STOCK PRICE PREDICTION

Name:- Sumit Raj

Subject:- Machine Learning

Duration:- 1 months and 5 days

Stock Price Prediction using Machine Learning

Stock Price Prediction using machine learning is the process of predicting the future value of a stock traded on a stock exchange for reaping profits. With multiple factors involved in predicting stock prices, it is challenging to predict stock prices with high accuracy, and this is where machine learning plays a vital role.

Stock Price as a Time Series Data

Treating stock data as time-series, one can use past stock prices (and other parameters) to predict the stock prices for the next day or week. Machine learning models such as Recurrent Neural Networks (RNNs) or LSTMs are popular models applied to predicting time series data such as weather forecasting, election results, house prices, and, of course, stock prices. The idea is to weigh out the importance of recent and older data and determine which parameters affect the “current” or “next” day prices the most. The machine learning model assigns weights to each market feature and determines how much history the model should look at to predict future stock prices.

Stock Price Prediction using Moving Average Time Series

To begin with, we can use moving averages (or MA) to understand how the amount of history (or the number of past data points) considered affects the model's performance. A simple moving average computes the mean of the past N data points and takes this value as the predicted N+1 value.

So,

$$SMA = \frac{P1 + P2 + ... + Pn}{N}$$

Where P1 to Pn are n immediate data points that occur before the present, so to predict the present data point, we take the SMA of the size n (meaning that we see up to n data points in the past). The SMA is our predicted value. The precision of the model will vary significantly with the choice of n. Higher n would mean that we are willing to go deeper into the past to compute the present value. For example, n=2 means that we take the average of the stock price of the past two days, while n=50 would consider 50 days' worth of stock prices. Obviously, 50 days’ worth of data will have more information about the trends of the stock and would lead to better predictions. However, based on context, a large n can also destabilize the model as the more granular fluctuations are smoothened off – looking at prices from the past 300 days would be sub-optimal.

Another moving average is the exponential moving average (EMA), giving more weight to the more recent samples. With this, we can look at more data points in the past and still not diminish the more recent trends in fluctuations.

$$EMA = P_t * k + EMA_{t-1} * (1 - k)$$

Where Pt is the price at time t and k is the weight given to that data point. EMA(t-1) represents the value computed from the past t-1 points. Clearly, this would perform better than a simple MA. The weight k is computed as k = 2/(N+1).

While implementing these methods, we will see how EMA performs better than SMA, proving that assigning higher weights to more recent data points will yield more fruitful results. But for now, let us assume that that is the case with stock prices as time series data.

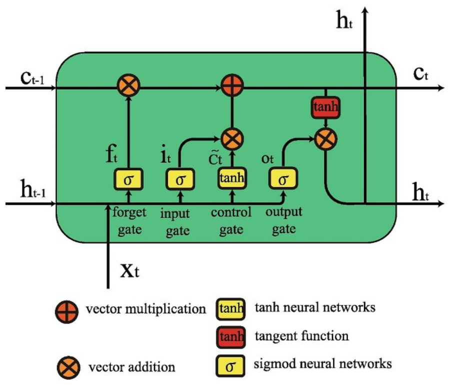
So considering more past data and giving more importance to newer samples, EMA performs better than SMA. However, given the static nature of its parameters, EMA might not perform well for all cases. In EMA, we have fixed the value of k (the weight/significance of past data), and it is linked with the window size N (how much past we wish to consider).

It can be difficult to set these parameters manually and impossible to optimize. Thus, we can use more complex models that can compute the significance of each past data point and optimize our predictions. This can be achieved with weight updation while training a machine learning model. And thinking of using past data to compute the future, the most immediate model that comes to mind is the LSTM model!

Understanding Long Short Term Memory Network for Stock Price Prediction

LSTM is a Recurrent Neural Network that works on data sequences, learning to retain only relevant information from a time window. New information the network learns is added to a “memory” that gets updated with each timestep based on how significant the new sample seems to the model. Over the years, LSTM has revolutionized speech and handwriting recognition, language understanding, forecasting, and several other applications that have become the new normal today.

A standard LSTM cell comprises of three gates: the input, output, and forget gate. These gates learn their weights and determine how much of the current data sample should be remembered and how much of the past learned content should be forgotten. This simple structure is an improvement over the previous and similar RNN model.



As seen in the equations below, i, f, and o represent the three gates: input, forget, and output. C is the cell state that preserves the learned data, which is given as output h. All of this is computed for each timestamp t, considering the learned data from timestamp (t-1).

$$
\begin{aligned}
i_t &= \sigma\left(x_t U^i + h_{t-1} W^i\right) \\
f_t &= \sigma\left(x_t U^f + h_{t-1} W^f\right) \\
o_t &= \sigma\left(x_t U^o + h_{t-1} W^o\right) \\
\tilde{C}_t &= \tanh\left(x_t U^g + h_{t-1} W^g\right) \\
C_t &= \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right) \\
h_t &= \tanh(C_t) * o_t
\end{aligned}
$$

The forget gate decides what information and how much of it can be erased from the current cell state, while the input gate decides what will be added to the current cell state. The output gate, used in the final equation, controls the magnitude of output computed by the first two gates.

So, as opposed to standard feed-forward neural nets, LSTMs have the potential to remember or erase portions of the past data windows actively. Its feature of reading and training on windows (or timesteps) of data makes its training unique. Let’s build the model in Python.

Evaluating Prediction Performance for Stock Price Prediction

Before putting the algorithms into practice, let’s clarify the metric to measure the performance of our models. Stock price prediction being a fundamental regression problem, we can use RMSE (Root Mean Squared Error) or MAPE (Mean Absolute Percentage Error) to measure how close or far off our price predictions are from the real world.

Looking closely at the formula of RMSE, we can see how we will be able to consider the difference (or error) between the actual (At) and predicted (Ft) price values for all N timestamps and get an absolute measure of error.

$$RMSE = \sqrt{\frac{1}{N} * \sum_{t=1}^{N} (At - Ft)^2}$$

On the other hand, MAPE looks at the error concerning the true value – it will measure relatively how far off the predicted values are from the truth instead of considering the actual difference. This is a good measure to keep the error ranges in check if we deal with too large or small values. For instance, RMSE for values in the range of 10e6 might blow out of proportion, whereas MAPE will keep error in a fixed range.

$$MAPE = \frac{1}{N} * \sum_{t=1}^{N} \left| \frac{At - Ft}{At} \right|$$

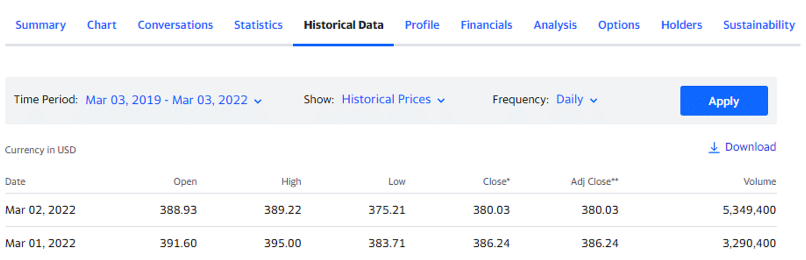
Stock Price Prediction Project using Machine Learning in Python with Source Code

First, we will implement a simple LSTM network using Keras in Python. Let’s take a look at the dataset. We can work on actual stock data from major public companies such as Facebook, Microsoft, or Apple by simply downloading the data from finance.yahoo.com.

Downloading the Stock Prices Dataset

Go to finance.yahoo.com/ and search the company you wish to predict the stock of. For our example, we will look at the Netflix (NFLX) stock over 3 years.

Going to finance.yahoo.com/quote/NFLX/history?p=NFLX in the “Historical Data” section, we see the stock data listed each day. We can filter out the time for which we wish to analyse and download the CSV file using the download button on the right.



Time Period: Mar 03, 2019 - Mar 03, 2022 Show: Historical Prices Frequency: Daily

Currency in USD

Date	Open	High	Low	Close*	Adj Close**	Volume
Mar 02, 2022	388.93	389.22	375.21	380.03	380.03	5,349,400
Mar 01, 2022	391.60	395.00	383.71	386.24	386.24	3,290,400

The download CSV file will contain the data for Open, High, Low, Close, Adj Close, Volume for each date, as shown in the image above.

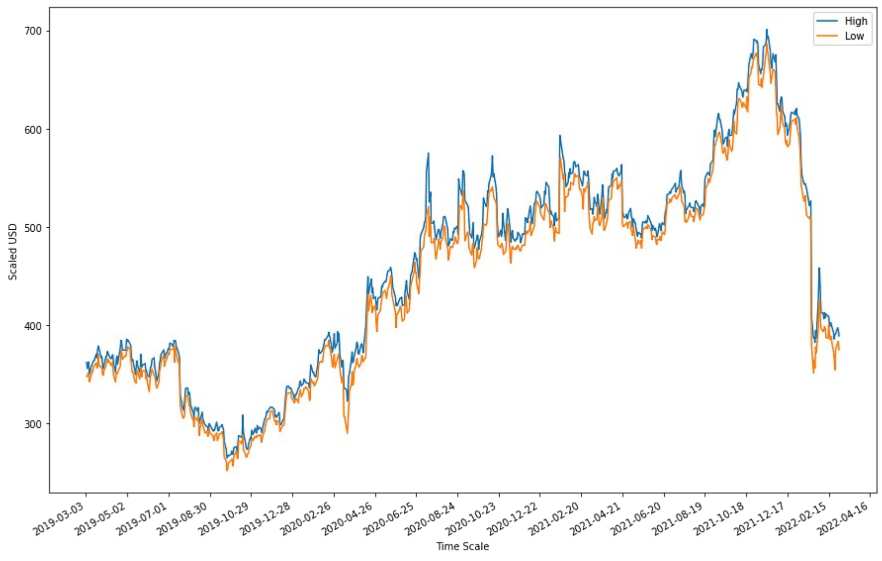
Loading the Stock Prices Dataset

Load the CSV file as a DataFrame using Pandas. Since the data is indexed by date (each row represents data from a different date), we can also index our DataFrame by the date column. We have taken the data from March 2019 to March 2022. This will also challenge our model to work with the unpredictable changes caused by the COVID-19 pandemic.

```
import pandas as pd
stock_data = pd.read_csv('./NFLX.csv',index_col='Date')
stock_data.head()
```

```
                  Open        High         Low       Close   Adj Close   Volume
      Date
2019-03-04  359.720001  362.250000  348.040009  351.040009  351.040009  7487000
2019-03-05  351.459991  356.170013  348.250000  354.299988  354.299988  5937800
2019-03-06  353.600006  359.880005  351.700012  359.609985  359.609985  6211900
2019-03-07  360.160004  362.859985  350.500000  352.600006  352.600006  6151300
2019-03-08  345.750000  349.920013  342.470001  349.600006  349.600006  6898800
```

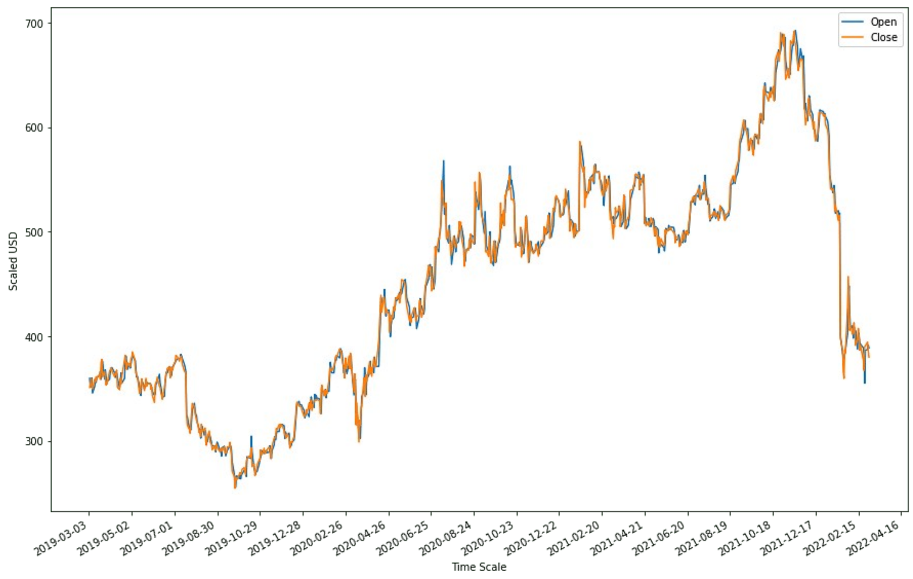
Plotting the High and Low points of Netflix stock over 3 years, we see the below graph.



As noticeable, around March 2020, we see a sudden drop in the price, after which it reports steady growth until recently.

It will be challenging for a machine learning model to correctly estimate the rapid changes that we can see in March 2020 and February 2022. We will focus on evaluating the model performance in predicting the more recent values after training it on the past data.

Similarly, plotting the Open and Close value of the stock for each day gives equivalent observations.



The code for plotting these graphs is as shown below. We use matplotlib to plot the DataFrame columns directly against the Date index column. To make things flexible while plotting against dates, lines 6-8 convert our date strings into datetime format and plot them cleanly and legibly. The interval parameter in line 7 defines the interval in days between each tick on the date axis.

```
 1 import matplotlib.dates as mdates
 2 import matplotlib.pyplot as plt
 3 import datetime as dt
 4
 5 plt.figure(figsize=(15,10))
 6 plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
 7 plt.gca().xaxis.set_major_locator(mdates.DayLocator(interval=60))
 8 x_dates = [dt.datetime.strptime(d,'%Y-%m-%d').date() for d in stock_data.index.values]
 9
10 plt.plot(x_dates, stock_data['High'], label='High')
11 plt.plot(x_dates, stock_data['Low'], label='Low')
12 plt.xlabel('Time Scale')
13 plt.ylabel('Scaled USD')
14 plt.legend()
15 plt.gcf().autofmt_xdate()
16 plt.show()
```

We will use the Open, High, and Low columns to predict the Closing value of the Netflix stock for the next day.

Importing the Libraries

We will be building our LSTM models using Tensorflow Keras and preprocessing our data using scikit-learn. These imports are used in different steps of the entire process, but it is good to club these statements together. Whenever we wish to import something new, just add the statement arbitrarily to the below group.

```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import *
from tensorflow.keras.callbacks import EarlyStopping

from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.model_selection import train_test_split
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
```

Data Preprocessing

As with any other machine learning model, it is always good to normalize or rescale the data within a fixed range when dealing with real data. This will avoid features with larger numeric values to unjustly interfere and bias the model and help achieve rapid convergence.

First, we define the features and the target as discussed above.

```python
target_y = stock_data['Close']
X_feat = stock_data.iloc[:,0:3]
```

Next, we use a StandardScaler to rescale our values between -1 and 1.

```python
#Feature Scaling
sc = StandardScaler()
X_ft = sc.fit_transform(X_feat.values)
X_ft = pd.DataFrame(columns=X_feat.columns,
                    data=X_ft,
                    index=X_feat.index)
```

Scikit-learn also provides a popular MinMaxScaler preprocessing module. However, considering the context, stock prices might max out or minimise on different days, and using those values to influence others might not be great. The change in values from using either of these methods would not be much, so we stick to StandardScaler.

We have 757 data samples in the dataset.

So, the next step would be to split it into training and testing sets. As explained above, the training of an LSTM model requires a window or a timestep of data in each training step. For instance, the LSTM will take 10 data samples to predict the 10th one by weighing the first nine input samples in one step. So, we need a different approach than the train_test_split provided by scikit-learn.

Let’s define a splitting function called lstm_split() which will make windows of size “n_steps” starting from the first sample of data and ending at n_steps’th sample (if n_steps=10, then the 10th sample) from the end. We understand the latter part because, for each time step, LSTM will take n_steps-1 samples for training and predict the last sample. Loss calculation is done based on the error in this prediction. So if n_steps=10, you cannot use the last 9 samples to predict anything because the “10th” data point for the current step does not exist in the dataset.

The function below takes the entire data and creates windows of size n_steps starting from the beginning. The target y will contain the target value corresponding to the n_steps'th index. So if n_steps is 10, the first element in X will have features from 10 data samples, and y will contain the target of the 10th data sample.

```
def lstm_split(data, n_steps):
    X, y = [], []
    for i in range(len(data)-n_steps+1):
        X.append(data[i:i + n_steps, :-1])
        y.append(data[i + n_steps-1, -1])

    return np.array(X), np.array(y)
```

Train and Test Sets for Stock Price Prediction

We split our data into training and testing sets. Shuffling is not permitted in time-series datasets. In the beginning, we take two steps worth of past data to predict the current value. Thus, the model will look at yesterday's and today's values to predict today's closing price.

```
X1, y1 = lstm_split(stock_data_ft.values, n_steps=2)

train_split=0.8
split_idx = int(np.ceil(len(X1)*train_split))
date_index = stock_data_ft.index

X_train, X_test = X1[:split_idx], X1[split_idx:]
y_train, y_test = y1[:split_idx], y1[split_idx:]
X_train_date, X_test_date = date_index[:split_idx], date_index[split_idx:]

print(X1.shape, X_train.shape, X_test.shape, y_test.shape)
```

```
(755, 2, 3) (604, 2, 3) (151, 2, 3) (151,)
```

Note above that the size of X1 is n_steps less than that of the original dataset. As we explained above, you cannot use the last two samples of the original set during training or prediction as we do not have their corresponding ground truth values.

Building the LSTM model

We will use the Sequential and LSTM modules provided by Tensorflow Keras to build a simple, single-unit LSTM model.

```
lstm = Sequential()
lstm.add(LSTM(32, input_shape=(X_train.shape[1], X_train.shape[2]),
              activation='relu', return_sequences=True))
lstm.add(Dense(1))
lstm.compile(loss='mean_squared_error', optimizer='adam')
lstm.summary()
```

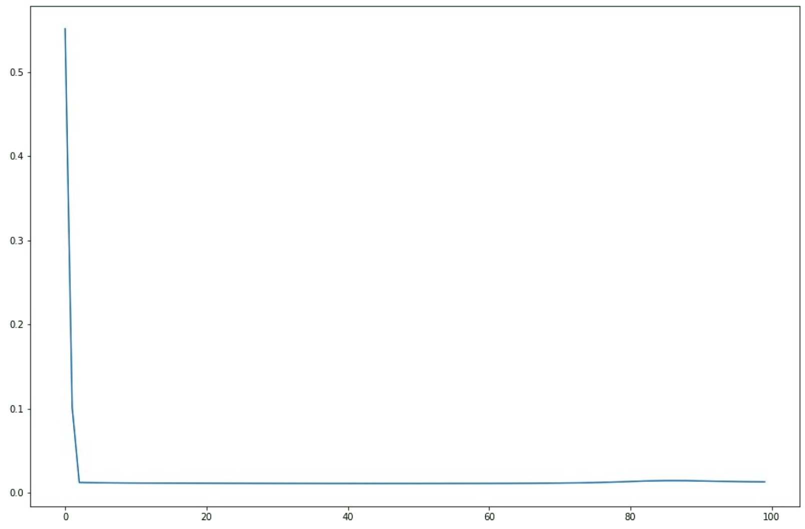
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 1, 32)             4608

 dense_2 (Dense)             (None, 1, 1)              33

=================================================================
Total params: 4,641
Trainable params: 4,641
Non-trainable params: 0
_________________________________________________________________
```

Now we can fit this simple model to the training data.

```
history=lstm.fit(X_train, y_train,
                 epochs=100, batch_size=4,
                 verbose=2, shuffle=False)
```

```
Epoch 1/100
152/152 - 2s - loss: 0.5510 - 2s/epoch - 16ms/step
Epoch 2/100
152/152 - 1s - loss: 0.1016 - 829ms/epoch - 5ms/step
Epoch 3/100
152/152 - 0s - loss: 0.0124 - 447ms/epoch - 3ms/step
Epoch 4/100
152/152 - 0s - loss: 0.0122 - 313ms/epoch - 2ms/step
```

Given the simplicity of the model and the data, we note that the loss reduction stagnates after only 20 epochs. You can observe this by plotting the training loss against the number of epochs, and LSTM does not learn much after 10-20 epochs.

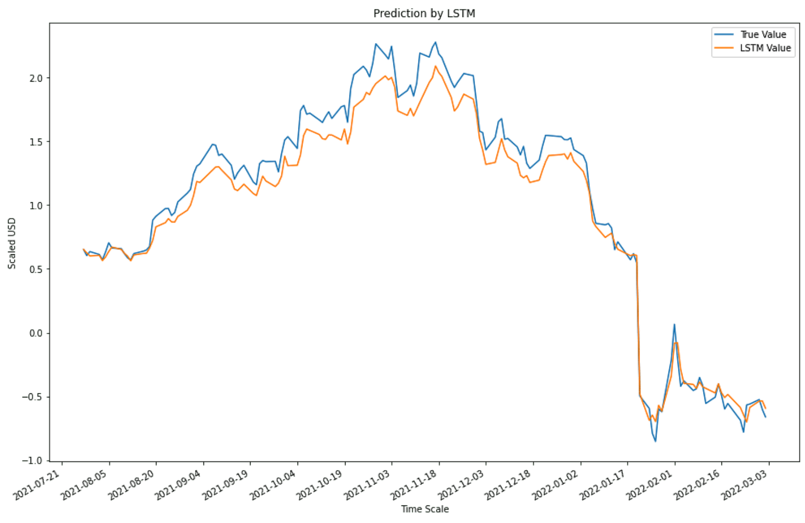


Performance Evaluation on Test Set

Nevertheless, we can check the performance of our model on a test set as below.

```
y_pred = lstm.predict(X_test)
```

To evaluate, first, we plot the curve for true values and overlap it with that for the predicted values.



Thus, we can see that LSTM can emulate the trends of the stock prices to a certain extent. Based on the recent dip in prices, it has also fit the dropping curve well.

As we decided earlier, we can also check the RMSE and MAPE values to evaluate the performance. We will use these values for future comparison.

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
mape = mean_absolute_percentage_error(y_test, y_pred)
print("RSME: ",rmse)
print("MAPE: ", mape)
```

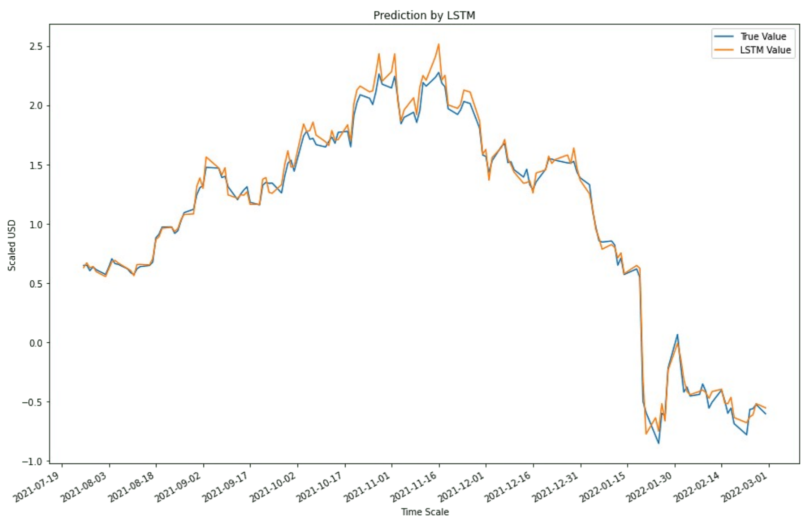
```
RSME:  0.16808551269723027
MAPE:  0.12614061132643917
```

Let’s try to get better results with the same dataset but a deeper LSTM model.

```
lstm = Sequential()
lstm.add(LSTM(50, input_shape=(X_train.shape[1], X_train.shape[2]),
              activation='relu', return_sequences=True))
lstm.add(LSTM(50, activation='relu'))

lstm.add(Dense(1))
lstm.compile(loss='mean_squared_error', optimizer='adam')
lstm.summary()
```

We added another LSTM layer and increased the number of LSTM units per layer to 50.

While the loss still converges early, the curve is better fitted to the true value.



Moreover, the RMSE and MAPE values are better too.

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
mape = mean_absolute_percentage_error(y_test, y_pred)
print("RSME: ",rmse)
print("MAPE: ", mape)
```

```
RSME:  0.0710094097230204
MAPE:  0.058775797917331896
```

Thus we observe substantial improvement by adding another LSTM layer to the model. However, further adding even more layers would not be fruitful as the model might overfit or stagnate during training.

Now we will try fitting the same model but with increased time steps. We’ll try for n_steps=10.

We change the value in the block below and rerun the entire process with the same model as before.

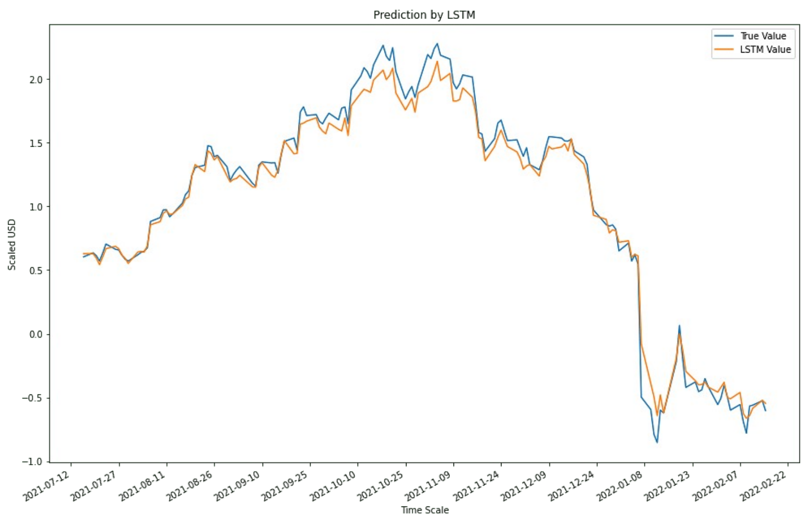
```
n_steps=10
X1, y1 = lstm_split(stock_data_ft.values, n_steps=n_steps)

train_split=0.8
split_idx = int(np.ceil(len(X1)*train_split))
date_index = stock_data_ft.index

X_train, X_test = X1[:split_idx], X1[split_idx:]
y_train, y_test = y1[:split_idx], y1[split_idx:]
X_train_date, X_test_date = date_index[:split_idx], date_index[split_idx:-n_steps]

print(X1.shape, X_train.shape, X_test.shape, X_test_date.shape, y_test.shape)
```

```
(747, 10, 3) (598, 10, 3) (149, 10, 3) (149,) (149,)
```

The model we used above:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 lstm_9 (LSTM)               (None, 10, 50)            10800     
                                                                 
 lstm_10 (LSTM)              (None, 50)                20200     
                                                                 
 dense_4 (Dense)             (None, 1)                 51        
                                                                 
=================================================================
Total params: 31,051
Trainable params: 31,051
Non-trainable params: 0
_________________________________________________________________
```

Surprisingly, we get similar performance as before!



```python
rmse = mean_squared_error(y_test, y_pred, squared=False)
mape = mean_absolute_percentage_error(y_test, y_pred)
print("RSME: ",rmse)
print("MAPE: ", mape)
```

```
RSME:  0.09550824106541782
MAPE:  0.07162436280154859
```

We note that LSTM was able to achieve decent RMSE and MAPE values despite the data complexity. Further, we note that creating even deeper networks did not help improve the test performance.

Before we conclude, as promised earlier, let’s look at how better or worse LSTMs perform compared with statistical techniques such as SMA and EMA.

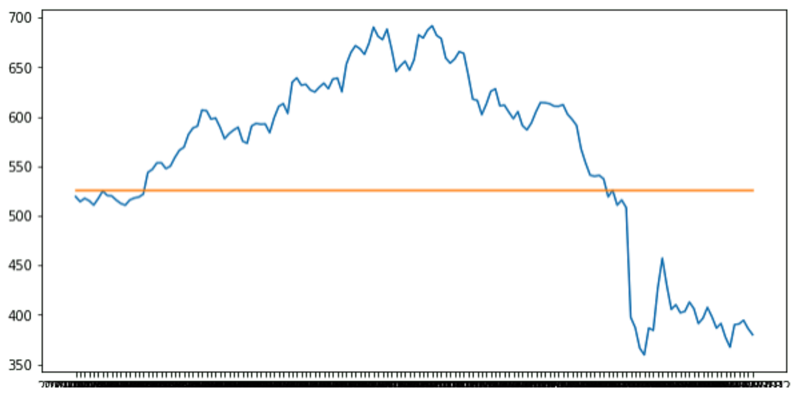
LSTM vs. Simple Moving Average vs. Exponential Moving Average for Stock Price Prediction

Simple Moving Average

```python
train_split = 0.8
split_idx = int(np.ceil(len(stock_data)*train_split))
train = stock_data[['Close']].iloc[:split_idx]
test = stock_data[['Close']].iloc[split_idx:]

test_pred = np.array([train.rolling(10).mean().iloc[-1]]*len(test)).reshape((-1,1))

print('Test RMSE: %.3f' % mean_squared_error(test, test_pred, squared=False))
print('Test MAPE: %.3f' % mean_absolute_percentage_error(test, test_pred))

plt.figure(figsize=(10,5))
plt.plot(test)
plt.plot(test_pred)
plt.show()
```

Based on this, we find the results as below:

```
Test RMSE: 98.507
Test MAPE: 0.159
```



LSTMs perform better under similar conditions.

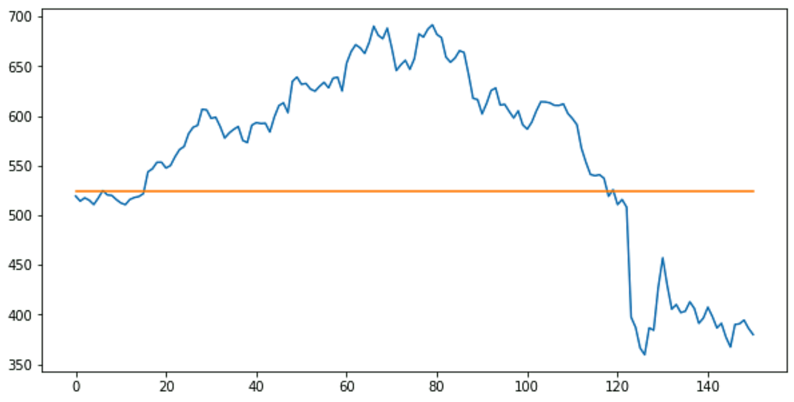
Similarly, let us also check how the exponential moving average performs. As we noted initially, it is supposed to perform better than SMA.

Exponential Moving Average

Rather than implementing EMA from scratch, we can use the module provided by statsmodels in Python. For now, we can use the SimpleExpSmoothing module of the TSA API from statsmodels. While fitting this model, we can tune the smoothing_level parameter to get optimal performance – we note that a relatively lower value yields better results.

```python
from statsmodels.tsa.api import SimpleExpSmoothing

X = stock_data[['Close']].values
train_split = 0.8
split_idx = int(np.ceil(len(X)*train_split))
train = X[:split_idx]
test = X[split_idx:]
test_concat = np.array([]).reshape((0,1))

for i in range(len(test)):
    train_fit = np.concatenate((train, np.asarray(test_concat)))
    fit = SimpleExpSmoothing(np.asarray(train_fit)).fit(smoothing_level=0.2)
    test_pred = fit.forecast(1)
    test_concat = np.concatenate((np.asarray(test_concat), test_pred.reshape((-1,1))))

print('Test RMSE: %.3f' % mean_squared_error(test, test_concat, squared=False))
print('Test MAPE: %.3f' % mean_absolute_percentage_error(test, test_concat))
```

```
Test RMSE: 99.268
Test MAPE: 0.160
```



So in the case of predicting stock price data, SMA and EMA have similar performance and are far behind when compared to LSTMs. We can improve our LSTM model by finetuning the hyperparameters such as the number of cells, batch size, or the loss function. However, using data beyond 2019 (up to 5-10 years' worth of data) would greatly help the model.