# Stock Price Prediction Project

*Egar Garcia*

*4/17/2019*

## Contents



# 1 Introduction

## 1.1 Overview

A stock market, also called equity market or share market, is a network of economic transactions, where the stocks of public companies can be bought and sold. The equity market offers companies the ability to access capital in exchange of a portion in the ownership of the company for interested outside parties.

In the stock market, other financial securities like exchange traded funds (ETF), corporate bonds and derivatives based on stocks, commodities, currencies, bonds, etc. can be traded. However, for purpose of this project only the exchange of stocks will be considered.

It is common to use stock market and stock exchange interchangeably, but the stock market is a general super set of the stock exchange. If someone is trading in the stock market, it means that it buys and sells stock/shares/equity on one (or more) of the stock exchange(s) that are part of the overall stock market.

The stock market offers an opportunity to investors to increase their income without the high risk of entering into their own businesses with high overheads and start up costs. On the other hand, selling stocks helps companies themselves to expand exponentially, when a company's shares are purchased it is generally associated with the increased in the company's worth. Therefore, trading on the stock market can be a win-win for both investor and owner.

The stock exchange that are leading the U.S. include the New York Stock Exchange (NYSE), Nasdaq, BATS and Chicago Board Options Exchange (CBOE). The Dow Jones Industrial Average (DJIA) is a price-weighted average of 30 significant stocks traded on the NYSE and the Nasdaq, it is the most closely watched market indicator in the world and it is generally perceived to mirror the state of American economy.

Projecting how the stock market will perform is a very difficult thing to do, there are so many factors involved in the prediction, some of them emotional or irrational, which combined with the prices volatility make difficult to predict with a high degree of accuracy. Abundant information is available in the form of historical stock prices, which make this problem suitable for the use of machine learning algorithms.

Investment firms, hedge funds and individuals have been using financial models to better understand the market behavior and attempt to make projections in order to make profitable investments and trades.

## 1.2 Objective

The purpose of this project is to build stock price predictors. More specifically, the problem is to predict the closing prices of the trading days existing in a queried date range for a given company's stock. For the scope of this project only the companies included in the Dow Jones Industrial Average are considered.

To address the problem, different machine learning methods are used, they take historical stock data for a particular company over a certain date range (in the past) as training input, and output projected estimates for a given queried date range (in the future).

## 1.3 Dataset

There are multiple options to obtain the historical stock records needed to feed the machine learning algorithms, a popular source (and the one used in this project) is an API provided by IEX Group Inc. (https://iextrading.com), which provides access to stock historical records to developers and engineers for free.

The API provided by IEX (which documentation can be found at https://iextrading.com/developer/docs/#chart) allows to retrieve historical stock price information for a maximum of 5 years back to the current date. The API can provide historical records for several companies, but for this project only the records for the ones in the Dow Jones are considered.

Each one of the historical records corresponds to the information of a trading day for a specific company. An important aspect to notice is that the market is closed on weekends and some defined holidays (for which there are no records). The historic stock price records contain the following columns:

- `date`: Trading day
- `open`: Opening price
- `high`: Highest price
- `low`: Lower price
- `close`: Closing price
- `volume`: Number of shares traded
- `unadjustedVolume`: Number of shares traded also considering companies adjustments
- `change`: Change of closing price relative to the day before

- `changePercent`: Change in percent value
- `vwap`: Volume Weighted Average Price
- `label`: Formatted version of the date
- `changeOverTime`: Metric considered to measure the changes bases on weighted dates

### 1.4 Key steps

1. Building the mechanisms to retrieve the dataset into the local machine's storage, and to keep it updated.
2. Performing an analysis and exploration of the dataset, aiming to select only the features that are relevant to perform predictions.
3. Building models that represent the machine learning methods used in this project to address the problem.
4. Evaluation of the different models with common training and test sets, and doing the comparison based on the RMSE against the ground truth.
5. Analyzing the predicting performance of each one of the models, and discussing the results.

## 2 Data analysis

### 2.1 Data exploration

The API provided by IEX retrieves the data in JSON format, which at the end contains records, where each record corresponds to the information of a trading day for a specific company, a company is identified by a ticker symbol (also known as stock symbol).

For example to retrieve the historical stock prices for the ticker symbol "MSFT" (Microsoft Corporation) of the last 5 trading days, the call to the API would be https://api.iextrading.com/1.0/stock/msft/chart/5d. The following code is an example of how to retrieve those historical records and how they look like:

```
as.data.frame(
  fromJSON(
    content(GET('https://api.iextrading.com/1.0/stock/msft/chart/5d'), 'text'),
    flatten = TRUE))
```

```
##         date   open   high      low  close   volume unadjustedVolume
## 1 2019-04-12 120.64 120.98 120.3700 120.95 19745143         19745143
## 2 2019-04-15 120.94 121.58 120.5700 121.05 15792572         15792572
## 3 2019-04-16 121.64 121.65 120.1000 120.77 14071787         14071787
## 4 2019-04-17 121.24 121.85 120.5400 121.77 19300938         19300938
## 5 2019-04-18 122.19 123.52 121.3018 123.37 27990998         27990998
##   change changePercent     vwap  label changeOverTime
## 1   0.62         0.515 120.6888 Apr 12   0.0000000000
## 2   0.10         0.083 121.0236 Apr 15   0.0008267879
## 3  -0.28        -0.231 120.9954 Apr 16  -0.0014882183
## 4   1.00         0.828 121.2869 Apr 17   0.0067796610
## 5   1.60         1.314 122.6257 Apr 18   0.0200082679
```

As in can be seen, each historic stock price record contains the columns: `date`, `open`, `high`, `low`, `close`, `volume`, `unadjustedVolume`, `change`, `changePercent`, `vwap`, `label` and `changeOverTime`. The purpose of this project is to predict the future values of the closing price, which corresponds to the column `close`.

To make predictions is necessary to chose a subset of columns which values can be known a priory and used for the prediction, unfortunately all the column values except `date` and `label` (which finally is a variant of `date`) are unknown before the occurrence of the respective trading day. That means that the only data that

can be used to predict the future closing prices are the past closing prices and the company itself, and then the only relevant columns are `date` and `close`.

Per year there are from 251 to 253 trading days, since the market is closed on weekends and some holidays:

- 2014 had 252 trading days
- 2015 had 252 trading days
- 2016 had 252 trading days
- 2017 had 251 trading days
- 2018 had 251 trading days
- 2019 is expected to have 252 trading days
- 2020 is expected to have 253 trading days

That means that each company in the Dow Jones would have from 251 to 253 historical records per year, except for "DOW" (Dow Inc.) since its records started to appear on 2019-04-02.

For example, in a 5 years period from 2014-04-14 to 2019-04-12, there are 1259 historical records per company, with the exception of "DOW" which has 18. If the historical records of the actual Dow Jones Industrial Average (which ticker symbol is "DIA") are also included in the dataset they would add another 1259 records. Then for this 5 year period there are 37788 historical stock price records in total ($1259 \times 29 + 18 + 1259 = 37788$).

## 2.2 Dataset retrieval

The strategy to retrieve the dataset is to get the historical stock price records for each one of the 30 stocks in the Dow Jones (the list of stocks is retrieved from the file `dow_jones_stocks.csv`), plus the ones for the Dow Jones Industrial Average, and then combining them in a single dataframe.

For the historical stock price records only the columns `date` and `close` are picked, and one new column `symbol` is added to indicate the ticker symbol of the company which the record belongs to.

The following code is the implementation of the mechanism to retrieve the dataset from the IEX API, the main function is `get_dow_jones_dataframe`.

```
#' Loading the set of ticker symbols of the companies contemplated in
#' the  Dow Jones Industrial Average.
dow_jones_stocks <- read.csv('dow_jones_stocks.csv', stringsAsFactors = FALSE)

#' This is the symbol used by the actual average, i.e. the Dow Jones Industrial Average.
djia_symbol <- 'DIA'

#' This is the template to create the URL to extract historical stock prices
#' from the IEX API.
iex_api_url_template <- 'https://api.iextrading.com/1.0/stock/%s/chart/%s'

#' Retrieves the historic prices for a particuler stock from the data source.
#'
#' @param ticker_symbol The ticker symbol or symbols to filter the data.
#' @param hist_period   The period to retrieve historical records,
#'    p.e '5y' for 5 years, '1y' for 1 year, '1m' for 1 month, etc.
#' @return The dataframe containing the historic prices.
get_historical_records <- function(ticker_symbol, hist_period = '5y') {
  # Getting the historic records from IEX into a dataframe
  call_url <- sprintf(iex_api_url_template, ticker_symbol, hist_period)
  resp <- GET(call_url)
  historical_prices <- as.data.frame(fromJSON(content(resp, 'text'), flatten = TRUE))

  # Adding the ticker symbol as a column
```

```
  historical_prices['symbol'] = ticker_symbol
  # Converting the date column to a Date type
  historical_prices$date <- as.Date(historical_prices$date, format='%Y-%m-%d')

  # Only chosing the columns: symbol, date, close
  historical_prices %>% select(symbol, date, close)
}

#' Gets a dataframe containing historic prices for stocks in
#' the Dow Jones Industrial Average.
#'
#' @param hist_period The period to retrieve historical records,
#'    p.e '5y' for 5 years, '1y' for 1 year, '1m' for 1 month, etc.
#' @return The dataframe containing the historic prices.
get_dow_jones_dataframe <- function(hist_period = '5y') {
  # Getting the historic records of the Dow Jones Industrial Average
  historical_prices <- get_historical_records(djia_symbol, hist_period = hist_period)

  # Retrieves the historic records for each one of the ticker symbols in the
  # Dow Jones Industrial Average
  for (tickers_symbol in dow_jones_stocks$symbol) {
    historical_prices <- rbind(historical_prices,
                               get_historical_records(tickers_symbol,
                                                      hist_period = hist_period))
  }

  historical_prices
}
```

The IEX's API only allows to get historical records for a maximum of the previous 5 years, however the management of the dataset does not have to be limited by this constraint, since an old dataset can be updated just by adding the missing records. In this way a mechanism can be implemented to keep data beyond 5 years old and keeping a dataset updated.

The following is the implementation of the function `update_dow_jones_dataframe` which is in charge of updating a dataset by adding the missing recent records.

```
#' Updates a dataframe containing historic prices for stocks in
#' the Dow Jones Industrial Average,
#' by retrieving the most recent records from the information source.
#'
#' @param historical_prices The dataframe containing stock price historical records.
#' @return The dataframe containing the historic prices.
update_dow_jones_dataframe <- function(historical_prices) {
  # Getting the amount of days that need to be updated
  last_recorded_day = max(historical_prices$date)
  today = Sys.Date()
  days_to_update = difftime(today, last_recorded_day, units="days")

  # Deciding the historic period to request the source according
  # to the days that need to be updated
  hist_period = '5y'
  if (days_to_update < 1) {
    return(historical_prices)
  } else if (days_to_update < 28) {
```

```
    hist_period = '1m'
  } else if (days_to_update < 365) {
    hist_period = '1y'
  }

  # Getting the data frame containing the missing records
  last_historical_prices <- get_dow_jones_dataframe(hist_period = hist_period)

  # Adding the missing records and removing duplicates
  historical_prices <- rbind(historical_prices, last_historical_prices)
  historical_prices[!duplicated(historical_prices[c('symbol', 'date')]),]
}
```

The following is a helper function to get a subset of the dataset by filtering by a given ticker symbol and/or a date range.

```
#' Gets a dataframe containing a subset of the records of the current dataset,
#' which is obtained by filtering by a ticker symbol and/or a date range.
#'
#' @param ticker_symbol The ticker symbol to filter the data.
#' @param from_date The minimum date to appear in the records of the subset.
#' @param to_date The maximum date to appear in the records of the subset.
#' @return The dataframe with the subset resulted of filtering the dataset.
filter_historical_records <- function(historical_prices,
                                      ticker_symbol = NULL,
                                      start_date = NULL, end_date = NULL) {
  df <- historical_prices

  # Filtering by ticker symbol if exists
  if (!is.null(ticker_symbol)) {
    df <- df %>% filter(symbol == ticker_symbol)
  }
  # Filtering by start date if exists
  if (!is.null(start_date)) {
    df <- df %>% filter(date >= start_date)
  }
  # Filtering by end date if exists
  if (!is.null(end_date)) {
    df <- df %>% filter(date <= end_date)
  }

  df
}
```

The following variable `dow_jones_dataframe_filename` specifies the file's name `dow_jones_dataframe.Rda`, which is where the dataset is being stored.

```
#' The file's name where the dataset is stored.
dow_jones_dataframe_filename <- 'dow_jones_dataframe.Rda'
```

### Examples of data retrieval

The following is an example to create a brand new dataset, using the default historical period of 5 years back from the current date. Then the dataset is saved to the file `dow_jones_dataframe.Rda`.

```
dow_jones_historical_records <- get_dow_jones_dataframe()
save(dow_jones_historical_records, file = dow_jones_dataframe_filename)
```

In the following code the dataset is loaded from the file `dow_jones_dataframe.Rda` into the variable `dow_jones_historical_records` (assuming that this was the one previously saved, p.e. by running `save(dow_jones_historical_records, file = dow_jones_dataframe_filename)`), then the dataset is updated by adding the missing recent records, and finally the dataset is saved again in the same file.

```
load(file = dow_jones_dataframe_filename)
dow_jones_historical_records <-
  update_dow_jones_dataframe(dow_jones_historical_records)
save(dow_jones_historical_records, file = dow_jones_dataframe_filename)
```

The following is an example for just loading and existing dataset from the file `dow_jones_dataframe.Rda` into the variable `dow_jones_historical_records` (assuming that this variable was previously saved, p.e. by running `save(dow_jones_historical_records, file = dow_jones_dataframe_filename)`).

```
load(file = dow_jones_dataframe_filename)
```

## 2.3 Data visualization

The following code generates a visualization in figure 1 of the historical closing prices for the 30 stocks of the companies in the Dow Jones, displaying the five years' period up to April 12th, 2019. It displays the prices along the time for all the Dow Jones companies, in addition includes a listing of their respective ticker symbols and names.

```
dow_jones_historical_records %>%
  filter(symbol != djia_symbol) %>%
  left_join(dow_jones_stocks, by = 'symbol') %>%
  select(date, close, symbol, company) %>%
  ggplot(aes(x = date, y = close, group = symbol,
             color = sprintf('%s (%s)', symbol, company))) +
  geom_line(size = 0.3) +
  labs(colour = '') +
  theme(legend.position = 'bottom') +
  geom_dl(aes(label = symbol), method = 'angled.boxes')
```

The following code generates the visualization in figure 2 which displays the historical prices of the same stocks, but this time grouped by industry type.

```
dow_jones_historical_records %>%
  filter(symbol != djia_symbol) %>%
  left_join(dow_jones_stocks, by = 'symbol') %>%
  select(date, close, symbol, industry, company) %>%
  filter(symbol != djia_symbol) %>%
  ggplot(aes(x = date, y = close)) +
  geom_line(aes(group = symbol, color = symbol)) +
  geom_dl(aes(label = symbol, color = symbol), method = 'smart.grid') +
  facet_wrap(~industry, ncol = 4) +
  theme(legend.position = 'none')
```

The following code generates the visualization in figure 3, which contrasts the behavior of all Dow Jones stocks with the actual Dow Jones Industrial Average index (highlighted in black).

```
dow_jones_historical_records %>%
  mutate(index = ifelse(symbol == djia_symbol, 'DJIA', 'Stock in Dow Jones'),
```

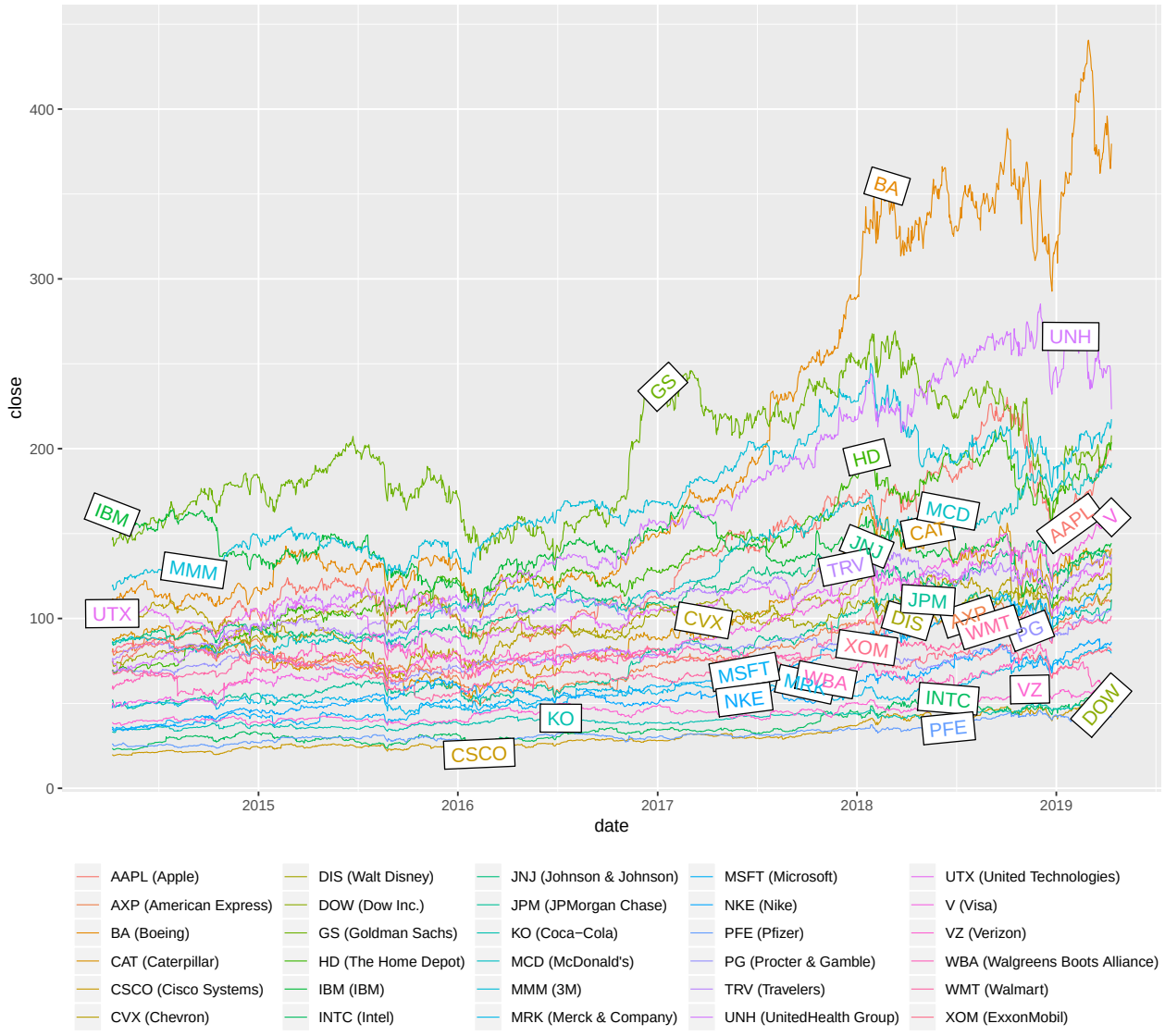


Figure 1: Dow Jones 5 year historic prices

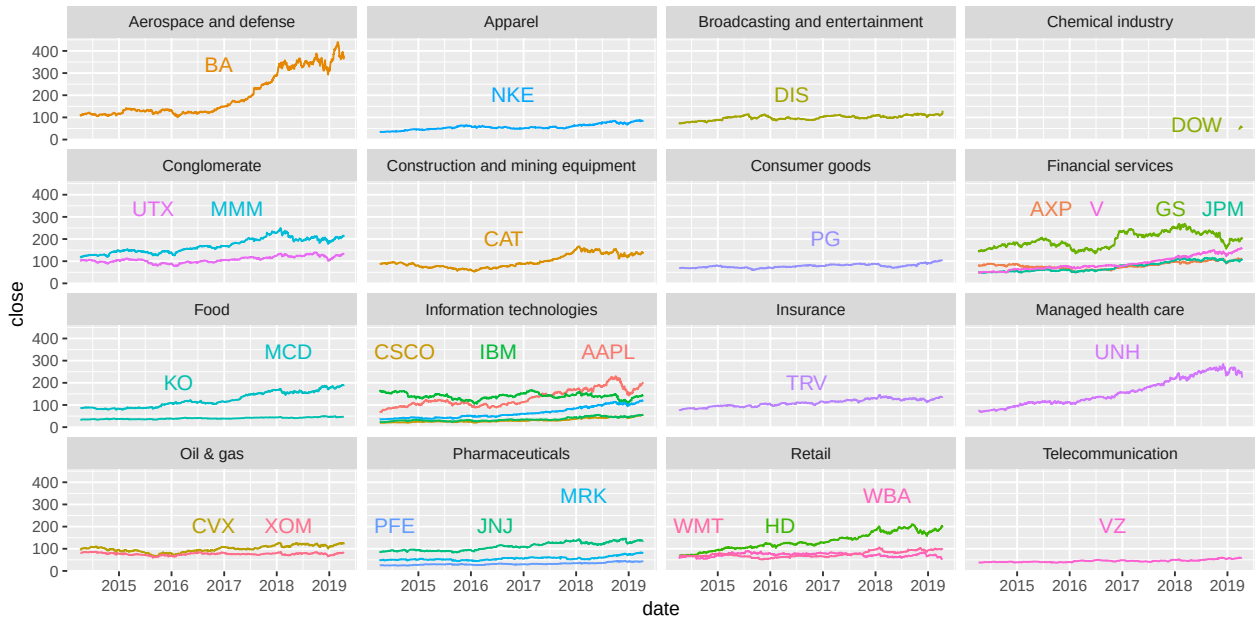


Figure 2: Dow Jones 5 year historic prices grouped per industry

```
        # Hack alert: Prefixing the DJIA with 'zzz_' in order be plotted at the end
        symbol = ifelse(symbol == djia_symbol, paste('zzz_', djia_symbol), symbol)) %>%
  ggplot(aes(x = date, y = close, group = symbol, color = index, size = index)) +
  geom_line() +
  scale_color_manual(values = c('black', 'gray')) +
  scale_size_manual(values = c(0.5, 0.25)) +
  labs(colour = '') +
  theme(legend.position = 'bottom') +
  guides(size = FALSE)
```

The DJIA is calculated with the stock prices of 30 selected public large companies, to calculate the DJIA the prices are added and then divided by the Dow divisor, which is constantly modified, then it is not surprising that most of these stocks seems to behave in a similar way to the DJIA.

In figure 4 is presented a visualization generated for the following code, which shows the correlation of each one of the stocks in the Dow Jones against each other and against the actual DJIA (identified with the ticker symbol "DIA"). It can be observed that in most of the cases there is a high correlation, observed by a dominance of the red color in the matrix, which is the color used to indicate a high correlation, i.e. close to 1.

```
dow_jones_historical_records %>%
  # Hack alert: Prefixing the DJIA with ' ' in order to place it firts
  mutate(symbol = ifelse(symbol != djia_symbol, symbol, paste(' ', djia_symbol))) %>%
  spread(symbol, close) %>%
  select(-date) %>%
  cor(method = 'pearson', use = 'complete.obs') %>%
  ggcorrplot(lab = TRUE)
```

To illustrate the interpretation of the correlations, let's take the following stocks in regards with their correlation respect to the DJIA, the respective visualization is generated by the code that follows the list:

- "CAT" the stock with the highest correlation (correlation of 97%). As it can be seen in figure 5 its behavior looks to be very similar to the DJIA, i.e. up-trends, down-trends, big-rises and big-drops seem to match.

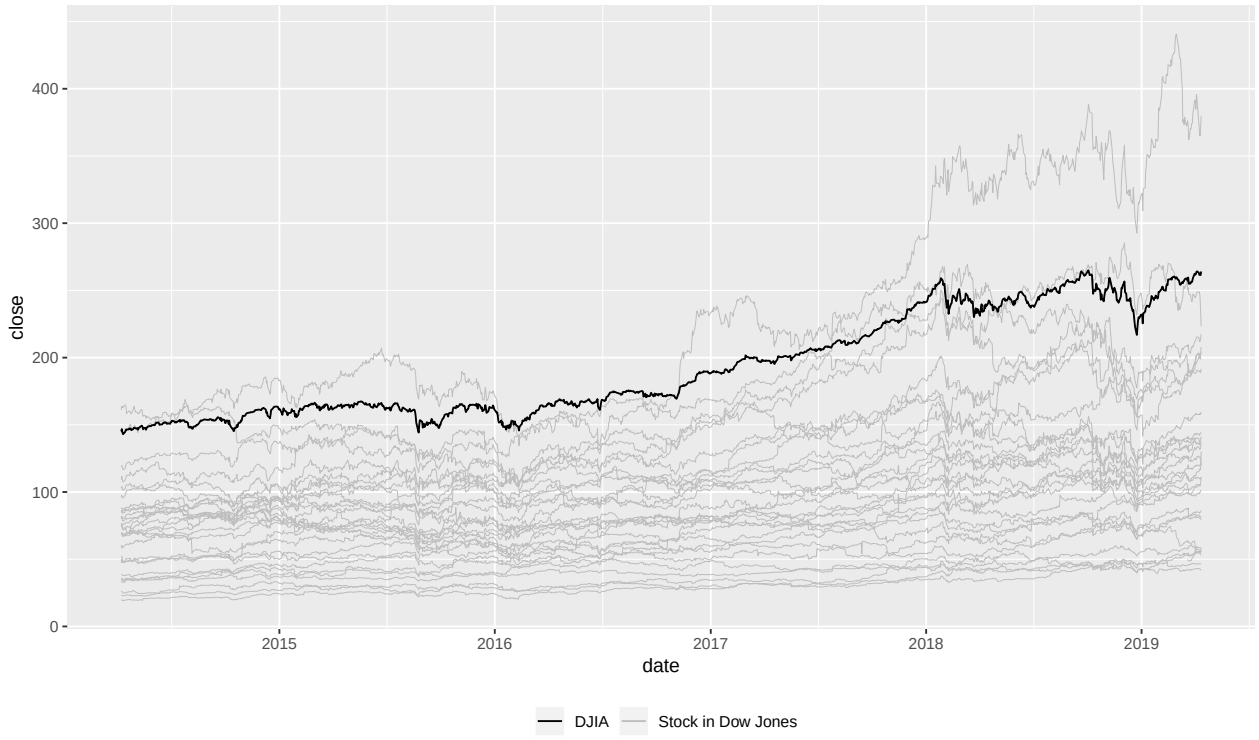


Figure 3: Dow Jones stoks in contrast with the Dow Jones Industrial Average index

- "WBA" the one with the highest inverse correlation (correlation of -76%). As it can be seen in figure 5, its behavior looks to be somehow contrary to the DJIA, p.e. from early 2016 the DJIA has a general tendency to rise, but "WBA" has a general tendency to drop.
- "UNH" the closest to not having correlation at all (correlation of -19%). As it can be seen in figure 5, looks like "UNH" in general presents its own tendency.

```
dow_jones_historical_records %>%
  filter(symbol == djia_symbol | symbol == 'CAT' | symbol == 'WBA' | symbol == 'UNH') %>%
  mutate(index = ifelse(symbol == djia_symbol, 'DJIA', 'Dow Jones stock')) %>%
  ggplot(aes(x = date, y = close, group = symbol, color = symbol, size = index)) +
  geom_line() +
  scale_color_manual(values = c(DIA='black', CAT='red', WBA='blue', UNH='gray')) +
  scale_size_manual(values = c(0.5, 0.25)) +
  labs(colour = '') +
  theme(legend.position = 'none') +
  geom_dl(aes(label = symbol, color = symbol), method = 'last.polygons')
```

When looking at the stocks one at a time, it seems like the stock prices fluctuates a lot and a clear pattern is not perceived, at least not at a human comprehensible level. This is understandable since the stock prices depend on a lot of different factors like the company's financial health, economic supply-demand and even involving human emotions like trust, euphoria or panic.

At a macro level it can be observed that they are common events that seem to affect the stock prices as a whole, like a rise and sudden fall of prices around the beginning of 2018, or a generalized drop on the stock prices at the end of 2018. However these kind of events also do not seem to have a (humanly) comprehensible pattern either, here is where machine learning methods can provide a lot help to face the forecasting problem.

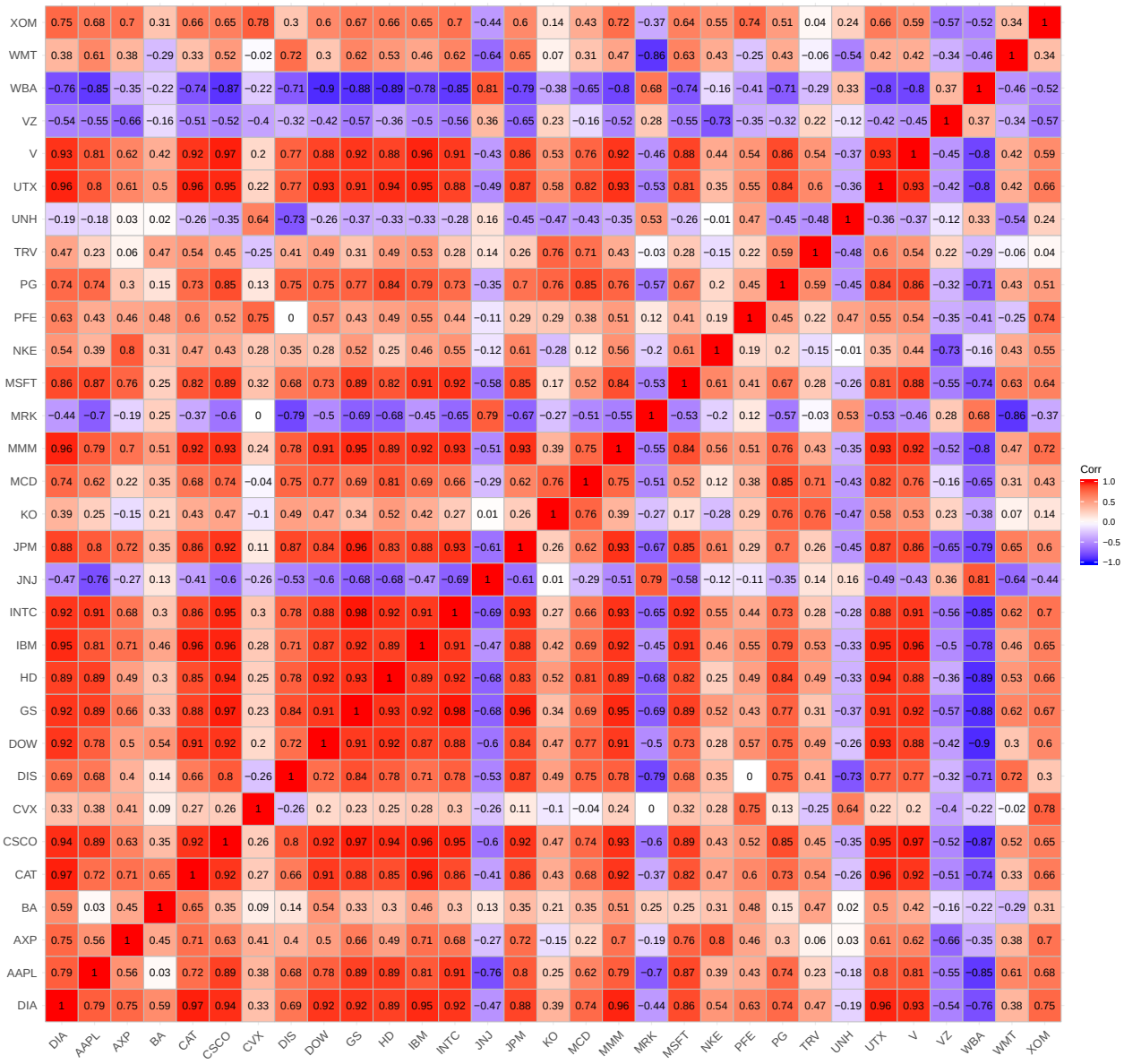


Figure 4: Correlation among Dow Jones stocks

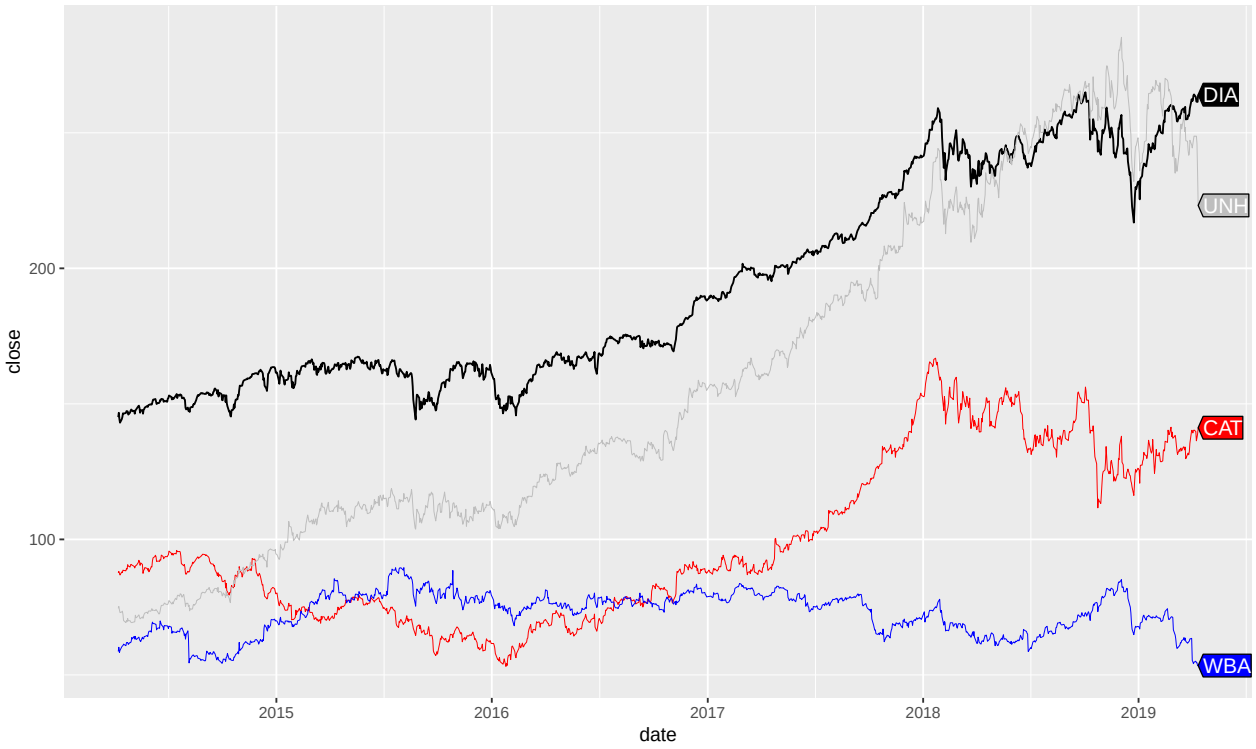


Figure 5: Example of some stocks highly, inverse and non correlated in comparison to the DJIA

# 3 Methods

A normal machine learning dataset is a collection of observations where time does not necessarily play a main role, predictions are made for new/unknown data (which might be considered as predicting the future), however all the prior observations are pretty much treated equally, and the order or the observations is not taken in consideration, even is a good practice to shuffle the observations to perform the training.

A time series dataset is different, because they have an explicit dependence of the order between observations, thus the time plays a main role. For these datasets the time is both a constraint and also a structure that provides additional information. During the exploration of the dataset it was realized that the only data that we can know a priory and use to make the predictions are the dates, then this characteristic makes the problem to forecast the stock closing prices to be a time series forecasting problem.

Even though the problem of forecasting stocks can be seen as a particular case of time series forecasting, in the "Classical ML methods" section some classical (or general purpose) machine learning methods are used to attempt giving a solution to the stock forecaster problem. Then in the "Time series based methods" specific methods to forecast time series are applied to address the problem. Both kind of methods are going to be evaluated and analyzed in this report.

## 3.1 Evaluation methodology

To evaluate the different (machine learning) methods tested in this project, an object-oriented approach is used, each method is going to be represented by a model which would be generated through the construction of an object.

To generate a model, the constructor function receives a training set consisting of stock historical records in a given date range (in the past), this is used to fit the model and construct the model's object. The model's

object includes a `predict` function used to perform a prediction for a given date range (which could be the one for testing, validation, or production ideally in the future).

The evaluation consist in randomly select a date range for training, it should contain 750 trading days (which is around 3 years of historical records) and being old enough to leave at least 120 trading days in the dataset following the training end date to use for validation. Also it is selected a random ticker symbol in the Dow Jones, except 'DOW', because it doesn't have enough historical records to do an evaluation with 750 trading days back.

The performance of prediction is based on the RMSE, which is calculated against the ground truth in the test set for the following 1, 5, 10, 20, 40, 60 and 120 trading days after the training end date.

## 3.2 Preparation

### 3.2.1 Trading days retrieval

To perform predictions in a date range, it is necessary to identify the trading days (which are the days when the stock market is open) that exist in that given range, because only on them it makes sense to perform predictions.

The following code loads from the file `market_holidays.csv` the list of market holidays, i.e. the days that the market is closed without considering weekends.

```
#' Market holidays loaded from file 'market_holidays.csv'
market_holidays <- read.csv('market_holidays.csv') %>%
  mutate(date = as.Date(date, format='%Y-%m-%d'))
```

The following is the implementation of a function to get the trading days existing in a date range. The strategy is to get the sequence of all the existing days in that range and then remove weekends and market holidays .

```
#' Getting the trading days existing in a date range.
#'
#' @param from_date The starting date of the range.
#' @param datetime to_date The ending date of the range.
#' @return A list of the trading days in the specified date range.
get_trading_days_in_range <- function(from_date, to_date) {
  # Checking if the data range is valid
  if (from_date > to_date) {
    stop('Invalid date range')
  }

  data.frame('date' = seq(from_date, to_date, 'days')) %>% # Sequene of dates in the range
    filter(!(wday(date) %in% c(1, 7))) %>% # Filtering by weekdays (Mon to Fri)
    anti_join(market_holidays, by = 'date')  # Removing holidays
}
```

The following function is used to get a determined number of trading days after a given date, which is very useful for testing and measuring the performance of the different methods tested in this project, by performing the tests for different given numbers of days ahead after training.

```
#' Getting a specific number of trading days after a given date.
#'
#' @param after_date The date after which training days are going to be retrieved.
#' @param num_trading_days The number of training days to get.
#' @return A list containing the trading days.
get_trading_days_after <- function(after_date, no_trading_days) {
```

```
  trading_days <- data.frame(date = as.Date(character()))
  counter <- 0
  current_date <- after_date

  # Getting each day after the given date till getting
  # the requested number of days
  while (counter < no_trading_days) {
    current_date <- current_date + ddays(1)

    # If not in a weekend nor a holiday adding the date to the set of trading days
    if (!(wday(current_date) %in% c(1, 7) | current_date %in% market_holidays$date)) {
      counter <- counter + 1
      trading_days[counter,] <- c(current_date)
    }
  }

  trading_days
}
```

### 3.2.2 Generating training and test sets

The following function generates a list containing the training and test sets from a given dataset, they are generated by filtering the records by a specific ticker symbol, then for the training set the records in the given date range are chosen, and finally the test set is created by choosing the historic records corresponding to the following given number of trading days after the training's end date.

```
#' Extracts training and test sets from a given data set.
#' The training set is created with records in the given date range
#' for the training. The test set is created with records
#' from the day after the training day and to the date in which
#' the number of requested test days is covered with trading days.
#'
#' @param historical_prices The dataset of stock historical prices used
#'    to extract the training and test sets from.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param start_training The minimum date for the records used in the training set.
#' @param end_training The maximum date for the records used in the training set.
#' @param num_test_days The number of trading days after end_training
#'    used to create the test set.
#' @return A list containing the ticker symbol, training set and test set.
get_train_and_test_sets <- function(
  historical_prices, ticker_symbol, start_training, end_training, num_test_days) {

  # Filtering the data set to only contains the records related to the
  # given ticker symbol
  df <- historical_prices %>%
    filter(symbol == ticker_symbol)

  # Getting the training set
  training_set <- df %>%
    filter(date >= start_training & date <= end_training)

  # Getting the specific dates for test which size is 'test_days'
```

```
  test_days <- get_trading_days_after(end_training, num_test_days)

  # Getting the test set
  test_set <- df %>%
    filter(date >= min(test_days$date) & date <= max(test_days$date))

  list(symbol = ticker_symbol, training = training_set, test = test_set)
}
```

The following is the code to do the random selection of the ticker symbol and date range used create the training and test set, which are used to perform the evaluation for the machine learning methods studied in this project.

```
# Choosing a random ticker symbol, except 'DOW' since it doesn't have enough data
eval_ticker_symbol <- sample(filter(dow_jones_stocks, symbol != 'DOW')$symbol , 1)
# Getting the available dates for that ticker symbol
random_dates <- filter(dow_jones_historical_records, symbol == eval_ticker_symbol)
# Getting a random training end date such as there are
# at least 750 historical records up to it (for training) and
# at least 120 after (for testing )
random_idx_end_training <- sample(750:(nrow(random_dates) - 120), 1)
eval_training_end <- random_dates[random_idx_end_training, 'date']
# Calculating the training start date such as there are 750 records for training set
eval_training_start <- random_dates[(random_idx_end_training - 750 + 1), 'date']

# Extracting train and test sets
eval_sets <- get_train_and_test_sets(dow_jones_historical_records,
                                     eval_ticker_symbol,
                                     eval_training_start,
                                     eval_training_end,
                                     120)

# Getting the test date range
eval_test_start <- min(eval_sets$test$date)
eval_test_end <- max(eval_sets$test$date)

# Cleaning intermediate variables
rm(random_dates, random_idx_end_training)
```

The result of the random selection used for evaluation purposes can be seen by running the following code.

```
# Printing the values obtained by the random selection
print(sprintf('Symbol: %s, Training: [%s, %s], Test: [%s, %s]',
              eval_ticker_symbol, eval_training_start, eval_training_end,
              eval_test_start, eval_test_end))
```

```
## [1] "Symbol: MSFT, Training: [2014-07-24, 2017-07-14], Test: [2017-07-17, 2018-01-04]"
```

A visualization of the generated training and test sets, which are being used to perform the evaluation of the methods/models, can be observed in figure 6 and it is plotted by the next code.

```
ggplot() +
  geom_line(data = eval_sets$training, aes(x = date, y = close, color = 'Training Set')) +
  geom_line(data = eval_sets$test, aes(x = date, y = close, color = 'Test Set')) +
  scale_color_manual(values = c('Training Set' = 'blue', 'Test Set' = 'green')) +
  labs(color = '') +
```

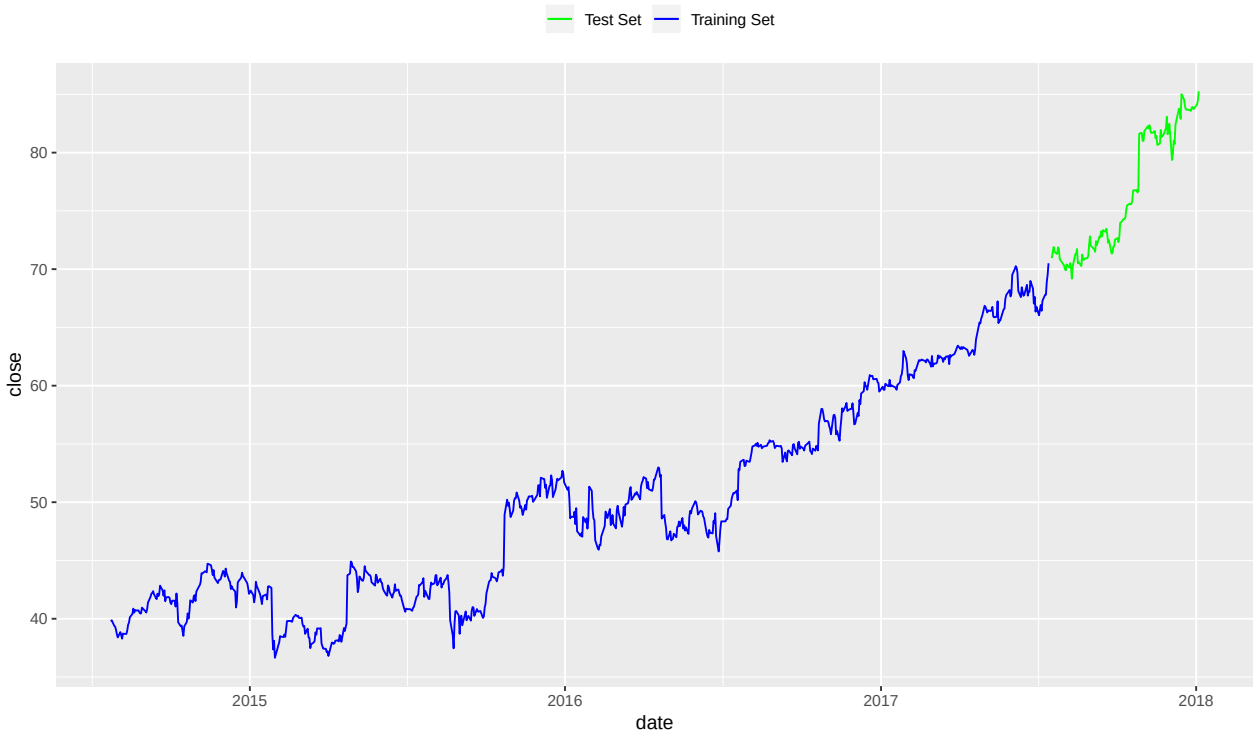


Figure 6: Generated training and test sets for evaluation

```
  theme(legend.position = 'top')
```

### 3.2.3 Supporting the presentation of evaluation results

In here some useful functions are defined to support the visualizations (included in this report) of the evaluation results, which are obtained after evaluate each one of the methods/models studied in this project.

The following function plots the comparison of the predictions given for a particular method against the ground truth in the training and test sets.

```
#' Plots the predictions in comparison with the training and validation sets,
#' in order to provide a visualization of a model's prediction performance.
#'
#' @param sets A list containing the training and test sets.
#' @param predictions The predictions against the test set.
#' @param training_predictions The predictions against the training set (optional).
plot_predictions <- function(sets, predictions, training_predictions = NULL) {
  plot <- ggplot()

  if (!is.null(training_predictions)) {
    plot <- plot +
      geom_line(data = training_predictions,
                aes(x = date, y = close, color = 'Training Prediction'))
  }

  plot <- plot +
    geom_line(data = sets$training, aes(x = date, y = close, color = 'Training Set')) +
```

```
    geom_line(data = sets$test, aes(x = date, y = close, color = 'Test Set')) +
    geom_line(data = predictions, aes(x = date, y = close, color = 'Test Prediction')) +
    scale_color_manual(values = c('Training Set' = 'blue',
                                  'Training Prediction' = 'cyan',
                                  'Test Set' = 'green',
                                  'Test Prediction' = 'red')) +
    labs(color = '') +
    theme(legend.position = 'top')

  plot
}
```

The following code creates a dataframe to store the evaluation results for all the methods considered in this project, evaluations are based on the RMSE against the ground truth in the test set applied to the following 1, 5, 10, 20, 40, 60 and 120 tradings days after the end of the training. Then a function is defined to support the displaying (and inclusion in this report) of the results for an individual method/model.

```
#' Creates an empty dataframe to report the evaluation results
create_results_dataframe <- function() {
  data.frame('Method' = character(),
             'Number of trading days ahead' = integer(),
             'RMSE' = numeric(),
             stringsAsFactors = FALSE, check.names = FALSE)
}

#' Dataframe to store all the evaluation results (for each method)
results <- create_results_dataframe()

#' This is a helper function used to support the displaying of the
#' evaluation results for a particular method, and at the same time
#' adding them to the dataframe that contains all the results.
get_evaluation_results <- function(method, predictions) {
  eval_results <- create_results_dataframe()

  for (i in c(1, 5, 10, 20, 40, 60, 120)) {
    n <- nrow(eval_results) + 1
    eval_results[n, 'Method'] <- method
    eval_results[n, 'Number of trading days ahead'] <- i
    eval_results[n, 'RMSE'] <- RMSE(predictions$close[1:i], eval_sets$test$close[1:i])
  }

  results <<- rbind(results, eval_results)

  eval_results %>% select(-'Method')
}
```

## 3.3 Classical ML methods

### 3.3.1 Linear Regression

Linear Regression is the first and most naive machine learning method to implement, the idea is just to obtain the regression line for the closing prices against the dates. Linear Regression is is a very popular method used in several domains, in spite of its simplicity it can be be very useful as a reference to benchmark

the performance of the other method.

For this project's purposes, Linear Regression follows the model:

$$\hat{c} = \alpha d + \beta$$

Where $\hat{c}$ is the predicted closing price of a stock for a specific trading day's date $d$ (actually a numerical representation of the date). $\alpha$ and $\beta$ are the linear coefficients (slope and intercept).

The following code is the implementation of the model based on Linear Regression.

```
#' This function represents a constructor
#' for a stock forecaster model based on Linear Regression.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on Linear Regression.
LinearRegressionStockForecaster <- function(
  base_dataset, ticker_symbol, training_start = NULL, training_end = NULL) {

  model <- list()

  # Extracting the training set from the base dataset,
  # i.e. filtering by ticker symbol and date range
  training_set <- filter_historical_records(
    base_dataset, ticker_symbol, training_start, training_end) %>%
    select(date, close)

  # Fitting a Linear Regression model where
  # 'date' is the predictor and 'close' is the predicted value
  model$model <- train(close~date, data = training_set, method = 'lm')

  #' The predicting function
  #'
  #' @param from_date The initial date of the date range to predict.
  #' @param to_date The final date of the date range to predict.
  #' @return A dataframe containing the dates in the range to predict
  #'    with their respective predicted closing price.
  model$predict <- function(from_date, to_date = NULL) {
    # If final date is null, then using the initial date, i.e predicting for 1 day
    if (is.null(to_date)) {
      to_date <- from_date
    }

    # Getting a dataframe containing the trading days to make predictions for
    trading_days <- get_trading_days_in_range(from_date, to_date)
    # Getting the predicted stock closing values
    preds <- predict(model$model, trading_days)
    # Creating the dataframe with the predicted values per trading day
    data.frame(date = trading_days$date, close = preds)
  }

  model
```

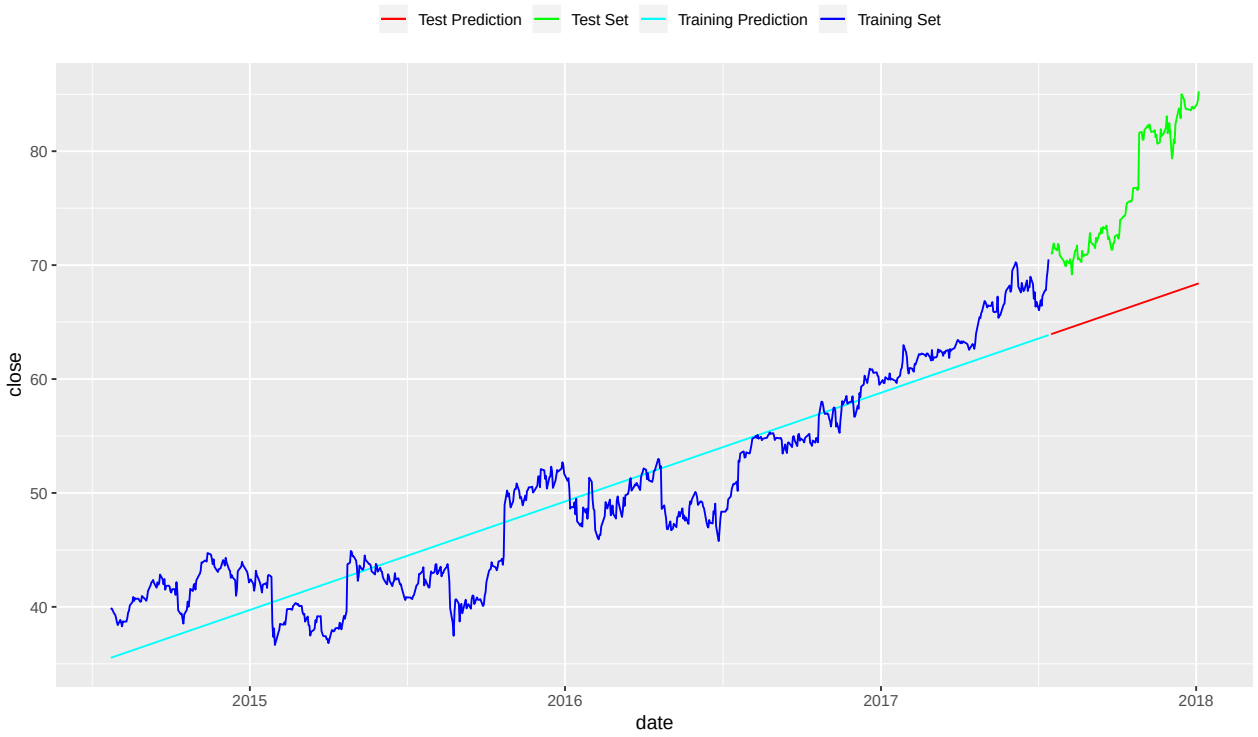


Figure 7: Prediction with Linear Regression

```
}
```

In the following code a Linear Regression model is created and fitted/trained using the defined training set for evaluation purposes, then it makes the predictions for the data range given for the test set.

```
lr_forecaster <- LinearRegressionStockForecaster(eval_sets$training, eval_ticker_symbol)
lr_predictions <- lr_forecaster$predict(eval_test_start, eval_test_end)
```

The visualization of the prediction performance of Linear Regression can be seen in figure 7, the results of the evaluation are presented in the table 1.

Table 1: Evaluation of prediction with Linear Regression

| Number of trading days ahead | RMSE |
|---:|---:|
| 1 | 7.120728 |
| 5 | 7.417800 |
| 10 | 7.285820 |
| 20 | 6.541369 |
| 40 | 6.506058 |
| 60 | 6.698735 |
| 120 | 10.956366 |

### 3.3.2 Generalized Additive Model

Generalized Additive Models (GAMs) are a way to model non-linear responses with in the framework of a linear logistic model or other generalized linear model, i.e. GAM relaxes the linearity constraint by finding a set of smooth curve functions applied to the predictors and a constant term $a$, such that the model is represented by the sum of them [1].

For the purpose of this project, a GAM model looks like:

$$\hat{c} = a + f(d)$$

Where $\hat{c}$ is the predicted closing price of a stock for a specific trading day's date $d$ (actually a numerical representation of the date), $a$ is a numeric constant and $f$ is a spline (i.e. a smooth curve function build up from polynomials).

The following code is the implementation of the model based on GAM, the package `caret` is being used to perform the tuning of the parameters.

```
#' This function represents a constructor
#' for a stock forecaster model based on Generalized Additive Model.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on Generalized Additive Model.
GeneralizedAdditiveModelStockForecaster <- function(
  base_dataset, ticker_symbol, training_start = NULL, training_end = NULL) {

  model <- list()

  # Extracting the training set from the base dataset,
  # i.e. filtering by ticker symbol and date range
  training_set <- filter_historical_records(
    base_dataset, ticker_symbol, training_start, training_end) %>%
    select(date, close)

  # Fitting a GAM model using caret to tune the parameters,
  # 'date' is the predictor and 'close' is the predicted value
  model$model <- train(close~date, data = training_set, method = 'gam',
                       trControl = trainControl(method = 'cv', number = 6,
                                                summaryFunction = defaultSummary),
                       tuneGrid = expand.grid(select = c(TRUE, FALSE),
                                              method = c("GCV.Cp", "REML", "P-REML",
                                                         "ML", "P-ML")),
                       metric = 'RMSE')

  #' The predicting function
  #'
  #' @param from_date The initial date of the date range to predict.
  #' @param to_date The final date of the date range to predict.
  #' @return A dataframe containing the dates in the range to predict
  #'    with their respective predicted closing price.
  model$predict <- function(from_date, to_date = NULL) {
    # If final date is null, then using the initial date, i.e predicting for 1 day
    if (is.null(to_date)) {
      to_date <- from_date
    }

    # Getting a dataframe containing the trading days to make predictions for
    trading_days <- get_trading_days_in_range(from_date, to_date)
    # Getting the predicted stock closing values
```

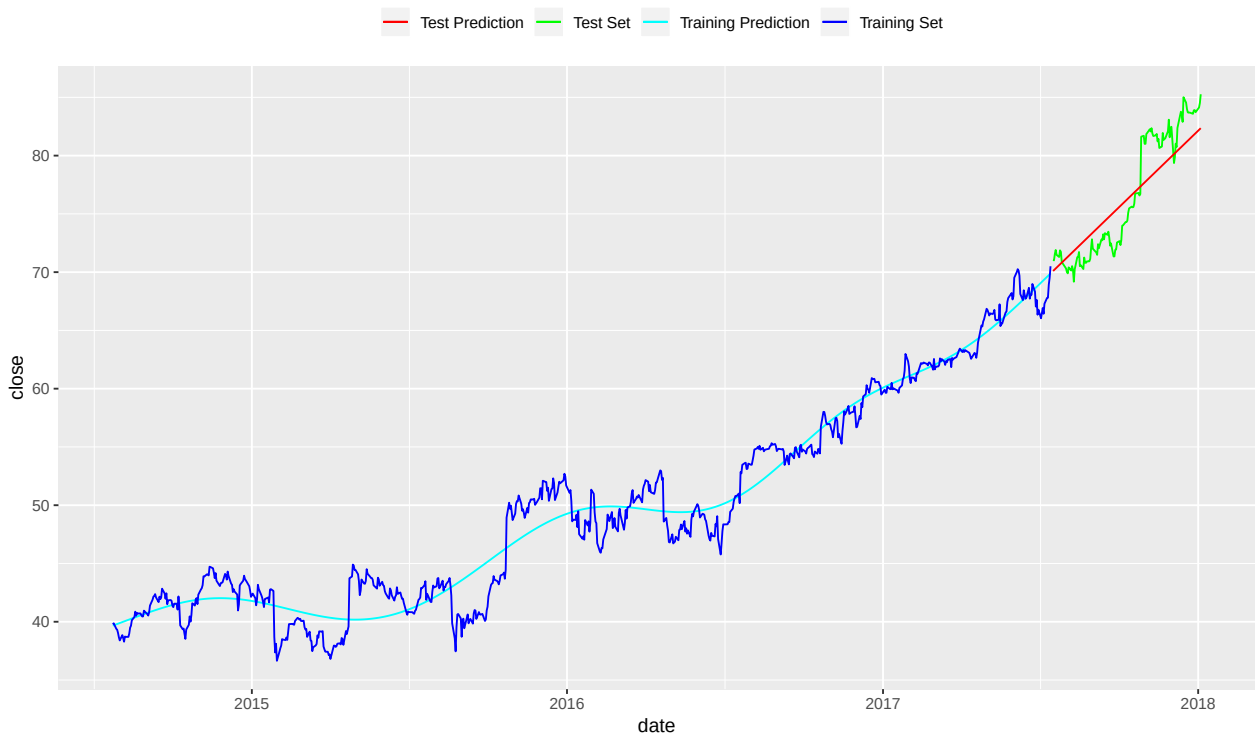


Figure 8: Prediction with Generalized Additive Model

```
    preds <- predict(model$model, trading_days)
    # Creating the dataframe with the predicted values per trading day
    data.frame(date = trading_days$date, close = preds)
  }

  model
}
```

In the following code a Generalized Additive Model is created and fitted/trained using the defined training set for evaluation purposes, then it makes the predictions for the data range given for the test set.

```
gam_forecaster <- GeneralizedAdditiveModelStockForecaster(
  eval_sets$training, eval_ticker_symbol)
gam_predictions <- gam_forecaster$predict(eval_test_start, eval_test_end)
```

The visualization of the prediction performance for GAM can be seen in figure 8 and the results of the evaluation are presented in the table 2.

Table 2: Evaluation of prediction with Generalized Additive Model

| Number of trading days ahead | RMSE |
|---|---|
| 1 | 0.9579269 |
| 5 | 1.1875802 |
| 10 | 0.9989334 |
| 20 | 1.2513098 |
| 40 | 1.4787244 |
| 60 | 1.8804725 |

| Number of trading days ahead | RMSE |
|---|---|
| 120 | 2.2275526 |

### 3.3.3 Support-Vector Machine

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection, they are among the best "out of the box" supervised learning techniques. The idea behind SVMs is to use entire training examples as classification (or regression landmarks) called support vectors [1]. The strategy of these techniques is to perform a data separation (or regression) by a hyper-plane when taking the data points to a higer dimension.

For this project's problem, a SVM model looks like:

$$\hat{c} = \sum_{i=1}^{m} a_i k(d_i, d) + b$$

Where $\hat{c}$ is the predicted closing price of a stock for a specific trading day's date $d$ (actually a numerical representation of the date), $k$ is a kernel function (i.e. $k(u, v) = \phi(u) \cdot \phi(v)$ for a given function $\phi$), $a_1, \ldots, a_m$ are numeric scalars, $b$ is a numeric scalar, and $d_1, \ldots, d_m$ are the support vectors (which in this case are other dates).

A radial based kernel (used for this project's implementation) is given by the formula:

$$k(u, v) = e^{(-\frac{||u-v||^2}{2\sigma^2})}$$

The following code is the implementation of the model based on SVM with a radial based kernel, the package `caret` is being used to perform the tuning of the parameters.

```
#' This function represents a constructor
#' for a stock forecaster model based on Support-Vector Machine.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on Support-Vector Machine.
SupportVectorMachineStockForecaster <- function(
  base_dataset, ticker_symbol, training_start = NULL, training_end = NULL) {

  model <- list()

  # Extracting the training set from the base dataset,
  # i.e. filtering by ticker symbol and date range
  training_set <- filter_historical_records(
    base_dataset, ticker_symbol, training_start, training_end) %>%
    select(date, close)

  # Fitting a Linear Regression model where
  # 'date' is the predictor and 'close' is the predicted value
  model$model <- train(close~date, data = training_set, method = 'svmRadial',
                       trControl = trainControl(method = 'cv', number = 6,
                                                summaryFunction = defaultSummary),
```

```
                        tuneGrid = expand.grid(sigma= 2^c(-25, -20, -15,-10, -5, 0),
                                               C= 2^c(0:5)),
                        metric = "RMSE")

  #' The predicting function
  #'
  #' @param from_date The initial date of the date range to predict.
  #' @param to_date The final date of the date range to predict.
  #' @return A dataframe containing the dates in the range to predict
  #'    with their respective predicted closing price.
  model$predict <- function(from_date, to_date = NULL) {
    # If final date is null, then using the initial date, i.e predicting for 1 day
    if (is.null(to_date)) {
      to_date <- from_date
    }

    # Getting a dataframe containing the trading days to make predictions for
    trading_days <- get_trading_days_in_range(from_date, to_date)
    # Getting the predicted stock closing values
    preds <- predict(model$model, trading_days)
    # Creating the dataframe with the predicted values per trading day
    data.frame(date = trading_days$date, close = preds)
  }

  model
}
```

In the following code a Support-Vector Machine model is created and fitted/trained using the defined training set for evaluation purposes, then it makes the predictions for the data range given for the test set.

```
svm_forecaster <- SupportVectorMachineStockForecaster(
  eval_sets$training, eval_ticker_symbol)
svm_predictions <- svm_forecaster$predict(eval_test_start, eval_test_end)
```

The visualization of the prediction performance for SVM can be seen in figure 9 and the results of the evaluation are presented in the table 3.

Table 3: Evaluation of prediction with Support-Vector Machine

| Number of trading days ahead | RMSE |
|---:|---:|
| 1 | 2.211493 |
| 5 | 2.481180 |
| 10 | 2.306069 |
| 20 | 1.687804 |
| 40 | 1.402610 |
| 60 | 1.421239 |
| 120 | 6.467348 |

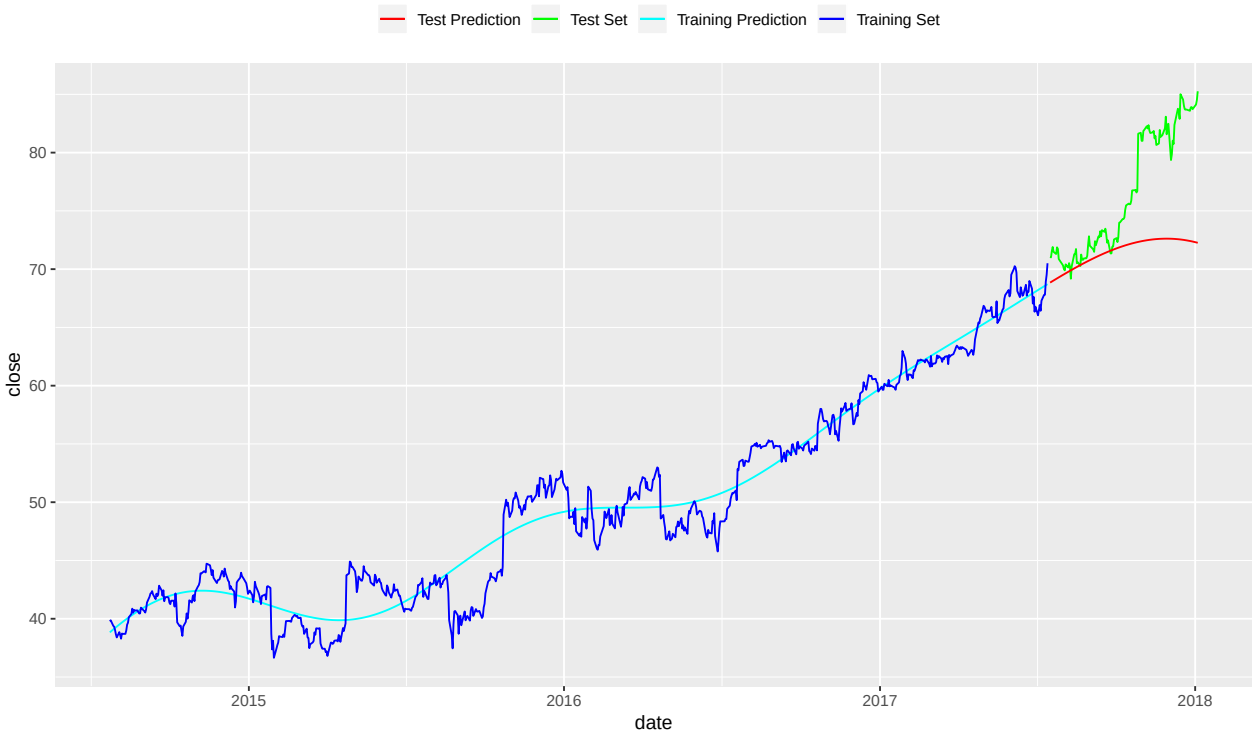


Figure 9: Prediction with Support-Vector Machine

## 3.4 Time series based methods

### 3.4.1 ARIMA

AutoRegressive Integrated Moving Average (ARIMA) It is a very popular statistical method for time series analysis and forecasting, its acronym is descriptive, capturing the key aspects of the model itself [2]:

- AR (Autoregression): A model that uses the dependent relationship between an observation and some number of lagged observations.
- I (Integrated). The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- MA (Moving Average): A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

The mathematical equations to describe ARIMA models are fairly complex and beyond the scope of this report, furter details can be found on [3]. In summary ARIMA has there are three important parameters to tune [4]:

- $p$: Past values used for forecasting the next value.
- $q$: Past forecast errors used to predict the future values.
- $d$: Order of differencing.

Parameter tuning for ARIMA consumes a lot of time, fortunately the module "Auto ARIMA" (used for this project) is available and has the advantage of automatically selecting the best combination of the values of ($p$, $q$, $d$) that provides the least error, this selection is done during the fitting process.

To be fitted, ARIMA only needs a sequence of ground truth consecutive values in the time series to do the training, then to prepare the training set only the column `close` needs to be selected, however an important aspect is that the order must be preserved.

The following code is the implementation of the model based on ARIMA using `auto.arima` to automatically tune the parameters.

```
#' This function represents a constructor
#' for a stock forecaster model based on ARIMA.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on ARIMA.
ArimaStockForecaster <- function(
  base_dataset, ticker_symbol, training_start = NULL, training_end = NULL) {

  model <- list()

  # Extracting the training set from the base dataset,
  # i.e. filtering by ticker symbol and date range
  training_set <- filter_historical_records(
    base_dataset, ticker_symbol, training_start, training_end) %>%
    select(date, close)

  # Keeping track of the training end date
  model$training_end <- max(training_set$date)

  # Fitting an ARIMA model
  # (using auto.arima which automatically tunes the parameters)
  # to predict it takes a time series containing the stock closing prices
  model$model <- auto.arima(
    training_set %>% column_to_rownames(var = 'date') %>% .$close,
    start.p = 0, start.q = 0, max.p = 5, max.q = 5,
    start.P = 0, start.Q = 0, max.P = 5, max.Q = 5,
    d = 1, D = 1,
    seasonal = TRUE,
    trace = FALSE)

  #' The predicting function
  #'
  #' @param from_date The initial date of the date range to predict.
  #' @param to_date The final date of the date range to predict.
  #' @return A dataframe containing the dates in the range to predict
  #'    with their respective predicted closing price.
  model$predict <- function(from_date, to_date = NULL) {
    # If final date is null, then using the initial date, i.e predicting for 1 day
    if (is.null(to_date)) {
      to_date <- from_date
    }
    # Checking that date range is valid
    if (from_date > to_date) {
      stop('Invalid date range')
    }
    # Checking that prediction range is after training
    if (from_date <= model$training_end) {
```

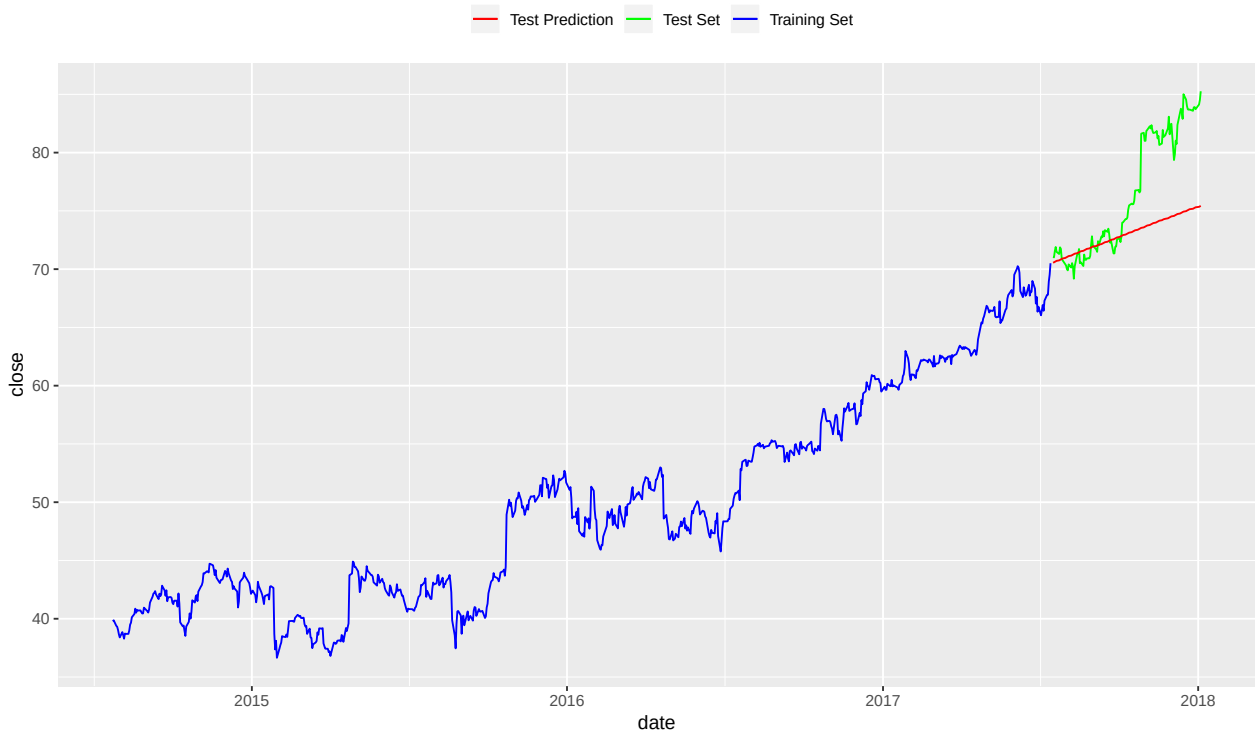


Figure 10: Prediction with ARIMA

```
      stop('Prediction range should be after training')
    }

    # Getting a dataframe containing the trading days to make predictions for,
    # including the days that might be missing after the end of training
    # and the begining of the predicting range
    trading_days <- get_trading_days_in_range(model$training_end + ddays(1), to_date)
    # Getting the predicted stock closing values
    preds <- forecast(model$model, nrow(trading_days))
    # Creating the dataframe with the predicted values per trading day
    # and filtering to include just the trading days in the given range
    data.frame(date = trading_days, close = preds$mean) %>%
      filter(date >= from_date)
  }

  model
}
```

In the following code an ARIMA model is created and fitted using the defined training set for evaluation purposes, then it makes the predictions for the data range given for the test set.

```
arima_forecaster <- ArimaStockForecaster(eval_sets$training, eval_ticker_symbol)
arima_predictions <- arima_forecaster$predict(eval_test_start, eval_test_end)
```

The visualization of the prediction performance for ARIMA can be seen in figure 10, note that the prediction on the training set does not appear since the model do not support predictions on a date range, it predicts for a number of time steps (days in this case) ahead. The results of the evaluation are presented in the table 4.

Table 4: Evaluation of prediction with ARIMA

| Number of trading days ahead | RMSE |
|---|---|
| 1 | 0.5111737 |
| 5 | 0.8251918 |
| 10 | 0.7555108 |
| 20 | 0.9401079 |
| 40 | 0.8155093 |
| 60 | 0.8172873 |
| 120 | 5.1225938 |

### 3.4.2 Prophet

Prophet is an open source forecasting tool developed by Facebook, it is optimized for the business forecast tasks encountered at Facebook. They claim that the default settings produce forecasts that are often as accurate as those produced by skilled forecasters, with much less effort [5].

For training, this model receives the date as predictor and the ground truth of the value to predict as outcome, however their columns should be named `ds` and `y` respectively.

The following code is the implementation of the model based on Prophet, the parameter `daily.seasonality` is turned on to reflect that the time series to forecast are based on time steps of days.

```
#' This function represents a constructor
#' for a stock forecaster model based on Prophet.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on Prophet
ProphetStockForecaster <- function(
  base_dataset, ticker_symbol, start_date = NULL, end_date = NULL) {

  model <- list()

  # Extracting the training set from the base dataset)
  # (i.e. filtering by ticker symbol and date range),
  # and changing the column names to 'ds' and 'y' which are the ones Prophet uses
  training_set <- filter_historical_records(
    base_dataset, ticker_symbol, start_date, end_date) %>%
    select(date, close) %>%
    setnames(old = c('date', 'close'), new = c('ds', 'y'))

  # Fitting an Prophet model
  model$model <- prophet(training_set, daily.seasonality = TRUE)

  #' The predicting function
  #'
  #' @param from_date The initial date of the date range to predict.
  #' @param to_date The final date of the date range to predict.
  #' @return A dataframe containing the dates in the range to predict
  #'    with their respective predicted closing price.
```

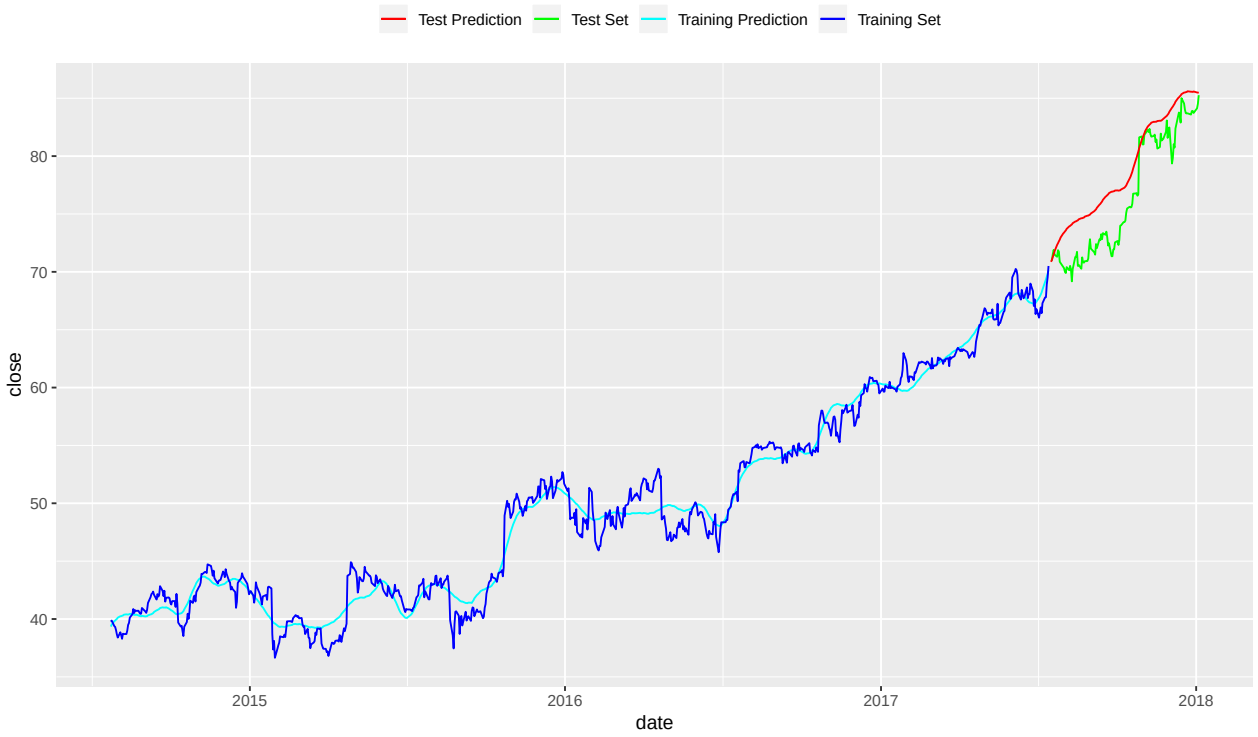


Figure 11: Prediction with Prophet

```
  model$predict <- function(from_date, to_date = NULL) {
    # If final date is null, then using the initial date, i.e predicting for 1 day
    if (is.null(to_date)) {
      to_date <- from_date
    }

    # Getting a dataframe containing the trading days to make predictions for,
    # and changing the column name to 'ds' which is the one Prophet uses
    trading_days <- get_trading_days_in_range(from_date, to_date) %>%
      setnames(old = c('date'), new = c('ds'))
    # Getting the predicted stock closing values
    preds <- predict(model$model, trading_days)
    # Creating the dataframe with the predicted values per trading day
    data.frame(date = trading_days$ds, close = preds$yhat)
  }

  model
}
```

In the following code a Prophet model is created and fitted using the defined training set for evaluation purposes, then it makes the predictions for the data range given for the test set.

```
prophet_forecaster <- ProphetStockForecaster(eval_sets$training, eval_ticker_symbol)
prophet_predictions <- prophet_forecaster$predict(eval_test_start, eval_test_end)
```

The visualization of the prediction performance for Prophet can be seen in figure 11. The results of the evaluation are presented in the table 5.

Table 5: Evaluation of prediction with Prophet

| Number of trading days ahead | RMSE |
|---|---|
| 1 | 0.1949709 |
| 5 | 0.2592360 |
| 10 | 1.0540464 |
| 20 | 2.7064137 |
| 40 | 3.1106034 |
| 60 | 3.4517933 |
| 120 | 2.8519843 |

#### 3.4.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a deep learning method (i.e. that uses neural networks) used to forecast time series.

A Recurrent Neural Network (RNN) can be thought of as multiple copies of the same network, each passing a message to a successor, aiming for them to learn from the past [6]. RNNs are good in handling sequential data but they have two main problems, the first one called "Vanishing/Exploding Gradient problem" presented as a result of the weights being repeated several times, and the second one called "Long-Term Dependencies problem" that happens when the context is far away [7].

Long Short-Term Memory networks are a special kind of RNNs (introduced by Hochreiter & Schmidhuber in 1997 [8]) with capability of handling Long-Term dependencies and also provide a solution to the Vanishing/Exploding Gradient problem [7]. They are currently used to address difficult sequence problems in machine learning and achieve state-of-the-art results [9].

For this model is necessary to create an LSTM neural, that can be done through Keras. Keras is a high-level neural networks API capable of running on top of multiple back-ends like TensorFlow, CNTK or Theano. It was developed with a focus on enabling fast experimentation, being able to go from idea to result with the least possible delay is key to doing good research [10]. Even though Keras is written in Python, the package `keras` provides an interface for using it with R.

The implementation of this model is significantly more complex than the previous ones, because it had to be constructed from the scratch, since there is no library that already have it available.

LSTM takes as predictors sequences of a given length (known as time steps) containing consecutive values in a time series and the outcome is the next value in the time sequence, i.e. the predictor is a subsequence of the time series instead of a single value. To prepare the training set only the values in the column close are needed, the date is not necessary since the important factor is the order. The preprocessing for this method consists in creating the subsequences with the closing prices previous to the date of the respective outcome. In addition, as recomended in most deep-learning approaches the values to predict (i.e. the closing prices) are scaled to the range from 0 to 1.

The preprocessing mechanism for the prediction set is complex as well, since it is required to create the subsequences containing the closing prices previous to the date to predict, and normalized to the range $[0, 1]$. That means that in order to do long term predictions the missing values of the subsequence need to be fulfilled with other predicted values.

In contrast with other methods, a trainied LSTM model can take advance of an updated dataset to make further predictions, basically by constructing the subsequence used as predictor with updated data, the advantage is that the LSTM model can be kept in use without the need of training it again. For example and ideal case would be to train the LSTM model, then updating the dataset on daily bases and performing the prediction of the closing price for the next day.

The following code is the implementation of the LSTM model, an important aspect to notice is that in the prediction the it first tries to fill the subsequences used as predictors with the existing records in the given dataset, if there are missing records the data is fulfilled with predictions for the previous dates.

```
#' This function represents a constructor
#' for a stock forecaster model based on Long Short-Term Memory.
#'
#' @param base_dataset The dataframe used to extract the training set
#'    in accordance with the date range.
#' @param ticker_symbol The ticker symbol to perform predictions for.
#' @param training_start The minimum date for the records used in the training set.
#' @param training_end The maximum date for the records used in the training set.
#' @return The model based on Long Short-Term Memory
LongShortTermMemoryStockForecaster <- function(
  base_dataset, ticker_symbol, start_date = NULL, end_date = NULL) {

  model <- list()

  # The ticker symbol used to fit the model
  model$ticker_symbol <- ticker_symbol

  # Size of the time sub-series used to feed the LSTM network
  model$timesteps <- 60
  # Number of epochs used to train the LSTM network
  model$epochs <- 2

  # Extracting the training set from the base dataset)
  # i.e. filtering by ticker symbol and date range.
  model$training_set <- filter_historical_records(
    base_dataset, model$ticker_symbol, start_date, end_date) %>%
    select(date, close)

  # Keeping track of the training end date
  model$training_end <- max(model$training_set$date)

  # Scale factor used to normalize the timeseries used as input for the
  # LSTM network to contain values in the range [0, 1]
  model$scale_factor <- max(model$training_set$close) * 2.0

  # Constructing the inputs used to traing the LSTM network
  # The predictor is a collection of time sub-series of size 'timesteps'
  train_X <- t(sapply((model$timesteps + 1): nrow(model$training_set),
                      function(i) {
                        model$training_set$close[(i - model$timesteps) : (i - 1)]
                      }))
  train_X <- train_X / model$scale_factor
  dim(train_X) <- c(dim(train_X)[1], dim(train_X)[2], 1)
  # The outcome are the closing prices normalized/scaled to the range [0, 1]
  train_Y <- model$training_set$close[(model$timesteps + 1) : nrow(model$training_set)]
  train_Y <- train_Y / model$scale_factor

  # Constructing the LSTM neural network, with an architecture of two LSTM layers
  model$model <- keras_model_sequential() %>%
    layer_lstm(units = 50, return_sequences = TRUE,
```

```
              input_shape = c(model$timesteps, 1)) %>%
  layer_lstm(units = 50) %>%
  layer_dense(1) %>%
  compile(loss = 'mean_squared_error', optimizer = 'adam')

# Training the LSTM network
model$training_history <- model$model %>%
  fit(x = train_X, y = train_Y, epochs = model$epochs, batch_size = 1, verbose = 2)

#' The predicting function
#'
#' @param from_date The initial date of the date range to predict.
#' @param to_date The final date of the date range to predict.
#' @param base_dataset The data set used to support the prediction process.
#' @return A dataframe containing the dates in the range to predict
#'    with their respective predicted closing price.
model$predict <- function(from_date, to_date = NULL, base_dataset = NULL)  {
  # If final date is null, then using the initial date, i.e predicting for 1 day
  if (is.null(to_date)) {
    to_date <- from_date
  }
  # Checking that date range is valid
  if (from_date > to_date) {
    stop('Invalid date range')
  }

  # If not base dataset to support predictions is provided,
  # then using the training set
  if (is.null(base_dataset)) {
    base_dataset <- model$training_set
  } else {
    base_dataset <- filter(base_dataset, symbol == model$ticker_symbol)
  }

  # Only the historical records on or before the end prediction date are needed
  base_dataset <- filter(base_dataset, date <= to_date)

  # Reducing the base dataset just to contain the historical records
  # necessary to perform the prediction
  idx <- which(base_dataset$date >= from_date)
  if (length(idx) > 0) {
    # If there are records that already exist in the given date range ...
    idx <- min(idx)

    if (idx <= model$timesteps) {
      stop('Not enough records to perform predictions')
    }

    # ... keeping the previous 'timesteps' records before the begining of
    # of the prediction date range
    base_dataset <- base_dataset[(idx - model$timesteps) : nrow(base_dataset),]
  } else {
    # If there are not records already existing in the given date range,
```

```
  # just taking the last 'timesteps' records
  base_dataset <- tail(base_dataset, n = model$timesteps)

  if (nrow(base_dataset) < model$timesteps) {
    stop('Not enough records to perform predictions')
  }
}

# Getting the missing days (for which there are not historical records)
# needed to fulfill the predictions in the given range
missing_start <- max(base_dataset$date) + ddays(1)
if (missing_start <= to_date) {
  missing_days <- get_trading_days_in_range(missing_start, to_date)
} else {
  missing_days <- NULL
}

# Normalizing the time series used to perform the predictions
# to be in the range [0, 1]
inputs <- base_dataset$close / model$scale_factor

trading_days <- c() # The trading days in the date range to predict
preds <- c() # The predicted closing prices in the date range

count <- 1

# First using the historical records that already exist to
# suuport the prediction in the given date range
i <- model$timesteps + 1
while (i <= nrow(base_dataset)) {
  trading_days[count] <- base_dataset[i, 'date']

  # Feeding the LSTM network with the previous subseries of size 'timesteps'
  x <- inputs[(i - model$timesteps) : (i - 1)]
  dim(x) <- c(1, model$timesteps, 1)
  y <- predict(model$model, x)

  # Scaling back the output to be the prediction
  preds[count] <- y * model$scale_factor

  i <- i + 1
  count <- count + 1
}

# Secondly, fulfiling the missing days by completing the timeseries
# with the prediction for the previous day
j <- 1
while (!is.null(missing_days) && j <= nrow(missing_days)) {
  trading_days[count] <- missing_days[j, 'date']

  # Feeding the LSTM network with the previous subseries of size 'timesteps'
  x <- inputs[(length(inputs) - model$timesteps + 1) : length(inputs)]
  dim(x) <- c(1, model$timesteps, 1)
```

```
      y <- predict(model$model, x)

      # Completing the timeseries with the predicted value for the current day
      inputs[length(inputs) + 1] <- y

      # Scaling back the output to be the prediction
      preds[count] <- y * model$scale_factor

      j <- j + 1
      count <- count + 1
    }

    # Returning the results just for the given prediction data range
    data.frame(date = as_date(trading_days), close = preds) %>%
      filter(date >= from_date)
  }

  model
}
```

In the following code an LSTM model is created and trained using the defined training set for evaluation purposes.

```
lstm_forecaster <- LongShortTermMemoryStockForecaster(eval_sets$training, eval_ticker_symbol)
```

Then in the next code, the predictions are performed for the data range given for the test set, in here is represented the case when the closing prices of the predicting range are completly unknown (p.e. when a long term prediction in the future is attempted).

```
lstm_predictions <- lstm_forecaster$predict(eval_test_start, eval_test_end)
```

In figure 12 it can be seen the visualization of the prediction performance for LSTM when the predicting closing prices are completely unknown. The results of the evaluation are presented in the table 6.

Table 6: Evaluation of prediction with LSTM

| Number of trading days ahead | RMSE |
|---:|---:|
| 1 | 5.254464 |
| 5 | 5.656434 |
| 10 | 6.199327 |
| 20 | 7.081838 |
| 40 | 10.700527 |
| 60 | 14.040237 |
| 120 | 25.484415 |

The previoUs evaluation does not seem very encouraging, however the resuls are completly different when the prediction is performed when the dataset is kept updated, p.e. in the situation where the dataset is daily updated and the predictions performed for the next day. The following code simulates this scenario by providing an updated dataset to support the prediction process.

```
lstm_daily_predictions <- lstm_forecaster$predict(eval_test_start, eval_test_end,
                                                  dow_jones_historical_records)
```

In figure 13 it can be seen the visualization of the prediction performance for LSTM when the dataset is kept updated and the prediction performed for the next day. The results of the evaluation for this approach are

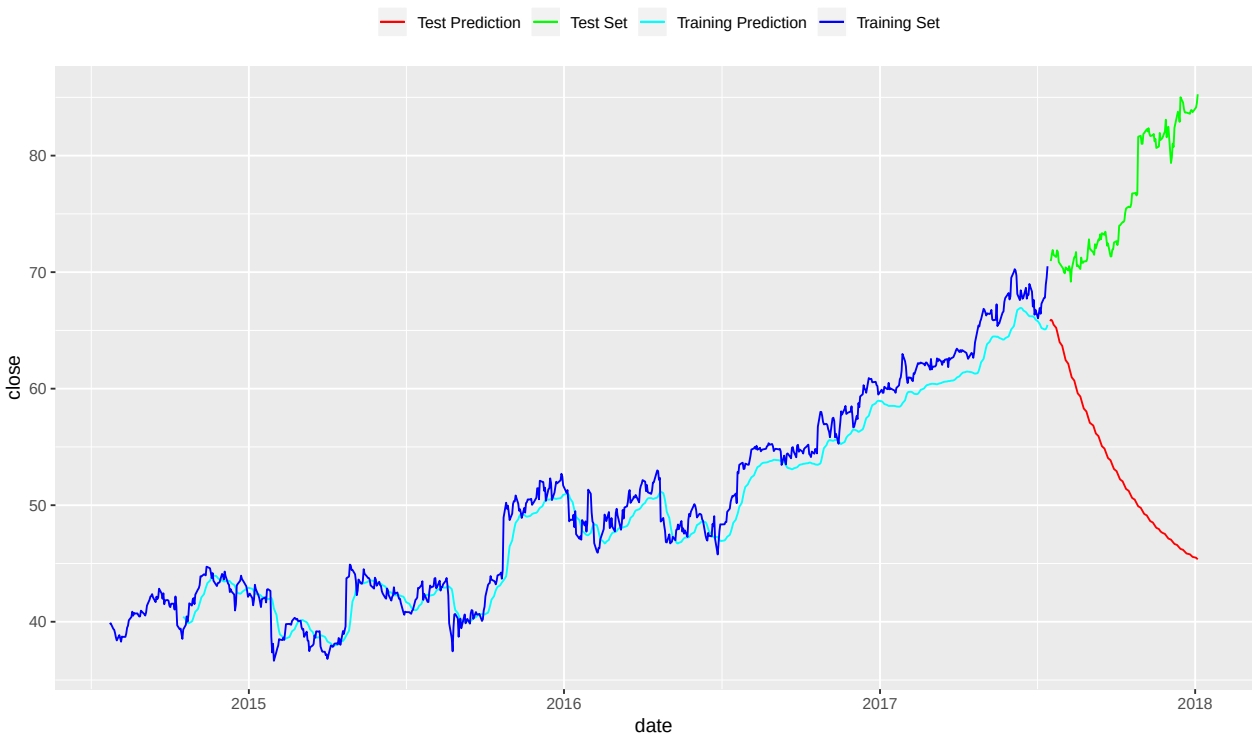


Figure 12: Prediction with LSTM

presented in the table 7.

Table 7: Evaluation of prediction with LSTM doing daily update and predict

| Number of trading days ahead | RMSE |
|---|---|
| 1 | 5.254464 |
| 5 | 4.752300 |
| 10 | 3.919453 |
| 20 | 2.971955 |
| 40 | 2.884936 |
| 60 | 2.837608 |
| 120 | 3.423760 |

# 4 Results

The results of the evaluation for all the models tested in this report are gathered in table 8.

Table 8: Results of the evaluation

| Method | 1 | 5 | 10 | 20 | 40 | 60 | 120 |
|---|---|---|---|---|---|---|---|
| ARIMA | 0.5111737 | 0.8251918 | 0.7555108 | 0.9401079 | 0.8155093 | 0.8172873 | 5.122594 |
| GAM | 0.9579269 | 1.1875802 | 0.9989334 | 1.2513098 | 1.4787244 | 1.8804725 | 2.227553 |
| Linear Regression | 7.1207281 | 7.4178000 | 7.2858196 | 6.5413686 | 6.5060576 | 6.6987347 | 10.956366 |
| LSTM | 5.2544642 | 5.6564339 | 6.1993272 | 7.0818379 | 10.7005270 | 14.0402367 | 25.484415 |
| LSTM - daily update/predict | 5.2544642 | 4.7522998 | 3.9194533 | 2.9719552 | 2.8849360 | 2.8376081 | 3.423760 |
| Prophet | 0.1949709 | 0.2592360 | 1.0540464 | 2.7064137 | 3.1106034 | 3.4517933 | 2.851984 |
| SVM | 2.2114934 | 2.4811800 | 2.3060693 | 1.6878038 | 1.4026105 | 1.4212387 | 6.467348 |



# References

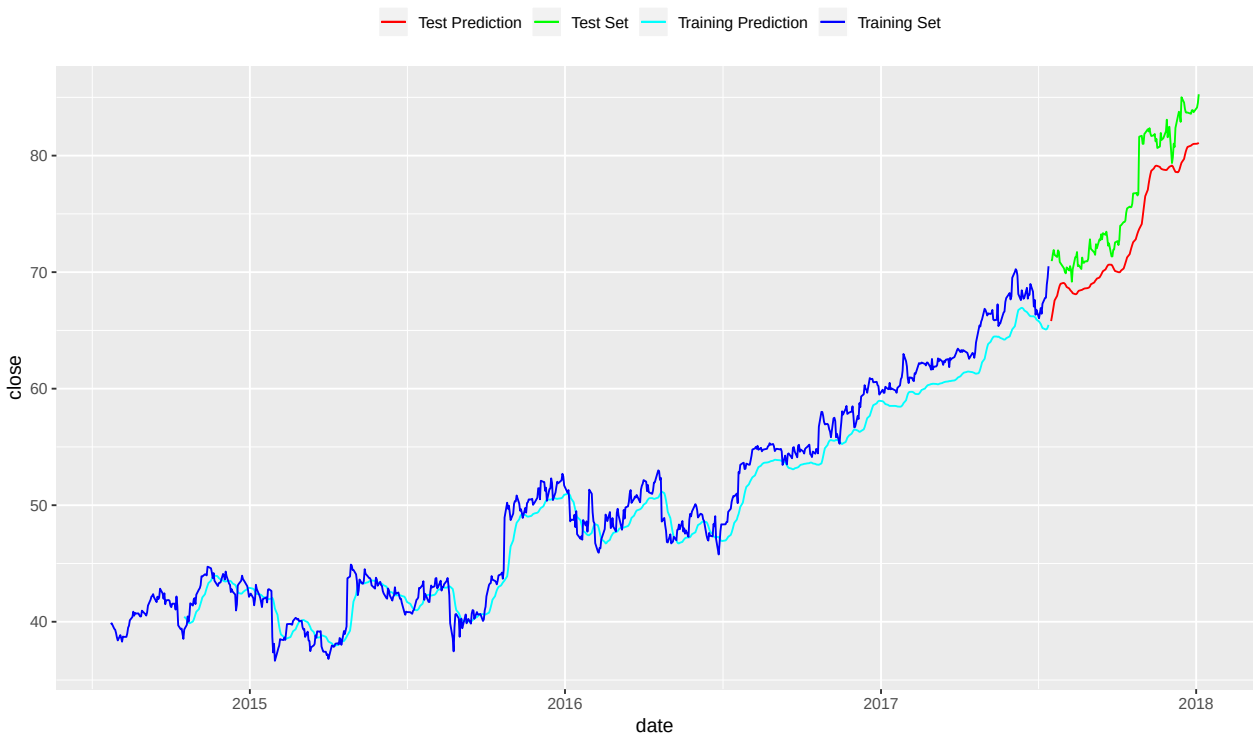


Figure 13: Prediction with LSTM doing daily update and predict

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 6, pp. 1735–1780, 1997 [Online]. Available: http://www.bioinf.jku.at/publications/older/2604.pdf

[9] J. Brownlee, "Time series prediction with lstm recurrent neural networks in python with keras." Machine Learning Mastery, 21-Jul-2016 [Online]. Available: https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

[10] J. Allaire and F. Chollet, "R interface to keras." RStudio, Google [Online]. Available: https://keras.rstudio.com/