# Machine Learning & Statistical Data Mining

## - Stock Trend Prediction Analysis -
## The JPMorgan Chase & Co (JPM) case

# Table of Contents

## Abstract

This aim of this study is to establish whether the stock trend is influenced by past prices only, or whether external factors also play a role. In this report, the analysis only focus one of stock; namely JPMorgan Chase & Co (JPM). The trend at date *t* is generated from the log returns between *t+1* and *t*. When the log return is positive, then the stock trend is classified as *Up*, when it is negative, it is classified as *Down*. Otherwise, it is considered as *Neutral*. For this, several supervised classification models, such as LDA, QDA, SVM, etc. have been selected to assess the predictive power of the past prices, the volume as well as numerous technical indicators to forecast the trend. The result of the experiment indicates that, after feature selection and model optimizations, the best model test performance is obtained by the Quadratic Discriminant Analysis (QDA), the Ridge and the Linear Discriminant Analysis (LDA) models, with an average test accuracy of approximately 80% (and standard deviation of 1.4%) across these models. Consequently, it can be concluded that most of the price variation is mainly attributable to past prices fluctuations (endogenous factors). However, there is still a large proportion (20.00%) that does not seem to be explained by price movements. The potential impact of exogenous factors (such as fundamental or sentiment indicators) is not part of this experiment.


## Definition of Terms

- Basis Point (bp): This represents a percent of one percent, i.e. 0.0001.
- Technical Analysis: This a trading tool employed to forecast securities future movement by analyzing statistics gathered from trading activity [1].
- Technical Indicator: A statistical or otherwise fabricated metric (usually using the underlying price/volume information) to help predict the underlying future price move [2]. Technical indicators used in this study are detailed in the *Data Pre-processing* section.
- API: The application program interface (API) defines how software components should interact. [3]
- XLF Index: this is an exchange traded fund based on the financial sector industry. It contains approximately 60 stocks. The XLF index price moves is an indicator of the sector performance.
- Fundamental analysis: it is a method of evaluating a security net asset value by measuring its intrinsic value, examining related economic, financial and other qualitative and quantitative factors [4].
- Sentiment analysis: this is the process of opinion mining to gauge positive, neutral or negative 'feeling' the market holds, at given point time, relating to a stock performance.

## Software Dependencies

R - Version 3.3.3 & R Studio - Version 0.99.903

## Hardware

Windows 10 64bits platform, supported by a Intel Core i7-67000HQ processor / RAM: DDR4 8GB.

## How to Run the Code

The code can be run in its entirety (or section by section) in R studio. For this, the following files need to be stored under a unique directory:

- Stock_Trend_Following_Analysis_Assignment2.rmd
- JPM.csv
- XLF.csv

These extra files and folder are not required for the correct running of the R code

- StockPriceCropper.ipynb
- The result folder contains the Training/Optimisation/Testing accuracy rate for each model, as well the QDA confusion matrix which lists the sensitivity, precision, F1, etc. measures of the tested model.

The file configuration is shown in *Code Snippet 0*. This is for information only; the code should not be altered.

````
###########################################################################
###########################################################################
#######                Setup / User Configuration
###########################################################################
###########################################################################
```{r set_up}
index_name = "XLF"
constituents_full_path = paste(getwd(),"/",index_name,".csv",sep="")
stock_mk_price_dir = paste(getwd(),"/",sep="")
out_dir = paste(getwd(),"/",sep="")
````

Code Snippet 0 – Configuration
````

## Introduction

The aim of this study is to evaluate the prediction power of lagged prices, the volume and numerous technical indicators for forecasting stock price movements (*Up*/*Down* and *Neutral*). This experiment only focuses on one stock, namely JPM, but the methodology can be extended to other stocks, indices, and asset classes (such as FX, Commodities, etc). To achieve this, the stock price data is downloaded from the Yahoo Finance web site [5] using a hand-crafted python web scraping program (c.f. section *Data Description*). The daily log stock returns are transformed into *Up*/*Down* and *Neutral* categories, indicating respectively whether the stock increased/decreased or remained unchanged between a date *t* and *t+1*. These categories are then stored under a new explained variable called *Direction*. Numerous technical indicators are generated (c.f. section *Data Pre-processing*) and used as attributes, amongst other explanatory variables (such as lagged prices and the volume), to attempt to predict the trend *Direction*.

Missing data produce a prediction bias. Therefore, records containing missing data are removed (c.f. section *Missing Data*). Following this, the analysis starts with a feature selection. The aim is to reduce the model complexity and focus on attributes concentrating the most prediction power (i.e. the most variance). The next step focuses on training and optimising several supervised classification models (such as LDA, QDA, SVM, etc.,) against the JPM times series *Direction* response variable. The training and optimisation phase are performed on a sliding time window, as the usual cross validation methodology should be avoided in this instance. Indeed, time series contain patterns that need to be chronologically preserved (c.f. section *The Sliding Time Windows*). The last phase involves choosing the best fitting model on training data and optimising the model hyper-parameters. Once the best training model is selected, the model is ready for testing against an unseen test data set.

## Data Description

The JPM stock historical price data (a.k.a. the time series) is at the heart of the analysis. It was downloaded from the *Yahoo IChart API*, via a custom-made *Python* program. The following piece of code describes the common functions used for downloading data from the Yahoo API (c.f. '*In [6]*' the *pull_historical_data()* function). The main calling function lives in section '*In [8]*' below. The code is fully commented in the *Code Snippet 1* section below. A fully running version of the code is available in *StockPriceCropper.ipynb*. The result of running this code is the production of the JPM.csv file that contains the historical price/volume information.

The time series for the JPM stock starts on 30-Dec-1983 and ends on 14-Dec-2016. It contains the *Open/High/Low/Close/Volume* and *Adj Close* for each date. They represent successively: i) the daily opening price, ii) the highest/lowest daily price, iii) the daily price close level, iv) the daily closing price amendment (i.e. adjustment), i.e. the close stock price including any distributions and corporate actions that occurred at any time prior to the next day's opening, as well as iv) the volume. The volume represents the number of shares that changed hands during the trading hours.

### Stock Prices Cropper Functions

In [6]:
```
print ("Start Stock Prices Cropper Functions")

base_url = "http://ichart.finance.yahoo.com/table.csv?s="
input_path = output_path = os.getcwd()

#Create the directory, if it does not exist
def mkdir(directory_full_path):
    if not os.path.exists(directory_full_path):
        os.makedirs(directory_full_path)

#Generate the full URL basd on the base_url and the ticker name (e.g. JPM)
def make_url(ticker_symbol):
    return base_url + ticker_symbol

#Returns the directory output name as well as the full file path
def make_output_filename(ticker_symbol, directory="Unknown"):
    return output_path + "/" + directory + "/", output_path + "/" + directory + "/" + ticker_symbol + ".csv"

def pull_historical_data(ticker_symbol, directory="Unknown"):
    try:
        #Generate the output directory and full path for the stock file name (e.g. ./JPM.csv)
        directory_full_path, file_full_path = make_output_filename(ticker_symbol, directory)
        #Generate the full URL
        ticker_url = make_url(ticker_symbol)
        #Make the directory if it does not exist
        mkdir(directory_full_path)
        #Get the data from the url and store it in the defined file path
        urlretrieve(ticker_url, file_full_path)
    except Exception as e: # catch *all* exceptions
        directory_full_path, file_full_path = make_output_filename(ticker_symbol, directory)
        outfile = open(file_full_path, "w")
        print(e)
        outfile.write(str(e))
        outfile.close()
print ("End Stock Prices Cropper Functions")
```

```
Start Stock Prices Cropper Functions
End Stock Prices Cropper Functions
```

# Run Stock Prices Cropper...

The data is sourced for free from Yahoo, via the Yahoo API.

In [8]:
```
print ("Start Cropping...")
#Get the JPM historical data from beg of time....
pull_historical_data ('JPM', "")
print ("End Cropping...")
```

```
Start Cropping...
End Cropping...
```

Code Snippet 1 – JPM Time Series Download

## Data Pre-processing

The process of analysis trends in times series usually involves the fabrication of numerous indicators directly derived from the data under analysis. In this case, the historical stock Close price. Trend filtering methods uses econometric/ statistical estimators to extract trends from time series [6]. In this study, we generate and study the impact of a selection of the indicators presented in [7]. Unless stated otherwise, the Close price is used to generate technical indicators. Some metrics, such as the *Average True Range (ATR)* may use the HLC (high/Low/Close) price. The HLC price aggregate is built as follows: HLC = (High+Low+Close)/3. The following sections details all the technical indicators and other derived data formulas that are used as part of the model feature selection. All technical indicators have been implemented by calling the relevant R functions from the *quantmod* library *[7bis]* (c.f. Code Snippet 2). In the following sections *[-1]* means today minus one business day. This is also known as *t-1*.

```r
#Add a the indicators column (the average value of the High,Low and Close)
sma_df["Hlc"] = (sma_df["High"] + sma_df["Low"] + sma_df["Close"])/3
sma_df["Hl"] = (sma_df["High"] + sma_df["Low"]) /2
sma_df["Sma20"] = SMA(sma_df$Close, n=20)
sma_df["Sma50"] = SMA(sma_df$Close, n=50)
sma_df["Sma100"] = SMA(sma_df$Close, n=100)
sma_df["Sma200"] = SMA(sma_df$Close, n=200)
sma_df["Ema20"] = EMA(sma_df$Close, n=20)
sma_df["Ema50"] = EMA(sma_df$Close, n=50)
sma_df["Ema100"] = EMA(sma_df$Close, n=100)
sma_df["Ema200"] = EMA(sma_df$Close, n=200)
sma_df["Rsi"] = RSI(sma_df$Close, n=14)
sma_df["Atr_tr"] = ATR(HLC(sma_df[,c("High","Low","Close")]))[][,1]
sma_df["Atr_atr"] = ATR(HLC(sma_df[,c("High","Low","Close")]))[][,2]
sma_df["Atr_trueHigh"] = ATR(HLC(sma_df[,c("High","Low","Close")]))[][,3]
sma_df["Atr_trueLow"] = ATR(HLC(sma_df[,c("High","Low","Close")]))[][,4]
sma_df["Smi_smi"] = SMI(HLC(sma_df[,c("High","Low","Close")]))[][,1]
sma_df["Smi_signal"] = SMI(HLC(sma_df[,c("High","Low","Close")]))[][,2]
sma_df["Macd_macd"] = MACD((sma_df[,c("Close")]))[][,1]
sma_df["Macd_signal"] = MACD((sma_df[,c("Close")]))[][,2]
sma_df["Bb_dn"] = BBands(HLC(sma_df[,c("High","Low","Close")]))[][,1]
sma_df["Bb_mavg"] = BBands(HLC(sma_df[,c("High","Low","Close")]))[][,2]
sma_df["Bb_up"] = BBands(HLC(sma_df[,c("High","Low","Close")]))[][,3]
sma_df["Bb_pctB"] = BBands(HLC(sma_df[,c("High","Low","Close")]))[][,4]
sma_df["Mfi"] = MFI(HLC(sma_df[,c("High","Low","Close")]), sma_df[,c("Volume")])
sma_df["Sar"] = SAR(sma_df[,c("High","Close")])
sma_df["Volatility"] = volatility(OHLC(sma_df[,c("Open","High","Low","Close")]), calc = "garman")
```

Code Snippet 2 – Technical Indicators implementations

### Simple Moving Average (Sma) 20/50/100/200

Each output value is the average of the previous *n* day values (e.g. 20/50/100/200 days). As each value in the period carries equal weight, it makes it less responsive to recent changes [8].

$$SMA = \frac{\sum_{1}^{n} price}{n}$$

With n the number of lag day for the moving average (e.g. 20/50/100/200)

### Exponential Moving Average (Ema) 20/50/100/200

Exponential Moving Average is a cumulative calculation, including all data (even values outside of the period). More recent values have a greater contribution to the average, past values have a diminishing contribution [8][9].

Multiplier: (2 / (time periods + 1) )
EMA: {Close - EMA(previous day)} x multiplier + EMA(previous day).
With *time periods* being the number of lag day for the moving average (e.g. 20/50/100/200)

### Relative Strength Index (RSI)

It produces a ratio of the recent upward price movements to the absolute price movement. The index ranges from 0 to 100. A RSI strictly greater 70 is interpreted as an overbought indicator. When the RSI is strictly below 30, it is interpreted as an oversold indicator [8].

$$
\begin{aligned}
&\text{If } close > close_{[-1]} \text{ then}\\
&\quad up = close - close_{[-1]}\\
&\quad dn = 0\\
&\text{else}\\
&\quad up = 0\\
&\quad dn = close_{[-1]} - close\\
&upavg = \frac{upavg \times (n-1) + up}{n}\\
&dnavg = \frac{dnavg \times (n-1) + dn}{n}\\
&RMI = 100 \times \frac{upavg}{upavg + dnavg}
\end{aligned}
$$

### Average True Range (ATR)

The ATR relates to the True Range Welles Wilder moving average. The ATR is a measure of volatility. High ATR values indicate high volatility, and low values indicate low volatility. The True range is used to determine the usual trading range of a stock [8].

$$
\begin{aligned}
&TrueHigh = \text{Highest of } high_{[0]} \text{ or } close_{[-1]}\\
&TrueLow = \text{Lowest of } low_{[0]} \text{ or } close_{[-1]}\\
&TR = TrueHigh - TrueLow
\end{aligned}
$$

The formula is sometimes stated as:

$$
\begin{aligned}
TR = \text{The greatest of the following:}&\\
&|\, high_{[0]} - low_{[-0]} \,|\\
&|\, high_{[0]} - close_{[-1]} \,|\\
&|\, low_{[0]} - close_{[-1]} \,|
\end{aligned}
$$

$$
\begin{aligned}
&TrueHigh = \text{Highest of } high_{[0]} \text{ or } close_{[-1]}\\
&TrueLow = \text{Lowest of } low_{[0]} \text{ or } close_{[-1]}\\
&TR = TrueHigh - TrueLow\\
&ATR = \frac{TR_{[-1]} \times (n-1) + TR}{n}
\end{aligned}
$$

### Stochastic Momentum Index (SMI)

The SMI is built on the Stochastic Oscillator. The Stochastic Oscillator calculates the position of the close price in relation to the high/low range. The SMI calculates the position of the Close price in relation to high/low range midpoint. SMI values range between +100 and -100. The SMI is greater than zero, when a close is greater than the midpoint (and reverse).

Extreme high/low SMI values indicate overbought/oversold conditions. A buy signal is issued when the SMI rises above -50, or when it crosses above the signal line (and reverse).

$$
\begin{aligned}
&cm = close - \frac{highesthigh - lowestlow}{2}\\
&hl = highesthigh - lowestlow\\
&cm = \text{EMA}(\text{EMA}(cm))\\
&hl = \text{EMA}(\text{EMA}(hl))\\
&SMI = 100 \times \left(\frac{cm}{hl/2}\right)\\
&Signal = \text{EMA}(SMI)
\end{aligned}
$$

### Moving Average Convergence/Divergence (MACD)

The MACD is the difference between two Exponential Moving Averages; a short and long one. The Signal line is an Exponential Moving Average of the MACD [8].
High/low values indicate an overbought/oversold asset. When the MACD line crosses above/below the signal line a buy/sell signal is generated. The signal is confirmed when the MACD is above/below zero buy/sell.

$$shortema = 0.15 \times price + 0.85 * shortema_{[-1]}$$
$$longema = 0.075 \times price + 0.925 * longema_{[-1]}$$
$$MACD = shortema - longema$$

### Bollinger Bands

Bollinger Bands consist of three lines: the simple moving average and he upper/lower bands. The later are usually two standard deviations above/below [8].
Bollinger Bands are not, in themselves, buy or sell signals. They indicate overbought or oversold conditions. A price near to the upper or lower band indicates a potential imminent reversal. The moving average becomes a support or resistance level.

$$TP = \frac{high + low + close}{3}$$
$$MidBand = \mathrm{SimpleMovingAverage}(TP)$$
$$UpperBand = MidBand + F \times \sigma(TP)$$
$$LowerBand = MidBand - F \times \sigma(TP)$$

With F usually set to 2.

### Money Flow Index (MFI)

The MFI calculates the ratio of money flowing into and out of a security [8]. Money Flow Index ranges between 0 and 100. Values above 80/below 20 indicate market tops/bottoms.

$$TypicalPrice = \frac{high + low + close}{3}$$
$$MoneyFlow = Typical\ Price \times volume$$
$$\text{If } Typical\ Price > Typical\ Price_{[-1]}$$
$$\quad PositiveMoneyFlow = PositiveMoneyFlow_{[-1]} + MoneyFlow$$
$$\text{else}$$
$$\quad NegativeMoneyFlow = NegativeMoneyFlow_{[-1]} + MoneyFlow$$
$$MoneyRatio_i = \frac{\sum_{i-n}^{i} PositiveMoneyFlow}{\sum_{i-n}^{i} NegativeMoneyFlow}$$
$$MoneyFlowIndex = 100 - \left(\frac{100}{1 + MoneyRatio}\right)$$

### Parabolic Stop And Reverse (SAR)

SAR calculates a trailing stop [8]. SAR advises to exit when the price crosses the SAR.

$$\text{If long and } high > xp \text{ then}$$
$$\quad xp = high$$
$$\quad af = af + step$$
$$\text{If short and } low < xp \text{ then}$$
$$\quad xp = low$$
$$\quad af = af + step$$
$$SAR = (xp - SAR_{[-1]}) \times af + SAR_{[-1]}$$

xp is extreme point
af is acceleration factor

### Volatility

This indicator generates the Close price degree if variation over a business 260 days' period.

$$\sigma_{cl} = \sqrt{\frac{Z}{n-2}\sum_{i=1}^{n-1}(r_i - \bar{r})^2}$$

$$where \quad r_i = \log\left(\frac{C_i}{C_{i-1}}\right)$$

$$and \quad \bar{r} = \frac{r_1 + r_2 + \ldots + r_{n-1}}{n-1}$$

### The price lags

The *t-1, t-2, t-3, t-4* and *t-5* are the price lags, i.e. the *t-n* prices used at date *t*. These price lags are used as attributes to establish whether they have an explanatory power on the trend at date *t*.

```
  #Add day lags
  sma_df = add_close_price_day_lag(sma_df, 1)
  sma_df = add_close_price_day_lag(sma_df, 2)
  sma_df = add_close_price_day_lag(sma_df, 3)
  sma_df = add_close_price_day_lag(sma_df, 4)
  sma_df = add_close_price_day_lag(sma_df, 5)
```

```
#Add the Close price day lag to the analysis
add_close_price_day_lag = function (stock_df, nb_days){
  #Ensure the dataset is ordered in data descending order,
  #as the calculations assume the ordering for the calculation

  #Generate the column dynalically based on the nb_days
  close_price_lag = paste("Close_price_", nb_days, "day_lag", sep="")
  for (k in 1:nrow(stock_df)) {
    stock_df[k,close_price_lag] = stock_df[k+nb_days,"Close"]
  }
  return (stock_df)
}
```

Code Snippet 3 – The Price lag main function

### The Volume

The volume corresponds to the amount of stock traded during each day. It is provided as part of the original data. No data transformation is performed on this quantity.

## Dummification

As the study focuses on predicting stock market trends, the price is not used directly as the regressor. The main reason is that the price is not log normally distributed. However, the log return is. Prices have therefore been transformed into log returns for each date t, as follows:

$$Return_t = \log (Price_{t+1}/Price_t).$$

A new attribute named *Direction* is generated from $Return_t$. It is defined as per the formula below. The ε (*Epsilon*) is a user defined constant, currently set to 0.0025 (i.e. 25 basis points). This was the level chosen for the JPM stock to ensure a balanced *Direction* class. More details about class balancing are provided in the section 'Skew Reduction'.

***A date t:***

$$\begin{cases} \text{Return @t} > \varepsilon \text{ then } Direction\ is\ set\ to\ Up \\ \text{Return @t} < -\varepsilon \text{ then } Direction\ is\ set\ to\ Down \\ Abs(\text{Return @t}) \leq \varepsilon \text{ then } Direction \text{ is set to } Neutral \end{cases}$$

The *Direction* attribute is the predicted variable that is used in all models.   The code implementation can be found in *Code Snippet 4* below.

```
#This function the log_returns. It indicates whether the market when up, down or sideways (neutral) from one day to the next.
#Please ensure the dataset is ordered in data descending order, as the calculations assume the ordering for the calculation
generate_log_returns_and_direction = function (stock_df, epsilon){
  stock_df["Direction"] = NA
  stock_df["Log_returns"] = NA

  for (k in 1:nrow(stock_df)) {
      #Generate the log return value
      stock_df[k,"Log_returns"] =  log10(stock_df$Close[k+1] / stock_df$Close[k])
      #Indicate whether it is going up, down or neutral
      if (k < nrow(stock_df)) {
        if (!is.na( stock_df[k,"Log_returns"])){
          #This is the default values of the 'Direction' and 'Direction_Flag'
          stock_df[k,"Direction"] = 'NA'
          stock_df[k,"Direction_Flag"] = -1
          #Do the Epsilon check
          if (abs(stock_df[k,"Log_returns"]) <= epsilon){
            stock_df[k,"Direction"] = 'Neutral'
            stock_df[k,"Direction_Flag"] = 1
          }
          else {
            #when the return is greater today compared to the day before,
            #then set the Direction to Up and Direction_Flag to 2
            if (stock_df[k,"Log_returns"] > epsilon){
                stock_df[k,"Direction"] = 'Up'
                stock_df[k,"Direction_Flag"] = 2
            }
            #Else set the Direction to Up and Direction_Flag to 2
            else {
                stock_df[k,"Direction"] = 'Down'
                stock_df[k,"Direction_Flag"] = 0
            }
          }
        }
      }
    }
  return(stock_df)
```

Code Snippet 4 – The predicted variable Dummification

## Skewness Reduction

As mentioned in [10], skew or imbalance in data affect most performance metrics (e.g. Kappa, F-1 scores, etc.). The below tables show the share of each trend category in the *Direction* class. The first table shows that once rows containing missing data are removed, and ε is set to 0, the *Neutral* category is greatly underrepresented in the *Direction* class (only 4.17%). To rebalance the class distribution, several values for ε were tested. The one offering the best-balanced ratio between the classes was retained, where ε equals to 0.0025.

| Row Labels | Count of Epsilon 0.0000 | % Class Allocation |
|---|---|---|
| DOWN | 3853 | 47.53% |
| NEUTRAL | 338 | 4.17% |
| UP | 3916 | 48.30% |
| **Grand Total** | **8107** | **100.00%** |

The following tables shows the *Direction* class balance for different level of ε.

| Row Labels | Count of Epsilon 0.0015 | % Class Allocation |
|---|---|---|
| DOWN | 3214 | 39.65% |
| NEUTRAL | 1579 | 19.48% |
| UP | 3314 | 40.87% |
| **Grand Total** | **8107** | **100.00%** |

| Row Labels | Count of Epsilon 0.0020 | % Class Allocation |
|---|---|---|
| DOWN | 2979 | 36.75% |
| NEUTRAL | 2065 | 25.46% |
| UP | 3063 | 37.79% |
| **Grand Total** | **8107** | **100.00%** |

| Row Labels | Count of Epsilon 0.0025 | % Class Allocation |
|---|---|---|
| DOWN | 2763 | 34.08% |
| NEUTRAL | 2500 | 30.84% |
| UP | 2844 | 35.08% |
| **Grand Total** | **8107** | **100.00%** |

| Row Labels | Count of Epsilon 0.0030 | % Class Allocation |
|---|---|---|
| DOWN | 2538 | 31.31% |
| NEUTRAL | 2937 | 36.22% |
| UP | 2632 | 32.47% |
| **Grand Total** | **8107** | **100.00%** |

## Missing Data

The initial raw market data, downloaded from the 'Yahoo API', do not contain any missing data. However, the data pre-processing step generates missing data, for many dates. For example, the Moving Average indictors need numerous historical dates to produce a value for a given date. Consequently, lagging indicators cannot produce data for the initial number of days corresponding to their day range lag. Data imputation, using median or K-Nearest Neighbours methods were considered. However, this would have created 'synthetic' patterns in the time series data, that do not correspond to reality. Indeed, the literature [11] suggests that financial returns do not follow a Brownian motion, i.e. a random motion. On the contrary, in case of financial crisis, volatility clustering can be observed. Volatility can also display some multifractal behaviour, which intrinsically implies regularities and volatilities patterns. Consequently, the preferred solution was simply to delete the missing data rows. The *Code Snippet 5* shows the code relating to the deletion of missing data. Approximately 12% of the rows contained in the original dataset were removed.

```
##########################################################################
##########################################################################
#######            Missing Data
##########################################################################
##########################################################################

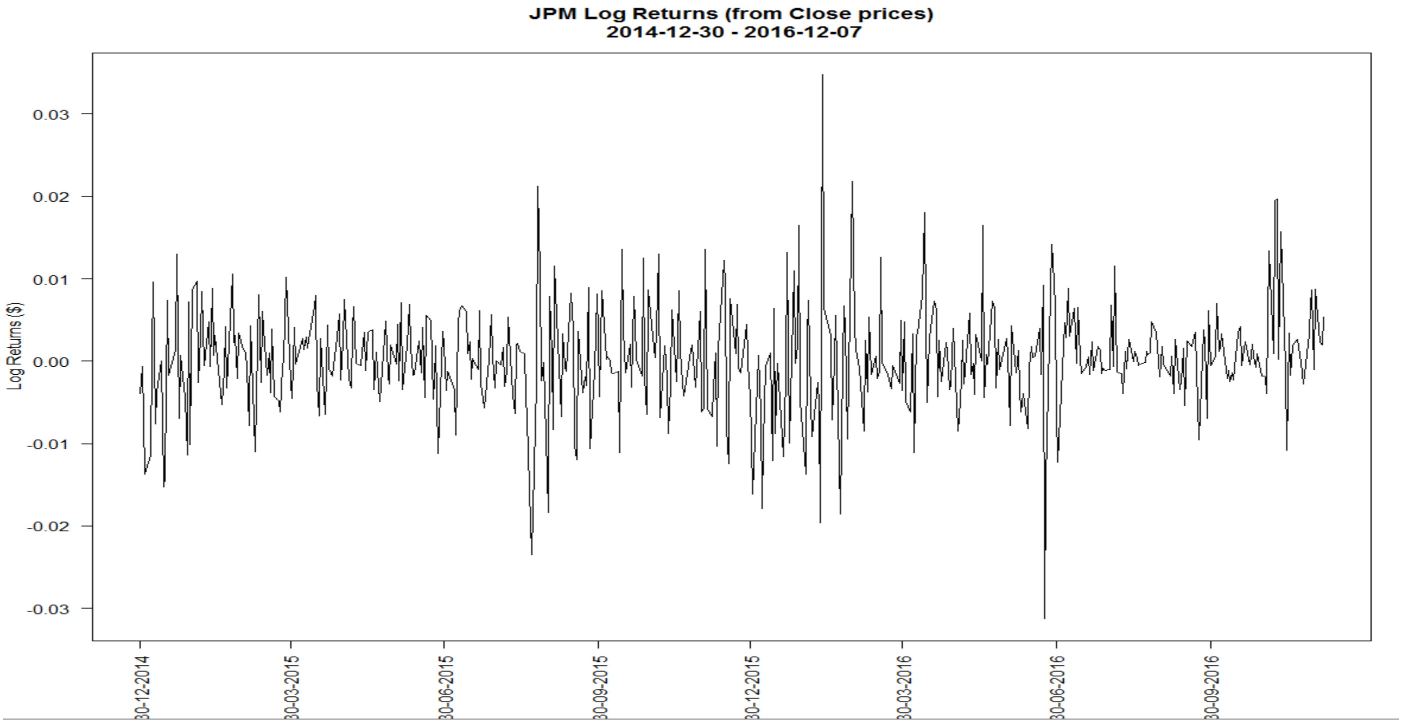
#Remove all rows containing NA values
#count the number of na in the entire dataset
count_na = sum(is.na(sma_df))
#remove all the rows containing na values
sma_df = na.omit(sma_df)
#count the perc of removed row and perform some sanity check
removed_missing_values_perc = 100 * count_na / nrow(sma_df)
cat("Removed missing values represent ", removed_missing_values_perc, " % of the dataset\n", sep ="" )
count_na = sum(is.na(sma_df))
if (count_na > 0) {
  cat("**** WARNING... **** \n")
  cat("The NA count in the dataframe should be zero. The actual result is: ", count_na, "\n", sep ="" )
}

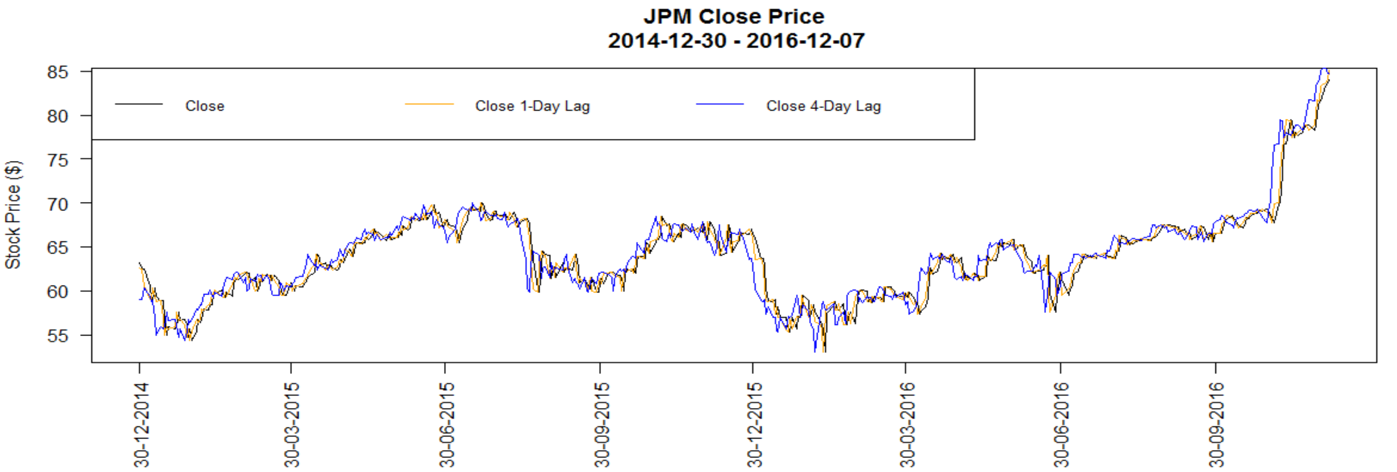
```

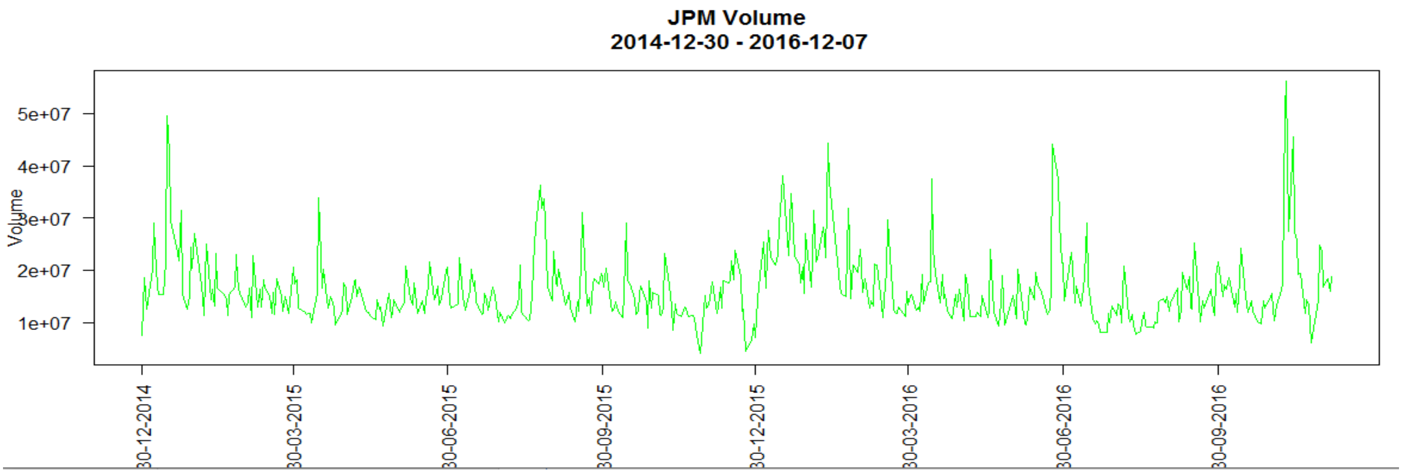
Code Snippet 5 – Missing Data Removal

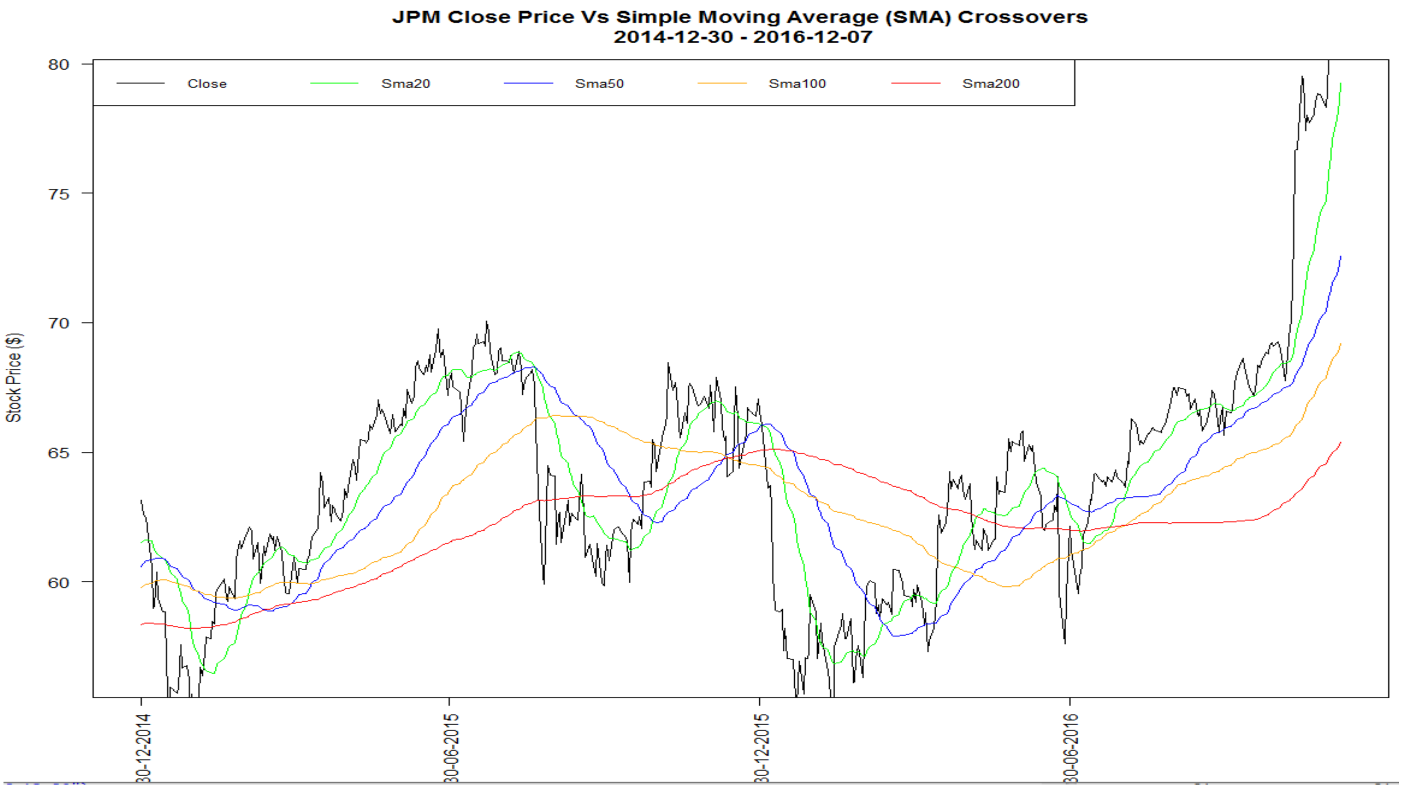
## Data Visualisation

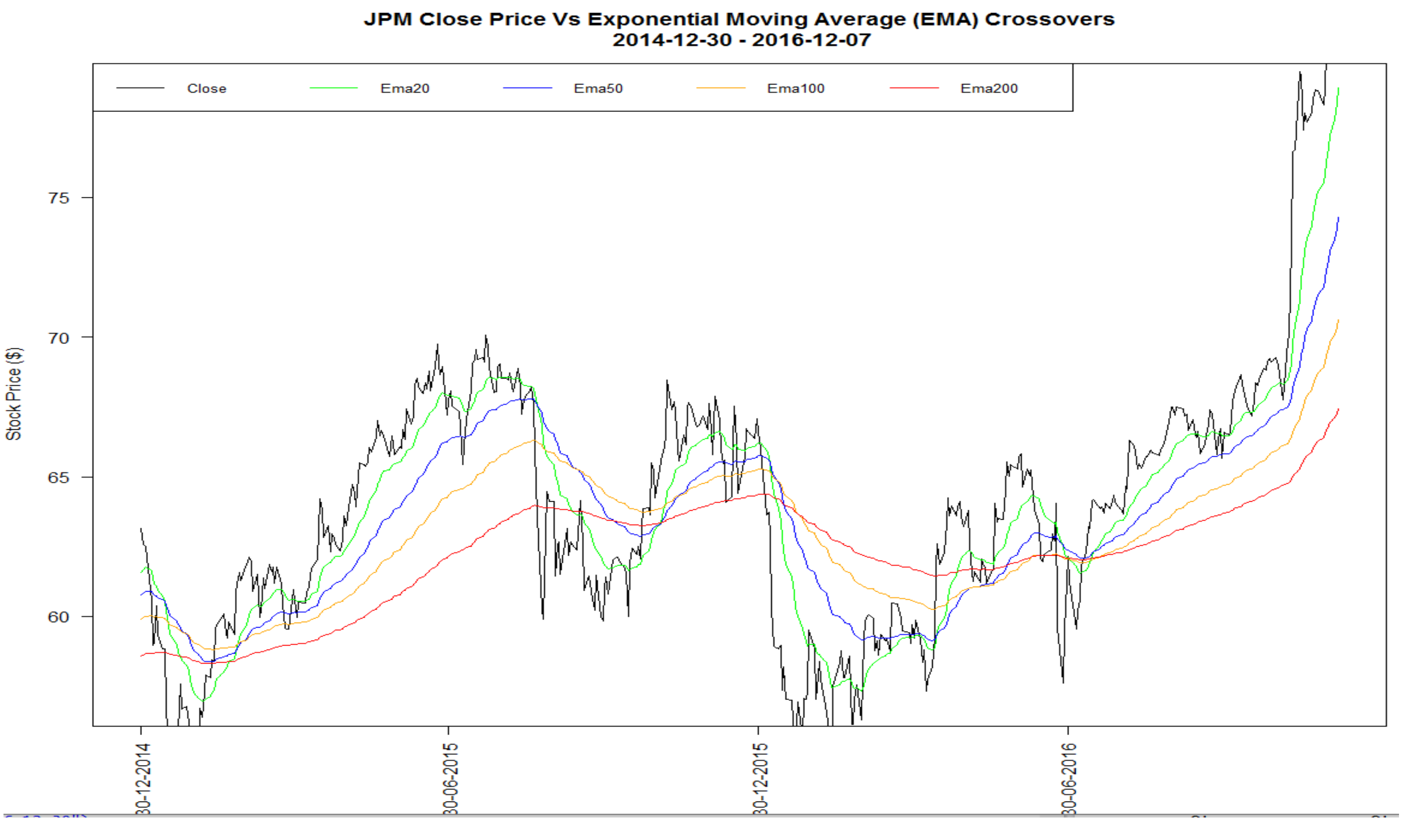
The purpose of this section is to provide several graphics to visualise the Log Returns pattern (i.e. the implicit direction) over a two-year period, alongside several of the main attributes that are part of the feature selection process. The R code for the graphics generation can be found in Appendix B.

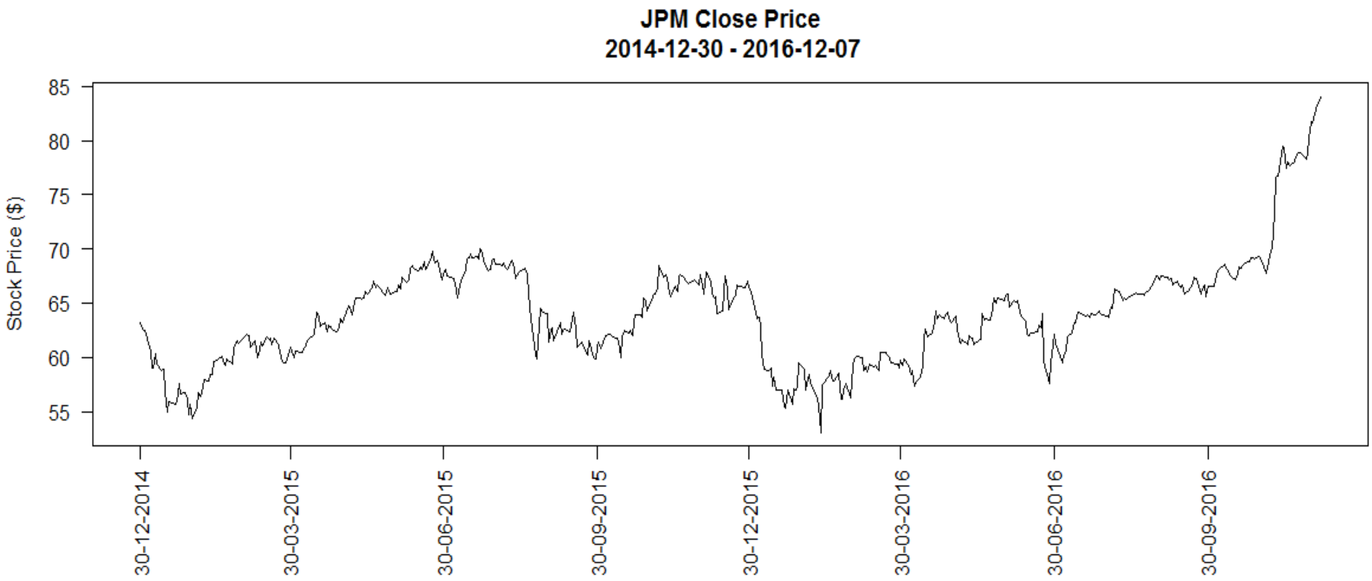


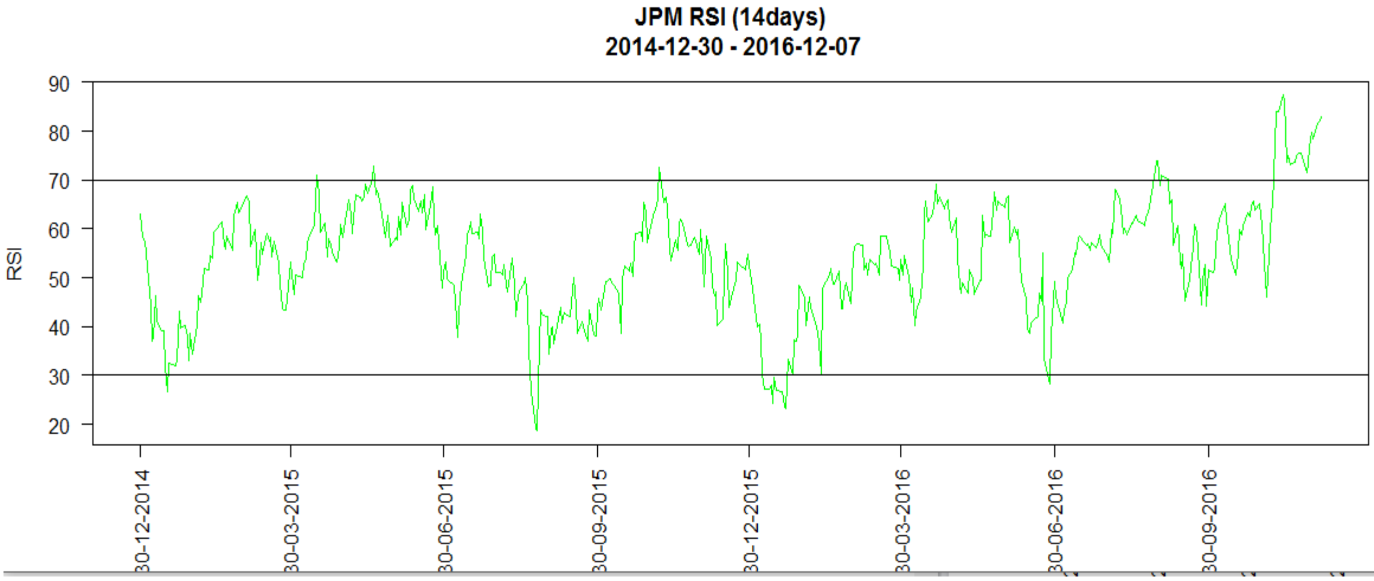




JPM Close Price Vs Simple Moving Average (SMA) Crossovers
2014-12-30 - 2016-12-07
Close
Sma20
Sma50
Sma100
Sma200
Stock Price ($)
80
75
70
65
60
30-12-2014
30-06-2015
30-12-2015
30-06-2016


JPM Close Price Vs Exponential Moving Average (EMA) Crossovers
2014-12-30 - 2016-12-07
Close
Ema20
Ema50
Ema100
Ema200
Stock Price ($)
75
70
65
60
30-12-2014
30-06-2015
30-12-2015
30-06-2016




The RSI horizontal top line represents the 70% boundary. The bottom line represents the 30% boundary.

## The Sliding Time Windows

Financial time series contains historical patterns, particularly during turbulent periods such as wars, economic crisis, presidential election, etc. The aim of machine learning is to recognise these patterns to form a prediction. A frequent approach to reduce model overfitting is the cross-validation methodology (either the K-fold or Leave-One-Out). However, the re-sampling/shuffling nature of these methodologies do intrinsically modify historical patterns. Therefore, it has been ruled out for this experiment. A time window sliding approach is used instead, over a training/validation and test sets. The detail of the methodology is as follows:

- Step1
    - Create a training data set over a given time period (here a continuous 260 business days).
    - Create a validation data set, starting the business day after the end of the training data set, which represents 25% of the dates contained in the training data set (here 65 business days)
    - Create a test set that starts one day after the end of the validation data set, and last for 5 consecutive business days.
- Step2
    - Feature select on the last training set (i.e. the one closest to the end date of the dataset). A discussion of this choice is provided in the section *'Challenges and Potential Improvements'*.
- Step3
    - For each designated model (e.g. LDA, QDA, SVM, etc.):
        - Perform the model training, for a list of different attributes and modified polynomial attributes, on a 100 training/validation data sets sliding windows. In this experiment, there are 100 sliding windows of 5 days in length. In other words, when the model fitting has been performed on one sliding window, the next model fitting starts 5 business days after the previous training window start date. The process is performed 100 times. The aim is to find the set and shape of attributes that provide the best average prediction accuracy on the validation data, over 100 iterations.
        - Select the best attribute list and optimise the model on the last training/validation time window. The aim is to discover the parameters level that optimise the model on the best attribute list.
        - Train the model, its best attribute list and optimised parameters on the entire training and validation data set time frame (i.e. training start date to validation end date). Then test the model against the test data to generate the test accuracy. Run this over the 100 times windows and the average of the test accuracy is produced.

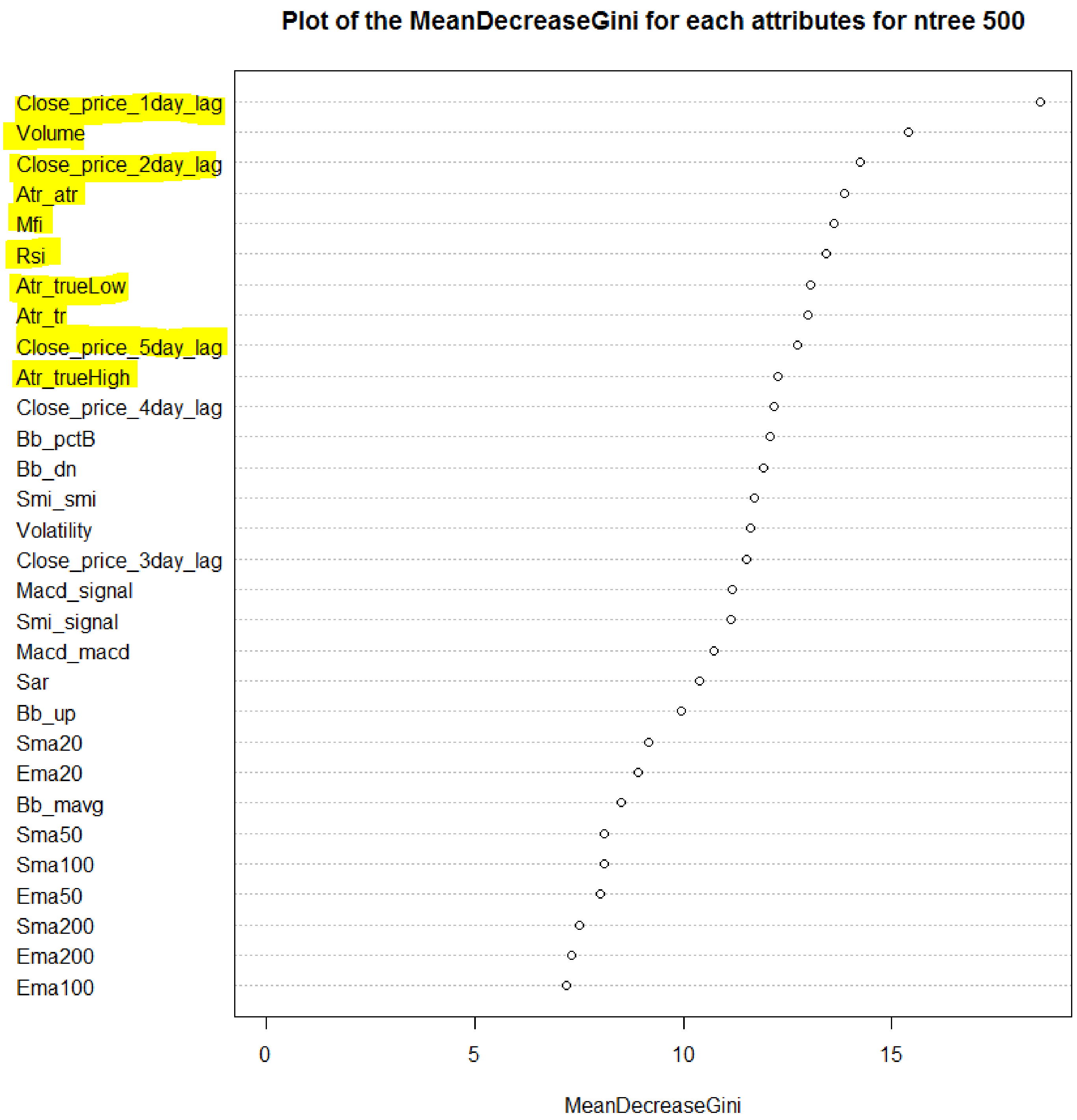
    A discussion relating to the 'out of step optimisation' and choice the 100 iterations is provided in section *'Challenges and Potential Improvements*'.

Initial experiments showed that, when tested with smaller time windows, such as 80 or 160 days; the model fitting in R breaks with the following error messages: 'Need at least two classes to do classification' or 'Some group is too small for *[the selected model]*'. Therefore, an arbitrary training time period of 260 days was selected (after experimentation). It has several advantages:

- It represents a time period slightly greater than one year. (Usually 252 days are used for the calculation of the annualized volatility [12]). Therefore, it enables to capture a larger amount of time series patterns.
- Running the LDA and QDA models on both an 80 days' time windows and a 260-day showed an increase of test performance from approximately 50% to 80% in each case.

## Feature Extraction

A Random Forest Classification model is used to perform the feature selection. This is model is selected as it is a popular machine learning method which couple a relatively good accuracy, robustness and ease of interpretation. The feature selection is performed only the training data of the last time window (the one closer to the dataset end date). A discussion relating to the limitation of this approach is available in section *'Challenges and Potential Improvements'*. The attributes are ordered in descending order of the *'MeanDecreaseGini'* index. The general interpretation is that a higher decreasing Gini indicates that some predicators play a greater role in classifying the data, than others. Currently, the top 10 most impacting attributes have been selected and stored in 'feature_reduction_summary.csv' file. There are also highlighted in yellow in the below picture.



## Normal Distribution Test

Some of the models listed below (e.g. the Linear Discriminant Analysis, the Quadradic Discriminant Analysis models, etc.) require that the explanatory variables follow a normal distribution. Although the lack of compliance should not prevent the use of these models for prediction purpose, it is interesting to establish whether these models are intrinsically statistically weak (or not). If this is the case, this may have an impact of the model stability and performance accuracy. For each of the 10 feature selected attributes, the following steps are involved:

- Create one dataset per *Direction* class type. Using the 'Close_price_1day_lag' feature as an example. It is broken in three distinct data set; one containing only rows where the *Direction* attribute is set to *Up*, a second where the rows contains the *Direction* attribute is set to *Neutral*, and the third one contains the rows where the *Direction* attribute is set to *Down*. In total, 30 datasets (attributes = 10* class types= 3) have been created.

- For each dataset, the attribute is normalized using the *scale(..) function.* It centers and scales the elements in the dataset. In other words, each element x in an attribute is normalized following this formula:

| |
|---|
| (x-avg(x))/std(x), where avg(x) is the mean of x and std(x) is the standard deviation of x. |

- Then a Kolmogorov Smirnov Test is run on each of the dataset to establish whether H0 can be rejected (the null hypothesis) that states; the data distribution follows a normal distribution. H0 is rejected when the Kolmogorov Smirnov Test returns a *p-value* inferior to 0.05. The below code shows the scaling (Line 907), and the Kolmogorov Smirnov Test in action (Line 909-919).

-

```
904  #Produce the results of the KS test
905  kolmogorov_smirnov_normal_distribution_test = function(data,stock_name,msg) {
906    #This is the z-score scaling: (x-avg)/std as default
907    scaled_data = scale(data)
908    #Ensure all data is unique, else it breaks the ks test
909    res = ks.test(unique(scaled_data), pnorm)
910    cat(paste(stock_name, msg, sep =" "))
911    cat("\n")
912    cat("H0 = the data is normally distributed.\n", sep="")
913    if(res$p.value > 0.05){
914        cat("The ks p_value: ", res$p.value, " > 0.05 -> H0 (the null hypothetsis) is NOT rejected. There is not enough evidence to reject the hypothesis that the distribution is normal. Therefore, the data seems to follow normal distribution\n",  sep="")
915    } else
916    {
917        cat("The ks p_value: ", res$p.value, " < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.\n",  sep="")
918    }
919  }
920
```

Code Snippet 6 – Kolmogorov Smirnov Test

- QQPlots are plotted for each attribute to visually check the shape of the attribute distribution against a normally distributed QQ plot. The more the data shape match the normal QQ plot (the yellow line), the more likely is the data to follow a normal distribution. The code relating the QQPlot is shown below:

```
#Produce a qqplot
plot_qqplot = function (data, stock_name, ylab_param) {
  data = data * 100.0
  title = paste(stock_name, ylab_param, sep =" ")
  qqnorm(data, main=paste(title, " \n (centered & scaled) - QQplot", sep=""), ylab=paste(ylab_param, "(%)", paste=""))
  qqline(data, col="gold", lwd=2)
}
```

Code Snippet 7 –The QQplot

The below code snippet shows an end to end example with 'Close_price_1_day_lag' attribute being broken down in three datasets. For each of them the Kolmogorov Smirnov Test and QQPlot is generated. Please refer to the R code to see all the other 27 cases.

```
        ############################ Close_price_1day_lag KS Test ################################################

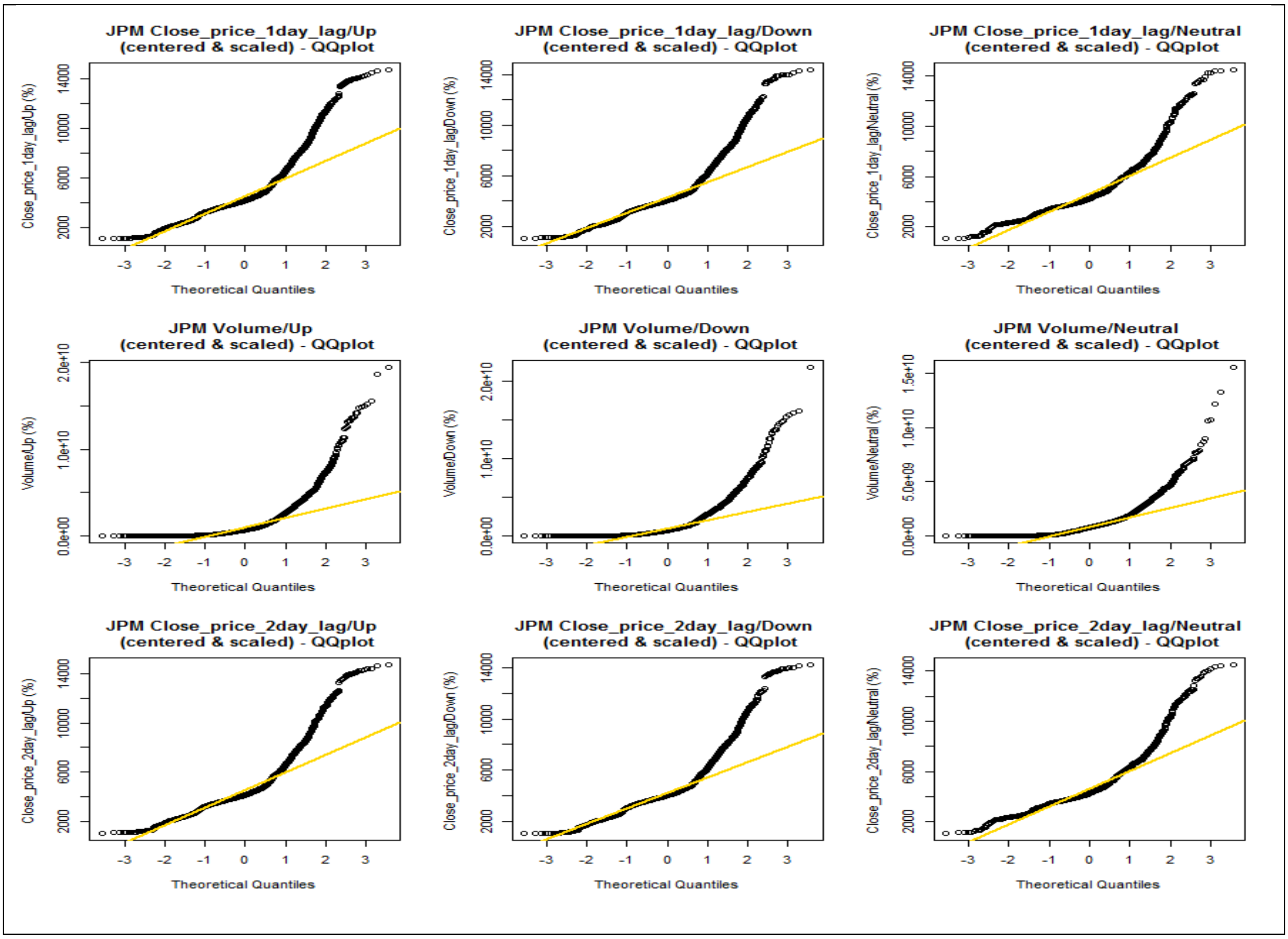
        df =  subset(subset(normality_check_df, select=c(Direction,Close_price_1day_lag)), Direction == 'Up')
        kolmogorov_smirnov_normal_distribution_test(df$Close_price_1day_lag, stock_name, paste("Check ", stock_name, " 'Close_price_1day_lag/Up' is normally distributed", sep=""))
        plot_qqplot (df$Close_price_1day_lag, stock_name, "Close_price_1day_lag/Up")

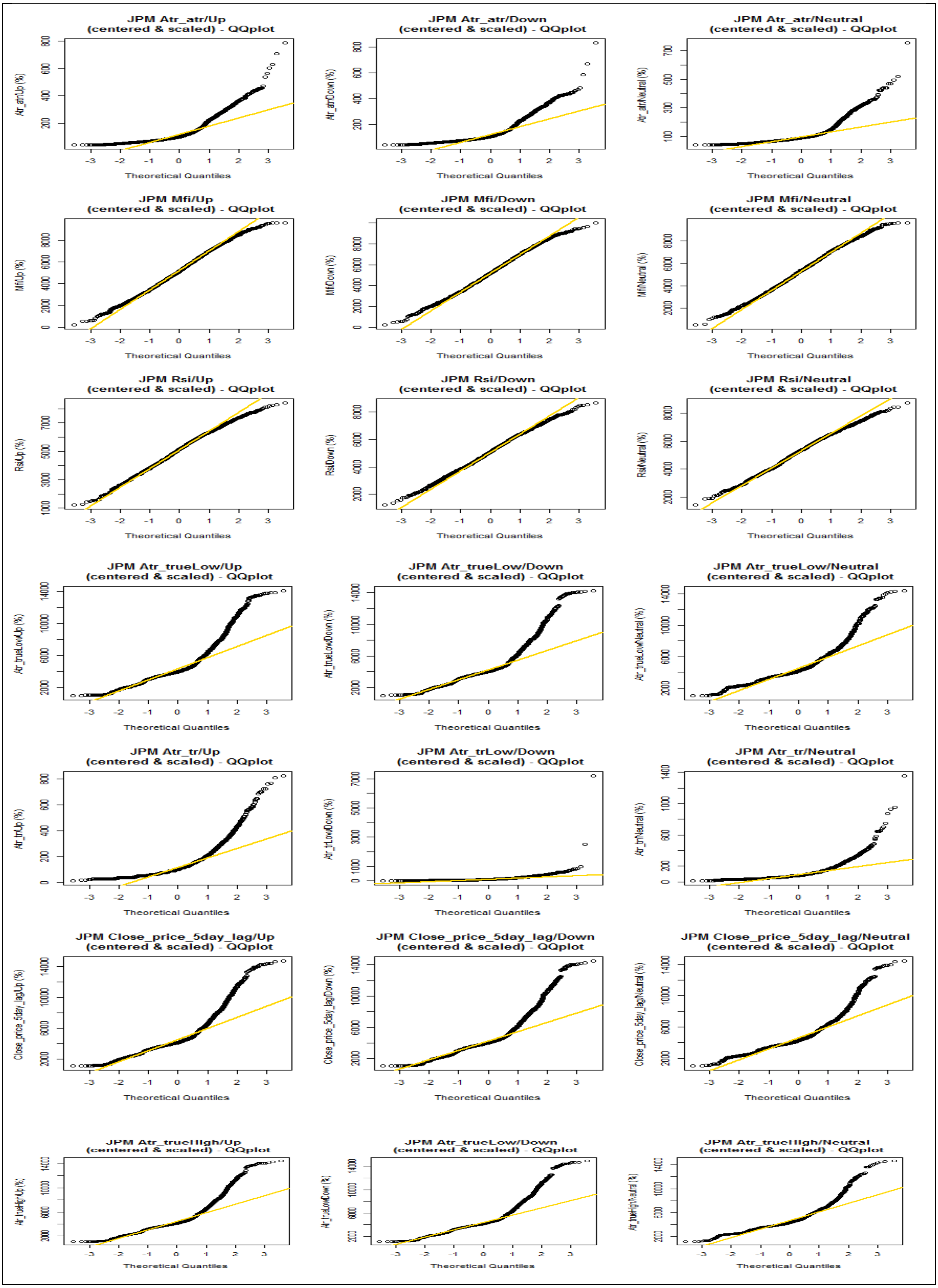
        df =  subset(subset(normality_check_df, select=c(Direction,Close_price_1day_lag)), Direction == 'Down')
        kolmogorov_smirnov_normal_distribution_test(df$Close_price_1day_lag,stock_name, paste("Check ", stock_name, " 'Close_price_1day_lag/Down' is normally distributed", sep=""))
        plot_qqplot (df$Close_price_1day_lag,stock_name, "Close_price_1day_lag/Down")

        df =  subset(subset(normality_check_df, select=c(Direction,Close_price_1day_lag)), Direction == 'Neutral')
        kolmogorov_smirnov_normal_distribution_test(df$Close_price_1day_lag,stock_name, paste("Check ", stock_name, " 'Close_price_1day_lag/Neutral' is normally distributed", sep=""))
        plot_qqplot (df$Close_price_1day_lag,stock_name, "Close_price_1day_lag/Neutral")
```

Code Snippet 8 – Kolmogorov Smirnov Test and QQ plot calling code

The detailed results of the Kolmogorov Smirnov Test can be found in Appendix A. In a nutshell, only the JPM 'Rsi/Down', 'Mfi/Up', 'Mfi/Down' and 'Mfi/Neutral' data set seem to follow a normal distribution. None of the others do. This is also visible from the below following graphs.




JPM Atr_atr/Up (centered & scaled) - QQplot
JPM Atr_atr/Down (centered & scaled) - QQplot
JPM Atr_atr/Neutral (centered & scaled) - QQplot
JPM Mfi/Up (centered & scaled) - QQplot
JPM Mfi/Down (centered & scaled) - QQplot
JPM Mfi/Neutral (centered & scaled) - QQplot
JPM Rsi/Up (centered & scaled) - QQplot
JPM Rsi/Down (centered & scaled) - QQplot
JPM Rsi/Neutral (centered & scaled) - QQplot
JPM Atr_trueLow/Up (centered & scaled) - QQplot
JPM Atr_trueLow/Down (centered & scaled) - QQplot
JPM Atr_trueLow/Neutral (centered & scaled) - QQplot
JPM Atr_tr/Up (centered & scaled) - QQplot
JPM Atr_trLow/Down (centered & scaled) - QQplot
JPM Atr_tr/Neutral (centered & scaled) - QQplot
JPM Close_price_5day_lag/Up (centered & scaled) - QQplot
JPM Close_price_5day_lag/Down (centered & scaled) - QQplot
JPM Close_price_5day_lag/Neutral (centered & scaled) - QQplot
JPM Atr_trueHigh/Up (centered & scaled) - QQplot
JPM Atr_trueLow/Down (centered & scaled) - QQplot
JPM Atr_trueHigh/Neutral (centered & scaled) - QQplot
Theoretical Quantiles

## Models Training & Hyper Parameters Tuning

Due to the cyclical nature of the stock market time series, it is not advisable to use cross-validation, neither for the model fitting or the hyper-parameter optimisation. Each model is therefore performance tested following the methodology described in the above section named *The Sliding Time Windows.* It is worth mentioning that during the training phase, each model is tested against a different list of feature selected attributes. The first training model lists all the 10 features that have been feature selected. Each following model removes one attribute at a time; the one with the least predictive power. The last instance represents a polynomial and interaction between a few of the most predictive power attribute, in an attempt to improve model fitting.

For each model, the following list of attributes are tested (in the order presented):

- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr
- Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag
- Direction ~ Close_price_1day_lag + Volume
- Direction ~ Close_price_1day_lag
- Direction ~ Close_price_1day_lag^3 + Volume^2 + Close_price_1day_lag + Volume + (Atr_atr * Mfi) + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh

Unless otherwise stated, the R default value of each hyper-parameter is used during the training phase. The optimisation phase takes the best trained model and attempt to optimise its hyper-parameters. As a general principal, when several hyperparameter configurations, for a given model, generate the same accuracy rate, the more computationally efficient parameter set is chosen. For example, when the bagging model returns an accuracy rate of 54.62% for trees number set to 1000 or 1500, and the same lambda (a.k.a. shrinkage), then the model is tested with the number of trees set to 1000.

The remaining sections i) describe each model, ii) provide details of any necessary data transformation prior to running the model, iii) list parameters that need tuning, iv) explain the code required for training/optimising and testing each model and v) offers the average test performance accuracy rate for each model.

## The Ridge

### *Model Description*

The Ridge is a method that regularise (i.e. constraints) the coefficient estimates of *p* predicators of a linear model. In other words, it shrinks the coefficients estimates towards zero [14]. Looking at the formula below, the Ridge regression coefficients are the one that minimise:

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}\beta_j^2=\mathrm{RSS}+\lambda\sum_{j=1}^{p}\beta_j^2,$$

Where $y_i$ is the expected value, β0 is the intercept, βj are the coefficients of each variable $x_j$, λ (λ >=0) is the tuning parameter.

The aim of the Ridge model is to fit the data and make the RSS small, by making the shrinkage penalty, $\lambda\sum_j \beta_j^2$, as small as possible.

### *Model Assumptions*

No specific requirement.

### *Further data transformation*

The glmnet. *glmnet (x,…)* function implementation does not require any further data transformation. It accepts the regressor as a list of character classes. The parameters can be of type character or numeric.

### *Parameter Tuning*

The main parameter tuning is the λ. The *α* is set to 0 for the Ridge regression, in the glmnet package.

### *Code Snippet Explanation*

<u>Model Training</u>

The *for-loop*, line 1404 iterates through the training data time windows (100 iterations). At each iteration, a training data slice and validation data sets are built for the time window in question (line 1405-1409). The column list is created line 1411 and then passed to the *run_mlr_model(…)* function, alongside a few parameters (e.g. the training and the validation datasets for the time slide period and *the α is set to 0*). On successful run, the model results are stored into an in-memory data frame named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*. The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 1418. In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 1416).

The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
1398        model_name = "Ridge"
1399        model_desc = "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh"
1400        model_type = "TRAIN"
1401        the_alpha = 0
1402        uuid = UUIDgenerate()
1403
1404         for (tw_index in time_window_seq){
1405           #Get the Training and Validation data for the given time window
1406           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
1407           training_data = final_df[training_range,]
1408           validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
1409           validation_data = final_df[validation_range,]
1410
1411          columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr","Mfi","Rsi","Atr_trueLow","Atr_tr","Close_price_5day_lag","Atr_trueHigh")
1412
1413          possibleError = tryCatch( run_mlr_model(columns, the_alpha, NULL, uuid, stock_name, model_name, model_desc,model_type,
1414                                      model_comparison_summary_df,training_data, validation_data),
1415                               error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1416          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1417        }
1418        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
1419
```

Code Snippet 9 – Calling the Ridge model

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* First a sparse matrix for the training and test model are created, lines *1301\1303*. Then the glmnet(..) function is called, line 1306. The *family* parameter is set to multinomial and the $\alpha$ is passed as part of the function call. In this case, $\alpha$ is set to 0 (Ridge regression). Depending on the caller need, the *test_data_param*, could correspond to test data (testing phase) or validation data (training and optimisation phase). The *lambda_param* is also defined by the caller. When it is set to NULL (training phase), line 1309 then the minimum λ is used, else the *lambda_param* value is used (optimisation/testing phases). The prediction is performed (line 1312). The confusion matrix is then built and added as an extra row to the *model_comparison_summary_df* dataframe (line 1320).

```
1290 run_mlr_model = function(colum_list, alpha_param, lambda_param, uuid, stock_name, model_name, model_desc,model_type, model_comparison_summary_df,training_data_param,
     test_data_param){
1291
1292   set.seed(1)
1293
1294   #Seleted the necessary columns for the training and test set
1295   mlr_training_data = training_data_param[columns]
1296   mlr_test_data = test_data_param[columns]
1297
1298   #Get the list of attributes col names
1299   col_names = colnames(mlr_training_data[-1])
1300   #Create a model for each set
1301   training_model = sparse.model.matrix( as.formula(paste("Direction ~", paste(col_names, sep = "", collapse=" +"))),
1302                                         data = mlr_training_data)
1303   test_model = sparse.model.matrix( as.formula(paste("Direction ~", paste(col_names, sep = "", collapse=" +"))),
1304                                          data = mlr_test_data)
1305
1306   mlr_fit = glmnet(training_model[1:nrow(mlr_training_data),], mlr_training_data$Direction, family = "multinomial", alpha=alpha_param)
1307
1308   #Geneate the model prediction depending on the lambda level
1309   if (is.null(lambda_param)) {
1310     lambda_param = min(mlr_fit$lambda)
1311   }
1312   mlr.pred = predict(mlr_fit,as.matrix(test_model), type="class",s=lambda_param)
1313
1314   #Generate the confusion matrix and calculate the classification error rate on the test/valdation data
1315   mlr.confusion_table =  table(mlr.pred, test_data_param$Direction)
1316   mlr.accuracy_rate = accuracy_rate_perc(mlr.confusion_table)
1317   mlr.error_rate = error_rate_perc(mlr.confusion_table)
1318
1319   #Add a row in the model comparison dataframe
1320   model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1321                                                             uuid,
1322                                                             stock_name,
1323                                                             "TRUE",
1324                                                             "",
1325                                                             model_name, model_desc,model_type,
1326                                                             mlr.accuracy_rate, mlr.error_rate)
1327
1328
1329
1330   return (model_comparison_summary_df)
1331 }
```

Code Snippet 9 – Calling the Ridge model for the training phase.

Model Optimisation

This code is similar to the training phase above. This time a list of λ are tested, line 1630/1633 on the last training window. The λ that generates the minimum accuracy error is retained for the testing phase.

```
1624         ############################################################################
1625         ############################################################################
1626         #######   The Ridge Model - Model Optimisation
1627         ############################################################################
1628         ############################################################################
1629
1630        lambda_list = c(0.0001, 0.0005, 0.0010,0.0015,0.0020,0.0050, 0.0055, 0.0060, 0.01, 0.02, 0.03, 0.04, 0.5,1)
1631
1632         #The optimisation is performed on the last sliding window
1633         for (the_lambda in lambda_list){
1634            model_name = "Ridge"
1635            model_desc = paste( "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr +
     Close_price_5day_lag + Atr_trueHigh | lambda = ", toString(the_lambda),sep="")
1636           model_type = "OPTIMISATION"
1637           the_alpha = 0
1638           uuid = UUIDgenerate()
1639
1640           #Get the Training and Validation data for the given time window
1641           training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
1642           training_data = final_df[training_range,]
1643           validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
1644           validation_data = final_df[validation_range,]
1645
1646            columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr","Mfi","Rsi","Atr_trueLow","Atr_tr",
1647                        "Close_price_5day_lag","Atr_trueHigh")
1648
1649            possibleError = tryCatch( run_mlr_model(columns, the_alpha, the_lambda, uuid, stock_name, model_name, model_desc,model_type,
1650                                   model_comparison_summary_df,training_data, validation_data),
1651                         error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1652            add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1653         }
1654
```

Code Snippet 10 – Calling the Ridge model for the optimisation phase

Model Testing

The best training model is chosen alongside its hyper-parameter list. In this case *lambda_param* (line 1661) is set to the best optimised value (i.e. 0.001). It is then fitted against the entire training data set, and finally tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same *run_mlr_model()* function is called as for the training phase, line 1678. This time, the test data is used in lieu of the validation data.

```
1656            ##########################################################################
1657            #######   The Ridge Model - Model Testing
1658            ##########################################################################
1659            ##########################################################################
1660           model_name = "Ridge"
1661           the_lambda = 0.001
1662           model_desc = paste( "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr +
      Close_price_5day_lag + Atr_trueHigh | lambda = ", toString(the_lambda),sep="")
1663           model_type = "TEST"
1664           the_alpha = 0
1665           uuid = UUIDgenerate()
1666
1667           #This time the model is tained on the Traning + Validation data, then it is tested against the Testing data.
1668           for (tw_index in time_window_seq){
1669             #Get the Training and Validation data for the given time window
1670             training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
1671             training_data = final_df[training_range,]
1672             test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
1673             test_data = final_df[validation_range,]
1674
1675             columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr","Mfi","Rsi","Atr_trueLow","Atr_tr",
1676                         "Close_price_5day_lag","Atr_trueHigh")
1677
1678              possibleError = tryCatch( run_mlr_model(columns, the_alpha, the_lambda, uuid, stock_name, model_name, model_desc,model_type,
1679                                        model_comparison_summary_df,training_data, validation_data),
1680                                 error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1681              add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1682           }
1683           generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
1684
1685          cat ("\n")
1686          cat (paste ("Save Comparison Summary: ", model_name, " \n", sep=""))
1687          selected_model_df = model_comparison_summary_df[model_comparison_summary_df$model_name == model_name, ]
1688          print(selected_model_df)
1689          save_to_csv(selected_model_df, out_dir,paste("Model_Results_", model_name, ".csv", sep=""))
1690
```

Code Snippet 10 – Calling the Ridge model for the testing phase

### *Results*

As shown in the below table, the Ridge model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh* generates the highest training accurate rate at 63.17%. The optimisation model shows the best accuracy for λ = 0.001, at 73.85%. The test performance of the Ridge model, run against the selected list of attributes and the optimised λ, produces a 75.39% accuracy rate, with a standard deviation of 1.44%. The data is available in the file: *.../final results/Model_Results_Ridge.xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| ⊟TEST | | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.001 | 75.39 | 1.44 |
| ⊟TRAIN | | |
| Direction ~ Direction ~ Close_price_1day_lag | 28.58 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume | 29.20 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 29.37 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 31.13 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 32.41 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 32.60 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 59.08 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 59.27 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 60.28 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 63.17 | |

| aggregation_type | (blank) |
|---|---|

| Row Labels | Sum of model_accuracy_rate |
|---|---|
| ⊟OPTIMISATION | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.001 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.0015 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.002 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.005 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.0055 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.006 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.01 | 73.08 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.02 | 71.54 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.03 | 70.00 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.04 | 69.23 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.5 | 59.23 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.0001 | 62.31 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 1e-04 | 73.85 |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 5e-04 | 73.85 |

## The Lasso

### *Model Description*

Like the Ridge, the Lasso is a method that regularise (i.e. constraints) the coefficient estimates of *p* predicators of a linear model. Looking at the formula below, the Lasso regression coefficients are the one that minimise:

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}|\beta_j| \;=\; \mathrm{RSS}+\lambda\sum_{j=1}^{p}|\beta_j|.$$

Where $y_i$ is the expected value, $\beta_0$ is the intercept, $\beta_j$ are the coefficients of each variables $x_j$, λ (λ >=0) is the tuning parameter. This time, the ℓ1 parameter: $|\beta_j|$ is used instead of the ℓ2 parameter: $\beta_j$ (as per the Ridge mode). The advantage is that the ℓ1 parameter has the effect of forcing the coefficient estimate to zero when λ is large enough [14].

### *Model Assumptions*

No specific requirement.

### *Further data transformation*

The glmnet. *glmnet (x,…)* function implementation does not require any further data transformation. It accepts the regressor as a list of character classes. The parameters can be of type character or numeric.

### *Parameter Tuning*

The main parameter tuning is the λ. The α is set to 1 for the Lasso regression, in the glmnet package.

### *Code Snippet Explanation*

Model Training

Same description as the Ridge section. The only difference is that α is set to 1 (Line 1700).

```
1692          #########################################################################
1693          #########################################################################
1694          #######   The Lasso Model - Model Training
1695          #########################################################################
1696          #########################################################################
1697         model_name = "Lasso"
1698         model_desc = "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + 
     Atr_trueHigh"
1699         model_type = "TRAIN"
1700         the_alpha = 1
1701         uuid = UUIDgenerate()
1702
1703          for (tw_index in time_window_seq){
1704            #Get the Training and Validation data for the given time window
1705            training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
1706            training_data = final_df[training_range,]
1707            validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
1708            validation_data = final_df[validation_range,]
1709
1710           columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr","Mfi","Rsi","Atr_trueLow","Atr_tr","Close_price_5day_lag","Atr_trueHig
     h")
1711
1712           possibleError = tryCatch( run_mlr_model(columns, the_alpha, NULL, uuid, stock_name, model_name, model_desc,model_type,
1713                                         model_comparison_summary_df,training_data, validation_data),
1714                                  error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1715           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1716         }
1717         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 11 – Calling the Lasso model (training)

Model Optimisation

*Same description as the Ridge section. The only difference is that α is set to 1 (Line 1936).*

```
1922
1923         ############################################################################
1924         ############################################################################
1925         #######   The Lasso Model - Model Optimisation
1926         ############################################################################
1927         ############################################################################
1928
1929        lambda_list = c(0.0001, 0.0005, 0.0010,0.0015,0.0020,0.0050, 0.0055, 0.0060, 0.01, 0.02, 0.03, 0.04, 0.5,1)
1930
1931         #The optimisation is performed on the last sliding window
1932         for (the_lambda in lambda_list){
1933            model_name = "Lasso"
1934            model_desc = paste( "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | lambda = ",
toString(the_lambda),sep="")
1935           model_type = "OPTIMISATION"
1936           the_alpha = 1
1937           uuid = UUIDgenerate()
1938
1939           #Get the Training and Validation data for the given time window
1940           training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
1941           training_data = final_df[training_range,]
1942           validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
1943           validation_data = final_df[validation_range,]
1944
1945            columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr")
1946
1947            possibleError = tryCatch( run_mlr_model(columns, the_alpha, the_lambda, uuid, stock_name, model_name, model_desc,model_type,
1948                                        model_comparison_summary_df,training_data, validation_data),
1949                               error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1950            add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1951         }
1952
```

Code Snippet 12– Calling the Lasso model (Optimisation)

Model Testing

The best training model is chosen alongside its hyper-parameter list. In this case lambda_param (line 1960) is set to the best optimised value (i.e. 0.005). It is then fitted against the entire training data set, and finally tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same run_mlr_model() function is called as for the training phase, line 1976. This time, the test data is used in lieu of the validation data.

```
1953         ############################################################################
1954         ############################################################################
1955         #######   The Lasso Model - Model Testing
1956         ############################################################################
1957         ############################################################################
1958         model_name = "Lasso"
1959         the_alpha = 1
1960         the_lambda = 0.005
1961         model_desc = paste( "Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr +
Close_price_5day_lag + Atr_trueHigh | lambda = ", toString(the_lambda),sep="")
1962         model_type = "TEST"
1963         uuid = UUIDgenerate()
1964
1965         #This time the model is tained on the Traning + Validation data, then it is tested against the Testing data.
1966         for (tw_index in time_window_seq){
1967           #Get the Training and Validation data for the given time window
1968           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
1969           training_data = final_df[training_range,]
1970           test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
1971           test_data = final_df[validation_range,]
1972
1973           columns = c("Direction","Close_price_1day_lag","Volume","Close_price_2day_lag","Atr_atr","Mfi","Rsi","Atr_trueLow","Atr_tr",
1974                       "Close_price_5day_lag","Atr_trueHigh")
1975
1976            possibleError = tryCatch( run_mlr_model(columns, the_alpha, the_lambda, uuid, stock_name, model_name, model_desc,model_type,
1977                                        model_comparison_summary_df,training_data, validation_data),
1978                               error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1979            add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1980         }
1981         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
1982
1983        cat ("\n")
1984        cat (paste ("Save Comparison Summary: ", model_name, " \n", sep=""))
1985        selected_model_df = model_comparison_summary_df[model_comparison_summary_df$model_name == model_name, ]
1986        print(selected_model_df)
1987        save_to_csv(selected_model_df, out_dir,paste("Model_Results_", model_name, ".csv", sep=""))
1988
```

Code Snippet 12– Calling the Lasso model (Testing)

### *Results*

As shown in the below table, the Lasso model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh* generates the highest training accurate rate at 73.59%. The optimisation model shows the best accuracy for lambda= 0.005, at 22.31%. The test performance of the Lasso model, run against the selected list of attributes and the optimised λ, produces a 80.71% accuracy rate, with a standard deviation of 1.38%. The data is available in the file: *Model_Lasso.xlsx*

| K | L | M |
|---|---|---|
| aggregation_type | AVG | |
| | | |
| **Row Labels** | **Sum of model_accuracy_rate** | **Std Dev** |
| **TEST** | | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| lambda = 0.005 | 80.71 | 1.38 |
| **TRAIN** | | |
| Direction ~ Direction ~ Close_price_1day_lag | 28.58 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume | 29.19 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 29.61 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 30.98 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 32.62 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 32.56 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 71.86 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 73.88 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 73.62 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 73.59 | |

| | | |
|---|---|---|
| aggregation_type | (blank) | |
| | | |
| **Row Labels** | **Sum of model_accuracy_rate** | |
| **OPTIMISATION** | | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.001 | 20.00 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.0015 | 20.77 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.002 | 21.54 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.005 | 22.31 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.0055 | 22.31 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.006 | 22.31 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.01 | 19.23 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.02 | 16.92 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.03 | 16.15 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.04 | 13.85 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 0.5 | 13.85 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 1 | 13.85 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 1e-04 | 19.23 | |
| Direction ~ Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| lambda = 5e-04 | 19.23 | |

## Linear Discriminant Analysis (LDA)

### *Model Description*

As described in [14], the multivariate Gaussian density function for the LDA model when *p>1* (*p* is the number of attributes) can described as follows.  We assume that *X = (x1,x2, ….,xn)* is a vector of predictors. This model assumes that each individual predictor follows a one-dimensional normal distribution, with some correlation between each pair of predictors”. Therefore, $X \sim N(\mu, \Sigma)$ , where $E(X) = \mu$ is the mean of *X* (a vector of p components) and $Cov(X) = \Sigma$ is the covariance matrix of *X*. The estimates *f(x)* are then fed into the *Bayes’ Theorem* to perform the class prediction.

$$f(x) = \frac{1}{(2\pi)^{p/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T \boldsymbol{\Sigma}^{-1}(x-\mu)\right).$$

### *Model Assumptions*

Each attribute class should follow a normal (Gaussian) distribution.

### *Further data transformation*

The MASS.*lda (x,…)* function implementation does not require any further data transformation. It accepts the regressor as a list of character classes. The parameters can be of type character or numeric.

### *Parameter Tuning*

There is no need for parameter tuning for this model.

### *Code Snippet Explanation*

<u>Model Training</u>

The *for-loop*, line 2001 iterates through the training data time windows (100 iterations). At each iteration, the the *lda(x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 2008). It produces the *lda_fit* object, i.e. the function fitting the model. The *lda_fit* model is then passed to the *run_lda_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period. On successful run, the model results are stored into an in-memory object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*. The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated (line 2024).  In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 2022).

The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
1990         ##########################################################################
1991         ##########################################################################
1992         #######    Linear Descriminent Analysis (LDA) - Model Training
1993         ##########################################################################
1994         ##########################################################################
1995
1996         model_name = "Linear Descriminent Analysis (LDA)"
1997         model_desc = "Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag +
     Atr_trueHigh"
1998         model_type = "TRAIN"
1999         uuid = UUIDgenerate()
2000
2001         for (tw_index in time_window_seq){
2002           #Get the Training and Validation data for the given time window
2003           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
2004           training_data = final_df[training_range,]
2005           validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
2006           validation_data = final_df[validation_range,]
2007
2008           lda_fit =  lda (Direction ~ Close_price_1day_lag +
2009                                       Volume +
2010                                       Close_price_2day_lag +
2011                                       Atr_atr +
2012                                       Mfi +
2013                                       Rsi +
2014                                       Atr_trueLow +
2015                                       Atr_tr +
2016                                       Close_price_5day_lag +
2017                                       Atr_trueHigh,
2018                             data=training_data)
2019
2020           possibleError = tryCatch( run_lda_model(lda_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
2021                                     error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
2022           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
2023         }
2024         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 13– Calling the LDA model (Training)

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (Line 1114/1120).

```
1102 #the test_data_param could be a traning/validation or test dataset
1103 run_lda_model = function(fit_param, uuid, stock_name, model_name, model_desc, model_type, model_comparison_summary_df, test_data_param, save_confusion_matrix=FALSE){
1104
1105     lda.fit = fit_param
1106
1107     #Generate the confusion matrix and calculate the training/validation error rate
1108     lda.pred = predict(lda.fit, test_data_param)$class
1109     lda.confusion_table =  table(lda.pred, test_data_param$Direction)
1110     lda.accuracy_rate = accuracy_rate_perc(lda.confusion_table)
1111     lda.error_rate = error_rate_perc(lda.confusion_table)
1112
1113     #Add a row in the model comparison dataframe
1114     model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1115                                                              uuid,
1116                                                              stock_name,
1117                                                              "TRUE",
1118                                                              "",
1119                                                              model_name, model_desc,model_type,
1120                                                              lda.accuracy_rate, lda.error_rate)
1121     if (save_confusion_matrix == TRUE){
1122
1123       save_to_csv(lda.confusion_table, out_dir,paste("lda_confusion_matrix_", UUIDgenerate() ,".csv",sep=""))
1124     }
1125
1126     return (model_comparison_summary_df)
1127 }
```

Code Snippet 14– The *run_lda_model()* function implementation

<u>Model Testing</u>

The best training model is chosen and is fitted against the entire training data set. It is then tested against the test set (i.e. the last 5 business days). The confusion matrix that evaluates the test prediction vs the expected test data provides the test accuracy rate. The same *run_lda_model()* function is called as for the training phase. This time, the test data is used in lieu of the validation data.

```
        model_name = "Linear Descriminent Analysis (LDA)"
        model_desc = "Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr"
        model_type = "TEST"
        uuid = UUIDgenerate()

        #This time the model is tained on the Traning + Validation data, then it is tested against the Testing data.
        for (tw_index in time_window_seq){
          #Get the Training and Validation data for the given time window
          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
          training_data = final_df[training_range,]
          test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
          test_data = final_df[validation_range,]

          lda_fit =  lda (Direction ~ Close_price_1day_lag +
                                     Volume +
                                     Close_price_2day_lag +
                                     Atr_atr +
                                     Mfi +
                                     Rsi +
                                     Atr_trueLow +
                                     Atr_tr,
                               data=training_data)

          possibleError = tryCatch( run_lda_model(lda_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,test_data, TRUE),
                                     error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
        }
        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)

        cat ("\n")
        cat (paste ("Save Comparison Summary: ", model_name, " \n", sep=""))
        selected_model_df = model_comparison_summary_df[model_comparison_summary_df$model_name == model_name, ]
        print(selected_model_df)
        save_to_csv(selected_model_df, out_dir,paste("Model_Results_", model_name, ".csv", sep=""))
```

*Code Snippet 15– Calling the LDA model (Testing)*

### *Results*

As shown in the below table, the LDA model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr* generates the highest training accurate ate at 72.98%. The test performance of the LDA model, run against the selected list of attributes, produces an 80.20% accuracy rate, with standard deviation of 1.06%. The data is available in the file: *Model_Results_Linear Descriminent Analysis (LDA).xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| **TEST** | | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 80.20 | 1.06 |
| **TRAIN** | | |
| Direction ~ Close_price_1day_lag | 39.45 | |
| Direction ~ Close_price_1day_lag + Volume | 41.52 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 42.32 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 41.38 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 39.99 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 39.68 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 69.98 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 72.98 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 72.41 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 72.41 | |

## Quadratic Discriminant Analysis (QDA)

### *Model Description*

As described in [14], the QDA classifier model follows the same assumption as the LDA model, and is also dependent on the Bayes' theorem to perform class predictions. The main difference resides in the assumption that that each class has its own covariance matrix. Therefore, $X \sim N(\mu k, \sum k)$ where $k$ represents the *kth* class. The class separator becomes quadratic instead of being linear, hence the name.

### *Model Assumptions*

Each class should follow a normal (Gaussian) distribution.

### *Further data transformation*

The R *MASS.qda (x,…)* function implementation does not require any further data transformation. It accepts the regressor as a list of character classes. The parameters can be of type character or numeric.

### *Parameter Tuning*

There is no need for parameter tuning for this model.

### *Code Snippet Explanation*

<u>Model Training</u>

The *for-loop*, line 2372 iterates through the training data time windows (100 iterations). At each iteration, the the *qda(x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 2379). It produces the *qda_fit* object, i.e. the function fitting the model. The *qda_fit* model is then passed to the *run_qda_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period. On successful run, the model results are stored into an in-memory object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*. The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 2394. In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 2392).
The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
2367        model_name = "Quadratic Descriminent Analysis (QDA)"
2368        model_desc = "Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag"
2369        model_type = "TRAIN"
2370        uuid = UUIDgenerate()
2371
2372        for (tw_index in time_window_seq){
2373          #Get the Training and Validation data for the given time window
2374          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
2375          training_data = final_df[training_range,]
2376          validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
2377          validation_data = final_df[validation_range,]
2378
2379          qda_fit =  qda (Direction  ~ Close_price_1day_lag +
2380                                       Volume +
2381                                       Close_price_2day_lag +
2382                                       Atr_atr +
2383                                       Mfi +
2384                                       Rsi +
2385                                       Atr_trueLow +
2386                                       Atr_tr +
2387                                       Close_price_5day_lag,
2388                    data=training_data)
2389
2390          possibleError = tryCatch( run_qda_model(qda_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
2391                             error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
2392          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
2393        }
2394        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 16 – Calling the QDA model (Training)

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (line 1140/1146).

```
1129 run_qda_model = function(fit_param, uuid, stock_name, model_name, model_desc,model_type, model_comparison_summary_df,test_data_param, save_confusion_matrix=FALSE){
1130
1131     qda.fit = fit_param
1132
1133     #Generate the confusion matrix and calculate the training/validation error rate
1134     qda.pred = predict(qda.fit, test_data_param)$class
1135     qda.confusion_table =  table(qda.pred, test_data_param$Direction)
1136     qda.accuracy_rate = accuracy_rate_perc(qda.confusion_table)
1137     qda.error_rate = error_rate_perc(qda.confusion_table)
1138
1139     #Add a row in the model comparison dataframe
1140     model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1141                                                               uuid,
1142                                                               stock_name,
1143                                                               "TRUE",
1144                                                               "",
1145                                                               model_name, model_desc,model_type,
1146                                                               qda.accuracy_rate, qda.error_rate)
1147
1148     if (save_confusion_matrix == TRUE){
1149       save_to_csv(qda.confusion_table, out_dir,paste("qda_confusion_matrix_", UUIDgenerate() ,".csv",sep=""))
1150     }
1151
1152     return (model_comparison_summary_df)
1153 }
```

Code Snippet 17 – The *run_qda_model()* function implementation

Model Testing

The best training model is chosen and is fitted against the entire training data set. It is then tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same *run_qda_model()* function is called as for the training phase. This time, the test data is used in lieu of the validation data.

```
        ############################################################################
        ############################################################################
        #######   Quadratic Descriminent Analysis (QDA) - Model Testing
        ############################################################################
        ############################################################################

        model_name = "Quadratic Descriminent Analysis (QDA)"
        model_desc = "Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag"
        model_type = "TEST"
        uuid = UUIDgenerate()

        #This time the model is tained on the Traning + Validation data, then it is tested against the Testing data.
        for (tw_index in time_window_seq){
          #Get the Training and Validation data for the given time window
          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
          training_data = final_df[training_range,]
          test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
          test_data = final_df[validation_range,]

          qda_fit =  qda (Direction ~ Close_price_1day_lag +
                                      Volume +
                                      Close_price_2day_lag +
                                      Atr_atr +
                                      Mfi +
                                      Rsi +
                                      Atr_trueLow +
                                      Atr_tr +
                                      Close_price_5day_lag,
                          data=training_data)

          possibleError = tryCatch( run_qda_model(qda_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,test_data, TRUE),
                                    error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
        }
        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

*Code Snippet 18– Calling the QDA model (Testing)*

### *Results*

As shown in the below table, the LDA model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag* generates the highest training accurate rate at 68.48%. The test performance of the LDA model run against the selected list of attributes produces a 80.90% accuracy rate, with a standard deviation of 1.64%. The data is available in the file: *Model_Results_Quadratic Descriminent Analysis (QDA).xlsx*

| aggregation_type | AVG | |
|---|---|---|
| | | |
| **Row Labels** | **Sum of model_accuracy_rate** | |
| **⊟TEST** | | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 80.90 | 1.64 |
| **⊟TRAIN** | | |
| Direction ~ Close_price_1day_lag | 36.97 | |
| Direction ~ Close_price_1day_lag + Volume | 38.71 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 40.96 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 40.52 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 40.24 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 39.32 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 66.49 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 67.92 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 68.48 | |

## Decision Tree

### *Model Description*

The goal is to find regions R1,…,Rn that minimise the classification error rate. It corresponds to the fraction of the training observations in that region that do not belong to the most common class. It is given by the equation from [14], $pHat_{mk}$ represents the portion of training observations in the *mth* region that are from the *kth* class.

$$E = 1 - \max_{k}(\hat{p}_{mk}).$$

### *Model Assumptions*

No specific requirements.

### *Further data transformation*

The *tree.tree (x,…)* function implementation requires that the regressor variable is encoded as a factor (i.e. an enumerated type). Therefore, the *factor()* function has been applied on the *Direction* regressor. None of the other parameters need to be adapted.

### *Parameter Tuning*

The tree size parameter is optimised.

### *Code Snippet Explanation*

<u>Model Training</u>

The *for-loop*, line 3144 iterates through the training data time windows (100 iterations). At each iteration, the the *tree(x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 3151). It produces the *tree_fit* object, i.e. the function fitting the model. The *tree_fit* model is then passed to the *run_tree_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period. On successful run, the model results are stored into an in-memory object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*.The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 3167. In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 3165).
The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
3134        ##########################################################################
3135        ##########################################################################
3136        #######   Decision Tree - Model Training
3137        ##########################################################################
3138        ##########################################################################
3139        model_name = "Decision Tree"
3140        model_desc = "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + 
    Atr_trueHigh"
3141        model_type = "TRAIN"
3142        uuid = UUIDgenerate()
3143
3144        for (tw_index in time_window_seq){
3145          #Get the Training and Validation data for the given time window
3146          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
3147          training_data = final_df[training_range,]
3148          validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
3149          validation_data = final_df[validation_range,]
3150
3151          tree_fit =  tree (factor(Direction) ~ Close_price_1day_lag +
3152                                               Volume +
3153                                               Close_price_2day_lag +
3154                                               Atr_atr +
3155                                               Mfi +
3156                                               Rsi +
3157                                               Atr_trueLow +
3158                                               Atr_tr +
3159                                               Close_price_5day_lag +
3160                                               Atr_trueHigh,
3161                            data=training_data)
3162
3163          possibleError = tryCatch( run_tree_model(tree_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
3164                                    error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3165          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3166        }
3167        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 19 – Calling the Tree model (Training)

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (line 1217/1124).

```
1203 run_tree_model = function(fit_param,uuid, stock_name, model_name, model_desc,model_type, model_comparison_summary_df,test_data_param, can_prune=TRUE){
1204
1205     set.seed(1)
1206
1207     tree.fit = fit_param
1208     #Type="class" is selected as we are dealing with a classification problem
1209     tree.pred = predict(tree.fit, test_data_param, type="class")
1210
1211     #Generate the confusion matrix and calculate the training/validation error rate
1212     tree.confusion_table =  table(tree.pred, test_data_param$Direction)
1213     tree.accuracy_rate = accuracy_rate_perc(tree.confusion_table)
1214     tree.error_rate = error_rate_perc(tree.confusion_table)
1215
1216     #Add a row in the model comparison dataframe
1217     model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1218                                                               uuid,
1219                                                               stock_name,
1220                                                               "TRUE",
1221                                                               "",
1222                                                               model_name, model_desc,model_type,
1223                                                               tree.accuracy_rate, tree.error_rate)
1224
1225     return (model_comparison_summary_df)
1226 }
```

Code Snippet 20 – Calling the *run_tree_model()* function

<u>Model Optimisation</u>

This code is similar to the training phase above. This time the tree minimum size list (*min_size_list*) is provided (line 1434). The optimisation is run over the last training period and generate the accuracy for each tree minimum size element in the list. The highest accuracy is retained. The minimum tree size is selected for this accuracy level.

```
3434         min_size_list = c(1,2,5,10,20,40,60,80,100)
3435
3436         #The optimisation is performed on the last sliding window
3437         for (the_min_size in min_size_list){
3438           model_name = "Decision Tree"
3439           model_desc = paste( "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + 
     Close_price_5day_lag + Atr_trueHigh | min size = ", toString(the_min_size),sep="")
3440           model_type = "OPTIMISATION"
3441           uuid = UUIDgenerate()
3442
3443           #Get the Training and Validation data for the given time window
3444           training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
3445           training_data = final_df[training_range,]
3446           validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
3447           validation_data = final_df[validation_range,]
3448
3449           tree_fit =  tree (factor(Direction) ~ Close_price_1day_lag +
3450                                                 Volume +
3451                                                 Close_price_2day_lag +
3452                                                 Atr_atr +
3453                                                 Mfi +
3454                                                 Rsi +
3455                                                 Atr_trueLow +
3456                                                 Atr_tr +
3457                                                 Close_price_5day_lag +
3458                                                 Atr_trueHigh,
3459                             data=training_data)
3460
3461
3462           tree.pruned = prune.misclass(tree_fit, best = the_min_size)
3463
3464           possibleError = tryCatch( run_tree_model(tree.pruned,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
3465                                     error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3466           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3467         }
```

Code Snippet 21 – Calling the Tree model for the optimisation phase

<u>Model Testing</u>

The best training model is chosen alongside its hyper-parameter list. It is then fitted against the entire training data set, and finally tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same run_tree_model() function is called as for the training phase, line 3496. This time, the test data is used in lieu of the validation data.

### *Results*

As shown in the below table, the Ridge model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh* generates the highest training accurate rate at 44.33%. The optimisation model shows the best min tree size = 2

The test performance of the Tree model run against the selected list of attributes and the optimised min tree size produces a 70.80% accuracy rate, with a standard deviation of 21.45%. The data is available in the file: *Model_Results_Tree.xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| **TEST** | | |
| Direction ~ Close_price_1day_lag + Close_price_4day_lag + Close_price_2day_lag + Atr_trueHigh | 40.80 | 21.45 |
| **TRAIN** | | |
| factor(Direction) ~ Close_price_1day_lag | 38.07 | |
| factor(Direction) ~ Close_price_1day_lag + Volume | 38.18 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 38.31 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 37.22 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 37.39 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 38.90 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 42.19 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 41.52 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 43.79 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 44.33 | |

<table>
<tr><td>aggregation_type</td><td>(blank)</td></tr>
<tr><th>Row Labels</th><th>Sum of model_accuracy_rate</th></tr>
<tr><td>OPTIMISATION</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 10</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 100</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 2</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 20</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 40</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 5</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 60</td><td>54.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | min size = 80</td><td>54.62</td></tr>
</table>

## Bagging

### Model Description

Bagging is a procedure that reduces the statistical learning method variance [14]. This is achieved by making repeated samples from the training data set (bags) and averaging the prediction accuracy over the bags number.

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x).$$

With B, the number of bag and  f^*b (x) function, the prediction for each bag.
This methodology can be applied to classification trees, where the class predicted is the most commonly occurring class for a bag. Bagging is a special case randomForest where the number of variables randomly sampled as candidates at each split is equal to the number of attributes in the model (a.k.a. *mtry*).

### Model Assumptions

No specific requirements.

### Further data transformation

The *randomForest.randomForest (x,…)* function implementation requires that the regressor variable is encoded as a factor (i.e. an enumerated type). Therefore, the *factor()* function has been applied on the *Direction* regressor.

### Parameter Tuning

The tree size parameter is optimised.

### Code Snippet Explanation

<u>Model Training</u>

The *for-loop*, line 3520 iterates through the training data time windows (100 iterations). At each iteration, the the *randomForest (x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 3527). It produces the *bagging_fit* object, i.e. the function fitting the model. The *bagging _fit* model is then passed to the *run_tree_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period.  On successful run, the model results are stored into an in-memory object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*.
The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 3549.  In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 3547).
The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
3508         ########################################################################
3509         ########################################################################
3510         #######                Bagging Model
3511         ######  Specialised Random Forest where mtry = p (number of attributes)
3512         ########################################################################
3513         ########################################################################
3514         model_name = "Bagging"
3515         model_desc = "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + 
     Atr_trueHigh"
3516         model_type = "TRAIN"
3517         uuid = UUIDgenerate()
3518         m_try=10
3519 
3520         for (tw_index in time_window_seq){
3521           #Get the Training and Validation data for the given time window
3522           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
3523           training_data = final_df[training_range,]
3524           validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
3525           validation_data = final_df[validation_range,]
3526 
3527           bagging_fit = randomForest( factor(Direction) ~  Close_price_1day_lag +
3528                                                            Volume +
3529                                                            Close_price_2day_lag +
3530                                                            Atr_atr +
3531                                                            Mfi +
3532                                                            Rsi +
3533                                                            Atr_trueLow +
3534                                                            Atr_tr +
3535                                                            Close_price_5day_lag +
3536                                                            Atr_trueHigh,
3537                                      data=training_data,
3538                                      mtry=m_try,
3539                                      importance=TRUE)
3540 
3541           possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
3542                                                         uuid,stock_name,
3543                                                         model_name,model_desc,
3544                                                         model_type,model_comparison_summary_df,
3545                                                         validation_data),
3546                                   error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3547           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3548         }
3549         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
3550
```

Code Snippet 20 – Calling the Bagging model (Training)

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (line 1280/1286).

```
1265 run_rdmForest_model = function(fit_param, n_trees_param, uuid, stock_name, model_name, model_desc,model_type, model_comparison_summary_df,data_param){
1266
1267   set.seed(1)
1268
1269   rdmForest.fit = fit_param
1270
1271   #Geneate the model prediction.
1272   rdmForest.pred = predict(rdmForest.fit, data_param, n.trees = n_trees_param, type="response")
1273
1274   #Generate the confusion matrix and calculate the classification error rate on the test/valdation data
1275   rdmForest.confusion_table =  table(rdmForest.pred, data_param$Direction)
1276   rdmForest.accuracy_rate = accuracy_rate_perc(rdmForest.confusion_table)
1277   rdmForest.error_rate = error_rate_perc(rdmForest.confusion_table)
1278
1279   #Add a row in the model comparison dataframe
1280   model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1281                                                            uuid,
1282                                                            stock_name,
1283                                                            "TRUE",
1284                                                            "",
1285                                                            model_name, model_desc,model_type,
1286                                                            rdmForest.accuracy_rate, rdmForest.error_rate)
1287   return (model_comparison_summary_df)
1288 }
1289
```

Code Snippet 21 – Calling the *run_rmdForest_model()* function

<u>Model Optimisation</u>

This code is similar to the training phase above. This time the number of tree(s) to grow (n_tree_list) is provided (line 3890). The optimisation is run over the last training period and generates the accuracy for each number of tree element in the list. The highest accuracy is retained. The minimum tree to grow is selected for this accuracy level.

```
3884         #########################################################################
3885         #########################################################################
3886         #######             Bagging Model - Optimisation
3887         ######  Specialised Random Forest where mtry = p (number of attributes)
3888         #########################################################################
3889         #########################################################################
3890         n_trees_list = c(500,1000,2000,3000,4000, 5000)
3891
3892         for (n_trees in n_trees_list){
3893           model_name = "Bagging"
3894           model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | ntrees = ", n_trees,
      paste="")
3895           model_type = "OPTIMISATION"
3896           uuid = UUIDgenerate()
3897           m_try=7
3898
3899           #Get the Training and Validation data for the given time window
3900           training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
3901           training_data = final_df[training_range,]
3902           validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
3903           validation_data = final_df[validation_range,]
3904
3905           bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
3906                                                           Volume +
3907                                                           Close_price_2day_lag +
3908                                                           Atr_atr +
3909                                                           Mfi +
3910                                                           Rsi +
3911                                                           Atr_trueLow,
3912                                      data=training_data,
3913                                      ntree=n_trees,
3914                                      mtry=m_try,
3915                                      importance=TRUE)
3916
3917           possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
3918                                                          uuid,stock_name,
3919                                                          model_name,model_desc,
3920                                                          model_type,model_comparison_summary_df,
3921                                                          validation_data),
3922                                     error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3923           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3924
```

Code Snippet 22 – Calling the Tree model for the optimisation phase

Model Testing

The best training model is chosen alongside its hyper-parameter list. In this case *n_tree* (line 3934) is set to the best optimised value (i.e. 500). It is then fitted against the entire training data set, and finally tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same run_*rdmForest* _model() function is called as for the training phase, line 3496. This time, the test data is used in lieu of the validation data.

```
3927 #        ########################################################################
3928 #        ########################################################################
3929 #        #######            Bagging Model - Testing
3930 #        ######  Specialised Random Forest where mtry = p (number of attributes)
3931 #        ########################################################################
3932 #        ########################################################################
3933
3934       n_trees = 500
3935       model_name = "Bagging"
3936       model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag
 + Atr_trueHigh | ntrees = ", n_trees, paste="")
3937       model_type = "TEST"
3938       uuid = UUIDgenerate()
3939       m_try=7
3940
3941       for (tw_index in time_window_seq){
3942         #Get the Training and Validation data for the given time window
3943         training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
3944         training_data = final_df[training_range,]
3945         test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
3946         test_data = final_df[test_range,]
3947
3948         bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
3949                                                          Volume +
3950                                                          Close_price_2day_lag +
3951                                                          Atr_atr +
3952                                                          Mfi +
3953                                                          Rsi +
3954                                                          Atr_trueLow,
3955                                     data=training_data,
3956                                     ntree=n_trees,
3957                                     mtry=m_try,
3958                                     importance=TRUE)
3959
3960         possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
3961                                                        uuid,stock_name,
3962                                                        model_name,model_desc,
3963                                                        model_type,model_comparison_summary_df,
3964                                                        test_data),
3965                                 error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3966         add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3967       }
3968       generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
3969
```

Code Snippet 20 – Calling the Bagging model (Testing)

### *Results*

As shown in the below table, the Bagging model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow* generates the highest training accurate rate at 45.68%. The optimisation model shows the best tree to grow level at 500. The test performance of the Bagging model, run against the selected list of attributes and the optimised min tree size, produces a 37.00% accuracy rate, with a standard deviation of 27.47%. The data is available in the file: *Model_Results_Bagging.xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| **TEST** | | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 500 | 37.00 | 27.47 |
| **TRAIN** | | |
| factor(Direction) ~ Close_price_1day_lag | 33.72 | |
| factor(Direction) ~ Close_price_1day_lag + Volume | 35.45 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 33.31 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 32.08 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 34.99 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 37.32 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 45.68 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 44.09 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 42.40 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 44.50 | |

| aggregation_type | (blank) |
|---|---|

| Row Labels | Sum of model_accuracy_rate |
|---|---|
| **OPTIMISATION** | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 500 | 54.62 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 1000 | 52.31 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 2000 | 52.31 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 3000 | 52.31 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 4000 | 50.77 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow \| ntrees = 5000 | 50.77 |

## RandomForest

### *Model Description*

Bagging is a special case of Random Forest where the number of variables randomly sampled as candidates at each split is equal to the number of attributes in the model (a.k.a. *mtry*). Usually Random Forest have an *mtry* set approximately to SQRT(p) or p/2. Please refer to section *Bagging* for detailed information on the Bagging algorithm.

### *Model Assumptions*

No specific requirements.

### *Further data transformation*

The *randomForest.randomForest (x,…)* function implementation requires that the regressor variable is encoded as a factor (i.e. an enumerated type). Therefore, the *factor()* function has been applied on the *Direction* regressor. None of the other parameters need to be adapted.

### *Parameter Tuning*

The tree to grow parameter is optimised.

### *Code Snippet Explanation*

<u>Model Training</u>

Same comment as for the Bagging implementation, however this time there are two cases: the training case with mtry =  SQRT(p) and mtry = p/2.

```
3977        ##########################################################################
3978        ##########################################################################
3979        #######             Random Forest Model
3980        ######              mtry approx equal to SQRT(p)
3981        ##########################################################################
3982        ##########################################################################
3983        model_name = "Random Forest (mtry = SQRT(p))"
3984        model_desc = "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag +
    Atr_trueHigh"
3985        model_type = "TRAIN"
3986        uuid = UUIDgenerate()
3987        m_try=3
3988
3989        for (tw_index in time_window_seq){
3990          #Get the Training and Validation data for the given time window
3991          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
3992          training_data = final_df[training_range,]
3993          validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
3994          validation_data = final_df[validation_range,]
3995
3996          bagging_fit = randomForest( factor(Direction) ~  Close_price_1day_lag +
3997                                                           Volume +
3998                                                           Close_price_2day_lag +
3999                                                           Atr_atr +
4000                                                           Mfi +
4001                                                           Rsi +
4002                                                           Atr_trueLow +
4003                                                           Atr_tr +
4004                                                           Close_price_5day_lag +
4005                                                           Atr_trueHigh,
4006                                      data=training_data,
4007                                      mtry=m_try,
4008                                      importance=TRUE)
4009
4010          possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
4011                                                        uuid,stock_name,
4012                                                        model_name,model_desc,
4013                                                        model_type,model_comparison_summary_df,
4014                                                        validation_data),
4015                                  error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
4016          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
4017        }
4018        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 21a – Calling the Random Forest model, where m_try = SQRT(p), c.f. line 3987 (Training)

```
1450        #######################################################################
1451        #######################################################################
1452        #######             Random Forest Model
1453        ######              mtry approx equal to p/2
1454        #######################################################################
1455        #######################################################################
1456        model_name = "Random Forest (mtry = p_div_2)"
1457        model_desc = "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh"
1458        model_type = "TRAIN"
1459        uuid = UUIDgenerate()
1460        m_try=5
1461
1462        for (tw_index in time_window_seq){
1463          #Get the Training and Validation data for the given time window
1464          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
1465          training_data = final_df[training_range,]
1466          validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
1467          validation_data = final_df[validation_range,]
1468
1469          bagging_fit = randomForest( factor(Direction) ~  Close_price_1day_lag +
1470                                                           Volume +
1471                                                           Close_price_2day_lag +
1472                                                           Atr_atr +
1473                                                           Mfi +
1474                                                           Rsi +
1475                                                           Atr_trueLow +
1476                                                           Atr_tr +
1477                                                           Close_price_5day_lag +
1478                                                           Atr_trueHigh,
1479                                     data=training_data,
1480                                     mtry=m_try,
1481                                     importance=TRUE)
1482
1483          possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
1484                                                        uuid,stock_name,
1485                                                        model_name,model_desc,
1486                                                        model_type,model_comparison_summary_df,
1487                                                        validation_data),
1488                                     error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
1489          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
1490        }
1491        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```

Code Snippet 22b – Calling the Random Forest model, where m_try = p/2, c.f. line 1460 (Training)

<u>Model Optimisation</u>

Same comments as for the bagging case. The n_tree optimisation is performed for both m_try = p/2 and m_try = SRQT(p).

```
352        #######################################################################
353        #######################################################################
354        #######             Random Forest Model - Optimisation
355        ######              mtry approx equal to SQRT(p)
356        #######################################################################
357        #######################################################################
358        n_trees_list = c(500,1000,2000,3000,4000, 5000)
359
360        for (n_trees in n_trees_list){
361          model_name = "Random Forest (mtry = SQRT(p))"
362          model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | ntrees = ", n_trees, paste=""")
363          model_type = "OPTIMISATION"
364          uuid = UUIDgenerate()
365          m_try=3
366
367          #Get the Training and Validation data for the given time window
368          training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
369          training_data = final_df[training_range,]
370          validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
371          validation_data = final_df[validation_range,]
372
373          bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
374                                                          Volume +
375                                                          Close_price_2day_lag +
376                                                          Atr_atr +
377                                                          Mfi +
378                                                          Rsi +
379                                                          Atr_trueLow +
380                                                          Atr_tr +
381                                                          Close_price_5day_lag +
382                                                          Atr_trueHigh,
383                                    data=training_data,
384                                    ntree=n_trees,
385                                    mtry=m_try,
386                                    importance=TRUE)
387
388          possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
389                                                        uuid,stock_name,
390                                                        model_name,model_desc,
391                                                        model_type,model_comparison_summary_df,
392                                                        validation_data),
393                                    error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
394          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
```

Code Snippet 23b – Calling the Random Forest model, where m_try = SQRT(p), c.f. line 365 (Optimisation)

```
827         #######            Random Forest Model - Optimisation
828         ######             mtry approx equal to p_div_2
829         ##########################################################################
830         ##########################################################################
831         n_trees_list = c(500,1000,2000,3000,4000, 5000)
832 
833         for (n_trees in n_trees_list){
834           model_name = "Random Forest (mtry = p_div_2)"
835           model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + 
Close_price_5day_lag + Atr_trueHigh | ntrees = ", n_trees, paste=""")
836           model_type = "OPTIMISATION"
837           uuid = UUIDgenerate()
838           m_try=4
839 
840           #Get the Training and Validation data for the given time window
841           training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
842           training_data = final_df[training_range,]
843           validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
844           validation_data = final_df[validation_range,]
845 
846           bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
847                                                           Volume +
848                                                           Close_price_2day_lag +
849                                                           Atr_atr +
850                                                           Mfi +
851                                                           Rsi +
852                                                           Atr_trueLow +
853                                                           Atr_tr +
854                                                           Close_price_5day_lag +
855                                                           Atr_trueHigh,
856                                       data=training_data,
857                                       ntree=n_trees,
858                                       mtry=m_try,
859                                       importance=TRUE)
860 
861           possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
862                                                         uuid,stock_name,
863                                                         model_name,model_desc,
864                                                         model_type,model_comparison_summary_df,
865                                                         validation_data),
866                                   error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
867           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
868 
```
Code Snippet 23b – Calling the Random Forest model, where m_try = p/2, c.f. line 838 (Optimisation)

<u>Model Testing</u>

The best training model is chosen and is fitted against the entire training data set. It is then tested against the test set (i.e. the last 5 business days).

- For the m_try = SQRT(p) case, the *n_tree* is set to the best optimised value (1000).
- For the m_try = p/2 case, the *n_tree* is set to the best optimised value (4000).

The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same *run_rdmForest_model()* function is called as for the training phase, line 3496. This time, the test data is used in lieu of the validation data.

```
4399        ##########################################################################
4400        #######            Random Forest Model - Testing
4401        ######             mtry approx equal to SQRT(p)
4402        ##########################################################################
4403        ##########################################################################
4404        n_trees = 1000
4405        model_name = "Random Forest (mtry = SQRT(p))"
4406        model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag
 + Atr_trueHigh | ntrees = ", n_trees, paste=""")
4407        model_type = "TEST"
4408        uuid = UUIDgenerate()
4409        m_try=3
4410 
4411        for (tw_index in time_window_seq){
4412          #Get the Training and Validation data for the given time window
4413          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
4414          training_data = final_df[training_range,]
4415          test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
4416          test_data = final_df[test_range,]
4417 
4418          bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
4419                                                          Volume +
4420                                                          Close_price_2day_lag +
4421                                                          Atr_atr +
4422                                                          Mfi +
4423                                                          Rsi +
4424                                                          Atr_trueLow +
4425                                                          Atr_tr +
4426                                                          Close_price_5day_lag +
4427                                                          Atr_trueHigh,
4428                                      data=training_data,
4429                                      ntree=n_trees,
4430                                      mtry=m_try,
4431                                      importance=TRUE)
4432 
4433          possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
4434                                                        uuid,stock_name,
4435                                                        model_name,model_desc,
4436                                                        model_type,model_comparison_summary_df,
4437                                                        test_data),
4438                                  error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
4439          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
4440        }
4441        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
```
Code Snippet 23a – Calling the Random Forest model, where m_try = SQRT(p), c.f. line 4404 (Testing)

```r
4870
4871        ###########################################################################
4872        ###########################################################################
4873        #######             Random Forest Model  - Testing
4874        ######              mtry approx equal to p/2
4875        ###########################################################################
4876        ###########################################################################
4877        n_trees = 4000
4878        model_name = "Random Forest (mtry = p_div_2)"
4879        model_desc = paste("factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag
                + Atr_trueHigh | ntrees = ", n_trees, paste=""
4880        model_type = "TEST"
4881        uuid = UUIDgenerate()
4882        m_try=4
4883
4884        for (tw_index in time_window_seq){
4885          #Get the Training and Validation data for the given time window
4886          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
4887          training_data = final_df[training_range,]
4888          test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
4889          test_data = final_df[test_range,]
4890
4891          bagging_fit = randomForest( factor(Direction) ~ Close_price_1day_lag +
4892                                                          Volume +
4893                                                          Close_price_2day_lag +
4894                                                          Atr_atr +
4895                                                          Mfi +
4896                                                          Rsi +
4897                                                          Atr_trueLow +
4898                                                          Atr_tr +
4899                                                          Close_price_5day_lag +
4900                                                          Atr_trueHigh,
4901                                        data=training_data,
4902                                        ntree=n_trees,
4903                                        mtry=m_try,
4904                                        importance=TRUE)
4905
4906          possibleError = tryCatch( run_rdmForest_model(  bagging_fit,n_trees,
4907                                                        uuid,stock_name,
4908                                                        model_name,model_desc,
4909                                                        model_type,model_comparison_summary_df,
4910                                                        test_data),
4911                                    error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
4912          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
4913        }
```

Code Snippet 23b – Calling the Random Forest model, where m_try = p/2, c.f. line 4877 (Testing)

### *Results*

**<u>The m_try = SQRT(p) Case</u>**

As shown in the below table, the Random Forest model with the following configuration: *factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh generates* the highest training accurate rate at 45.08%. The optimisation model shows the best tree to grow level is at 1000. The test performance of the Tree model run against the selected list of attributes and the optimised min tree size produces a 34.80% accuracy rate, with a standard deviation of 26.30%. The data is available in the file: *Model_Results_Random Forest (mtry = SQRT(p)).xlsx*

| aggregation_type | AVG | |
|---|---|---|
| **Row Labels** | **Sum of model_accuracy_rate** | **Std Dev** |
| **TEST** | | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 1000 | 34.80 | 26.30 |
| **TRAIN** | | |
| factor(Direction) ~ Close_price_1day_lag | 33.72 | |
| factor(Direction) ~ Close_price_1day_lag + Volume | 35.14 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 33.18 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 31.94 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 34.87 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 37.22 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 44.12 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 42.28 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 41.69 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 45.08 | |

| aggregation_type | (blank) |
|---|---|
| **Row Labels** | **Sum of model_accuracy_rate** |
| **OPTIMISATION** | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 500 | 46.15 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 1000 | 50.00 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 2000 | 49.23 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 3000 | 48.46 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 4000 | 48.46 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 5000 | 48.46 |

**The m_try = SQRT(p) Case**

As shown in the below table, the Random Forest model with the following configuration: *factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh generates* the highest training accurate rate at 45.08%. The optimisation model shows the best tree to grow level is at 4000. The test performance of the Tree model run against the selected list of attributes and the optimised min tree size produces a 35.00% accuracy rate, with a standard deviation of 25.33%. The data is available in the file: *Model_Results_Random Forest (mtry = p_div_2).xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| **TEST** | | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 4000 | 35.00 | 25.33 |
| **TRAIN** | | |
| factor(Direction) ~ Close_price_1day_lag | 33.72 | |
| factor(Direction) ~ Close_price_1day_lag + Volume | 35.14 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 33.39 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 31.94 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 34.87 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 37.15 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 44.12 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 42.72 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 41.93 | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 44.70 | |

| aggregation_type | (blank) |
|---|---|

| Row Labels | Sum of model_accuracy_rate |
|---|---|
| **OPTIMISATION** | |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 1000 | 49.23 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 2000 | 50.77 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 3000 | 51.54 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 4000 | 52.31 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 500 | 49.23 |
| factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh \| ntrees = 5000 | 50.00 |

## Boosting

### *Model Description*

Boosting is similar to the Bagging approach, except that the trees are grown sequentially: each tree is grown using information from previously grown trees [14].

### *Model Assumptions*

No specific requirements.

### *Further data transformation*

No data transformation is required.

### *Parameter Tuning*

The tree to fit and the shrinkage parameters are optimised.

### *Code Snippet Explanation*

<u>Model Training</u>

The *for-loop*, line 2683 iterates through the training data time windows (100 iterations). At each iteration, the the *gbm (x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 2692). It produces the *boost_fit* object, i.e. the function fitting the model. The *boost _fit* model is then passed to the *run_boosting_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period. On successful run, the model results are stored into an object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*. The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 2712. In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 2710).
The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
2671        ##########################################################################
2672        ##########################################################################
2673        #######             Boosting Model
2674        ##########################################################################
2675        ##########################################################################
2676
2677        model_name = "Boosting"
2678        model_desc = "Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + 
    Atr_trueHigh"
2679        model_type = "TRAIN"
2680        uuid = UUIDgenerate()
2681        n_trees = 500
2682
2683        for (tw_index in time_window_seq){
2684          #Get the Training and Validation data for the given time window
2685          training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
2686          training_data = final_df[training_range,]
2687          validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
2688          validation_data = final_df[validation_range,]
2689
2690          #This is a mutinominal classification problem. Therefore, the distribution is set to "multinomial".
2691          #The gbm() function fit the Direction variable against all other selected predictors on the training set.
2692          boost_fit = gbm(Direction ~ Close_price_1day_lag +
2693                                      Volume +
2694                                      Close_price_2day_lag +
2695                                      Atr_atr +
2696                                      Mfi +
2697                                      Rsi +
2698                                      Atr_trueLow +
2699                                      Atr_tr +
2700                                      Close_price_5day_lag +
2701                                      Atr_trueHigh,
2702                        data = training_data, distribution = "multinomial", n.trees = n_trees)
2703
2704          possibleError = tryCatch( run_boosting_model(  boost_fit,uuid,
2705                                                         n_trees,stock_name,
2706                                                         model_name,model_desc,
2707                                                         model_type,model_comparison_summary_df,
2708                                                         validation_data),
2709                                    error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
2710          add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
2711        }
2712        generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
2713
```

Code Snippet 24 – Calling the Boosting model (Training)

This below function is responsible for generating the confusion matrix, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (line 1253/1259).

```
1229 run_boosting_model = function(fit_param, uuid, n_trees_param, stock_name, model_name, model_desc,model_type, model_comparison_summary_df,test_data_param){
1230
1231   set.seed(1)
1232
1233   boost.fit = fit_param
1234
1235   #Geneate the model prediction. Type=response returns the class prediction as probability
1236   boost.pred = predict(boost.fit, test_data_param, n.trees = n_trees_param, type="response")
1237
1238   #We know need to find the most probable class (the class showing the highest probability) for each row of the boost.pred
1239   #see http://stackoverflow.com/questions/29454883/in-gbm-multinomial-dist-how-to-use-predict-to-get-categorical-output
1240   boost.proba = apply(boost.pred, 1, which.max)
1241   #And now turn the class id (1,2 or 3) into the class name
1242   boost.pred_class = colnames(boost.pred)[boost.proba]
1243   #necessary as a non-compatible skrinkage could generate unpredictable classes
1244   boost.accuracy_rate = "NA"
1245   boost.error_rate = "NA"
1246   if (length(boost.pred_class) > 0){
1247     boost.confusion_table =  table(boost.pred_class, test_data_param$Direction)
1248     boost.accuracy_rate = accuracy_rate_perc(boost.confusion_table)
1249     boost.error_rate = error_rate_perc(boost.confusion_table)
1250   }
1251
1252   #Add a row in the model comparison dataframe
1253   model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1254                                                             uuid,
1255                                                             stock_name,
1256                                                             "TRUE",
1257                                                             "",
1258                                                             model_name, model_desc,model_type,
1259                                                             boost.accuracy_rate, boost.error_rate)
1260
1261
1262   return (model_comparison_summary_df)
1263 }
1264
```

Code Snippet 25 – Calling the *run_boosting_model()* function

<u>Model Optimisation</u>

This code is similar to the training phase above. The number of tree(s) to fit (tree_list) and their shrinkage (lambdas) parameters are is provided (line 3056/3058). The optimisation of these two parameters is run over the last training period and generate the accuracy. The highest accuracy is retained. The minimum tree/shrinkage level are selected for this accuracy level.

```
3044
3045         ########################################################################
3046         ########################################################################
3047         #######             Boosting Optimisation
3048         ########################################################################
3049         ########################################################################
3050         #The optimisation is performed on the last sliding window
3051         training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
3052         training_data = final_df[training_range,]
3053         validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
3054         validation_data = final_df[validation_range,]
3055
3056         tree_list = c (500,1000,1500,2000,3000,5000)
3057         #Generate a vector of sequence of staring from 0.1 and going to 1.0, with a 0.1 step
3058         lambdas = seq(0.1, 1, by = 0.1)
3059         len = length(lambdas)
3060         for (the_tree in tree_list){
3061           for (i in 1:len) {
3062             model_name = "Boosting"
3063             model_desc = paste("Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | n_trees =", the_tree, " |
     shrinkage =", lambdas[i] , sep= " ")
3064             model_type = "OPTIMISATION"
3065             uuid = UUIDgenerate()
3066
3067             boost_fit = gbm(Direction ~ Close_price_1day_lag +
3068                                         Volume +
3069                                         Close_price_2day_lag +
3070                                         Atr_atr +
3071                                         Mfi +
3072                                         Rsi +
3073                                         Atr_trueLow +
3074                                         Atr_tr
3075                             ,data = training_data, distribution = "multinomial", n.trees = the_tree, shrinkage = lambdas[i])
3076
3077             possibleError = tryCatch( run_boosting_model(  boost_fit,uuid,
3078                                                            n_trees,stock_name,
3079                                                            model_name,model_desc,
3080                                                            model_type,model_comparison_summary_df,
3081                                                            validation_data),
3082                                       error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3083             add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3084           }
3085         }
3086
```

Code Snippet 26 – Calling the Boosting model for the optimisation phase

## Model Testing

The best training model is chosen alongside its hyper-parameter list. In this case the *n_tree* (line 1092) is set to the best optimised value (1000), with its shrinkage level at 0.1 (line1093). It is then fitted against the entire training data set, and finally tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same run_mlr_model() function is called as for the training phase, line 1678. This time, the test data is used in lieu of the validation data.

```
3087         ##########################################################################
3088         ##########################################################################
3089         #######             Boosting Model - Testing
3090         ##########################################################################
3091         ##########################################################################
3092         n_trees = 500
3093         the_lambda = 0.1
3094         model_name = "Boosting"
3095         model_desc = paste("Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | n_trees =", the_tree, " |
     shrinkage =", the_lambda , sep= " ")
3096
3097         model_type = "TEST"
3098         uuid = UUIDgenerate()
3099
3100         for (tw_index in time_window_seq){
3101           #Get the Training and Validation data for the given time window
3102           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
3103           training_data = final_df[training_range,]
3104           test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
3105           test_data = final_df[test_range,]
3106
3107             boost_fit = gbm(Direction ~ Close_price_1day_lag +
3108                                         Volume +
3109                                         Close_price_2day_lag +
3110                                         Atr_atr +
3111                                         Mfi +
3112                                         Rsi +
3113                                         Atr_trueLow +
3114                                         Atr_tr
3115                                  ,data = training_data, distribution = "multinomial", n.trees = n_trees, shrinkage = the_lambda)
3116
3117             possibleError = tryCatch( run_boosting_model(  boost_fit,uuid,
3118                                                           n_trees,stock_name,
3119                                                           model_name,model_desc,
3120                                                           model_type,model_comparison_summary_df,
3121                                                           test_data),
3122                                         error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
3123             add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
3124         }
3125         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
3126
```

Code Snippet 27 – Calling the Boosting model (Testing)

### *Results*

As shown in the below table, the Boosting model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi+Atr_trueLow + Atr_tr* generates the highest training accurate rate at 35.53%. The optimisation model shows the number of trees and shrinkage should be set to respectively 1000 and 0.1. The test performance of the Boosting model, run against the selected list of attributes and the optimised parameters, produces a 47.60% accuracy rate, with a standard deviation of 25.71%. The data is available in the file: *Model_Results_Boosting.xlsx*

| Row Labels | Sum of model_accuracy_rate | Std Dev |
|---|---|---|
| ⊟TEST | | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 1 | 47.60 | 25.71 |
| ⊟TRAIN | | |
| Direction ~ Close_price_1day_lag + Volume | 31.00 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag | 30.94 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr | 32.76 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi | 32.08 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi | 31.94 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow | 32.37 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr | 35.53 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag | 33.83 | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | 34.02 | |
| factor(Direction) ~ Close_price_1day_lag^3 + Volume^2 + Close_price_1day_lag + Volume + (Atr_atr * Mfi) + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr - | 31.42 | |

| aggregation_type | (blank) |
|---|---|
| **Row Labels** | **Sum of model_accuracy_rate** |
| ⊟OPTIMISATION | |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.1 | 54.62 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.2 | 52.31 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.3 | 50.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.4 | 50.77 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.5 | 53.85 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.6 | 50.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.7 | 0.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.8 | 0.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 0.9 | 0.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1000 \| shrinkage = 1 | 0.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1500 \| shrinkage = 0.1 | 54.62 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1500 \| shrinkage = 0.2 | 52.31 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1500 \| shrinkage = 0.3 | 50.00 |
| Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr \| n_trees = 1500 \| shrinkage = 0.4 | 50.77 |

## Support Vector Machine (SVM)

### *Model Description*

The SVM is a generalisation of the classifier method named the maximal margin classifier [14]. The SVM finds a plane that separates the classes in feature space. It can accommodate non-linear class boundaries and multinomial classes.

### *Model Assumptions*

No specific requirements.

### *Further data transformation*

The R *e1071.svm (x,…)* function implementation requires that the regressor variable is encoded as a factor (i.e. an enumerated type). Therefore, the *factor()* function has been applied on the *Direction* regressor. None of the other parameters need to be adapted.

### *Parameter Tuning*

There are several parameters that are optimised: the kernel type (used for linear or nonlinear classification learning), the gamma (a parameter used for kernels) and the cost (i.e. 'C'-constant of the regularization term in the Lagrange formulation).

### *Code Snippet Explanation*

Model Training

The *for-loop*, line 933 iterates through the training data time windows (100 iterations). At each iteration, the the *svm (x,…)* function is called with a list of explanatory and explained variables, for a training data slice (line 940). It produces the *svm_fit* object, i.e. the function fitting the model. The *svm_fit* model is then passed to the *run_svm_model(…)* function, alongside a few parameters. One of them is the validation set for the time slide period. On successful run, the model results are stored into an object named *model_comparison_summary_df*. It is tagged with the state *model_run_success = TRUE*. The training accuracy rate is computed at each iteration. The average of the training accuracy is calculated on line 957. In case of a computational failure, the model description is added to the same in-memory data frame and tagged with the state *model_run_success = FALSE* (line 953).
The below code snippet only shows one example of a model trained against a given set of attributes. This example is repeated for each list of attributes. An average training accuracy rate is generated for each instance. The complete list of training case is available in the source code.

```
922
923         ##########################################################################
924         ##########################################################################
925         #######   Support Vector Machine (SVM) - Model Training
926         ##########################################################################
927         ##########################################################################
928         model_name = "Support Vector Machine (SVM)"
929         model_desc = "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag +
    Atr_trueHigh"
930         model_type = "TRAIN"
931         uuid = UUIDgenerate()
932
933         for (tw_index in time_window_seq){
934           #Get the Training and Validation data for the given time window
935           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"training_end_index"]
936           training_data = final_df[training_range,]
937           validation_range = time_window_df[tw_index,"validation_start_index"]:time_window_df[tw_index,"validation_end_index"]
938           validation_data = final_df[validation_range,]
939
940           svm_fit =  svm (factor(Direction) ~ Close_price_1day_lag +
941                                      Volume +
942                                      Close_price_2day_lag +
943                                      Atr_atr +
944                                      Mfi +
945                                      Rsi +
946                                      Atr_trueLow +
947                                      Atr_tr +
948                                      Close_price_5day_lag +
949                                      Atr_trueHigh,
950                            probability = TRUE, #indicates the model should allow for probability predictions
951                            data=training_data)
952
953           possibleError = tryCatch( run_svm_model(svm_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
954                                 error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
955           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
956         }
957         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
958
```

Code Snippet 28 – Calling the SVM model (Training)

The below code generates the confusion matrix and accuracy rate for the SVM model. Line 1162/1164 deals with predicting each class index (1,2,3) probabilities. This is necessary as the SVM is used in a multinomial mode. The class index with the highest probability is selected (line 1666/1176). Lines 1177/1183 deal with converting the class index into class names (*Up/Down/Neutral*). Then the confusion matrix is generated, which produces an accuracy rate of the predicted vs actual *Directions.* The validation data set is used for this purpose. The accuracy result is stored for each iteration in the in-memory table (Line 1192/1198).

```
1155 #the test_data_param could be a traning/validation or test dataset
1156 run_svm_model = function(svm_param, uuid, stock_name, model_name, model_desc, model_type, model_comparison_summary_df, test_data_param){
1157
1158     svm.fit = svm_param
1159
1160     #Generate the confusion matrix and calculate the training/validation error rate
1161     #decision.values = TRUE, probability = TRUE are necessary when multi class probabilities are required
1162     pred_prob = predict(svm.fit, test_data_param, decision.values = TRUE, probability = TRUE)
1163     #Get the class probabilities
1164     pred_prob_attr = attr(pred_prob, "probabilities")
1165     #Get the most probable class per row
1166     pred_class_as_factor = apply(pred_prob_attr, 1, which.max)
1167     pred_class_as_name = pred_class_as_factor
1168     #Get the name of each class
1169     col_name =  colnames(attr(pred_prob, "probabilities"))
1170     index = 1
1171     map = NULL
1172     #Create a list  that map an index to a class name
1173     for (cn in col_name){
1174         map = append(map,list(id = index, name = cn))
1175         index =index +1
1176     }
1177     #Find the predicted class name, given the predicted class factor, from the map list
1178     index = 1
1179     for(class_as_factor in pred_class_as_factor){
1180       pos = match(class_as_factor,map)
1181       pred_class_as_name[index] = map[pos+1]
1182       index = index +1
1183     }
1184     #Generate a dataframe for the predicted class names
1185     svm.pred_class_as_name_df = data.frame(matrix(unlist(pred_class_as_name)),stringsAsFactors=FALSE)
1186     #Generate the confusion matrix
1187     svm.confusion_table =  table(svm.pred_class_as_name_df$matrix.unlist.pred_class_as_name.., test_data_param$Direction)
1188     svm.accuracy_rate = accuracy_rate_perc(svm.confusion_table)
1189     svm.error_rate = error_rate_perc(svm.confusion_table)
1190
1191     #Add a row in the model comparison dataframe
1192     model_comparison_summary_df <<- add_row_to_model_summary( model_comparison_summary_df,
1193                                                               uuid,
1194                                                               stock_name,
1195                                                               "TRUE",
1196                                                               "",
1197                                                               model_name, model_desc,model_type,
1198                                                               svm.accuracy_rate, svm.error_rate)
1199
1200     return (model_comparison_summary_df)
1201 }
```

Code Snippet 29 – Calling the *run_svm_model()* function

## Model Optimisation

This code is similar to the training phase above. However, there are three parameters to optimise, namely the cost, gamma and kernel type (c.f. line 5237/8239). The optimisation of these three parameters, via 3 inner loops (line 5241/5239), is run over the last training period and generate the accuracy. The hyperparameter list with the highest accuracy is retained.

```r
5230         ##########################################################################
5231         ##########################################################################
5232         #######   Support Vector Machine (SVM) - Model Optimisation  (cost/gamma and kernel)
5233         ##########################################################################
5234         ##########################################################################
5235
5236         #The model above model will be run each cost in the below cost list:
5237         cost_list = c(0.001 , 0.01 , 0.1, 1 ,100 )
5238         gamma_list = c(0.01,0.03,0.05,0.5,1,100)
5239         kernel_list = c("linear", "radial", "sigmoid", "polynomial")
5240
5241         for (the_kernel in kernel_list){
5242           for (the_cost in cost_list){
5243             for (the_gamma in gamma_list){
5244               model_name = "Support Vector Machine (SVM)"
5245               model_desc = paste( "factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = ", toString(the_kernel), " | cost = ", toString(the_cost),"| gamma = ", toString(the_gamma), sep="")
5246               model_type = "OPTIMISATION"
5247               uuid = UUIDgenerate()
5248
5249               #The optimisation is performed on the last sliding window
5250               training_range = time_window_df[number_sliding_windows,"training_start_index"]:time_window_df[number_sliding_windows,"training_end_index"]
5251               training_data = final_df[training_range,]
5252               validation_range = time_window_df[number_sliding_windows,"validation_start_index"]:time_window_df[number_sliding_windows,"validation_end_index"]
5253               validation_data = final_df[validation_range,]
5254
5255               svm_fit =  svm (factor(Direction) ~ Close_price_1day_lag +
5256                                                    Volume +
5257                                                    Atr_tr +
5258                                                    Close_price_2day_lag +
5259                                                    Rsi +
5260                                                    Mfi +
5261                                                    Close_price_5day_lag +
5262                                                    Atr_trueHigh +
5263                                                    Atr_trueLow,
5264                                                    kernel = the_kernel,
5265                                                    cost = the_cost,
5266                                                    gamma = the_gamma,
5267                                          probability = TRUE, #indicates the model should allow for probability predictions
5268                                          data=training_data)
5269
5270               possibleError = tryCatch( run_svm_model(svm_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,validation_data),
5271                                         error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
5272               add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
5273             }
5274           }
```

239:1 Chunk 8: tree_model_training R Markdow

Code Snippet 30 – Calling the SVM model for the optimisation phase

## Model Testing

The best training model is chosen alongside the optimised parameters (Kernel = ‘linear’, gamma =0.01, cost =1). Tis model is fitted against the entire training data set. It is then tested against the test set (i.e. the last 5 business days). The confusion matrix, that evaluates the test prediction vs the expected test data, provides the test accuracy rate. The same *run_svm_model()* function is called as for the training phase, line 5313. This time, the test data is used in lieu of the validation data.

```
5277         ########################################################################
5278         ########################################################################
5279         #######   Support Vector Machine (SVM) - Model Testing
5280         ########################################################################
5281         ########################################################################
5282         the_kernel = "linear"
5283         the_cost = 1
5284         the_gamma = 0.01
5285         model_name = "Support Vector Machine (SVM)"
5286         model_desc = paste( "factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag
     + Atr_trueHigh | kernel = ", toString(the_kernel), " | cost = ", toString(the_cost),"| gamma = ", toString(the_gamma), sep="")
5287         model_type = "TEST"
5288         uuid = UUIDgenerate()
5289
5290         for (tw_index in time_window_seq){
5291           #Get the Training and Validation data for the given time window
5292           training_range = time_window_df[tw_index,"training_start_index"]:time_window_df[tw_index,"validation_end_index"]
5293           training_data = final_df[training_range,]
5294           test_range = time_window_df[tw_index,"test_start_index"]:time_window_df[tw_index,"test_end_index"]
5295           test_data = final_df[test_range,]
5296
5297           svm_fit =  svm (factor(Direction) ~ Close_price_1day_lag +
5298                                               Volume +
5299                                               Close_price_2day_lag +
5300                                               Atr_atr +
5301                                               Mfi +
5302                                               Rsi +
5303                                               Atr_trueLow +
5304                                               Atr_tr +
5305                                               Close_price_5day_lag +
5306                                                 Atr_trueHigh,
5307                                               kernel = the_kernel,
5308                                               cost = the_cost,
5309                                               gamma = the_gamma,
5310                            probability = TRUE, #indicates the model should allow for probability predictions
5311                            data=training_data)
5312
5313           possibleError = tryCatch( run_svm_model(svm_fit,uuid,stock_name,model_name,model_desc,model_type,model_comparison_summary_df,test_data),
5314                                 error = function(e) print(paste("MODEL ERROR: ", e, sep="")))
5315           add_failed_model(model_comparison_summary_df,uuid, stock_name, model_name,model_desc,model_type, possibleError)
5316         }
5317         generate_avg_model(model_comparison_summary_df,uuid, time_window_seq,model_type)
5318
5319         cat ("\n")
```

Code Snippet 30 – Calling the SVM model (Testing)

### *Results*

As shown in the below table, the SVM model with the following configuration: *Direction ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh* generates the highest training accurate rate at 49.85%. The optimisation model shows the kernel, cost and gamma to be respectively set to ‘linear’, 1 and 0.01.  The test performance of the SVM model run against the selected list of attributes and the optimised parameters produces a 68.40% accuracy rate, with a standard deviation of 26.39%. The data is available in the file: *Model_Results_Boosting.xlsx*

<table>
<tr><td>aggregation_type</td><td>AVG</td><td></td></tr>
<tr><td>Row Labels</td><td>Sum of model_accurac</td><td>Std Dev</td></tr>
<tr><td>TEST</td><td></td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh | kernel = linear | cost = 1| gamma = 0.01</td><td>68.40</td><td>26.39</td></tr>
<tr><td>TRAIN</td><td></td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag</td><td>28.74</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume</td><td>28.43</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag</td><td>28.23</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr</td><td>29.08</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi</td><td>32.42</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi</td><td>32.45</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow</td><td>47.18</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr</td><td>48.44</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr + Close_price_5day_lag + Atr_trueHigh</td><td>49.85</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Close_price_2day_lag + Atr_atr + Mfi + Rsi + Atr_trueLow + Atr_tr Close_price_5day_lag</td><td>47.35</td><td></td></tr>
</table>

<table>
<tr><td>aggregation_type</td><td>(blank)</td></tr>
<tr><td>Row Labels</td><td>Sum of model_accuracy_rate</td></tr>
<tr><td>OPTIMISATION</td><td></td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 0.01</td><td>60.77</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 0.03</td><td>63.85</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 0.05</td><td>60.77</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 0.5</td><td>60.00</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 1</td><td>63.08</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.001| gamma = 100</td><td>66.15</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 0.01</td><td>62.31</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 0.03</td><td>64.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 0.05</td><td>65.38</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 0.5</td><td>60.77</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 1</td><td>66.15</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.01| gamma = 100</td><td>60.77</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 0.01</td><td>74.62</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 0.03</td><td>73.85</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 0.05</td><td>73.85</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 0.5</td><td>73.85</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 1</td><td>73.85</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 0.1| gamma = 100</td><td>73.08</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 1| gamma = 0.01</td><td>80.00</td></tr>
<tr><td>factor(Direction) ~ Close_price_1day_lag + Volume + Atr_tr + Close_price_2day_lag + Rsi + Mfi + Close_price_5day_lag + Atr_trueHigh + Atr_trueLow | kernel = linear | cost = 1| gamma = 0.03</td><td>80.00</td></tr>
</table>

## Evaluation

The below table summarises the test performance obtained for each model, in decreasing order of average test performance accuracy. Initial experiments on a smaller time slice window of approximately 100 days (vs 260days here), and without class rebalancing, showed the test accuracy for LDA and QDA at approximately 50%. Bagging was approximately at 63%. With the current settings, the situation is reversed. The three best performing algorithms are the QDA, Ridge and LDA with approximatively an 80% test accuracy rate and a standard deviation at approximately 1.4%. The Lasso model follows in the tail of the first three ones, with a test performance accuracy of 75.39% and a low standard deviation at 1.44%. All the other model show test performance accuracy rates below 70%, with high variance, around 25% standard deviation.
The surprising fact is that most of attributes do not follow a normal distribution (c.f. *Normal Distribution Test* section), but at the time, the models that require attributes normal distribution, i.e. QDA, Ridge, LDA and Lasso, are showing the best average test performance accuracies and stabilities.

| **Model Name** | **Average Test Performance** |
|---|---|
| QDA | Avg: 80.90% - Std Dev:1.64% |
| Ridge | Avg: 80.71% - Std Dev:1.38% |
| LDA | Avg: 80.20% - Std Dev:1.06% |
| Lasso | Avg: 75.39% - Std Dev:1.44% |
| SVM | Avg: 68.40% - Std Dev: 26.39% |
| Boosting | Avg: 47.60% - Std Dev: 25.71% |
| Decision Tree | Avg: 40.80% - Std Dev:21.45% |
| Bagging (Random Forest where mtry= p) | Avg: 37.00% - Std Dev:27.47% |
| Random Forest<br>mtry= p/2<br>mtry= SQRT(p) | <br>Avg: 35.00% - Std Dev: 25.33%<br>Avg: 34.80% - Std Dev: 26.30% |

The next step involves looking at some of test confusion matrix measures for the model with the best test performance results, i.e. QDA. The aim is to establish whether other patterns can be uncovered. The description of the measures and formulas are provided below. The full implementation is available in the file named: *qda_confusion_matrix.xlsx*. The sensitivity levels showed in the below QDA table, indicates that the model is better at discovering *Neutral* trends (sensitivity =86.63%), and then it scores higher in discovering *Up* trends (sensitivity=76.45%) than *Down* trends (sensitivity=68.91%). The precision and F1 measure shows the same pattern for *Neutral* trends, but this time, precision for *Down* trend scores higher than for *Up* trends.

| **Total Population** | **Prediction Positive** | **Prediction Negative** |
|---|---|---|
| **Expected Positive** | True Positive (TP) | False Negative (FN) |
| **Expected Negative** | False Positive (FP) | True Negative (TN) |

| **Measure** | **Formula** | **Description** |
|---|---|---|
| Sensitivity (a.k.a. recall) | TP/(TP+FN) | It measures the proportion of positives that are correctly identified (i.e. how good is a test at detecting the positives). |
| Specificity | TN/(TN+FP) | It measures the proportion of negatives that are correctly identified (i.e. how good is a test at detecting false alarms). |
| Precision | TP/(TP+FP) | It measures the proportion of positives that were relevant. |
| F1-Score | 2*TP/(2*TP+FP+ FN) | It is a measure of the test accuracy. F1-Score is always between 0 (worst) and 1(best). |
| Accuracy | (TP+TN)/(TP+FN+FP+TN) | It is a measure of the statistical bias. Accuracy is between 0 (maximum bias) and 1 (no bias). |

| **QDA Measures / Scenarios** | **Down Vs (Neutral + Up)** | **Neutral Vs (Down + Up)** | **Up Vs (Down + Neutral)** |
|---|---:|---:|---:|
| Sensitivity/Recall (i.e. True Positive Rate) | 68.91% | 86.63% | 76.45% |
| Specificity (i.e. true negative rate) | 98.59% | 77.64% | 88.89% |
| Precision (i.e. positive predicted values) | 91.72% | 83.91% | 68.71% |
| F1-Score (i.e. is the harmonic mean of precision and sensitivity) | 78.69% | 85.25% | 72.37% |

## Challenges & Potential Improvements

- The Skewness reduction uses a constant set to 0.0025. This represents a +/-25bps buffer zone around zero, a.k.a. the ε range. This is quite a large number in trading terms, as spot trading margins are usually a few basis points. Furthermore, it is a synthetic buffer fabricated solely to rebalance the class instances. Indeed, the *Neutral* direction instances are usually underrepresented, and can cause models to break (e.g. QDA). It can also provoke skew in the training/test accuracy measures. The advantage of the approach is that rebalancing classes improve the model overall accuracy. However, the drawback relates to the potential for ignoring slow and continuous daily trend increment (decrement) over several days. Assuming there is a slow daily continuous trend increment of ε-0.01 on a daily basis for 10days, the data would be tagged with 10 days of *Neutral* directions. In reality, the cumulated effected of the 10days increase(decrease) induces a *Up (Down)* trend, over the time period. This could have a financial impact, as trading margins are much smaller than 25bps. This situation is unlikely but not entirely impossible. A better solution could be to use the daily asset volatility multiplied by a 'fudge' factor (e.g. 0.25) to try to stick closer to the asset real directionality.
- The feature selection was performed on the last time window, and was then used for all other time windows during the model training, optimisation and the test phases. This is a strong hypothesis that may need to be refined, as the data pattern carried in the last time window may not necessarily reflect the data pattern in each time window. More research needs to be carried out in this space.
- It would be interesting to investigate the test performance of neural networks such as multilayer perceptron (MLP), recursive neural networks (RNN), etc.
- The Ridge (α =0) and Lasso (α =1) models are subset of the Elastic Net regression. It would be interesting to see the impact of different levels of α (between 0 and 1) on the test accuracy rates.
- Initially, the *nnet.multinom()* function was selected to perform multinomial linear regression (due to the simplicity of its API). However, it requires that the training and validation/test sets are of the same dimension. This did not fit with the current sliding time window set-up. Therefore, it was dismissed in favour of the more complex *glmnet.glmnet()* function.
- Each model has been run against a number of different attributes. This is currently very a manual procedure. More investigation should be carried out (perhaps using the Caret library) to see whether code reduction could be implemented in this area. Also, dynamic formula creation should also be investigated [13].
- Currently the sliding time window supports 100 iterations. In order to improve the test performance accuracy, the time window should be increased (e.g. to 200, 300, etc. iterations).
- Some algorithms, such as the SVM are very slow when running the model with a matrix of optimisation factors, over multiple time windows. This could mean running multiple inner loops over the selected number of time windows. The number of loops depends on the number of factor to optimise. Therefore, it was decided to perform the optimisation on the last time window (the one closer to dataset end date). This may not be the best approach, as the optimisation should be run across all time widows for a given model. This can only be achieved with code refactoring, as explained below, as we need to move from a synchronous to a parallel model, to benefit fully from the server multi-cores.
- The current code has grown organically and has been built in a monolithic way. Each model is run after one another. This is not ideal for two reasons: i) running a model requires all the raw data to be load and massaged before any models can be run, ii) models cannot be run in parallel and iii) when a model fail, all following models may not run. The code refactoring should produce the following three independent modules:
    - An R module specialized in i) loading the raw dataset for an asset name (e.g. JPM), ii) removing the missing data row, iii) generating the derived data (e.g. technical indicators), iv) saving modified dataset into a file containing the asset name. This operation should be performed once (or anytime the derived data needs to regenerated).
    - A module specialised in carrying out data analysis (e.g. the Kolmogorov Smirnov Normal Distribution test) and the generation of visuals.
    - A R module per model that deals with both model training, optimisation and test performance/accuracy measures.

Final Note:

- While running the different model with the polynomial and interaction between variables, two error messages (depending on the model type) were consistently displayed: i) *'Error in [model_name].default(x, grouping, ...) : rank deficiency in group Down'* or ii) '*Error in tree(factor(Direction) ~ Close_price_1day_lag^3 + Volume^2 + : trees cannot handle interaction terms'.* The experiment relating to mixing polynomial and attribute interactions was therefore commented out for each model that threw this exception.
- Another error message appeared, in the case of the Boosting model. The model could not be trained against only one attribute as the code threw the following error: '*Boosting with Close_price_1day_lagas only attribute => Error in x[1:nTrain, , drop = FALSE] : incorrect number of dimensions'.* In that instance, the experiment was also removed.

## Conclusion

This experiment showed that for a sliding window of 100 iterations, over a 260 days' training, 65 days' validation and 5 days' testing contiguous data sets; the daily JPM stock trend can be explained endogenously at average test performance accuracy of approximately 80% (and standard deviation of 1.4%) for the Quadratic Discriminant Analysis (QDA), Ridge and Linear Discriminant Analysis (LDA) models. This means that approximately 20% of the price move is dependent on other factors. Therefore, it would be interesting to assess the impact of other variables such as fundamental or sentiment factors on the price trend. It would also be interesting to test the stability of the model accuracy over longer sliding windows and/or extra iterations, and to use other models such as neural network models (MPL, RNN, etc.).

.

## References

[1] Technical Analysis [Online], Available at: *https://www.tradingview.com/chart/technicalanalysis/*, [Accessed 27 March 2017]

[2] Technical Indicator[Online], Available at: *http://www.investopedia.com/terms/t/technicalindicator.asp*, [Accessed 27 March 2017]

[3] Vangie B., API - application program interface [Online], Available at: *http://www.webopedia.com/TERM/A/API.html*, [Accessed 27 March 2017]

[4] Fundamental Analysis [Online], Available at*: http://www.investopedia.com/terms/f/fundamentalanalysis.asp*, [Accessed 27 March 2017]

[5] Available at*:* http://ichart.finance.yahoo.com/table.csv?s=JPM, [Accessed 14 December 2016]

[6] Bruder B., Dao T.L., Richard J.C, Roncalli T. (December 2011), *Trend Filtering Methods for Momentum Strategies,* Available at*:* http://www.thierry-roncalli.com/download/lwp-tf.pdf, [Accessed 27 march 2017]

[7] Murphy J. (1986), *Technical Analysis of the Financial Markets: A Comprehensive Guide to Trading Methods and Applications*, New York Institute of Finance, pp.225-262.

[7bis] Quantmod Quantitative Financial Modelling & Trading Framework for R [Online], Available at: http://www.quantmod.com/, [Accessed 27 March 2017]

[8] Indicator Reference [Online], Available at: *http://www.fmlabs.com/reference/default.htm*, [Accessed 27 March 2017]

[9] Moving Averages – Simple and Exponential [Online], Available at: *http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators:moving_averages* , [Accessed 27 March 2017]

[10] Jeni L.A., Cohn J.F., De La Torre F, *Facing Imbalanced Data: Recommendations for the Use of Performance Metrics* [Online], Available at: http://www.pitt.edu/~jeffcohn/skew/PID2829477.pdf, [Accessed 27 March 2017]

[11] Morales R., Di Matteo T., Aste T [Online], Available at: Dependency structure and scaling properties of financial time series are related, http://www.nature.com/articles/srep04589, [Accessed 27 March 2017]

[12] Volatility (finance) [Online], Available at: *https://en.wikipedia.org/wiki/Volatility_(finance)*, [Accessed 27 March 2017]

[13] Dynamic formula creation in R? [Online], Available at: *http://stackoverflow.com/questions/29711599/dynamic-formula-creation-in-r*, [Accessed 27 March 2017]

[14] Gareth James , Daniela Witten , Trevor Hastie , Robert Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Publishing Company, Incorporated, 2014, pp. 143, 312

# Appendices

## Appendix A – The Kolmogorov Smirnov Test Details

***Check JPM 'Close_price_1day_lag/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 2.648992e-13 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_1day_lag/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 3.620437e-13 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_1day_lag/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Volume/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Volume/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Volume/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_2day_lag/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 9.880985e-15 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_2day_lag/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 1.338374e-12 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_2day_lag/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_atr/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_atr/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_atr/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Mfi/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0.3247685 > 0.05 -> H0 (the null hypothetsis) is NOT rejected. There is not enough evidence to reject the hypothesis that the distribution is normal. Therefore, the data seems to follow normal distribution

***Check JPM 'Mfi/Down' is normally distributed***

H0 = the data is normally distributed.
The ks p_value: 0.2228885 > 0.05 -> H0 (the null hypothetsis) is NOT rejected. There is not enough evidence to reject the hypothesis that the distribution is normal. Therefore, the data seems to follow normal distribution
***Check JPM 'Mfi/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0.3501365 > 0.05 -> H0 (the null hypothetsis) is NOT rejected. There is not enough evidence to reject the hypothesis that the distribution is normal. Therefore, the data seems to follow normal distribution
***Check JPM 'Rsi/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0.01189858 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Rsi/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0.08154547 > 0.05 -> H0 (the null hypothetsis) is NOT rejected. There is not enough evidence to reject the hypothesis that the distribution is normal. Therefore, the data seems to follow normal distribution
***Check JPM 'Rsi/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0.02306936 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trueLow/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 1.699751e-12 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trueLow/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 3.83249e-13 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trueLow/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_tr/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_tr/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trLow/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_5day_lag/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 1.598721e-14 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_5day_lag/Down' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 6.695755e-13 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Close_price_5day_lag/Neutral' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trueHigh/Up' is normally distributed***
H0 = the data is normally distributed.
The ks p_value: 2.742251e-14 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.
***Check JPM 'Atr_trueHigh/Down' is normally distributed***

H0 = the data is normally distributed.

The ks p_value: 4.746203e-13 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.

***Check JPM 'Atr_trueHigh/Neutral' is normally distributed***

H0 = the data is normally distributed.

The ks p_value: 0 < 0.05 -> H0 (the null hypothesis) is rejected. The data distribution does not seem to follow a normal distribution.

## Appendix B – Code for Graphics Generation

```
#This function plots the different moving averages
plot_simple_moving_averages = function (stock_df, from_date, to_date){
  #reset display to one graph per window
  par(mfrow=c(1,1))

  step_by = "6 mon"
  date_format = "%d-%m-%Y"

  from_date = get_from_date(stock_df, from_date);
  to_date = get_to_date(stock_df, to_date);

  #Only plot when dates are consistent
  if(to_date> from_date){
    #Reduce the data range to the required time range
    stock_df = stock_df[stock_df$DateAsDate >=from_date & stock_df$DateAsDate <=to_date, ]

    #Get the highest y value for the plot
    max_height = max(stock_df$Sma20,stock_df$Sma50,stock_df$Sma100,stock_df$Sma200, na.rm = TRUE)
    min_height = min(stock_df$Sma20,stock_df$Sma50,stock_df$Sma100,stock_df$Sma200, na.rm = TRUE)
    #Plot
    plot( stock_df$DateAsDate,
          stock_df$Close,
          col="black",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Stock Price ($)",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Close Price Vs Simple Moving Average (SMA) Crossovers\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)
    lines(stock_df$DateAsDate, stock_df$Sma20,col="green", type="l", ylab= "Sma20")
    lines(stock_df$DateAsDate, stock_df$Sma50,col="blue", type="l", ylab= "Sma50")
    lines(stock_df$DateAsDate, stock_df$Sma100,col="orange", type="l", ylab= "Sma100")
    lines(stock_df$DateAsDate, stock_df$Sma200,col="red", type="l", ylab= "Sma200")
    #inset=c(-0.2,0) to have the legend outside of the plot
    legend( x= "topleft",  legend=c("Close", "Sma20","Sma50", "Sma100", "Sma200"),
            col=c("black", "green", "blue", "orange", "red"), lty=1, cex=0.8, horiz = TRUE)

  }
}
```

```r
#This function plots the different moving averages
plot_exponential_moving_averages = function (stock_df, from_date, to_date){
  #reset display to one graph per window
  par(mfrow=c(1,1))

  step_by = "6 mon"
  date_format = "%d-%m-%Y"

  from_date = get_from_date(stock_df, from_date);
  to_date = get_to_date(stock_df, to_date);

  #Only plot when dates are consistent
  if(to_date> from_date){
    #Reduce the data range to the required time range
    stock_df = stock_df[stock_df$DateAsDate >=from_date & stock_df$DateAsDate <=to_date, ]

    #Get the highest y value for the plot
    max_height = max(stock_df$Ema20,stock_df$Ema50,stock_df$Ema100,stock_df$Ema200, na.rm = TRUE)
    min_height = min(stock_df$Ema20,stock_df$Ema50,stock_df$Ema100,stock_df$Ema200, na.rm = TRUE)
    #Plot
    plot( stock_df$DateAsDate,
          stock_df$Close,
          col="black",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Stock Price ($)",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Close Price Vs Exponential Moving Average (EMA) Crossovers\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)
    lines(stock_df$DateAsDate, stock_df$Ema20,col="green", type="l", ylab= "Ema20")
    lines(stock_df$DateAsDate, stock_df$Ema50,col="blue", type="l", ylab= "Ema50")
    lines(stock_df$DateAsDate, stock_df$Ema100,col="orange", type="l", ylab= "Ema100")
    lines(stock_df$DateAsDate, stock_df$Ema200,col="red", type="l", ylab= "Ema200")
    #inset=c(-0.2,0) to have the legend outside of the plot
    legend( x= "topleft",  legend=c("Close", "Ema20","Ema50", "Ema100", "Ema200"),
            col=c("black", "green", "blue", "orange", "red"), lty=1, cex=0.8, horiz = TRUE)

  }
}
```

```
#This function plots the relative strength index (rsi)
plot_relative_strength_index = function (stock_df, from_date, to_date){
  step_by = "3 mon"
  date_format = "%d-%m-%Y"

  from_date = get_from_date(stock_df, from_date);
  to_date = get_to_date(stock_df, to_date);

  #Display the graphs (one under the other), i.e. in a 2 rows/1 col grid system
  par(mfrow=2:1)

  #Only plot when dates are consistent
  if(to_date> from_date){
    #Reduce the data range to the required time range
    stock_df = stock_df[stock_df$DateAsDate >=from_date & stock_df$DateAsDate <=to_date, ]

    #Get the highest y value for the plot
    max_height = max(stock_df$Close, na.rm = TRUE)
    min_height = min(stock_df$Close, na.rm = TRUE)
    #Plot
    plot( stock_df$DateAsDate,
          stock_df$Close,
          col="black",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Stock Price ($)",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Close Price\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)

    max_height = max(stock_df$Rsi, na.rm = TRUE)
    min_height = min(stock_df$Rsi, na.rm = TRUE)
    plot( stock_df$DateAsDate,
          stock_df$Rsi,
          col="green",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "RSI",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"RSI (14days)\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)

    #Two vertical lines, at ordinate = 30 and 70, to show the RSI signal trigger
    abline(h=70)
    abline(h=30)

    #reset display to one graph per window
    par(mfrow=c(1,1))
  }
}
```

```r
#This function plots the relative strength index (rsi)
plot_prices_volume = function (stock_df, from_date, to_date){
  step_by = "3 mon"
  date_format = "%d-%m-%Y"

  from_date = get_from_date(stock_df, from_date);
  to_date = get_to_date(stock_df, to_date);

  #Display the graphs (one under the other), i.e. in a 2 rows/1 col grid system
  par(mfrow=2:1)

  #Only plot when dates are consistent
  if(to_date> from_date){
    #Reduce the data range to the required time range
    stock_df = stock_df[stock_df$DateAsDate >=from_date & stock_df$DateAsDate <=to_date, ]

    #Get the highest y value for the plot
    max_height = max(stock_df$Close, na.rm = TRUE)
    min_height = min(stock_df$Close, na.rm = TRUE)
    #Plot
    plot( stock_df$DateAsDate,
          stock_df$Close,
          col="black",
          type="l",
          lwd=1.5,
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Stock Price ($)",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Close Price\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)
    lines(stock_df$DateAsDate, stock_df$Close_price_1day_lag,col="orange", type="l", ylab= "1Day Price Lag")
    lines(stock_df$DateAsDate, stock_df$Close_price_4day_lag,col="blue", type="l", ylab= "4Day Price Lag")
    #inset=c(-0.2,0) to have the legend outside of the plot
    legend( x= "topleft",  legend=c("Close", "Close 1-Day Lag", "Close 4-Day Lag"),
    col=c("black", "orange", "blue"), lty=1, cex=0.8, horiz = TRUE)

    max_height = max(stock_df$Volume, na.rm = TRUE)
    min_height = min(stock_df$Volume, na.rm = TRUE)
    plot( stock_df$DateAsDate,
          stock_df$Volume,
          col="green",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Volume",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Volume\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
    axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)

    #reset display to one graph per window
    par(mfrow=c(1,1))
  }
}
```

```
#This function plots the relative strength index (rsi)
plot_log_returns = function (stock_df, from_date, to_date){
  #reset display to one graph per window
  par(mfrow=c(1,1))

  step_by = "3 mon"
  date_format = "%d-%m-%Y"

  from_date = get_from_date(stock_df, from_date);
  to_date = get_to_date(stock_df, to_date);

  #Only plot when dates are consistent
  if(to_date> from_date){
    #Reduce the data range to the required time range
    stock_df = stock_df[stock_df$DateAsDate >=from_date & stock_df$DateAsDate <=to_date, ]

    #Get the highest y value for the plot
    max_height = max(stock_df$Log_returns, na.rm = TRUE)
    min_height = min(stock_df$Log_returns, na.rm = TRUE)
    #Plot
    plot( stock_df$DateAsDate,
          stock_df$Log_returns,
          col="black",
          type="l",
          xlab= "",
          xlim=c(from_date,to_date),
          ylab= "Log Returns ($)",
          ylim=c(min_height,max_height),
          main=paste(stock_name,"Log Returns (from Close prices)\n", from_date, "-", to_date, sep=" "),
          las=2,
          xaxt='n') #this removes the default x-axis label
      axis.Date(1, at=seq(from_date,to_date, by=step_by), format=date_format, las=2)

  }
}
```

Code snippet relating to the generation of the Times Series graphs.