Capstone Project

Machine Learning Engineer Nanodegree

Parameswaran Iyer

March 18, 2019

Definition

Project Overview

Today almost every developed and developing countries have stock exchanges where in millions of transactions happens on a daily basis. A profit is made by the difference in the buying price and the selling price.

This project involves a machine learning model which predicts the closing stock price of a particular company for the next day using the data for previous days as inputs.

We can find similar researches in the following links:

- https://www.sciencedirect.com/science/article/pii/S1877050918307828
- https://www.researchgate.net/publication/259240183_A_Machine_Learning_Model_for_Stock_Market_Prediction
- https://acadpubl.eu/jsi/2017-115-6-7/articles/8/12.pdf

Problem Statement

The problem at hand is to *predict the closing stock price of a company using the historical data for analysis and data of previous n days as inputs.*

For solving the problem, the data set would be data for Yahoo! Finance [1] which contains the following columns:

- Date – The date for which the data is given.
- Open – The opening price of the stock for given date.
- High – The highest price of stock on that given date.
- Low – The lowest price of stock on that given date.
- Close – The closing price of the stock for given date.
- Adjusted Close – The close price which is adjusted according to the announcement of the company after the stock market closes.
- Volume – The number of stock traded on that given date.

To build the stock price predictor, we would use neural network for which we primarily have 2 models, first would be a simple neural network with Dense and Flatten Layers with RELU activation and the other would be a Recurring Neural Network consisting of LSTM architecture.

Metrics

The model will be evaluated with Mean Squared Error, Mean Absolute Percentage Error and Mean Absolute Error

Mean Squared Error [2]:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Mean Absolute Error [3]:

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

Mean Absolute Percentage Error [4]:

$$\mathrm{MAPE} = \frac{\sum \frac{|\mathrm{A-F}|}{\mathrm{A}} \times 100}{\mathrm{N}}$$

The above metrics were chosen because they give us an error value which lets us know the divergence of our output from the actual value. Minimizing this error in turn increases the probability of our model to predict the value which is much closer. It should be noted that it is near impossible to predict the exact value and hence accuracy metric will almost always return 0.
Mean Squared Error was selected because it returns a larger value if the error is large and a smaller value for small amount of error. At the same time, mean squared error converges quickly and hence it is used as loss function.
Mean Absolute Error was used to know what is the mean error i.e. the mean difference between the predicted and actual value.
Mean Absolute Percentage Error was used mainly because normalization of the data would not affect percentage error and it would be easier to analyze.

Data Exploration and Visualization

For this project, the data is taken from Yahoo! Finance [1]. Unfortunately, because of lack of official API any longer, we are using a third-party package called as 'fix-yahoo-finance' [5] which takes the data from the Yahoo! Finance webpage and returns the same. The data will be taken from 01-01-1997 to 31-12-2017. The dataset consists of 5196 non-null rows.

There are 3 inputs which is necessary for accessing the data using the package:

1. The ticker for the company whose stock price needs to be predicted. This has to be the same as that in Yahoo! Finance webpage for the corresponding ticker.
2. Start date – Date from which the data needs to be provided.
3. End date – Date till which the data needs to be provided.

Default value for Start and End date is provided as 01-01-1997 and 31-12-2017 respectively.

The dataset contains the following columns:

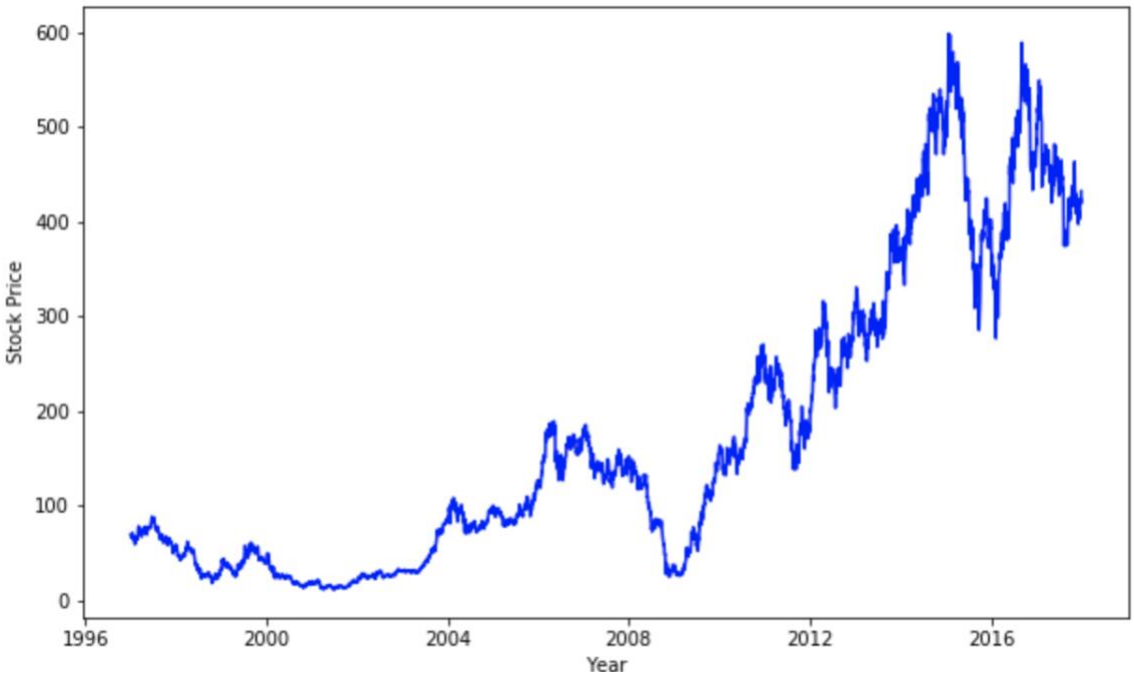
- Date – The date for which the data is given.
- Open – The opening price of the stock for given date.
- High – The highest price of stock on that given date.
- Low – The lowest price of stock on that given date.
- Close – The closing price of the stock for given date.
- Adjusted Close – The close price which is adjusted according to the announcement of the company after the stock market closes.
- Volume – The number of stock traded on that given date.

For example, the data set for the company TATAMOTORS looks as follows:

Date	Open	High	Low	Close	Adj Close	Volume
1997-01-01	65.73	69.18	65.54	69.06	15.61	11680285
1997-01-02	69.56	69.56	67.65	68.36	15.45	12850766
1997-01-03	67.65	68.98	67.48	68.71	15.53	7998983
1997-01-06	68.03	69.03	67.28	67.74	15.31	5613143
1997-01-07	67.60	67.60	65.39	65.91	14.90	11466071
1997-01-08	66.97	68.41	66.54	67.00	15.15	5246292
1997-01-09	67.00	67.55	66.30	66.79	15.10	5631408
1997-01-10	67.45	68.97	67.28	68.52	15.49	9725743
1997-01-13	68.03	68.03	66.94	67.65	15.29	4230535
1997-01-14	68.03	68.35	65.74	66.40	15.01	6508877

	Open	High	Low	Close	Adj Close	Volume
count	5196.000000	5196.000000	5196.000000	5196.000000	5196.000000	5.196000e+03
mean	173.379588	176.044580	170.397652	173.147794	148.808903	1.113871e+07
std	156.357692	158.197767	154.256158	156.161565	167.639822	9.562883e+06
min	11.110000	11.630000	11.030000	11.270000	4.030000	0.000000e+00
25%	44.080000	45.000000	43.055000	43.935000	14.487500	5.083488e+06
50%	120.245000	122.580000	117.940000	120.275000	75.335000	8.215621e+06
75%	270.025000	273.717500	264.835000	269.247500	264.477500	1.370871e+07
max	600.210000	605.900000	589.870000	598.130000	597.890000	1.107443e+08

From the data set, Adj. Close and Volume columns are dropped and not used in the project.



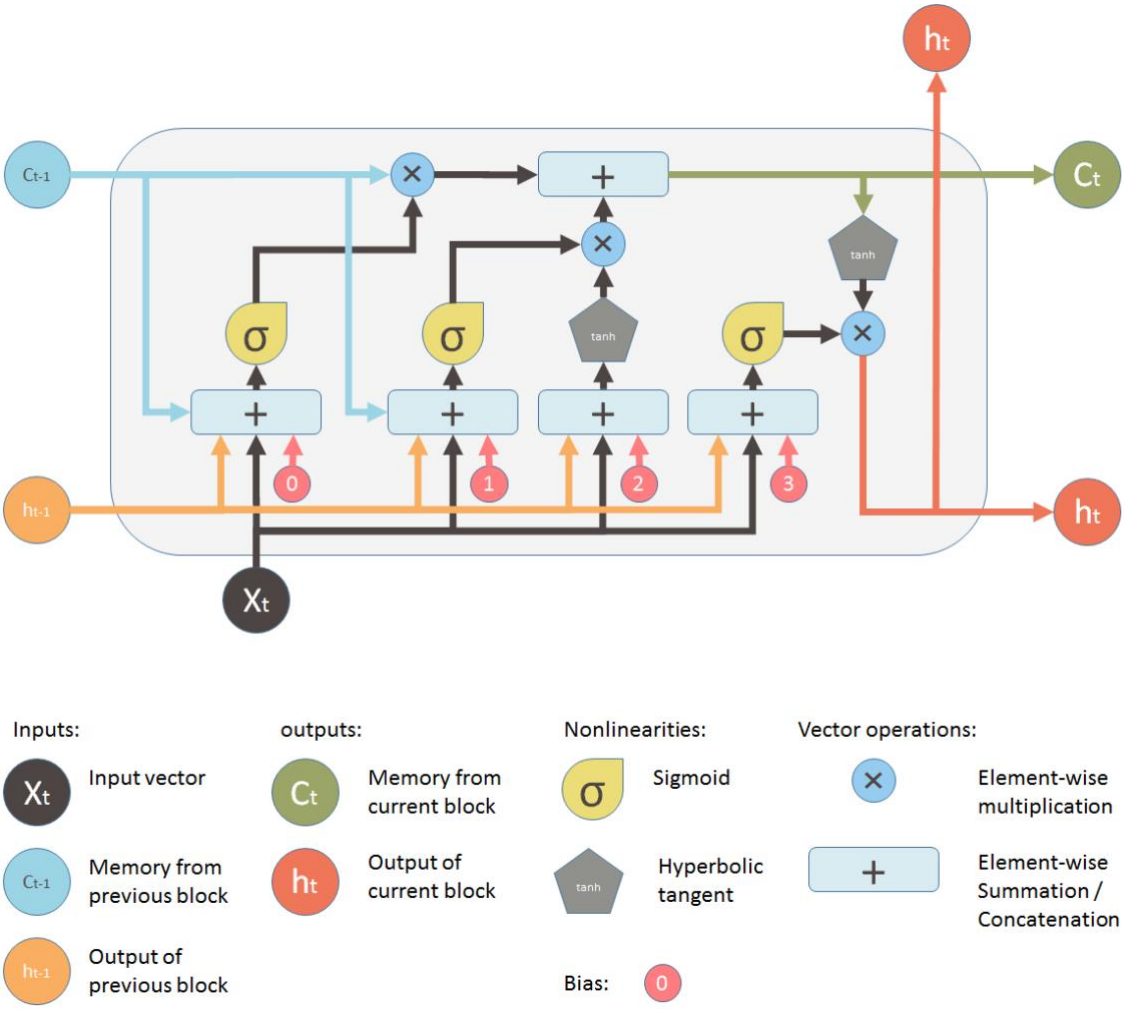
The above figure shows growth of TATAMOTORS and their increase in stock price overtime from 1997 to 2017. The max value of Closing Price which is 598, goes in line with the above graph.

Algorithms and Techniques

For the regressor, 2 neural network models are selected.
The first one would be a simple neural network, a multi-layer perceptron, containing of Dense and Flatten Layer with RELU activation functions.
The second layer would be a Recurrent Neural Network consisting of LSTM layer which stands for Long Short-Term Memory[6] which is a RNN architecture.
LSTM network are well suited for making predictions based on time series data and were developed to deal with the exploding and vanishing gradient problem that can be encountered when training traditional RNNs.
LSTM architecture[7] mainly comprises of 3 gates namely,

- Forget gate
- Input gate
- Output gate

In the first step of LSTM, we need to decide which information needs to be thrown away from the memory cell state, this is done in the ***forget gate***. The sigmoid function looks at h_{t-1} and X_t and outputs a number between 0 and 1 for each cell state $C_{t-1,}$ where 1 means to keep it and 0 means to delete it.



[8]

In the next step, we decide which information needs to be added to the cell state. This is done in 2 parts, first the sigmoid layer, called as ***input gate***, decides which values to update and second the tanh layer creates a vector of new candidate values. The combination of both the layers gives us the new candidate values scaled by how much we decided to update each state value.
Lastly, for output we again have 2 parts, first the sigmoid layer which is called as the ***output gate*** and the tanh layer. The sigmoid layer decides which parts to output from a new filtered cell state from the tanh layer. Combination of both these layers gives us all the parts which needs to be output

It should be noted that time is of the essence in this project and hence the time required to compile, train and predict will also be considered to choose the final model.

Before feeding the data, the data would be preprocessed which require a number of parameters

- num_of_days : Number of days which should be used to create batches of inputs. Eg: if num_of_days=4 then data of 4 days would be used to predict the output.
- train_valid_test_split_rate : Default Value 0.8 provided. The ratio in which training, validation data and testing data should be split. 0.8 means 80% of data will be used for training and validation whereas 20% for testing.
- train_valid_split_rate : Default Value 0.5. Split ratio between training and validation set.
- nomalization_factor: Default Value 100. The value which would be used to divide the entire data set and normalize.

Benchmark

The benchmark model is a K-Nearest Neighbours Classifier mainly because it works good with less amount of data and is fast to train.
The KNN model is fit using grid search to find the best n_neighbours value.

The best estimator comes out to be

```
knn.grid_search.best_estimator_
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=99, p=2,
          weights='uniform')
```

The metrics score comes to be as follows:

```
knn_score = knn.score(y_pred)
print('''Mean squared error is {},
Mean absolute error is {},
Mean absolute percentage error is {}'''.format(*knn_score))
```

```
Mean squared error is 2.1514170721810633,
Mean absolute error is 1.3137248325410023,
Mean absolute percentage error is 28.153548153361225
```

This model mainly does not work as expected because of one primary reason. The sklearn library expects the data set in form of 2-dimentional array where as our data set is in 3-dimentional array.

Data Preprocessing

For the Data preprocessing, firstly the columns of Volume and Adj. Close are dropped.

The data is normalized by using the ***normalization_factor.***

Date	Open	High	Low	Close	Adj Close	Volume
1997-01-01	65.73	69.18	65.54	69.06	15.61	11680285
1997-01-02	69.56	69.56	67.65	68.36	15.45	12850766
1997-01-03	67.65	68.98	67.48	68.71	15.53	7998983
1997-01-06	68.03	69.03	67.28	67.74	15.31	5613143
1997-01-07	67.60	67.60	65.39	65.91	14.90	11466071

Fig: Data before normalization and dropping columns

Date	Open	High	Low	Close	Adj Close	Volume
1997-01-01	0.6573	0.6918	0.6554	0.6906	0.1561	116802.85
1997-01-02	0.6956	0.6956	0.6765	0.6836	0.1545	128507.66
1997-01-03	0.6765	0.6898	0.6748	0.6871	0.1553	79989.83
1997-01-06	0.6803	0.6903	0.6728	0.6774	0.1531	56131.43

Fig: Data after normalization and dropping

After the normalization is done, the data needs to be split into features and output. Features would contain Open, High, Low, and Close of n number of days where n is an input.
Output consists of the Close value of the next day.
Sample input and output:

```
knn.x_train[0]
```

```
array([[0.6573, 0.6918, 0.6554, 0.6906],
       [0.6956, 0.6956, 0.6765, 0.6836],
       [0.6765, 0.6898, 0.6748, 0.6871],
       [0.6803, 0.6903, 0.6728, 0.6774]])
```

```
knn.y_train[0]
```

```
array([0.6591])
```

This processing is done in process_data function in data_utils.py module.

All data related processing is done in data_utils.py module in utils folder.

After the data is processed, the data is split using the split_data function, first into training, validation and testing set and then later into training and validation set.
It is to be noted that the data is not at all shuffled at any point. The first 40% remains as training set, next 40% as validation set and the remaining 20% as testing set. This is however the default setting and can be changed by setting the **train_valid_test_split_rate** and **train_valid_split_rate.**

Implementation

Firstly, the folder structure contains 3 main files or folders:

1. project.ipynb: This file is used for execution.
2. models: This folder contains all the models for the project. This has 4 files:
    a. KNNModel.py: Contains the implementation for KNN model.
    b. BaseModel.py: Abstract Base model class or Neural Networks model
    c. MLPModel.py: Inherits Base Model. Contains architecture for simple neural network
    d. LSTMModel.py: Inherits Base Model. Contains architecture for Recurrent Neural Network LSTM model.
3. utils: This folder contains data_utils.py for all the processing of data.

KNNModel.py
The file contains a class for KNNModel whose constructor is used to pass the data downloaded using the data utils. All the preprocessing happens in the constructor itself.
After the data preprocessing, an instance of KNN regressor is instantiated and passed to GridSearchCV. The instance of GridSearchCV is fit using the fit method. The GridSearchCV uses K-Fold validation with k=5

The class contains predict method which returns the predicted output for x values. It also contains score method used for calculating all the metrics i.e. Mean Squared Error, Mean Absolute Error and Mean Absolute Percentage Error and return all of them in that same order.

BaseModel.py
The file contains a base class model for both the neural network and contains methods summary, compile, fit, load_weights, evaluate and predict.
The constructor contains data preprocessing and the fit method has model checkpointer for saving the best weights to file and early stopping callback for avoiding overfitting.
The architecture for neural network does not reside here and is done in respective child classes

MLPModel.py and LSTMModel.py
The files contain class for MLPModel and LSTMModel respectively which inherits the BaseModel class and the model architecture is defined in the constructor.
Only model architecture definition lies here.
The models use the same BaseModel method for all the operations.

Refinement
The process of improving the models included a lot of trials with number of layers, their filters, normalization, and activation functions. The final model selected between the MLPModel and LSTMModel is the MLPModel because of good speed in training, better metric scores.
The initial MLPModel was made of 3 Dense Layers with 128, 256 and 512 filters respectively with relu activation, a flatten layer and finally a Dense Layer with 1 filter. This model contained Adam Optimizer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 4, 128)            640
_________________________________________________________________
dense_8 (Dense)              (None, 4, 256)            33024
_________________________________________________________________
dense_9 (Dense)              (None, 4, 512)            131584
_________________________________________________________________
flatten_2 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 2049
=================================================================
Total params: 167,297
Trainable params: 167,297
Non-trainable params: 0
_________________________________________________________________
```

Later on, 2 more Dense layers was added with 512 filters and relu activation. A variant of the model was also tried with no activation functions, except for one Dense layer before flatten.
In the end, the final model was as follows

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 4, 512)            2560
_________________________________________________________________
dense_12 (Dense)             (None, 4, 512)            262656
_________________________________________________________________
dense_13 (Dense)             (None, 4, 512)            262656
_________________________________________________________________
dense_14 (Dense)             (None, 4, 512)            262656
_________________________________________________________________
dense_15 (Dense)             (None, 4, 128)            65664
_________________________________________________________________
flatten_3 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_16 (Dense)             (None, 1)                 513
=================================================================
Total params: 856,705
Trainable params: 856,705
Non-trainable params: 0
_________________________________________________________________
```

Here too, a variant was tried using Dropout layers with 25% drop but the results were not better.

In case of LSTM Model the model started as follows

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 4, 20)             2000
_________________________________________________________________
lstm_2 (LSTM)                (None, 20)                3280
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 21
=================================================================
Total params: 5,301
Trainable params: 5,301
Non-trainable params: 0
_________________________________________________________________
```

Later different values of filters were tried such as 50, 128, 256, 512.
Dropout layers were introduced to reduce chances of overfitting and the final model looks as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_4 (LSTM)                (None, 4, 256)            267264
_________________________________________________________________
dropout_1 (Dropout)          (None, 4, 256)            0
_________________________________________________________________
lstm_5 (LSTM)                (None, 256)               525312
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 257
=================================================================
Total params: 792,833
Trainable params: 792,833
Non-trainable params: 0
_________________________________________________________________
```

Model Evaluation and Validation

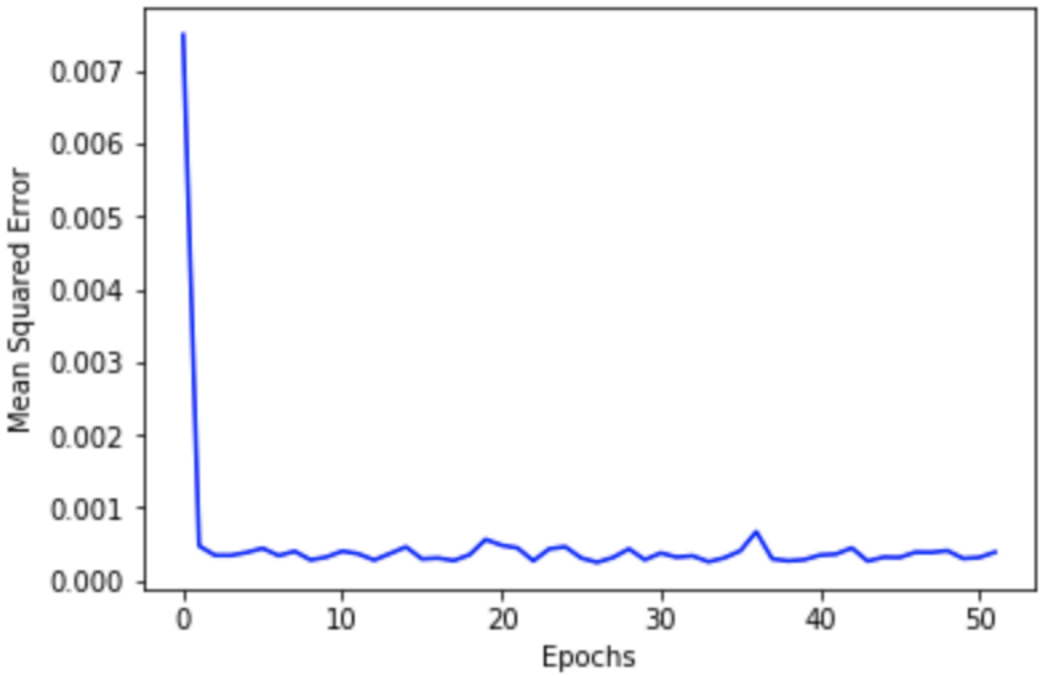


Fig: MSE for MLPModel while training.

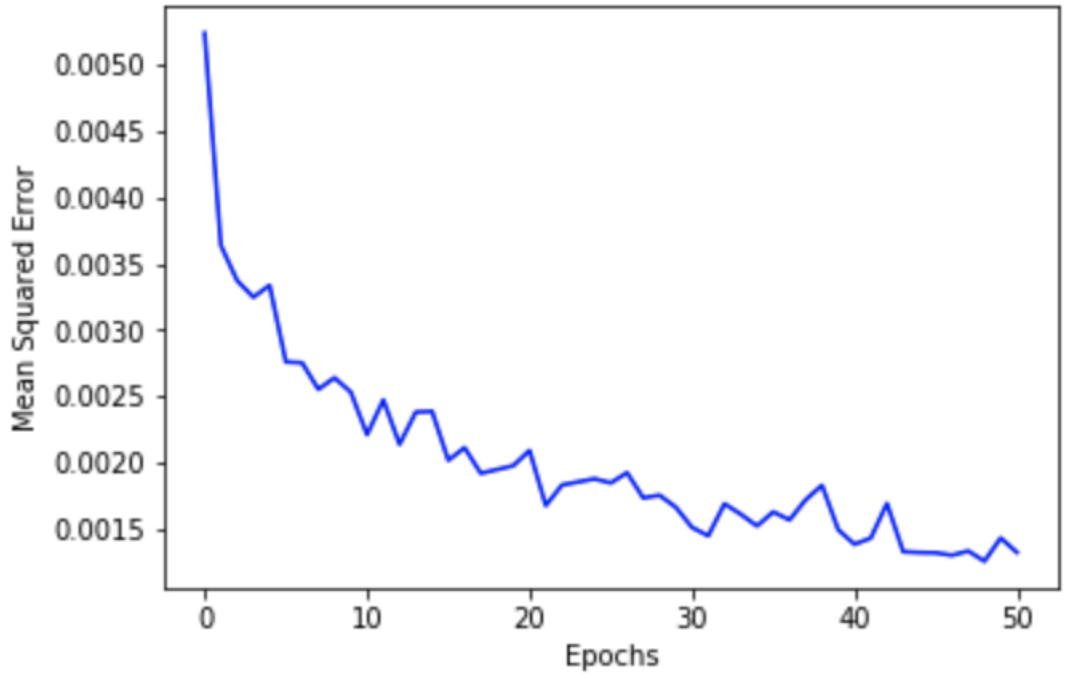


Fig: MSE for LSTMModel while training

From the above figures, it is clear that the metric score for MLPModel is better than that for LSTMModel. However, I believe that LSTMModel could have much better metric score but it takes a lot of time to compile as we add more parameters to it. Hence for the final model, I have chosen MLPModel since it is quick to train and has a good MSE, MAE and MAPE values.

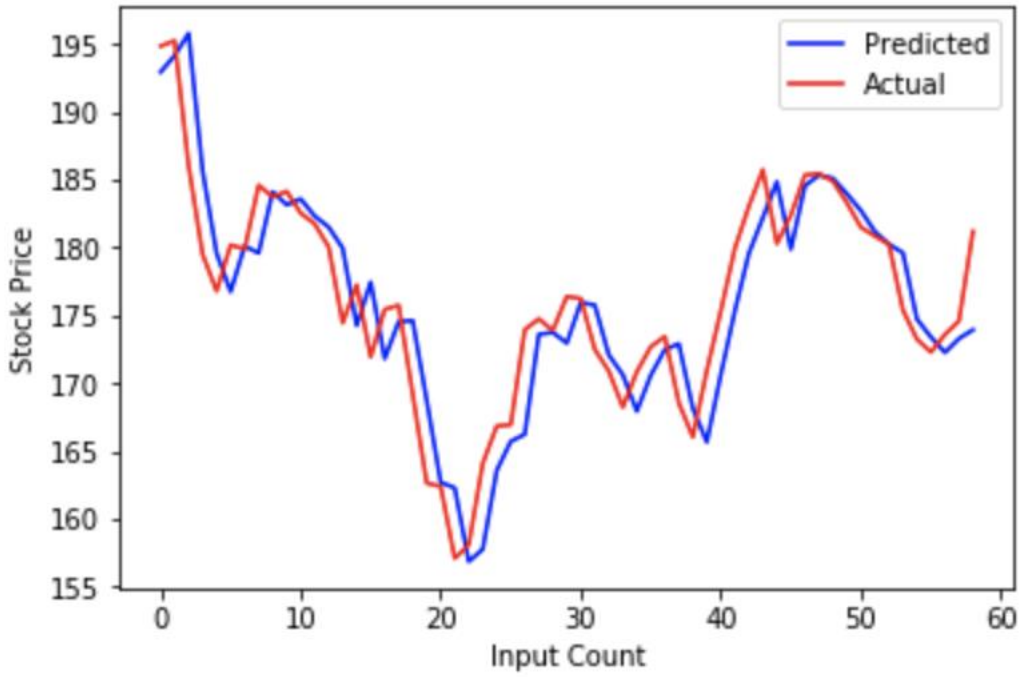


Fig: Predicted price vs Actual price on an unseen data for MLPModel

Here, the MLP Model fares well than our benchmark model, since the benchmark model fails to generalize well.

```
KNN Model:
Mean squared error is 2.1513720709391824,
Mean absolute error is 1.3137008535791017,
Mean absolute percentage error is 28.151909590095688


MLPModel:
Mean squared error is 0.008660921823597 29,
Mean absolute error is 1.6127322814691745,
Mean absolute percentage error is 0.0702271445423967
```

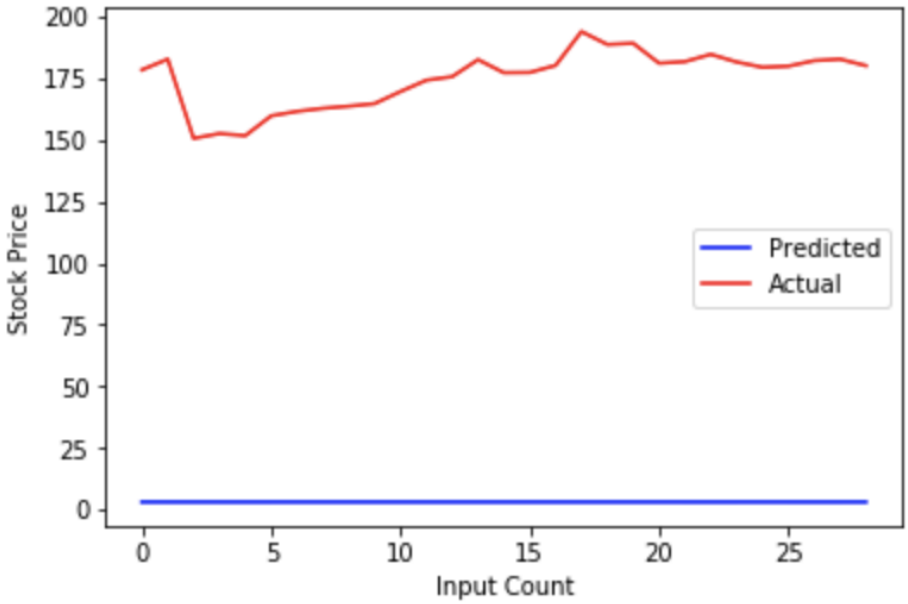


Fig: Predicted price vs Actual price on an unseen data for KNNModel

Conclusion

Reflection

The final result is satisfactory with mean absolute percentage error of less than 2%. Using the data of very recent time, specifically of 14th ,15th, 18th ,19th March 2019, the model predicted the exact value for the closing of 20th March.
One of the hard parts in making this project was to optimize the model for having a good training time as well as reducing the error of the model. There were lots of trials with different number of layers, different filters, activation function, etc. However, the hardest part of the project was the proposal stage since it involved getting started with the project.
It is important to understand that the models cannot have high accuracy as the stock market depends various real-time factors. Thus, finalizing the metrics and understanding why accuracy could not be used took time.
Instead of thinking of entire project at a time, taking one step at a time helped a lot. Initially the data retrieval and processing part was developed, then the model using a simple neural network, later on the data normalization was put in place.

Improvement

Stock trading is real time dealing of stocks. Unfortunately, the models can't work in real time. That in turn may be an improvement which give us the prediction real time. One big improvement will be a reinforcement learning agent which learns the stock market environment real time, and takes actions accordingly. This can even enable sentiment analysis of real world factors like news which directly, or indirectly affects the stock price and take the necessary action accordingly.

Reference:

1. https://finance.yahoo.com/
2. https://en.wikipedia.org/wiki/Mean_squared_error
3. https://en.wikipedia.org/wiki/Mean_absolute_error
4. https://en.wikipedia.org/wiki/Mean_absolute_percentage_error
5. https://pypi.org/project/fix-yahoo-finance/
6. https://en.wikipedia.org/wiki/Long_short-term_memory
7. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
8. https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714