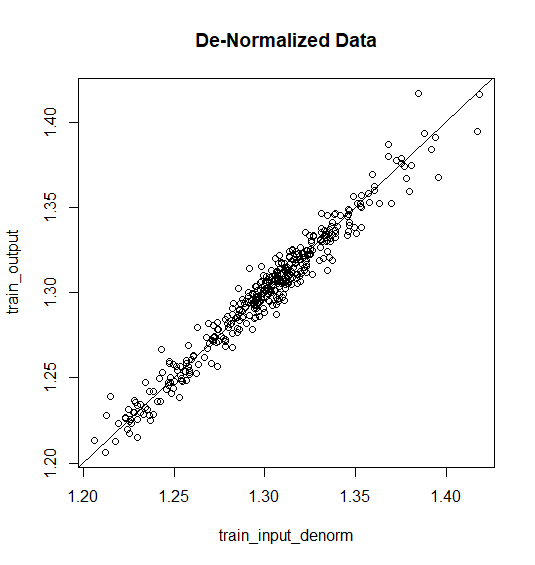
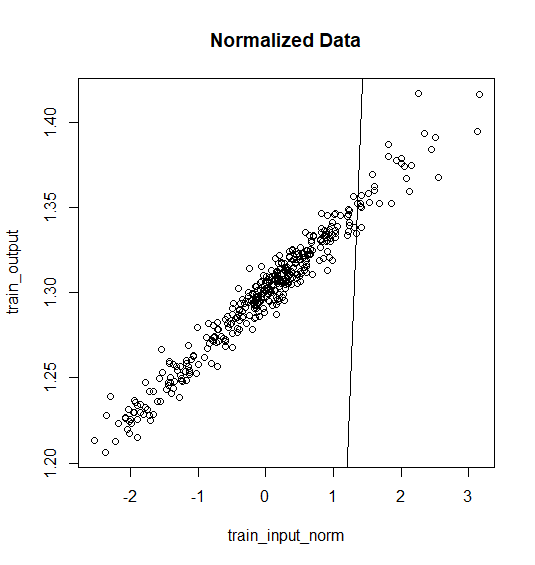
plot(train\_input\_denorm, train\_output, main = "De-Normalized Data")

abline(0, 1)

train\_input\_norm <- train\_input\_norm[1:length(train\_output)]

plot(train\_input\_norm, train\_output, main = "Normalized Data")

abline(0, 1)



Normalizing the data before using it in an MLP structure is a standard procedure, and it serves several important purposes:

**Scale Independence:** The scale of the input variables affects MLPs. The neural network might give more weight to variables with higher values when the scales of the input variables are very different. By normalizing the data, we ensure that each input variable contributes equally to learning by bringing them all to the same scale. This helps prevent bias in favor of some variables solely because of how big they are.

**More rapid convergence:** Normalizing the information can prompt quicker assembly during the preparation interaction. The MLP's weights and biases can adjust more effectively when the scales of the input variables are the same. Learning and convergence toward an ideal solution may speed up as a result.

**Numerical Stability Avoidance:** Numerical instability in the MLP can result from unnormalized data with extensive value ranges. The activation functions or gradient calculations may reach saturation with large values, or they may produce large gradients, which may impede the learning process. By normalizing the data, numerical instability issues like these are less likely to occur.

**Generalization**: The MLP's ability to generalize can benefit from normalization. The MLP is able to generalize to unseen data more effectively and becomes less sensitive to specific values when the data are normalized to a common range, such as between 0 and 1. This improves the model's ability to accurately predict new, untested samples and helps to prevent overfitting.



#ANS d

library(caret)

library(neuralnet)

# To calculate sMAPE (symmetric MAPE)

smape <- function(actual, predicted) {

200 \* mean(abs(actual - predicted) / (abs(actual) + abs(predicted)))

}

# Define a list of input vectors (time delays)

input\_vectors <- list(1:1, 1:2, 1:3, 1:4)

# Define a list of MLP models

mlp\_models <- list(

mlp1 = list(hidden = c(5), linear.output = TRUE, activation = "logistic", learningrate = 0.01),

mlp2 = list(hidden = c(10, 5), linear.output = TRUE, activation = "tanh", learningrate = 0.001),

mlp3 = list(hidden = c(15, 10, 5), linear.output = TRUE, activation = "relu", learningrate = 0.0001),

mlp4 = list(hidden = c(5), linear.output = FALSE, activation = "logistic", learningrate = 0.01),

mlp5 = list(hidden = c(10, 5), linear.output = FALSE, activation = "tanh", learningrate = 0.001),

mlp6 = list(hidden = c(15, 10, 5), linear.output = FALSE, activation = "relu", learningrate = 0.0001)

)

results <- list()

for (input\_vector in input\_vectors) {

for (i in 1:length(mlp\_models)) {

model <- neuralnet(

formula = train\_output ~ .,

data = cbind(train\_input[, input\_vector], train\_output),

hidden = mlp\_models[[i]]$hidden,

linear.output = mlp\_models[[i]]$linear.output,

learningrate = mlp\_models[[i]]$learningrate

)

predictions <- predict(model, cbind(test\_input[, input\_vector]))

# Calculate evaluation metrics

rmse <- RMSE(predictions, test\_output)

mae <- MAE(predictions, test\_output)

mape <- mean(abs(predictions - test\_output) / test\_output) \* 100

smape\_val <- smape(predictions, test\_output)

# Store the results

result <- list(

input\_vector = input\_vector,

model\_name = paste0("mlp", i),

RMSE = rmse,

MAE = mae,

MAPE = mape,

sMAPE = smape\_val

)

results[[length(results) + 1]] <- result

}

}

# The results

for (result in results) {

cat("Input Vector:", paste(result$input\_vector, collapse = "-"), "\n")

cat("Model:", result$model\_name, "\n")

cat("RMSE:", result$RMSE, "\n")

cat("MAE:", result$MAE, "\n")

cat("MAPE:", result$MAPE, "\n")

cat("sMAPE:", result$sMAPE, "\n")

cat("\n")

}

1. These are the 4 stat indices:  
   a) Root Mean Square Error (RMSE):

RMSE measures the average magnitude of the differences between predicted and actual values, taking into account both the direction and the magnitude of the errors. It is calculated by taking the square root of the average of the squared differences between the predicted and actual values. RMSE provides a measure of the overall model performance, where lower values indicate better accuracy. However, RMSE is sensitive to outliers and larger errors can disproportionately affect its value.

Mean Absolute Error (MAE):

MAE represents the average absolute difference between predicted and actual values. It measures the average magnitude of the errors without considering their direction. MAE is calculated by taking the average of the absolute differences between the predicted and actual values. Similar to RMSE, lower MAE values indicate better accuracy. MAE is less sensitive to outliers compared to RMSE.

Mean Absolute Percentage Error (MAPE):

MAPE is a relative measure of the forecast accuracy, expressed as a percentage. It calculates the average percentage difference between the predicted and actual values. MAPE is calculated by taking the average of the absolute percentage differences between the predicted and actual values. MAPE provides insights into the relative magnitude of errors, making it useful for comparing forecasting performance across different datasets. However, MAPE has some limitations, particularly when actual values are close to zero or when zero values are present.

Symmetric Mean Absolute Percentage Error (sMAPE):

sMAPE is another relative measure of forecast accuracy, which addresses some of the limitations of MAPE. It calculates the average percentage difference between the predicted and actual values, but unlike MAPE, it takes into account the average of the predicted and actual values. sMAPE is calculated by taking the average of the absolute percentage differences between the predicted and actual values, normalized by the average of the predicted and actual values. sMAPE provides a balanced measure of accuracy and is suitable for datasets with varying magnitudes. It is also symmetric, meaning it treats overestimations and underestimations equally.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model | Description | RMSE | MAE | MAPE | SMAPE |
| MLP1 | 1 input vector, 1 hidden layers, 5 nodes, linear output, learningrate = 0.01 and logistic activation. | 0.00639 | 0.00495 | 0.3742 | 0.3748 |
| MLP2 | 1 input vector, 2 hidden layers, 10, 5 nodes, linear output learningrate = 0.001, and tanh activation. | 0.00623 | 0.00478 | 0.3612 | 0.3617 |
| MLP3 | 1 input vector, 3 hidden layers, 15,10,5 nodes, linear output learningrate = 0.0001, and relu activation. | 0.00628 | 0.00484 | 0.3655 | 0.3659 |
| MLP4 | 1 input vector, 1 hidden layers, 5 nodes, non-linear output, learningrate = 0.01 and logistic activation. | 0.3249 | 0.3243 | 24.475 | 27.897 |
| MLP5 | 1 input vector, 2 hidden layers, 10, 5 nodes, non-linear output learningrate = 0.001, and tanh activation. | 0.3249 | 0.3243 | 24.4754 | 27.8975 |
| MLP6 | 1 input vector, 3 hidden layers, 15, 10, 5 nodes, non-linear output, learningrate = 0.0001 and relu activation. | 0.32495 | 0.3244 | 24.48 | 27.9 |
| MLP7 | 1-2 input vector, 1 hidden layers, 5 nodes, linear output, learningrate = 0.01 and logistic activation. | 0.00651 | 0.0050 | 0.3777 | 0.3783 |
| MLP8 | 1-2 input vector, 2 hidden layers, 10, 5 nodes, linear output learningrate = 0.001, and tanh activation. | 0.00649 | 0.00494 | 0.3735 | 0.3739 |
| MLP9 | 1-2 input vector, 3 hidden layers, 15, 10, 5 nodes, linear output learningrate = 0.0001, and relu activation. | 0.00765 | 0.00583 | 0.4405 | 0.4413 |
| MLP10 | 1-2 input vector, 1 hidden layers, 5 nodes, non-linear output, learningrate = 0.01 and logistic activation. | 0.3249 | 0.3243 | 24.475 | 27.897 |
| MLP11 | 1-2 input vector, 2 hidden layers, 10, 5 nodes, non-linear output, learningrate = 0.001 and tanh activation. | 0.3249 | 0.3243 | 24.4751 | 27.8971 |
| MLP12 | 1-2 input vector, 3 hidden layers, 15, 10, 5 nodes, non-linear output, learningrate = 0.0001 and relu activation. | 0.3250 | 0.3244 | 24.477 | 27.899 |
| MLP13 | 1-2-3 input vector, 1 hidden layers, 5 nodes, linear output, and logistic activation. | 0.00705 | 0.0057 | 0.4267 | 0.4271 |

1. To determine the efficiency of the best one-hidden layer and two-hidden layer networks, we can consider the total number of weight parameters per network. Generally, a more efficient network would have fewer parameters while maintaining good performance. Let's analyze the table to make a comparison:

Best one-hidden layer network: MLP1

Number of hidden layers: 1

Number of nodes in the hidden layer: 5

Total number of weight parameters for MLP1: (5\*1)+(1\*5)= 10

Best two-hidden layer network: MLP8

Number of hidden layers: 2

Number of nodes in the first hidden layer: 10

Number of nodes in the second hidden layer: 5

Total number of weight parameters for MLP8 = (2 \* 10) + (10 \* 5) + (5 \* 1)

= 20 + 50 + 5

= 75

Comparing the two networks, we can observe that MLP8 with two-hidden layers has more weight parameters compared to MLP1 with a single hidden layer. This is because MLP8 has additional connections between the first and second hidden layers, leading to a higher number of weights.

In terms of efficiency, then MLP1 is more efficient than MLP8 because it has a lower total number of weight parameters compared to MLP8.

However, I prefer the MLP8 approach because additional weights in MLP8 (two-hidden layer network) may provide more flexibility in capturing complex patterns in the data, potentially leading to improved forecasting accuracy. It also has a slightly lower MAE, MAPE and sMAPE than MLP1.



# ans h

library(neuralnet)

library(caret)

smape <- function(actual, predicted) {

200 \* mean(abs(actual - predicted) / (abs(actual) + abs(predicted)))

}

input\_vector <- 1:2

model <- neuralnet(

formula = train\_output ~ .,

data = cbind(train\_input[, input\_vector], train\_output),

hidden = c(10, 5),

linear.output = TRUE,

learningrate = 0.001

)

predictions <- predict(model, cbind(test\_input[, input\_vector]))

# Calculate stat. indices

rmse <- RMSE(predictions, test\_output)

mae <- MAE(predictions, test\_output)

mape <- mean(abs(predictions - test\_output) / test\_output) \* 100

smape\_val <- smape(predictions, test\_output)

# Create a scatter plot

plot(predictions, test\_output, main = "MLP8 Predictions vs. Desired Output", xlab = "Predictions", ylab = "Desired Output")

abline(0, 1)

# Print the stat. indices

cat("RMSE:", rmse, "\n")

cat("MAE:", mae, "\n")

cat("MAPE:", mape, "\n")

cat("sMAPE:", smape\_val, "\n")

