Machine Learning Engineer Nanodegree

Capstone Report

Using Deep Neural Networks to Predict Future Stock Prices

Jakob Salomonsson

March 10th, 2018

Table of Contents

1. Definition

1.1 Project Overview

Humans started to use money as a commodity to do trade for more than 2,500 years ago [1]. It can be claimed that ever since then, mankind has had an urge for making money, and arguably, at no other place is this more obvious than on the stock market. Both large institutions and private investors have today many alternatives when it comes to where and how to make their investments. Common for all however, is that they all want to sell for a higher price than what they bought for, why trying to predict the future is highly desirable. As a consequence, numerous strategies have been developed, where some are more efficient than others.

Machine Learning algorithms are no exception. A lot of research has been done within the field with various results. This blog post [2], where the author used a Deep Neural Network to predict the future price of Bitcoin and Ethereum, as well as this paper [3], influenced me in my choice.

Rather than predicting cryptocurrency prices, this project focus on stocks in the OMX Stockholm Mid Cap Index. Historical prices for each individual stock are processed and used to predict the stock price for the subsequent day and ten days ahead. The data is extracted from Yahoo! Finance [4].

My personal motivation for investigating this domain is the fact that I'm about to graduate from University, with quite some loans accumulated. Developing, or at least taking a step closer, to a system that can beat the market and help paying back the loans, would be great news for me.

1.2 Problem Statement

During the project, the historical stock price for each individual stock in the OMX Stockholm Mid Cap Index is processed. Logistic Regression and LSTM Deep Learning models are used, in this regression problem, to predict the price for each stock, both one and ten trading days into the future.

Input data are times series, covering prices (such as low and high, open and close) and volumes for each date. It is downloaded in an automated way from Yahoo! Finance to facilitate future updates. Before being fed into the model, the data needs to be processed to remove NaN-values and other disturbances. It is also normalised to facilitate comparison between different stocks, as well as scaled, to minimize training error and training time. The data is plotted and compared with OMX 30 to display the training and test sets as well as to reveal any eventual flaws.

The models themselves are built in Keras library with TensorFlow run as backend. Logistic Regression models can easily be imported from Scikit-learn, but to keep a homogeneous workflow they are built with the same library as the LSTM models. Different kind of architectures are tested out, with some inspiration from earlier work in the field [2] [5] [6] on how to tweak it. The Logistic Regression model constitutes of one single Dense layer, while the LSTM models make up of either one or two recurrent LSTM layers with Dropouts in between, before adding a core Dense layer. Combinations of various number of neurons, epochs, batch sizes, different activation functions and optimizers, as well as window sizes and Dropouts for the LSTM models, are evaluated during the project. Keras' documentation [7] [8] [9] provide some guidance in this task.

Subsequently, the scaled single-day predictions for both the Logistic Regression and LSTM Network are inverted before plotted and compared. The better alternative is chosen for making ten-day predictions as it's believed that unsatisfying single-day predictions will most certainly lead to even more unsatisfying ten-day predictions.

Ultimately, the top five predicted stock increases, the most recent ten days, serve as input to an optimizer algorithm (not in the scope of this project), which will allocate based on Sharpe Ratio.

1.3 Metrics

The models' performance on both the train and test sets are evaluated based on Mean Squared Error (MSE), which is commonly used for regression tasks. MSE is to prefer since it eliminates the effect of the sign (we want underestimates to be penalised similarly to overestimates) as well as it adds more weight to larger errors rather than smaller. It is defined as:

$$MSE(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta\left(x^{(i)}\right) - y^{(i)})^2$$

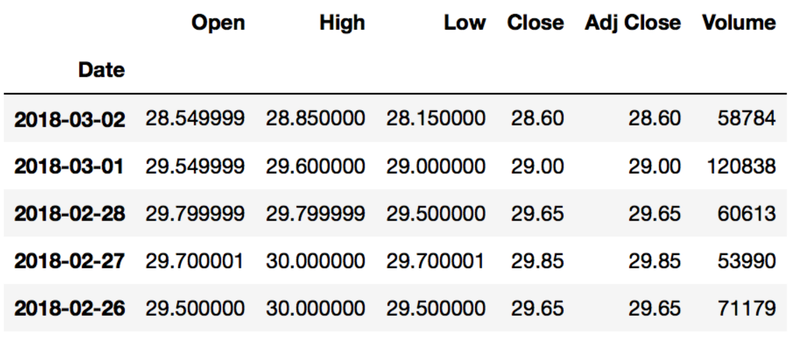
Where m is the number of training examples, $x^{(i)}$ and $y^{(i)}$ are the input vector and the class label for the i^{th} training example, θ are the chosen parameter values while $h_\theta\left(x^{(i)}\right)$ is the algorithm's prediction for the i^{th} training example using the parameter θ. Using an error function that calculates the absolute value, rather than the squared, would give the same result but it requires more complicated use. The top five stocks are chosen based on the highest predicted value increase, in percentage, the last ten days.

2. Analysis

2.1 Data Exploration

As of March 2018, OMX Stockholm Mid Cap Index constitutes in total of 144 shares [10], class A and class B shares included. However, many shares have missing values for long periods, while others are not even present at Yahoo! Finance. Stocks that don't have three years of history will be excluded. Shares that have absent values for two consecutive months or more, are also excluded as they are considered to be too unreliable to fit as input data. After this initial screening, the total amount of stocks used as input are 73, OMX Stockholm 30 Index included, where the latter will be used for comparison.

Table 1 displays the top five rows in the raw unprocessed data for share Acando B, ticker ACAN-B.ST. The data makes up of totally 1302 rows, but will increase daily as the starting date is fixed



	Open	High	Low	Close	Adj Close	Volume
Date						
2018-03-02	28.549999	28.850000	28.150000	28.60	28.60	58784
2018-03-01	29.549999	29.600000	29.000000	29.00	29.00	120838
2018-02-28	29.799999	29.799999	29.500000	29.65	29.65	60613
2018-02-27	29.700001	30.000000	29.700001	29.85	29.85	53990
2018-02-26	29.500000	30.000000	29.500000	29.65	29.65	71179

Table 1: Raw unprocessed data for share Acando B (ACAN-B.ST).

at December 27th 2012. This date is chosen arbitrary and for no other reason but to include five years of stock history in the input data. Only dates when the stock exchange was open and the shares were traded are presented, leading to an average of roughly 252 rows per year.

The data is represented by six columns, or features. These are *Open*, *High*, *Low*, *Close*, *Adjusted Close*, or *Adj Close*, and *Volume*. The first five are related to the stock price, where *Open* represents the opening price for each particular day and *High* the highest price for that particular stock at that particular date. *Low* and *Close* are, consequently, the lowest price and closing price for each particular date. *Adj Close*, is the adjusted closing price and has been adjusted to take into account changes in the stock price due to splits and dividends. It is crucial that this is done as a split can have huge consequences on the stock price, resulting in 10x changes without the actual value of the company alters. Running an algorithm on such data would most certainty lead to worthless results. These feature values are normally in the range of [1, 100]. Lastly, *Volume* is the number of shares traded each day for each specific share and it can be in the range of a couple of hundred thousand to some millions.

Volatility is engineered and added as an additional feature to augment the information on which the algorithm makes its predictions on. It has a distribution of normally [0,1]. With this extra feature, each stock now constitutes of roughly 9,100 data points, 1,300 per feature. The entire dataset amounts to roughly 665,000 data points.

In order to keep track on each individual stock, the stock name will be added to the Date column. The Date column also serves as index column.

As mentioned earlier, a lot of stocks were excluded due to NaN-values. However, there still exist such undefined data types among the chosen stocks. They will be solved through pandas' *fillna* function [11]. It is a normal procedure in this kind of task and is taught in Tucker Balch's *Machine Learning for Trading* course [12].

Subsequently, each feature will be normalised according to the first date in the dataset as well as scaled within the interval [0, 1]. The former is done to make it easier to compare different stocks as their original values might differ quite a lot, while the latter enables the solution to converge in the first place as well as to higher final performance [13].

	Open	**High**	**Low**	**Close**	**Adj Close**	**Volume**	**Volatility**
ACAN-B.ST__Date							
2018-03-02	0.799569	0.800857	0.793177	0.801724	0.856037	0.011421	0.219031
2018-03-01	0.842672	0.832976	0.829424	0.818965	0.872974	0.023478	0.181387
2018-02-28	0.853448	0.841542	0.850746	0.846983	0.900496	0.011777	0.089933
2018-02-27	0.849138	0.850107	0.859275	0.855603	0.908965	0.010490	0.090235
2018-02-26	0.840517	0.850107	0.850746	0.846983	0.900496	0.013829	0.151412

Table 2: Pre-processed input data for share Acando B (ACAN-B.ST).

Table 2 displays the pre-processed data for share Acando B after the steps described above have been applied.

Furthermore, the data will be split 80/20 into training and test sets, leading to 1042 rows of training data and 260 for testing purpose.

LSTM Networks require the input data to be 3D, in the order (*samples*, *window*, and *features*), where *samples* are the number of samples in the dataset and *window* how far back the model should base its predictions on. The longer window the more information is fed into the model but the longer the training time. *Features* are the number of features in the dataset (7). Hence it needs to be processed in several steps into a 3D vector in order to be fed into the model.

2.1 Exploratory Visualisation

In Figure 1 below, normalised Adjusted Close data for Acando B and OMX Stockholm 30 Index is plotted. The training and test sets for the stock are displayed in green and blue respectively, while OMX30 Index is shown in orange. Another 71 plots, similar to this and covering each individual stock, are plotted to display any disturbances in the data. They can all be accessed through this GitHub page.

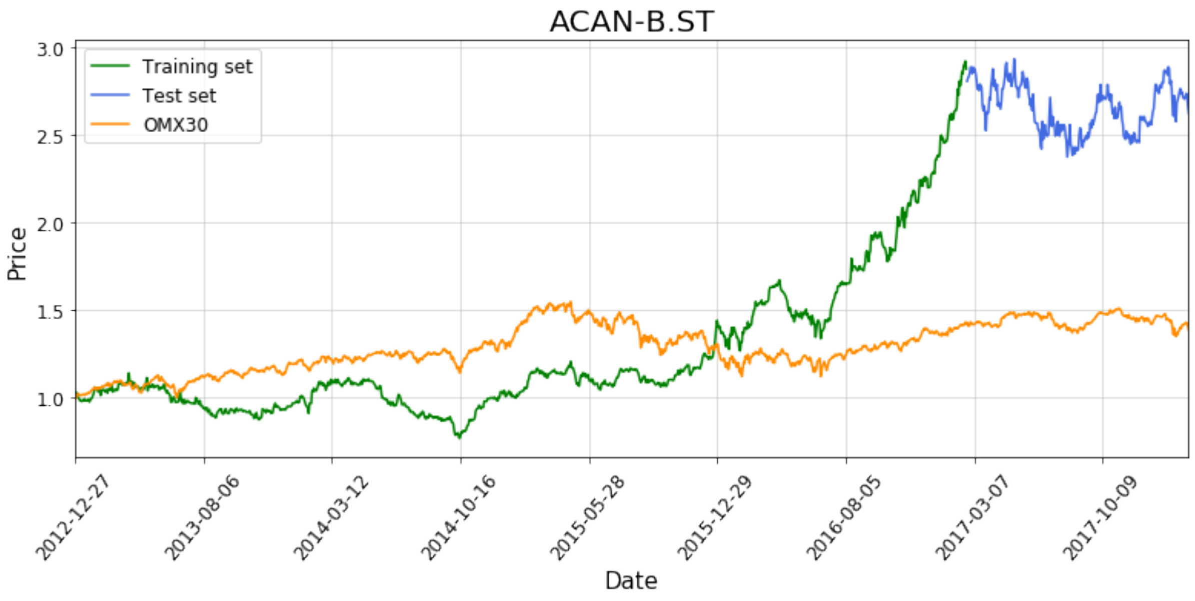
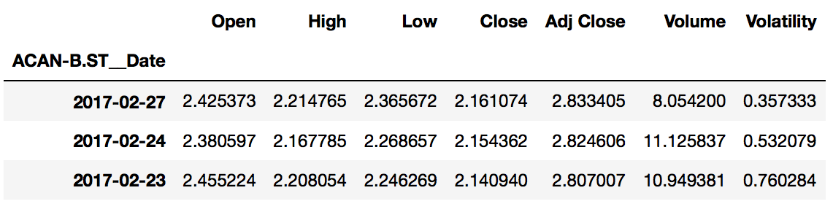


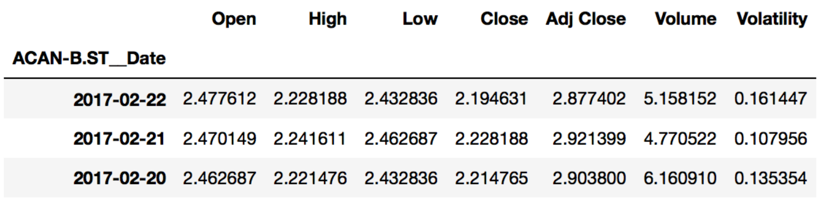
Figure 1: Visualisation of share Acando B, where green is the training set and blue is the test set. OMX30 Index is displayed in orange. All are normalised values for Adj Close.

As we can see from the plot, both graphs display continuous values. which is something our Machine Learning models require. There is a small gap between the training and test set, however this is just an ambiguity in the graph as printing the entire data frame shows us that there is no such gap in the actual data. This is displayed through Table 3, as both the last date (2017-02-22) in the training and the first date (2017-02-23) in the test sets are present.

Rather than displaying the scaled values, normalised data has been plotted. Doing so, makes it easier to identify the price differences. In the case of Acando B, the price has increased by roughly 160% (2,6 - 1,0) and for OMX30 by some 40% during the time period.



	Open	High	Low	Close	Adj Close	Volume	Volatility
ACAN-B.ST__Date							
2017-02-27	2.425373	2.214765	2.365672	2.161074	2.833405	8.054200	0.357333
2017-02-24	2.380597	2.167785	2.268657	2.154362	2.824606	11.125837	0.532079
2017-02-23	2.455224	2.208054	2.246269	2.140940	2.807007	10.949381	0.760284



	Open	High	Low	Close	Adj Close	Volume	Volatility
ACAN-B.ST__Date							
2017-02-22	2.477612	2.228188	2.432836	2.194631	2.877402	5.158152	0.161447
2017-02-21	2.470149	2.241611	2.462687	2.228188	2.921399	4.770522	0.107956
2017-02-20	2.462687	2.221476	2.432836	2.214765	2.903800	6.160910	0.135354

Table 3: Displaying that there is no gap between training set (below) and test set (above) in the actual data.

2.2 Algorithms and Techniques

2.2.1 LSTM Networks

A Long Short-Term Memory (LSTM) algorithm will be used for both the single and ten-day predictions. It was first presented by Hochreiter and Schmidhuber 1997 [14], and later improved by several other research teams. Just as the name implies, LSTM is a model for short-term memory which can last for a long period of time. It is widely used and works very well on a large amount of different problems and was explicitly designed to avoid long-term dependency issues. Through its ability to remember information over long time periods, it's commonly used for predicting time series. The input data will be fed into the LSTM as a 3D vector and therefore needs to be pre-processed before.

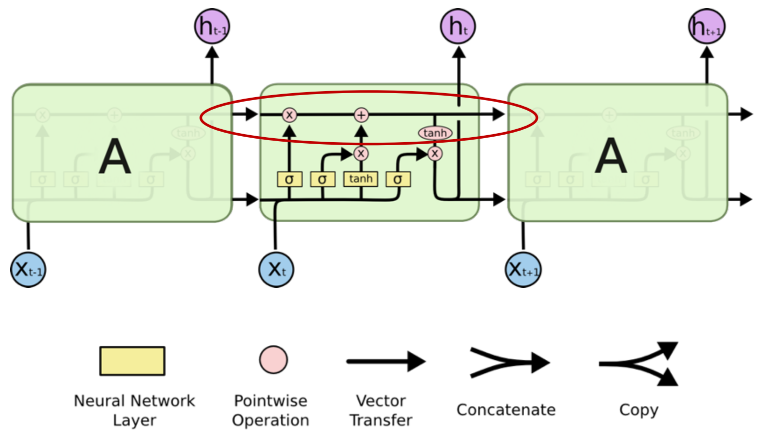


Figure 2: The LSTM chain-like structure (above) and its notation descriptions (below) [17].

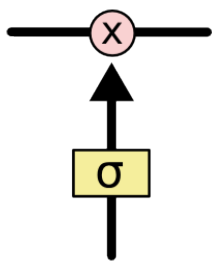


Figure 3: An LSTM gate.

Figure 2 illustrates the LSTM structure with its notations. The yellow boxes are neural network layers, like a sigmoid or tanh layer, while the pink circles represent pointwise operations, such as vector multiplication or addition. The black single lines represent vectors, carrying information from an output node to an input node, while merging and diverging lines represent concatenate and copy operations respectively.

Maybe the most important aspect of LSTMs is the horizontal line, marked with a red oval in the figure, running from left to right. It's called the *cell state*. Information can flow straight through it, virtually unchanged. However, structures called *gates*, displayed in Figure 3, can add or remove information with high precision to the cell state. A gate constitutes of a sigmoid neural network layer and a pointwise multiplication operation. It returns a number between zero and one, where "zero" means closing the gate and not letting any information through, while "one" means letting all information through. Three such gates are built into the LSTM to control and protect the cell state.

The following equations are the compact equivalents of the LSTM architecture described above [14]:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

$$h_t = o_t \circ \sigma_h(c_t)$$

Here W_z and U_z are the weight matrices and b_z is the bias vector which needs to be learned during training, where z can be either the forget gate *f*, the input gate *i*, the output gate *o* or the memory cell *c*. The operator $\circ$ refers to the Hadamard product and subscripts $_t$ refers to the time step.

2.2.2 Loss Function

What the loss function tells us is how close the predicted values are to the true ones. It is needed in order to compile the model. The loss function of choice is MSE as it takes into consideration negative numbers (by squaring them) as well as giving more weight to larger errors.

2.3 Benchmark

A logistic regression model will be trained and tested on the same data as the LSTM Network and used as benchmark. The input data is fed into the model as a 2D vector. A MSE of 0.01434 on the training set and 0.11133 on the test data is achieved. Figure 4 displays the predicted (pink train set and orange test set) as well as the true values (in green).

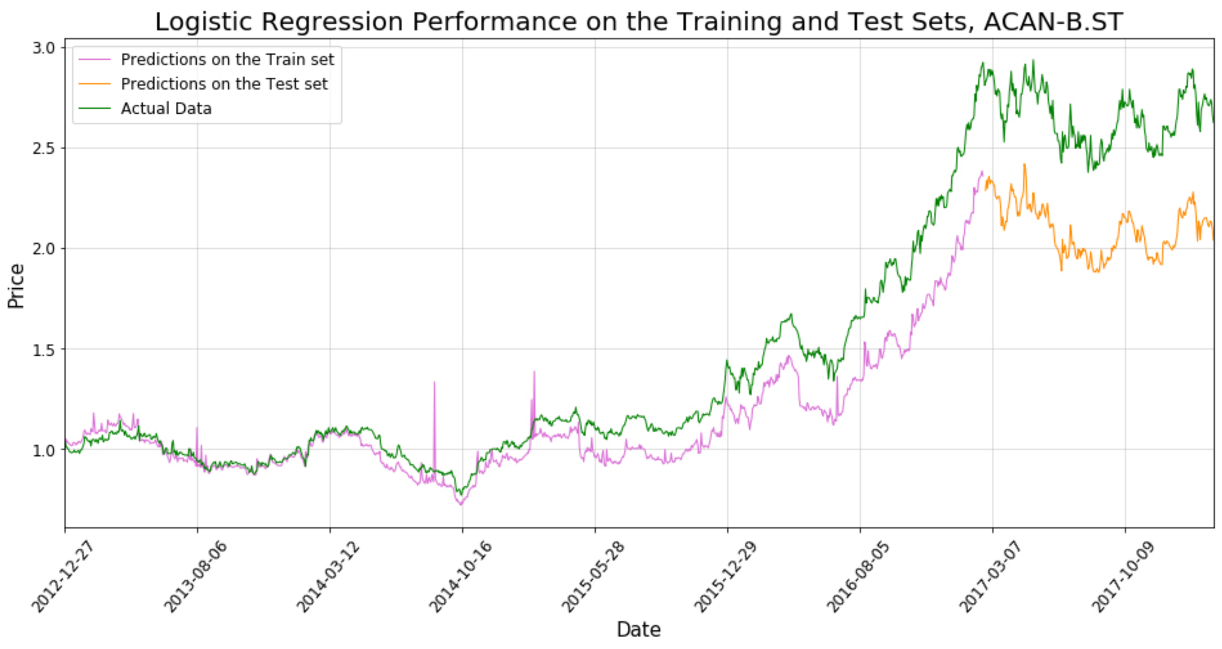


Figure 4: Logistic Regression performance on the train and test sets for stock Acando B.

3. Methodology

3.1 Data Pre-processing

First of all, the stock data needs to be downloaded. As explained earlier, this is done in an automated way through pandas' *get_data_yahoo* function. However, Yahoo! Finance seems to have become less reliable after the acquisition by Verizon in 2016 as several issues occurred during downloading. As of this, a *RemoteDataError* was returned several times when trying to do so. To mitigate the risk, a *@retry* function as well as a *try* and *except* statement was implemented, allowing ten tries with one second between each attempt. Each stock is subsequently downloaded in an individual named dataframe, values are filled forwards and backwards, respectively, through pandas' *fillna* function before they are saved in a .csv file. The code is displayed below.

```
### Download and save the data into .csv files ###
for i in tickers:
  stock_df = get_stock_data(i, dates)
  # Fill missing values forward, then, fill backward
  fill_missing_values(stock_df)
  # Save the files as .csv as well
  stock_df.to_csv(create_file_path(i))

#@retry function to minimize the risk of RemoteDataError.
# 10 retries, with one second's pause between each run.
@retry(stop_max_attempt_number=10)
def get_stock_data(ticker, dates):
  try:
    stock = data.get_data_yahoo(ticker, start, end)
    stock.sort_index(ascending=False, inplace=True)
    return stock
  except RemoteDataError:
    time.sleep(1)
    print('Trying again..')

def fill_missing_values(data_df):
  """Fill missing values forward, then backwards"""
  data_df.fillna(method="ffill", inplace=True)
  data_df.fillna(method="bfill", inplace=True)
```

Printing the data after these steps will result in what is displayed in Table 1.

The data is now stored in .csv files. The following code snip will download it into a data frame, calculate and add volatility as one of the features as well as change the index name to the stock's name + Date.

```
# Count the number of files in the input data directory
directory = '/Users/jakob/Desktop/Programming/Udacity Machine Learning Nano Degree/Capstone
Project/Data_Capstone/'
```

```
file_paths = glob(directory+"*.csv")                                # Get each .csv file in the directory

# Get all the file names
file_names = []
for root, dirs, files in os.walk(directory):
   for filename in files:
      filename = filename[:-4]                          # Just keep the ticker name, without the .csv file extention
      file_names.append(filename)
del file_names[0]

# Get the input data from the .csv files
loaded_stocks = []
for i in range(len(file_paths)):
   stock = pd.read_csv(file_paths[i], index_col='Date')
   stock['Volatility'] = (stock['High'] - stock['Low']) / stock['Open']  # Calculate the volatility
   stock.index.names = [file_names[i] + '__' + 'Date']                # Set the index name to stock name + Date
   loaded_stocks.append(stock)
```

The input data is subsequently normalised, per column, according to the first value through the *normalise_data* function. Any infinite numbers are set equal to zero and duplicated indices are removed.

```
def normalise_data(prices):
   """ Normalise data stored in prices"""
   if isinstance(prices, pd.DataFrame):     # Check if Dataframe
      prices = prices/prices.iloc[-1]          # normalise according to the first value
   else:                                       # if array
      prices = prices/prices[-1]
   return prices

# Normalize all the stock prices
norm_stock_prices = []
for i in loaded_stocks:
   norm_d = normalise_data(i)
   norm_d[norm_d == np.inf] = 0        # if any of the value in norm_d is an infinite value, set it equal to 0
   fill_missing_values(norm_d)
   norm_d = norm_d[~norm_d.index.duplicated(keep='last')]  # Remove duplicated indices (if any)
   norm_stock_prices.append(norm_d)
```

Plotting the data after these steps will result in what's displayed in Table 2.

Some more useful functions for getting the stock data as well as for plotting the data displayed in Figure 1, are also defined.

```
def get_stock(stock_list ,ticker):
   """ Returns the dataframe containing the stock with the ticker symbol ticker.
      stock_list is a list of stock data stored in dataframes. """
   for sd in stock_list:
      if sd.index.name[:-6] == ticker:
         return sd

def plot_3it(data_df1, data_df2, data_df3, label1='', label2='', label3='', title=''):
   """Plot the stock data stored in data_df1 and data_df2 in different colors.
      The entire dataset is stored in data_df3.
```

```
    label1, label2 and label3 are the label names while title is the plot title"""
  plt.figure(figsize=(10, 5))
  pl = data_df1.loc[::-1].plot(fontsize=12, figsize=(13, 5), label=label1, color='green')
  plt.plot([None for i in data_df1.loc[::-1]] + [x for x in data_df2.loc[::-1]], label=label2, color='royalblue')
  plt.plot(data_df3.loc[::-1], label=label3, color='darkorange')

  pl.set_title(label=title, fontsize=20)
  pl.set_xlabel('Date', fontsize=15)
  plt.autoscale(enable=True, axis='x', tight=True)
  pl.set_ylabel('Price', fontsize=15)
  plt.legend(fontsize=12, loc='upper left')
  plt.grid(axis='both', alpha=.5)
  pl.xaxis.set_major_locator(MaxNLocator(12))
  temp = pd.concat([data_df2, data_df1], axis=1)
  pl.xaxis.set_major_formatter(IndexFormatter(temp.index))
  plt.xticks(rotation=50, horizontalalignment='center', rotation_mode='default')
  plt.show()
```

The input data is now stored in a list, *norm_stock_prices*, with 73 indices, where each index contains a stock, stored in a data frame.

To keep the input data for the LSTM Network and the benchmark model apart, the list will be copied using *copy.deepcopy()*. This Python inbuilt function is necessary to use as it copies everything stored in the list and makes them completely independent of each other. Using the normal *copy()* function wouldn't work as changing the value in one would change the equivalent value in the other as well.

Furthermore, sklearn's *MinMaxScaler* function [15] will be used to scale the data in the interval [0,1]. Given that there are different min and max values in every column, each individual scaler will be tuned differently. As a result of this, one unique scaler for each column and stock needs to be created, resulting in a total of 511 scalers. They are stored in an array and can be accessed through the stock's name. Subsequently, the data is divided into training and test sets with the relation 80/20. Two lists will be created, one with normalised values and one with both normalised and scaled values. This will facilitate later on. The code is displayed below. The lists with the normalised and scaled stocks are named according to the Logistic Regression model, but up to this point there are no differences in the pre-processing steps between this and the LSTM model.

```
#Specify one scaler for each column and stock
many_MinMaxScalers = {}
for i_s in range(len(norm_stock_prices)):
  for j_s in range(7):
    many_MinMaxScalers["{0}".format(get_ticker(norm_stock_prices[i_s])+str(j_s))] =
MinMaxScaler(feature_range=(0,1))

def MMscale_data(data):
  """ A function for scaling the data in the dataframe data"""
  for i in range(len(data.columns)):
    data.iloc[:,i] = many_MinMaxScalers[get_ticker(data)+str(i)].fit_transform(data.iloc[:,i].values.reshape(-
1,1))
  return data

# Lists to store the normalised and scaled stocks
```

```
LOG_scaled_train_list, LOG_scaled_test_list = [], []

# Create training and test sets
for i in range(1, len(scaled_LOG_stock_prices)):
  test_size = int(len(scaled_LOG_stock_prices[i]) * 0.20)          # Specify the test set's size
  MMscale_data(scaled_LOG_stock_prices[i])

  # Scaled and normalized
  LOG_train, LOG_test = scaled_LOG_stock_prices[i][test_size:], scaled_LOG_stock_prices[i][0:test_size]
  # save each one into a list
  LOG_scaled_train_list.append(LOG_train)
  LOG_scaled_test_list.append(LOG_test)
```

As explained earlier, the input data for the LSTM Network needs to be a 3D array. In order to achieve this, the *create_LSTM_dataset* function is implemented as below. The function will also create the output data, which will be a 2D array of the Adjusted Close values, as these are the ones we wish to predict.

```
# convert an array of values into a dataset matrix.
def create_LSTM_dataset(dataset, window=10):
  dataX = [dataset[i:(i+window), :] for i in range(len(dataset)-window)]
  dataY = [dataset[j + window, 4] for j in range(len(dataset)-window)]
  return np.array(dataX), np.array(dataY)
```

The *create_LSTM_dataset* function, with window set to one, returns data equivalent to what's shown below.

```
  Input   Output
1  111     114
2  114     140
3  140     125
4  125     129
5  129     135
```

For visualisation purposes, the numbers displayed are not the actual ones. But as can be seen, the output at time *t* is equal to the input at time *t+1*.
Subsequently, the input data is reshaped to (*samples*, *window*, and *features*) using Numpy's *reshape()* function, before it is fed into the LSTM algorithm. The reshape operation is displayed below.

```
'''reshape the input to be [samples, time steps, features]'''
LSTM10_test_input = np.reshape(LSTM10_test_input, (LSTM10_test_input.shape[0],
LSTM10_test_input.shape[1], 7))

LSTM10_train_input = np.reshape(LSTM10_train_input, (LSTM10_train_input.shape[0],
LSTM10_train_input.shape[1], 7))
```

3.2 Implementation

3.2.1 Single Day Predictions with the LSTM Algorithm

The first step is to create the LSTM architecture. A Sequential model is used, followed by a LSTM recurrent layer with 20 neurons and input shape [1, 7] (one day's window size and 7 features). A 0.5 Dropout layer is included, to mitigate overfitting, before a core Dense layer, with output size one (we want to predict the Adjusted Close price for the next day) and a linear activation function is added. The code is displayed below.

```
def LSTM_model(inputs, output_size, neurons, activ_func="linear",
            dropout=0.5, loss="mean_squared_error", optimizer="adam"):
   model = Sequential()
   model.add(LSTM(neurons, input_shape=(inputs.shape[1], inputs.shape[2])))
   model.add(Dropout(dropout))
   model.add(Dense(units=output_size))
   model.add(Activation(activ_func))
   model.compile(loss=loss, optimizer=optimizer)
   model.summary()
   return model

# Create the model
model = LSTM_model(LSTM_train_input, output_size = 1, neurons=20)
```

The model is trained by calling the *fit()* function. Input parameters are, except the in- and output data, 20 epochs, one batch size and *shuffle* set to True. A 0.05 validation split is chosen, meaning that the model will use 5% of the training set to validate on. The model constitutes of a total amount of 2,261 parameters.

```
# Train the model
trained_LSTM = model.fit(LSTM_train_input, LSTM_train_output, epochs=20,
                         batch_size=1, verbose=1, shuffle=True, validation_split=0.05)
```

Some of the most difficult coding part in this section was actually to plot the predicted values. Developing the function to plot the results shown in Figure 7 was a tedious task that took many days of trial and error to get right. The axes need to make sense, the curves needed to get in line, the zoomed-in plot where calculations for min and max values needed to be calculated all took very long time. Reshaping the input data to a 3D vector for the LSTM was also a difficult task and took many days of trial and error to get right.

3.2.2 Ten-Day Predictions with the LSTM

The LSTM model for ten-day predictions is similar to the single day LSTM model but with an extra recurrent layer and Dropout added. The model constitutes of a total amount of 72,101 parameters.

```
# Define the LSTM model for 10 day predictions
def LSTM10_model(inputs, output_size, neurons, activ_func="linear",
                  dropout=0.2, loss="mean_squared_error", optimizer="rmsprop"):
  model = Sequential()

  model.add(LSTM(neurons, input_shape=(inputs.shape[1], inputs.shape[2]), return_sequences=True))
  model.add(Dropout(dropout))
  model.add(LSTM(neurons*2, return_sequences=False))
  model.add(Dropout(dropout))
  model.add(Dense(units=output_size))
  model.add(Activation(activ_func))

  model.compile(loss=loss, optimizer=optimizer)
  model.summary()
  return model

# Create the model
model_10 = LSTM10_model(LSTM10_train_input, output_size = 1, neurons=50)
```

The following function is developed to make predictions for *pred_len* amount of days (*predict_multiple_sequences*) [16].

```
def predict_multiple_sequences(model, data, window_size, pred_len):
  """ Make a sequence of predictions of pred_len steps before shifting prediction run forward by pred_len
steps."""
  prediction_seqs = []
  for i in range(int(len(data)/pred_len)):
    curr_frame = data[i*pred_len]
    predicted = []
    for j in range(pred_len):
      predicted.append(model.predict(curr_frame[np.newaxis,:,:])[0,0])
      curr_frame = curr_frame[1:]
      curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
    prediction_seqs.append(predicted)
  return prediction_seqs
```

A difficult task regarding the ten-day predictions was definitely the *predict_multiple_sequences* function displayed above. It took many days of searching and investigation before I found a previous solution (referenced to above) that I could benefit from.

3.3 Refinement

3.3.1 LSTM Single Day Predictions

Several combinations of different kind of parameters were tested before the final solution was obtained. As it turned out, using a *sigmoid* or *tanh* activation function resulted in substantial overfitting and, as a consequence, bad results on the test data. The training time was roughly the same as for the final choice. An advanced *LeakyReLU* activation layer resulted in slightly worse results (0.000262 on the train set and 0.000374 on the test set).

Adamax optimizer resulted in a higher MSE with slight overfitting and required more training time, while *RMSprop*, *Stochastic gradient decent* and *Adagrad* optimizers overall achieved worse results.

During fitting, Dropouts 0.2, 0.3, 0.4, 0.6 and 0.7 were tested out, all with worse results than the final choice of 0.5. Batch sizes of 2, 5 and 10, setting shuffle to False as well as different validation splits (0.1, 0.15 and 0.2) were tested out but with worse results. An architecture equivalent to the one used in the ten-day predictions was also tried out but with worse results in relation to training time.

It shall also be mentioned that training the model on normalised, rather than scaled, data resulted in roughly 250 times higher MSE.

3.3.2 LSTM Ten-Day Predictions

A total amount of 1,440 models were tested out for the ten-day predictions. As it turned out, each stock needed an individual model, and as a result 20 different models were tested out on the 72 stocks. This was a tedious task and required 36 hours of automated non-stop training, testing and plotting. A combination of different window lengths [10 or 20], batch sizes [1, 2, 10, 50, 100] and epochs [1, 2] were trained and compared. Slightly surprising was that fewer epochs in general resulted in better result. Figure 5 displays this as it is the final model trained with ten epochs rather than with one as is the final choice. The best model for each stock was subsequently saved for further use.

The simpler architecture, as for the single day predictions, was also tested out, but with higher errors a well as with bad visualized predictions when plotted.

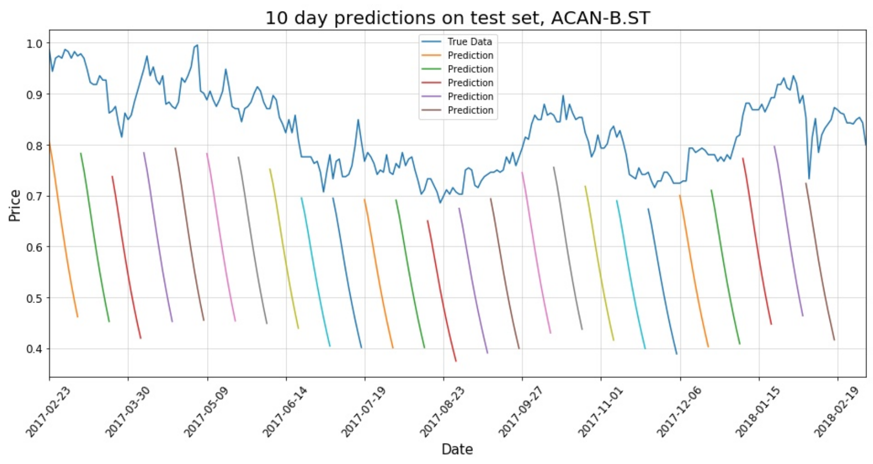


Figure 5: *LSTM ten-day predictions with ten training epochs.*

4. Results

4.1 Model Evaluation and Validation

4.1.1 LSTM Model for Single Day Predictions

After the wide combination of parameters, described in chapter 3.3.1, were tested out, the model with the lowest test error was chosen. The returned MSE on the training set was 0.000263 and 0.000363 on the test set. The error on both the training and test set is very small, allowing us to be

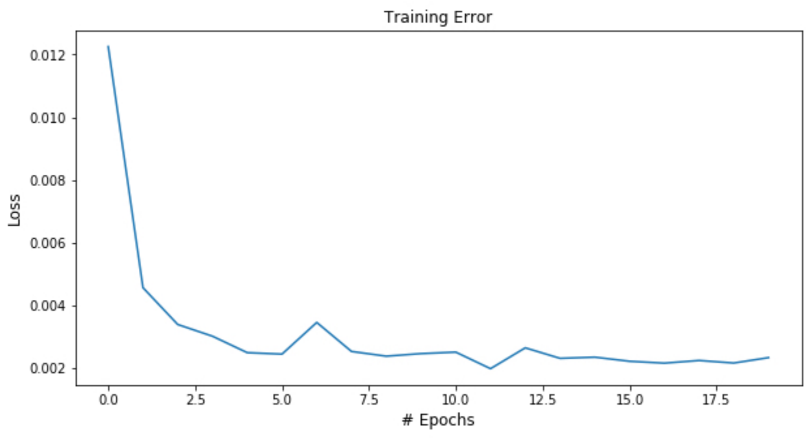


Figure 6: Training error on LSTM, single day predictions

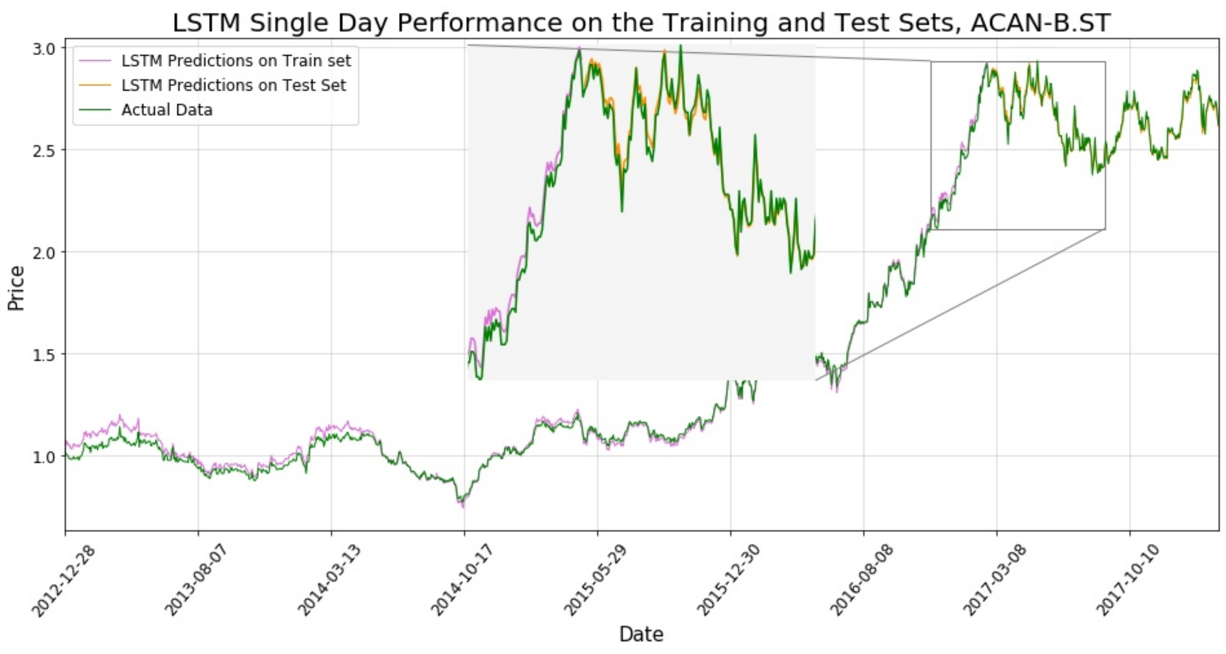


Figure 7: LSTM single day performance on the training and test sets on stock Acando B

fairly confident in our assumption that the model has not been over fitted on the train set. The training error is plotted and displayed in Figure 6. Already after 5 epochs the error has decreased significantly, with little difference the forthcoming 15 epochs.

The performance on both the train and test set is plotted in Figure 7, allowing us to confirm that the predictions matches the true data very well. There are some prediction misses in the beginning of the train data, and up to roughly May 2014, but overall, the visualised predictions are very good. The zoomed in square displays the result in higher resolution and confirms our initial observations.

Several runs with new and updated data (between the 15th of February and the 8th of March) have been tested out with similar results but some differences in training time and MSE. Comparing the reported results above (from the 2nd of March) with the ones obtained the 8th of March, the latter achieves a MSE of 0.000228 on the train set and 0.0011775 on the test set. It is a worse result, but still acceptable to be at least fairly trusted.

The same model with the same parameters can be used on all the 72 stocks. However, it needs to be trained on each one of them individually before it can return reliable predictions on that particular stock.

4.1.2 LSTM Model for Ten-Day Predictions

When it comes to the ten-day predictions, 72 slightly different models were necessary in order to return results that were, at least initially, acceptable. All the 1,440 combinations described in chapter 3.3.2 were tested out and the model with the best visual predictions, combined with the lowest MSE on the test set was hand-picked for each individual stock. That is, the model whose predictions matched the true values best, and hade the lowest MSE on the test set, was chosen. Surprisingly, the fewer number of epochs used for training, the better the results seemed to be, as displayed in chapter

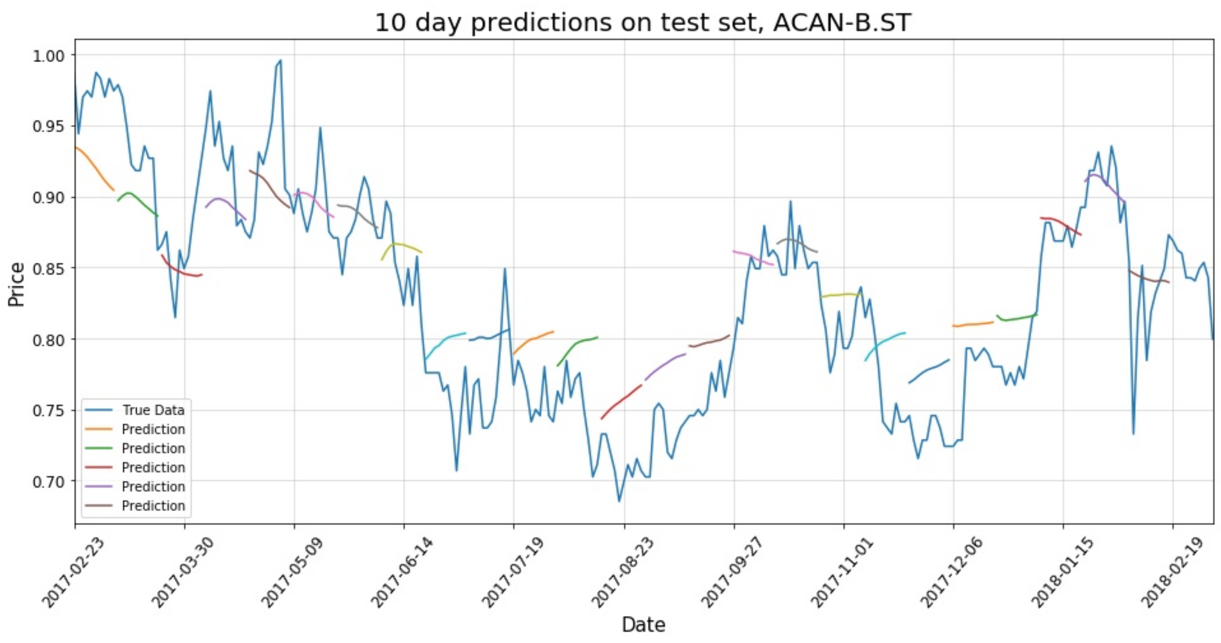


Figure 8: *LSTM ten-day predictions on the test set, one epoch, Acando B.*

3.3.2. This is very contradictory to what's usually the case for machine learning predictions and might raise some doubts concerning the models' robustness and validity.

All the models that in the end were chosen to make ten-day predictions were trained using either one or two epochs. For this reason, the training error wasn't plotted, as plotting one or two dots doesn't make much of a sense. In the case of Acando B, one epoch was used with a resulting MSE of 0.000725 on the training set and 0.001884 on the test set. Both errors are fairly small, allowing us to make the assumption that the model hasn't been over fitted. Figure 8 displays the ten-day predictions in relation to the true data for stock Acando B.

The models turned out to be very unstable and unreliable however. Training and testing the models on newer input, containing only a few extra lines of data, often resulted in something similar to what is displayed in Figure 5. This was a repeating problem, leading to the assumption that a new model needs to be found each time new data is presented. The conclusion must therefore be that the models don't generalise well on unseen data and, thus, cannot be trusted.

4.2 Justification

Comparing the MSE of the Logistic Regression's single day predictions (0,01434 on the train set and 0,11133 on the test set) with the LSTM model's single day predictions (0.000263 on the train set and 0.000363 on the test set) gives us a quite uniform picture of their performance. The Logistic Regression model returns a MSE roughly 55 times higher on the train set and 307 times higher on the test set than the LSTM model returned. This is a huge difference in performance between the two models.

Furthermore, looking at the plotted predictions (Figure 4 for Logistic Regression and Figure 7 for the LSTM model) confirms this difference visually. Not only does the LSTM model follow the true values much better, both on the train and test data, but also is it, in general, more reliable when the input data is altered or when other stock data is processed. The Logistic Regression model also predicts some complete erroneous spikes or noises in August 2014 and February 2015 roughly, as seen in the above-mentioned figure. For some stocks, this is even worse, as is the case for ANOD-B.ST for example, where the spikes' values are some 30 times higher than the actual data. This can be viewed through the following link. It happens despite the MSE values being very low.

However, the predictions returned by the single day LSTM model might be deceptive as it has all the earlier days to base its predictions on. As such, the next day's value will probably not be far from today's value. Even though the prediction will be wrong, it can still consider the true value and discard the prediction in order to make the next day's prediction. The model could by this just chose a random value, close to today's, for tomorrow's prediction, allowing the model to have virtually no substance behind it but still come up with close "predictions".

With this argument in mind, taking the much higher performing LSTM model for making ten-day predictions is an interesting analysis to make (plotted in Figure 8). The predictions don't seem to be completely random and there might actually be some substance behind the output. Many of the predictions seem reasonable. Taking the first, orange, prediction as an example, we'll find decreasing predicted values while at the same time the true values initially do so as well, before they bounce up.

The model failed to predict this last bounce up, but succeeds in catching the initial movement. The green, second, prediction seems promising as the line isn't just a straight one. Instead, it changes direction which raises some hope that the LSTM model has found some kind of pattern and isn't blindly throwing out predictions in one single direction. Nevertheless, many of the predictions are incorrect and some are completely useless, e.g. the fifth, brown, prediction. The true values are increasing intensely, while the model predicts a strong decrease.

It seems that roughly half of the predictions are fairly correct, something that, at least for me, isn't good enough in order to trust them. However, we shouldn't be too surprised that the returned predictions, based purely on a technical analysis are poor. There are so many underlying factors that influence the daily price movements. Market psychology, fundamental changes in the company, macro events, investors' decisions or market noise are just some factors that aren't considered in these predictions.

Taking these aspects into consideration, as well as those discussed in section 4.1.2, about the model's high instability, it must be concluded that the ten-day predictions aren't reliable and, thus, the model hasn't solved the initial problem.

The single day predictions for the LSTM Network outperforms the Logistic Regression model by a large margin. But there are still some concerns raised regarding the significance of the model and it can't be completely concluded that the model actually has solved the problem.

5. Conclusions

5.1 Reflections

During this project, 72 stocks in the Stockholm Mid Cap Index were downloaded along with the Stockholm OMX 30 Index. The 73 datasets were downloaded in an automated way to facilitate future updates and saved into .csv files on the computer. Subsequently, they were pre-processed by removing any disturbances such as NaN-values, before they were scaled and normalised for better model performance.

The next step was to reshape the data to a 3D vector to fit it into the LSTM Network, whose architecture was made up of two different types, one simple and one slightly more complex. The simple one was used for single day predictions, while the more complex was used for ten-day predictions. Different kind of architectures and more than 1,400 different parameter combinations were tested out to get the best performing models. For single day predictions, one model was reliable enough to be used on all the stocks, provided that it was trained on each stock before, while for the ten-day predictions an individual model for each one of the 72 stocks was needed.

The single day prediction model returned very good values, even though it can't be concluded that it solved the problem, while the ten-day predictions models were too unreliable to solve the problem.

One of the more difficult aspects in the project was actually to plot the predicted values. The *plot_LSTM* function, displayed in chapter 3.2.1, took very long time to develop as there were many small aspects to take into consideration. Clearly formatted and designed axes, get the plotted values at their correct positions, the zoomed in plot, where coordinates and locations needed to be elaborated

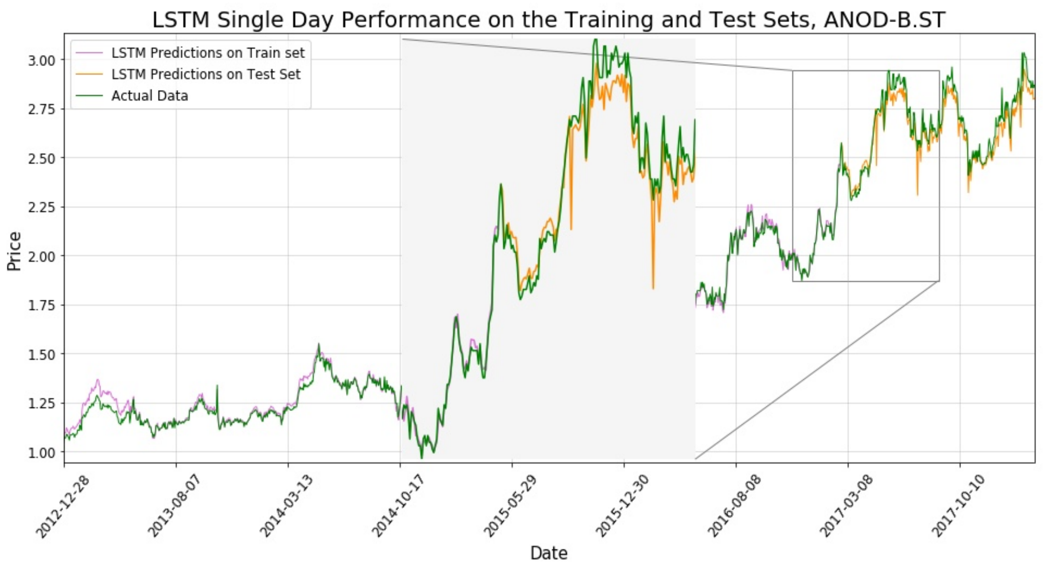


Figure 9: The function plot_LSTM used to plot the predicted values for stock Addnode Group B

and calculated properly, all took very long time to develop. Figure 9 displays the predicted values for stock Addnode Group B when this function was used. The displayed results are actually among the worse for the LSTM single day model.

Preparing the input data for the LSTM Network was another task that took a lot of time and many tries before I got it right.

The most interesting aspect was probably trying to create a model for the ten-day predictions. As it turned out, the models were way too unreliable and didn't generalise well to unseen data, but it was fun to see that the predictions weren't just straight, more or less, random lines.

I didn't have high expectations on whether I'd be able to predict the future stock prices even though there are more inefficiency on a Mid Cap Index which could be exploited. The single day model might be useful if elaborated some more, but the ten-day models should by no means be used for buying and selling on the stock market.

5.2 Improvements

I. **More features** as input data might improve the results, both engineered and downloaded, as well as longer stock history. With more input data for the model to base its predictions on the results might improve as this in general is the case in Machine Learning.

II. **Considering other input data** such as fundamental company information or market news and analyse it with deep learning. Google's Deep Mind project is doing something similar and it might add some more fundamental information to these purely technical based predictions.

III. **Try other ML algorithms** such as Facebook's Prophet algorithm, more complex LSTM architectures or other algorithms I haven't heard of. Or a combination of them.

Bibliography

[1] R. A. Mundell, “The Birth of Coinage,” Department of Economics, Columbia University, New York, 2002.

[2] D. Sheehan, “Predicting Cryptocurrency Prices With Deep Learning,” 21 11 2017. [Online]. Available: https://dashee87.github.io/deep%20learning/python/predicting-cryptocurrency-prices-with-deep-learning/. [Accessed 21 03 2018].

[3] A. C. Andersen and S. Mikelsen, “A Novel Algorithmic Trading Framework Applying Evolution and Machine Learning for Portfolio Optimization,” Department of Industrial Economics and Technology Management (IØT), Trondheim, 2012.

[4] “Yahoo! Finance,” 02 03 2018. [Online]. Available: https://finance.yahoo.com. [Accessed 02 03 2018].

[5] J. Brownlee, “machinelearningmastery.com,” 24 05 2016. [Online]. Available: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/. [Accessed 02 03 2018].

[6] J. Brownlee, “machinelearningmastery.com,” 14 08 2017. [Online]. Available: https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/. [Accessed 02 03 2018].

[7] “Core Layers,” [Online]. Available: https://keras.io/layers/core/#dense. [Accessed 03 03 2018].

[8] “Recurrent Layers,” [Online]. Available: https://keras.io/layers/recurrent/#lstm. [Accessed 03 03 2018].

[9] “Model (functional API),” [Online]. Available: https://keras.io/models/model/. [Accessed 03 03 2018].

[10] Avanza Bank Holding, “Avanza.se,” Avanza Bank Holding, [Online]. Available: https://www.avanza.se/aktier/lista.html. [Accessed 03 03 2018].

[11] Open Source, “pandas 0.22.0 documentation,” 30 12 2017. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html. [Accessed 04 03 2018].

[12] T. Balch and D. Dave, “Lesson 6, Incomplete data,” [Online]. Available: https://classroom.udacity.com/courses/ud501/lessons/3909458794/concepts/42693317700923. [Accessed 03 03 2018].

[13] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” Cornell University Library, 2015.

[14] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” München/Lugano, 1997.

[15] Pedregosa et al., “Scikit-learn: Machine Learning in Python, JMLR 12, pp. 2825-2830,,” 2011. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html. [Accessed 07 03 2018].

[16] J. Aungiers, “LSTM Neural Network for Time Series Prediction,” 21 12 2016. [Online]. Available: http://www.jakob-aungiers.com/articles/a/LSTM-Neural-Network-for-Time-Series-Prediction. [Accessed 8 03 2018].

[17] C. Olah, “Understanding LSTM Networks,” 27 08 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed 06 03 2018].