Using LSTM to Predict Changes in Stock Prices

Ali Ahmed 18L-0929

Bilal Ahmed 18L-1068

Ehtesham Arif 18L-0968

Hammad Abdullah 18L-1085

FAST-NU Lahore Campus

Abstract

Our task is to mimic the Long-short term memory management of a human brain with AI and deep learning-based system. Whenever a human is asked a question about a recent event, he/she isn’t able to answer until it’s something special, e.g. (what we ate a week ago). However, we do remember what was made last Eid. With the help of AI algorithms our system will be able to distinguish data elements, to store in long term memory or short-term memory. In this report, we use the same model to predict the changes in stock process over the span of years. Our goal is to enhance system capabilities by omitting trivial information (that may be required for a specific duration) to increase system efficiency.

Contents

Using LSTM to Predict Changes in Stock Prices

LSTM is a type of neural network which is basically an advanced form of RNN (Recurrent Neural Network). RNN is a variation of ANN (Artificial Neural Network) which basically increases the accuracy of the datasets of any type. First this model divides the set into test and train then uses its 3 gates in order to memorize the data in order to use this memory in future according to its weight. In our project we are designing a software application which first trains the Stock data and then tests the remaining sample. After that we add multiple hidden layers with the variation of different parameters in order to reduce the error of our model. This model basically tells us regarding future analysis of stock data of any company. This model also works on sudden and huge changes of data. Moreover, LSTM can work on any kind of data like images, videos, voice notes and any other kind.

Literature Review

[The first two heading levels get their own paragraph, as shown here. Headings 3, 4, and 5 are run-in headings used at the beginning of the paragraph.]

Solution 1 by Siddharth Yadav

[1] This implementation of LSTM is made for use in various tasks and applications such as time series data, such as stock prices

- Component of LSTM Cell in implementation
- Forget Gate “f” (a neural network with sigmoid)
- Candidate layer “C" (a NN with Tanh)
- Input Gate “I” (a NN with sigmoid)
- Output Gate “O” (a NN with sigmoid)
- Hidden state “H” (a vector)
- Memory state “C” (a vector)

- Inputs to the LSTM cell at any step are Xt (current input), Ht-1 (previous hidden state) and Ct-1 (previous memory state).
- Outputs from the LSTM cell are Ht (current hidden state) and Ct (current memory state)

Working of gates in LSTM

First, LSTM cell takes the previous memory state Ct-1 and does element wise multiplication with forget gate (f) to decide if present memory state Ct. If forget gate value is 0 then previous memory state is completely forgotten else if forget gate value is 1 then previous memory state is completely passed to the cell (Remember f gate gives values between 0 and 1).

Ct = Ct-1 * ft

Calculating the new memory state:

Ct = Ct + (It * C`t)

Now, we calculate the output:

Ht = tanh (Ct)

Code Implementation

This implementation uses LSTMs for predicting the price of stocks of IBM for the year 2017

Libraries Used:

- numpy
- matplotlib.pyplot
- pandas
- MinMaxScaler
- Sequential
- Dense, LSTM, Dropout, GRU, Bidirectional
- SGD
- math

The implementation reads data from a csv file and plots the graph for the read data. It then checks for missing values from both the model testing set and the test set. After scaling the training set, for each element of the training set there are 60 previous training set elements since the LSTM stores long term memory a data structure with 60 timesteps and 1 output is made.

The LSTM architecture is as follows

```
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))
 # Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```

After compiling, the model predicts the scores and plots them onto a graph.

Solution 2 by Pablo Castilla

[2] This implementation is to predict if a stock will rise based on previous information

Code Implementation

This implementation uses a Python notebook using data from New York Stock Exchange

Libraries Used:

- numpy
- matplotlib.pyplot
- pandas
- MinMaxScaler
- Time
- LSTM
- Sequential
- train_test_split
- Dense, Activation, Dropout
- check_output

The implementation reads data from a csv file, plots them and creates datasets with the data.

Then it builds the model:

```
model = Sequential()
model.add(LSTM(
        input_dim=1,
        output_dim=50,
        return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(
        100,
        return_sequences=False))
model.add(Dropout(0.2))

model.add(Dense(
        output_dim=1))
model.add(Activation('linear'))
start = time.time()
model.compile(loss='mse', optimizer='rmsprop')
print ('compilation time : ', time.time() - start)
```

The model after being compiled is used to predict results and plot them on a graph.

Solution 3 by Thushan Ganegedara

[3] This implementation reads data from a csv file and plots the data. It then splits the data into a training and test set. After scaling and normalizing the training data is “smoothed”. Then the implementation using an Averaging Mechanism, Standard average, which is; First you will try to predict the future stock market prices (for example, xt+1) as an average of the previously observed stock market prices within a fixed size window (for example, xt-N, ..., xt) (say previous 100 days). Thereafter you will try a bit more fancier "exponential moving average" method and see how well that does. Then you will move on to the "holy-grail" of time-series prediction; Long Short-Term Memory models.

First you will see how normal averaging works. That is you say,

In other words, you say the prediction at t+1t+1 is the average value of all the stock prices you observed within a window of tt to t−Nt−N.

Libraries Used:

Code Implementation

Libraries Used:

- numpy
- matplotlib.pyplot
- pandas
- MinMaxScaler
- datetime
- urlib.request
- tensorflow
- os

```
window_size = 100

N = train_data.size

std_avg_predictions = []
```

```
std_avg_x = []

mse_errors = []

for pred_idx in range(window_size,N):

        if pred_idx >= N:

        date = dt.datetime.strptime(k, '%Y-%m-%d').date() + dt.timedelta(days=1)

        else:

        date = df.loc[pred_idx,'Date']

   std_avg_predictions.append(np.mean(train_data[pred_idx-window_size:pred_idx]))

   mse_errors.append((std_avg_predictions[-1]-train_data[pred_idx])**2)

        std_avg_x.append(date)

print('MSE error for standard averaging: %.5f'%(0.5*np.mean(mse_errors)))
```

After this, the model is used to test and predict values and plot them on a graph

Solution 4 by Mrinal Walia

[4] This implementation makes a LSTM Model in python

Code Implementation

Libraries Used:

- numpy
- matplotlib.pyplot
- pandas
- MinMaxScaler
- math
- Sequential
- Dense, LSTM

This implementation splits the data set into a 75:25 ration for train and test data set. After scaling and readjustment of the data sets. The LSTM architecture is as follows:

```
# Initialising the RNN
model = Sequential()
model.add(LSTM(units = 50, return_sequences = True, input_shape = (x_train.shape[1], 1)))
model.add(Dropout(0.2))
# Adding a second LSTM layer and Dropout layer
model.add(LSTM(units = 50, return_sequences = True))
model.add(Dropout(0.2))
# Adding a third LSTM layer and Dropout layer
model.add(LSTM(units = 50, return_sequences = True))
model.add(Dropout(0.2))
# Adding a fourth LSTM layer and and Dropout layer
model.add(LSTM(units = 50))
model.add(Dropout(0.2))
# Adding the output layer
# For Full connection layer we use dense
# As the output is 1D so we use unit=1
model.add(Dense(units = 1))
```

It then compiles the data and the test data set is used to check predicted values.

Proposed Methodology

Working of LSTM Cell

[5] The function of LSTM is to mimic how a human brain handles, filters and retains information for short or long period of time. The LSTM architecture should be able to retain short-term memory and long-term memory separately but should be allowed to alter each other's

state. This architecture needs to retain its state at a current timestamp which will eventually be utilized to process information at a later timestamp.

LSTM's architecture will extend the RNN's architecture which helps in processing new information using previous information state but fails to retain long-term dependency information. LSTM will be able to overcome this weak area as it will retain both short-term and long-term memory.

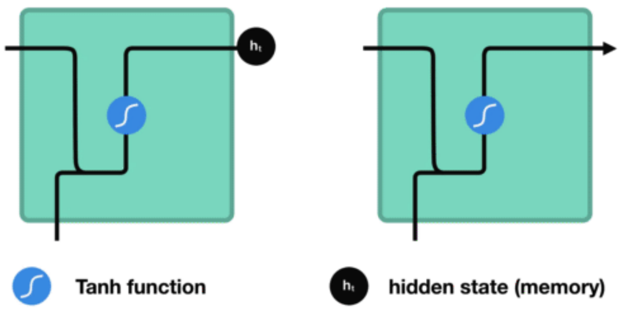


Figure 1: RNN Architecture

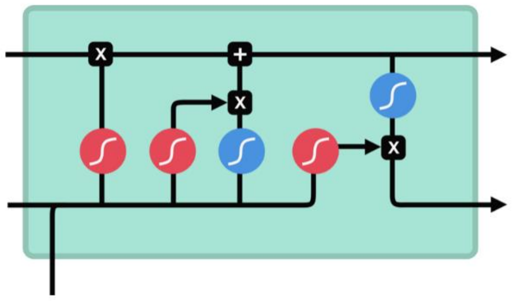


Figure 2: LSTM Architecture

It is evident from the above illustrations that in RNN there is only two input flows of information, namely Previous cell state and new input. On the other hand, LSTM network supports three input flows of information, namely Long-term state, Short-term state, and New input.

The cell state act as a transport highway that transfers relative information all the way down the sequence chain. The cell state carries relevant information throughout the sequence which allows that information to be processed at later time stamps.

The network, whichever it maybe RNN or LSTM, process information in the form of vectors. Following operations allow LSTM to retain or forget information by altering or manipulating vectors.

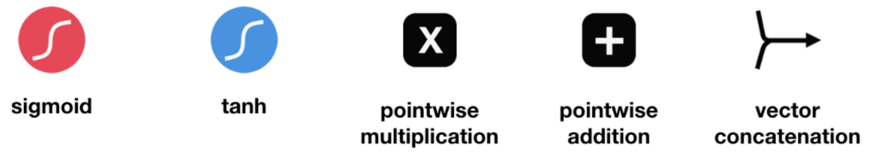


Figure 3: LSTM Operations

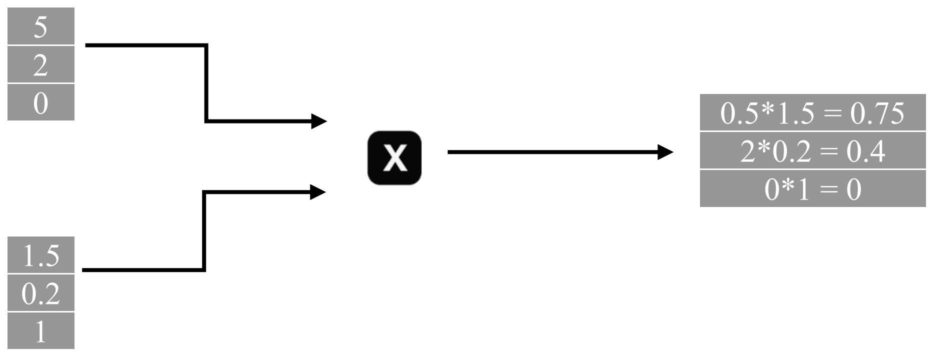
1. Tanh

The tanh activation is used to help regulate the vectors flowing through the network. The tanh function squishes values to always be between -1 and 1.

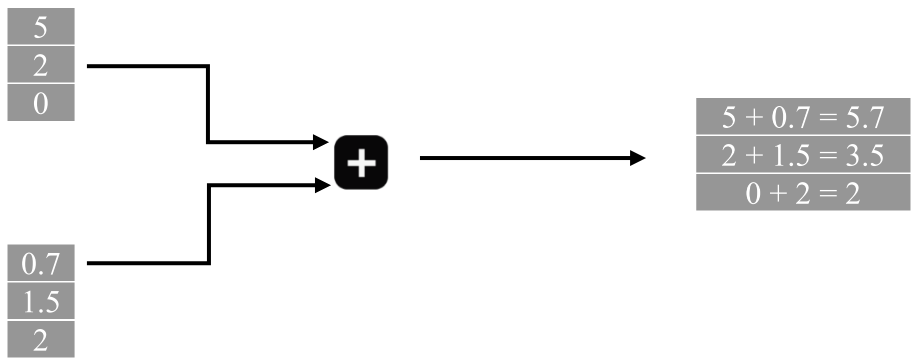
2. Sigmoid

A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1.

3. Pointwise Multiplication



4. Pointwise Addition



The LSTM have three different gates that regulate information flow in a cell. A Forget Gate, Input Gate, Output Gate.

The upper flow of information is called Cell State (Long-term Memory) and the lower flow of information is called Hidden State (Short-term Memory).

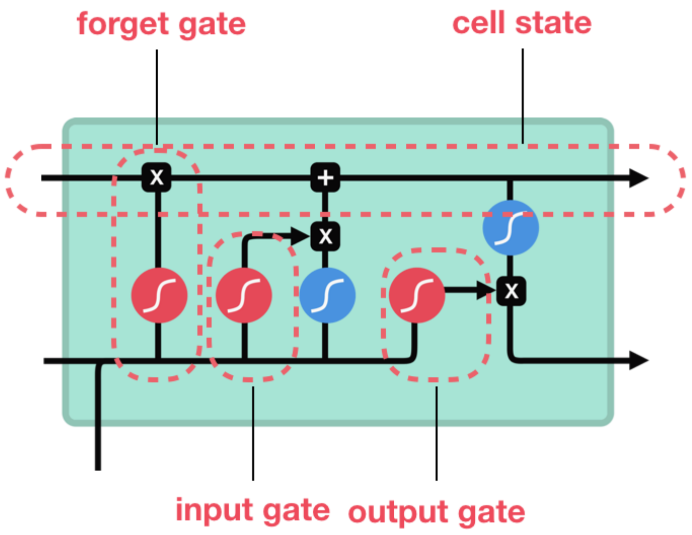


Figure 4: LSTM Gates

1. Forget Gate

The Forget gate decides what information should be retained or forgotten in the cell state. Information from the previous hidden state and current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

The sigmoid function output is pointwise-multiplied with cell state which alters the cell state vector – information in long-term memory is retained or forgotten.

2. Input Gate

Input gate is used to update cell state (Long-term Memory). First, the previous hidden state and current input are passed into a sigmoid function. Values come out between 0 and 1 – closer to 0 means information is not important and closer to 1 means information is important.

The hidden state and current input are also passed into the tanh function to squish values between -1 and 1 to help regulate the network. Then it pointwise-multiplies the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

Finally, pointwise-multiplication output is updated to cell state using pointwise-addition – new important information is added to long-term memory.

3. Output Gate

The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions.

The previous hidden state and current input are passed into a sigmoid function. Then, the updated cell state is passed to a tanh function. The sigmoid output is pointwise-multiplied with tanh output to update hidden state – information in short-term memory is retained or forgotten.

Finally, the new Cell State and Hidden State are then sent to next cells.

LSTM Model

LSTM's architecture has been implemented in a python library called "keras.layers". Thus, there was no need for implementing the architecture of a LSTM cell but a skeleton code was needed to create a model which would further needed to be modified and improved in order for us to train and test the model for predicting changes in the valuation of a stock at stock exchange.

Therefore, a basic level skeleton model was obtained from an online source which was then studied to understand its working flow and to improve the model. Consequently, we added extra hidden layers and altered parameters to reduce errors and improve its accuracy.

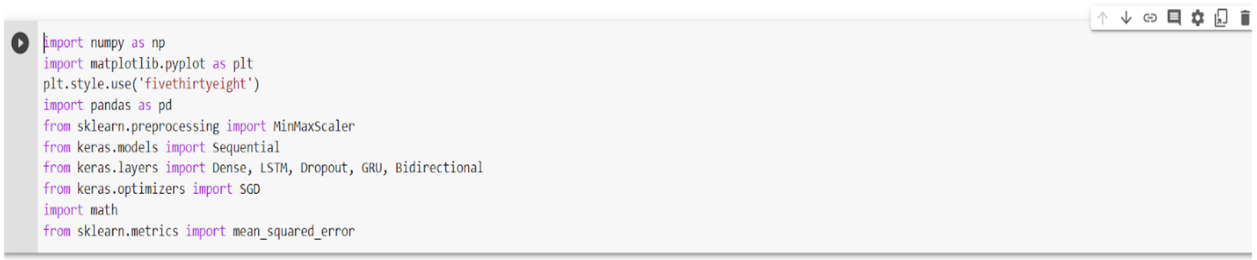
```
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
```

The above function was used to add extra hidden layers in the model to achieve more accurate results.

Implementation

Step 1- Built in Libraries

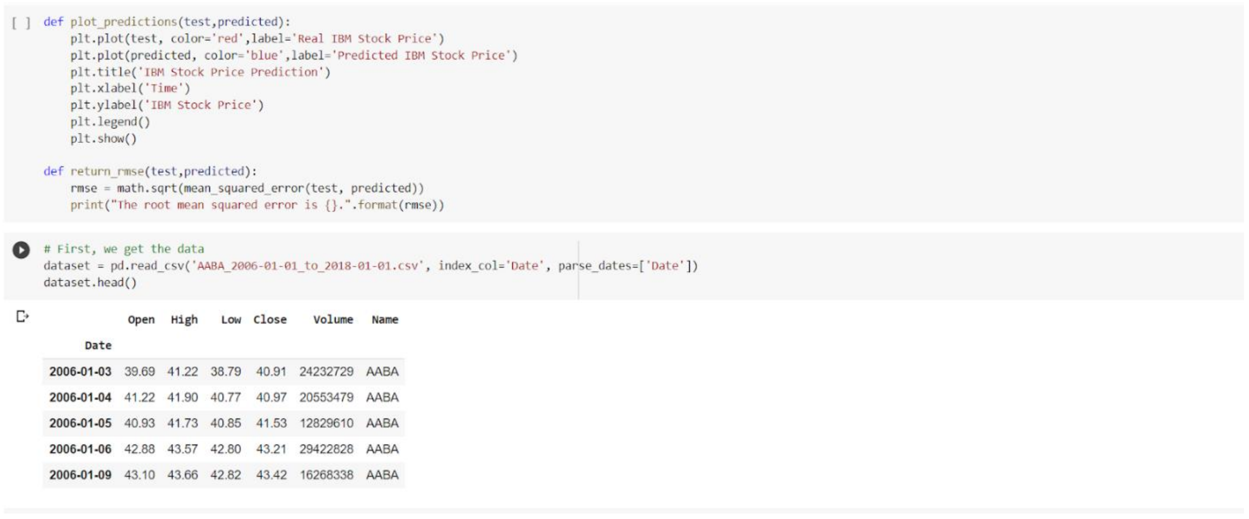
For the implementation we need some built in libraries of python in order to complete our system.



```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error
```

These libraries have their own functionalities in order to implement the machine learning projects.

Step 2- Import Data set.



```
def plot_predictions(test,predicted):
    plt.plot(test, color='red',label='Real IBM Stock Price')
    plt.plot(predicted, color='blue',label='Predicted IBM Stock Price')
    plt.title('IBM Stock Price Prediction')
    plt.xlabel('Time')
    plt.ylabel('IBM Stock Price')
    plt.legend()
    plt.show()

def return_rmse(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))
```

```
# First, we get the data
dataset = pd.read_csv('AABA_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=['Date'])
dataset.head()
```

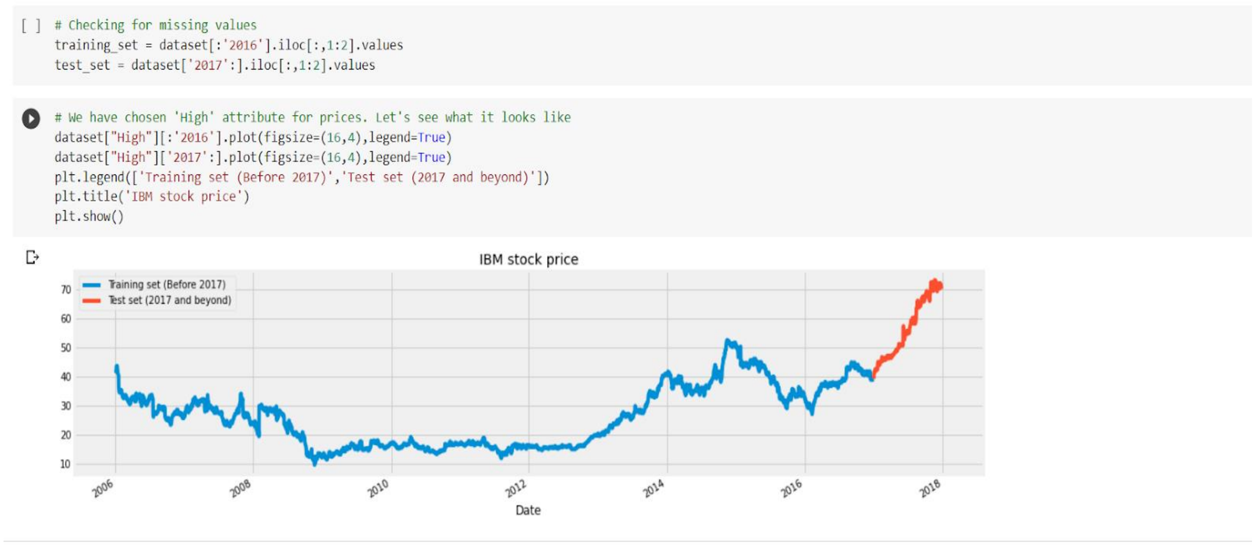
```
             Open   High    Low  Close    Volume  Name
      Date
2006-01-03  39.69  41.22  38.79  40.91  24232729  AABA
2006-01-04  41.22  41.90  40.77  40.97  20553479  AABA
2006-01-05  40.93  41.73  40.85  41.53  12829610  AABA
2006-01-06  42.88  43.57  42.80  43.21  29422828  AABA
2006-01-09  43.10  43.66  42.82  43.42  16268338  AABA
```

Step 3- Make Two parts of data set

In machine learning we have always made 2 data sets in order to complete our task or projects.

- Training Data set
- Test data set

So, in this case we have 2 data sets. Based upon first we train our data so that we can use these results in order to test data. For testing of data we have some part of remaining data. On which we can do experiments and then conclude;

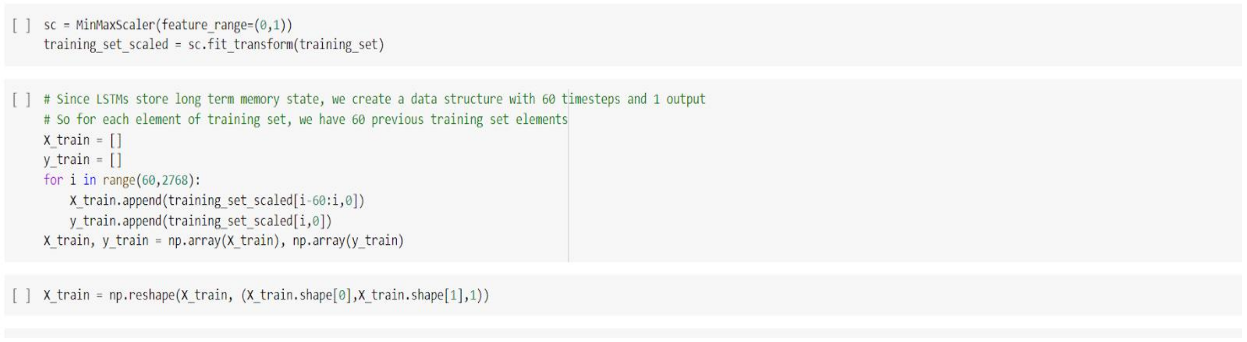


```
# Checking for missing values
training_set = dataset[:'2016'].iloc[:,1:2].values
test_set = dataset['2017':].iloc[:,1:2].values
```

```
# We have chosen 'High' attribute for prices. Let's see what it looks like
dataset["High"][:'2016'].plot(figsize=(16,4),legend=True)
dataset["High"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```

Step 4- Training of data

In this step we train our data by reshaping it in coordinates form so that we can use this trained data in our testing experiments.



```
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

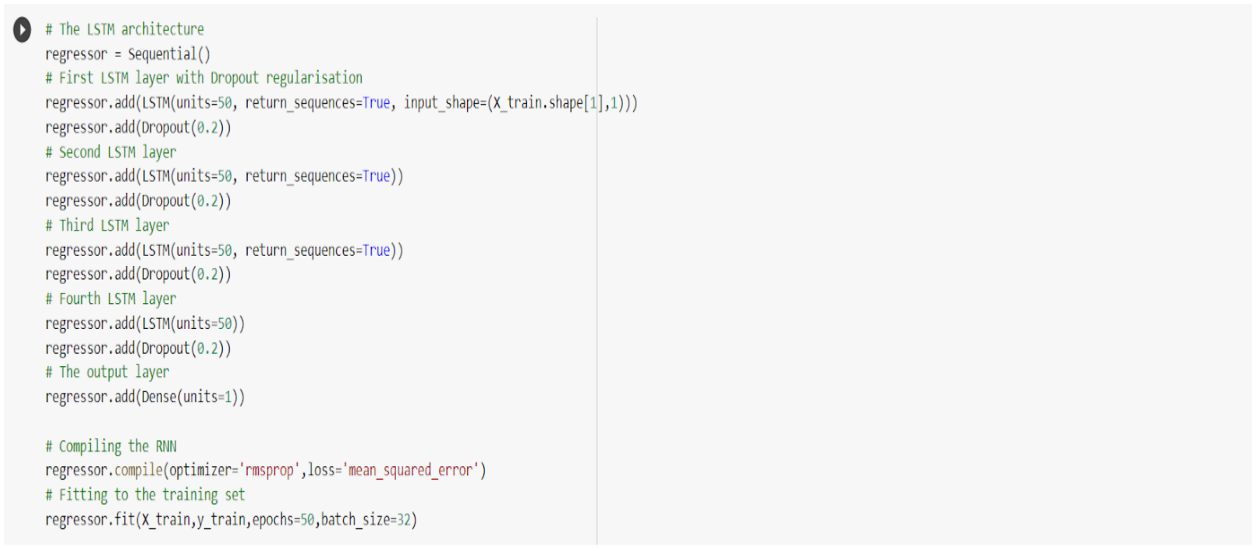
```
# Since LSTMs store long term memory state, we create a data structure with 60 timesteps and 1 output
# So for each element of training set, we have 60 previous training set elements
X_train = []
y_train = []
for i in range(60,2768):
    X_train.append(training_set_scaled[i-60:i,0])
    y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
```

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

Step 5- Applying LSTM Architecture on Training set

Now take the training data set and apply LSTM formulas on it. How LSTM works was discussed earlier. Furthermore, we are using several layers of LSTM in this part in order to make our calculations more effective towards final results.

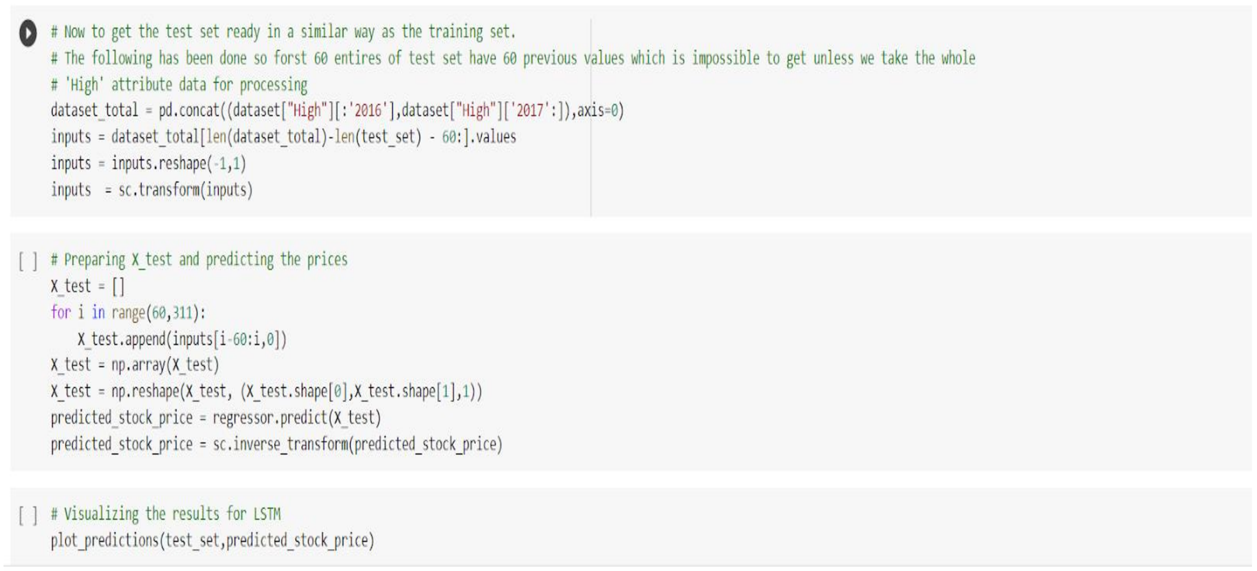
Apart from limiting our units in every layer so that this will work with accuracy, we are also using regression with dense no of units and dropout fractions with are basically key factors of the regressor function in order to apply different LSTM layers to our trained data.



```
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```

Step 6- Filter Useful data and Testing of data

Now our last step is to filter the useful data from the dataset and then test this data in order to plot our final results. These results are based on predictions which we conclude on the basis of our trained data.



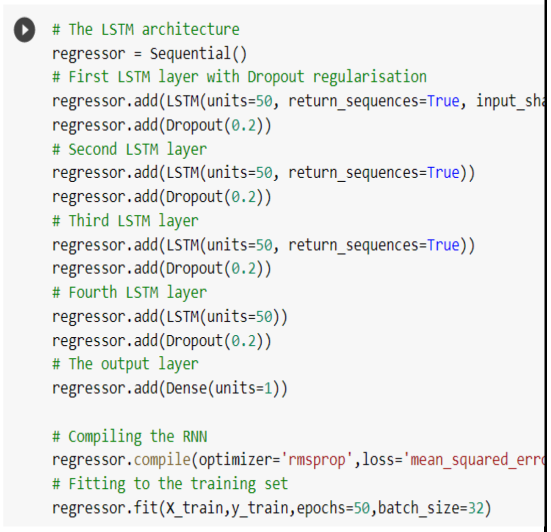
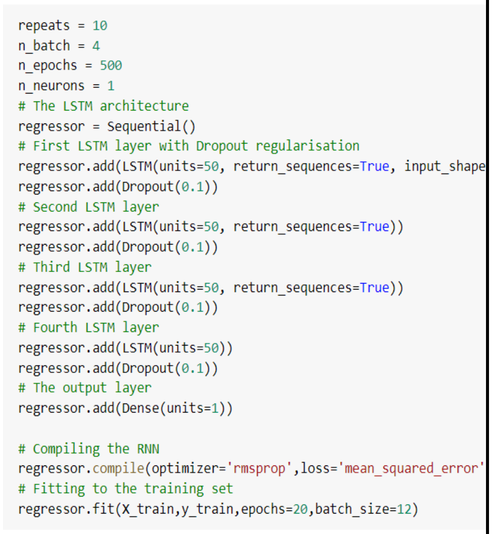
```
# Now to get the test set ready in a similar way as the training set.
# The following has been done so forst 60 entires of test set have 60 previous values which is impossible to get unless we take the whole
# 'High' attribute data for processing
dataset_total = pd.concat((dataset["High"][:'2016'],dataset["High"]['2017':]),axis=0)
inputs = dataset_total[len(dataset_total)-len(test_set) - 60:].values
inputs = inputs.reshape(-1,1)
inputs  = sc.transform(inputs)
```

```
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

```
# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price)
```

Experiments

We did two types of experiments. One is with the default model and the other with an improved model. Results showed that the improved model had better results;

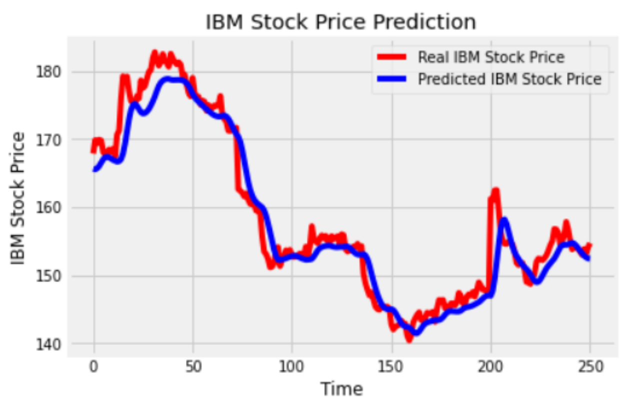
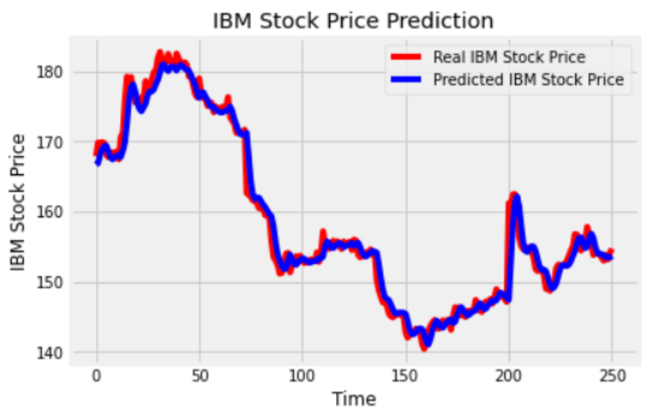
Default Model	Improved Model
	

```
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_sha
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_err
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```

```
repeats = 10
n_batch = 4
n_epochs = 500
n_neurons = 1
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape
regressor.add(Dropout(0.1))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.1))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.1))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.1))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error'
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=20,batch_size=12)
```

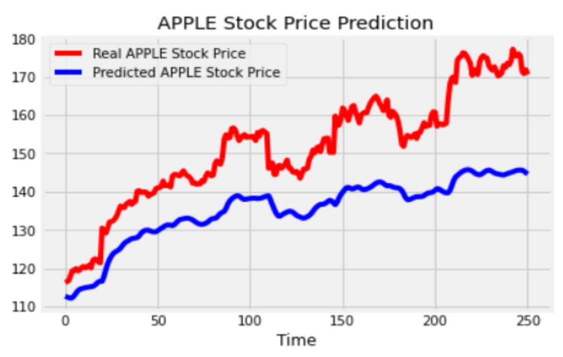
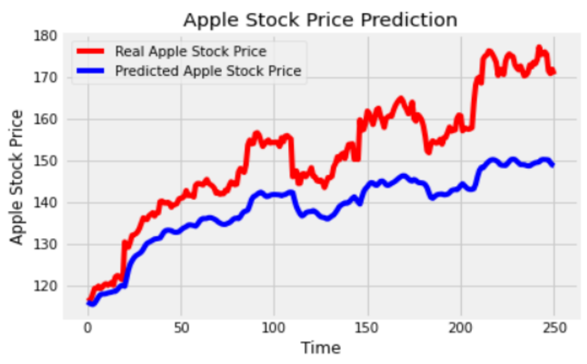
Data_set1: IBM Dataset

Default Model	Improved Model


`return_rmse(test_set,predicted_stock_price)`
`The root mean squared error is 3.0946246115428577.`
	
	`return_rmse(test_set,predicted_stock_price)`
	`The root mean squared error is 1.9054240910163065.`

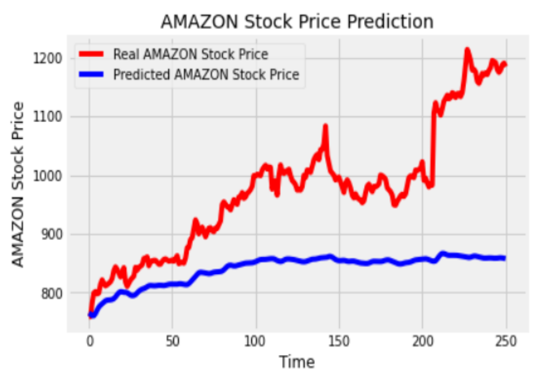
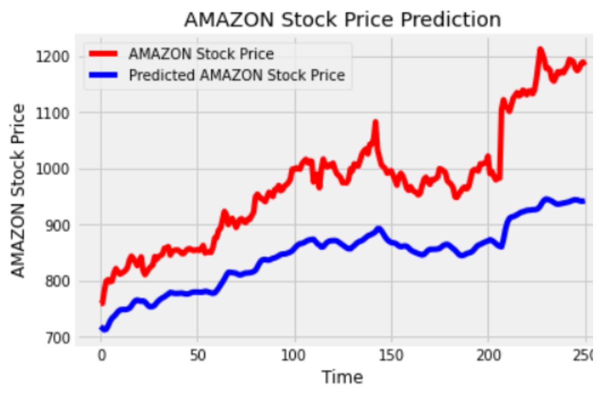
Data_set2: APPLE Dataset

Default Model	Improved Model


`turn_rmse(test_set,predicted_stock_price)`
`e root mean squared error is 17.6443291553704.`
	
	`return_rmse(test_set,predicted_stock_price)`
	`The root mean squared error is 14.27684775881528.`

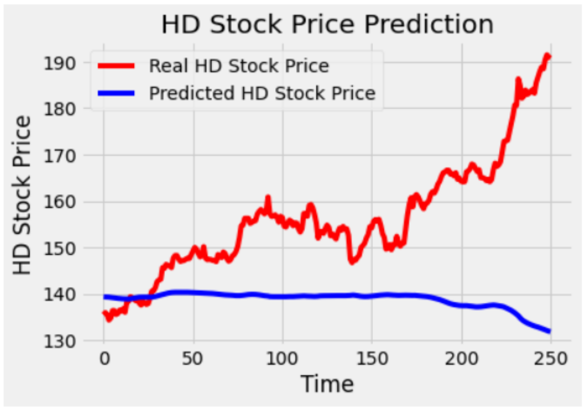
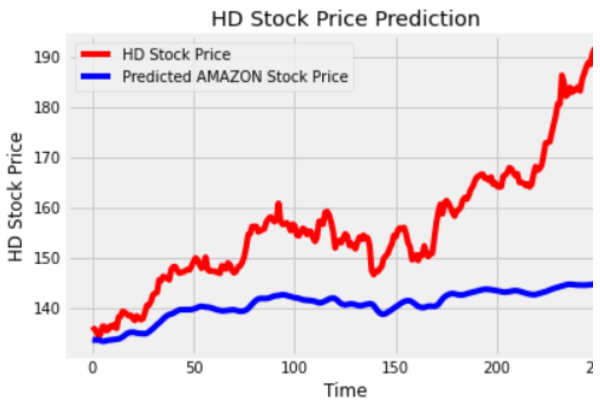
<u>Data_set3: AMAZON Dataset</u>

Default Model	Improved Model


`return_rmse(test_set,predicted_stock_price)`
`The root mean squared error is 160.6340946372162.`
	
	`return_rmse(test_set,predicted_stock_price)`
	`The root mean squared error is 140.33133301086835.`

<u>Data_set4: HD Dataset</u>

Default Model	Improved Model


`return_rmse(test_set,predicted_stock_price)`
`The root mean squared error is 22.490850762690812.`
	
	`return_rmse(test_set,predicted_stock_price)`
	`The root mean squared error is 18.48782820158653.`

Analysis

As it can be observed the LSTM model has made predictions on the basis of trained data on the test data. But there is always some error between actual data and data obtained from analysis of trained data. In the case of RNN we have an error factor, and that value is removed in LSTM layers by layers so that we can get accurate results.

Now there is mean square error in LSTM which depends on the training of data where less error means high accuracy in test data. We can observe from Experimentations that the minimum error decreased is 5% while the maximum is 25%. Accuracy has also been significantly improved.

Conclusion

We have predicted the stock prices using LSTM (Long Short-Term Memory) which is a RNN (Recurrent Neural Network). We have also discussed the solutions that are already available (with implementation). Our findings have been discussed in the Experiments section above, which shows that the result/ accuracy of our problem is significantly increased by using LSTM and tuning its parameters. Finally, a brief analysis is done, which concludes our report.

References

[1] https://www.kaggle.com/thebrownviking20/intro-to-recurrent-neural-networks-lstm-gru

[2] https://www.kaggle.com/pablocastilla/predict-stock-prices-with-lstm

[3] https://www.datacamp.com/community/tutorials/lstm-python-stock-market

[4] https://datascienceplus.com/long-short-term-memory-lstm-and-how-to-implement-lstm-using-python/

[5] https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21