In [1]:
```
# !pip install plotly==5.5.0
```

Importing necessary libraries

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pandas_datareader as data
import plotly.graph_objects as go
from sklearn.preprocessing import MinMaxScaler
```

Defining Start date and End date for our stock data

- We have taken data from 2010 up till 28th January 2022 to train our models.
- Did not consider data before 2010 as it the 2008 financial crisis was followed by a huge rise in price of every stock which might skew the stock prediction learning algorithm.
- Did consider 2020 Covid-19 crash as the stock market has still not fully recovered from it.

In [3]:
```
start_date = '2010-01-01'
end_date = '2022-01-28'
```

Taking user input for Stock

- Here, we have asked user to input the stock ticker which is the short way of representing a stock in stock exchanges and is unique
- For example,
  - Apple: AAPL
  - Alphabet: GOOGL

In [4]:
```
user_input = input('Enter Stock Ticker: ')
```

```
Enter Stock Ticker: AURIONPRO.NS
```

Reading data from Yahoo finance using Pandas data-reader

In [5]:
```
df = data.DataReader(user_input, 'yahoo', start_date, end_date)
df
```

Out[5]:

Date	High	Low	Open	Close	Volume	Adj Close
2010-01-04	253.335587	240.668808	247.424423	252.744461	1421.0	218.702499
2010-01-05	249.113327	246.579971	249.113327	248.691101	3427.0	215.195068
2010-01-06	255.868942	246.579971	249.113327	253.040024	3068.0	218.958252
2010-01-07	255.868942	249.113327	253.335587	255.868942	16389.0	221.406158
2010-01-08	255.868942	249.113327	249.113327	254.222260	16737.0	219.981232
...	...	...	...	...	...	...
2022-01-21	361.750000	344.649994	361.649994	344.649994	52379.0	344.649994
2022-01-24	336.000000	327.450012	335.000000	327.450012	35577.0	327.450012
2022-01-25	338.799988	311.100006	311.100006	334.000000	134434.0	334.000000
2022-01-27	350.700012	320.000000	322.250000	337.799988	141208.0	337.799988
2022-01-28	347.899994	325.000000	338.000000	329.899994	61824.0	329.899994

2980 rows × 6 columns

We don't need the dates of the stock prices as the index for our dataframe

In [6]:
```
df = df.reset_index()
df.head()
```

Out[6]:

	Date	High	Low	Open	Close	Volume	Adj Close
0	2010-01-04	253.335587	240.668808	247.424423	252.744461	1421.0	218.702499
1	2010-01-05	249.113327	246.579971	249.113327	248.691101	3427.0	215.195068
2	2010-01-06	255.868942	246.579971	249.113327	253.040024	3068.0	218.958252
3	2010-01-07	255.868942	249.113327	253.335587	255.868942	16389.0	221.406158
4	2010-01-08	255.868942	249.113327	249.113327	254.222260	16737.0	219.981232

Now, we can drop irrelevant collumns

In [7]:

```
df = df.drop(['Adj Close'], axis=1)
```

In [8]:

```
df.head()
```

Out[8]:

	Date	High	Low	Open	Close	Volume
0	2010-01-04	253.335587	240.668808	247.424423	252.744461	1421.0
1	2010-01-05	249.113327	246.579971	249.113327	248.691101	3427.0
2	2010-01-06	255.868942	246.579971	249.113327	253.040024	3068.0
3	2010-01-07	255.868942	249.113327	253.335587	255.868942	16389.0
4	2010-01-08	255.868942	249.113327	249.113327	254.222260	16737.0

Visualizing the closing stock price over the selected time period

In [9]:

```
plt.figure(figsize=(12, 6))
plt.plot(df.Date, df.Close, label='Closing Stock Price')
plt.title(user_input + (' Closing Price vs Time'))
plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.legend(fontsize=12)
```

Out[9]: <matplotlib.legend.Legend at 0x28407630eb0>

Defining 100-day and 200-day Moving Averages for both opening and closing price

In [10]:

```
closing_ma100 = df.Close.rolling(100).mean()
opening_ma100 = df.Open.rolling(100).mean()
```

In [11]:

```
closing_ma200 = df.Close.rolling(200).mean()
opening_ma200 = df.Open.rolling(200).mean()
```

In [12]:
```
closing_ma100 # First 99 values are null since it will be starting from 100th value
```

Out[12]:
```
0              NaN
1              NaN
2              NaN
3              NaN
4              NaN
           ...
2975    245.2980
2976    246.5400
2977    247.8000
2978    249.1075
2979    250.3980
Name: Close, Length: 2980, dtype: float64
```

In [13]:
```
opening_ma200 # First 199 values are null since it will be starting from 200th value
```

Out[13]:
```
0               NaN
1               NaN
2               NaN
3               NaN
4               NaN
           ...
2975    209.03275
2976    210.02775
2977    210.90325
2978    211.83500
2979    212.84325
Name: Open, Length: 2980, dtype: float64
```

Plotting daily closing price vs 100-day and 200-day Moving Averages

In [14]:
```
plt.figure(figsize=(12, 6))
plt.plot(df.Date, df.Close, label='Daily Closing Value')
plt.plot(df.Date, closing_ma100, label='100-day Moving Average')
plt.plot(df.Date, closing_ma200, label='200-day Moving Average')
plt.title(user_input + ' Daily Closing value vs 100- and 200-day moving averages')
plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.legend(fontsize=12)
```

Out[14]: <matplotlib.legend.Legend at 0x28407516df0>

Making candle-stick plots for the last 30 days

In [15]:
```
candlestick_fig = go.Figure(data=[go.Candlestick(x=df['Date'][-30:], open=df['Open'], close=df['Close'], high=df[

candlestick_fig.show()
```

Analysing our data

In [16]:
```
df.shape
```

Out[16]:
```
(2980, 6)
```

Splitting data into Training and Testing

In [17]:
```
train_set = pd.DataFrame(data=(df['Open'][:int(len(df)*0.80)], df['Close'][0:int(len(df)*0.80)], df['High'][0:int
test_set = pd.DataFrame(data=(df['Open'][int(len(df)*0.80):], df['Close'][int(len(df)*0.80):], df['High'][int(ler
```

In [18]:
```
print(train_set.shape)
print(test_set.shape)
```

```
(4, 2384)
(4, 596)
```

In [19]:
```
train_set = train_set.T
test_set = test_set.T
```

In [20]:
```
print(train_set.shape)
print(test_set.shape)
```

```
(2384, 4)
(596, 4)
```

In [21]:
```
train_set.head()
```

Out[21]:

	Open	Close	High	Low
0	247.424423	252.744461	253.335587	240.668808
1	249.113327	248.691101	249.113327	246.579971
2	249.113327	253.040024	255.868942	246.579971
3	253.335587	255.868942	255.868942	249.113327

4	249.113327	254.222260	255.868942	249.113327

In [22]:
```
test_set.head() # Note that the train set starts from index 2384, since this a sequential data
```

Out[22]:

	Open	Close	High	Low
2384	84.599998	88.349998	89.0	84.449997
2385	85.599998	88.800003	89.0	85.599998
2386	88.000000	86.800003	89.0	85.599998
2387	86.800003	85.599998	89.0	85.000000
2388	88.750000	88.849998	89.0	85.000000

Scaling data such that it is between 0 - 1 since the LSTMs require standardized data

In [23]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
```

In [24]:
```
train_set_arr = scaler.fit_transform(train_set)
test_set_arr = scaler.fit_transform(test_set)
```

In [25]:
```
train_set_arr
```

Out[25]:
```
array([[0.83937825, 0.87905499, 0.86355787, 0.8597123 ],
       [0.84628671, 0.86212978, 0.84628671, 0.88489215],
       [0.84628671, 0.88028914, 0.87392056, 0.88489215],
       ...,
       [0.19543351, 0.19845541, 0.21281814, 0.21535067],
       [0.18725251, 0.17841252, 0.19952401, 0.1836158 ],
       [0.17089051, 0.1756984 , 0.17845793, 0.18851447]])
```

In [26]:
```
train_set_arr.shape
```

Out[26]:
```
(2384, 4)
```

In [27]:
```
test_set_arr
```

Out[27]:
```
array([[0.15373922, 0.16820965, 0.1598218 , 0.16599013],
       [0.15651932, 0.16949153, 0.1598218 , 0.16932443],
       [0.16319155, 0.16379434, 0.1598218 , 0.16932443],
       ...,
       [0.78343065, 0.86796751, 0.85535288, 0.82313714],
       [0.81442869, 0.87879215, 0.88848674, 0.84894171],
       [0.85821518, 0.85628825, 0.8806905 , 0.86343866]])
```

In [28]:
```
test_set_arr.shape
```

Out[28]:
```
(596, 4)
```

Splitting train_set_arr into X_train and Y_train

- In this, x_train will have the prices of n consecutive days, where n is the step size.
- Step size denotes the number of previous days we want our model to predict the next day's values on
- For example, if we choose to predict the closing price of a stock based on the previous 100 days, x_train will have those 100 days of closing price data while y_train will have the 101th day data
- The step size is decided by us.

In [29]:
```
x_train = []
y_train = []

for i in range(100, train_set_arr.shape[0]):
```

```
    x_train.append(train_set_arr[i-100: i])
    y_train.append(train_set_arr[i])
```

In [30]:
```
print(x_train[0][0])
```

```
[0.83937825 0.87905499 0.86355787 0.8597123 ]
```

In [31]:
```
len(x_train), len(x_train[0]), len(x_train[0][0]) # 2284 instead of 2384 as the first 100 values are not counted.
```

Out[31]:
```
(2284, 100, 4)
```

In [32]:
```
print(y_train[0])
```

```
[0.84628671 0.74788436 0.84628671 0.77697848]
```

In [33]:
```
len(y_train), len(y_train[0]) # 2284 instead of 2384 as the first 100 values are not counted.
```

Out[33]:
```
(2284, 4)
```

Splitting test_set_arr into x_test and y_test

- This is done the same way as above

In [34]:
```
x_test = []
y_test = []

for i in range(100, test_set_arr.shape[0]):
    x_test.append(test_set_arr[i-100: i])
    y_test.append(test_set_arr[i])
```

In [35]:
```
x_test[0][0]
```

Out[35]:
```
array([0.15373922, 0.16820965, 0.1598218 , 0.16599013])
```

In [36]:
```
len(x_test), len(x_test[0]), len(x_test[0][0]) # 496 instead of 596 as the first 100 values are not counted.
```

Out[36]:
```
(496, 100, 4)
```

In [37]:
```
len(y_test), len(y_test[0]) # 496 instead of 596 as the first 100 values are not counted.
```

Out[37]:
```
(496, 4)
```

In [38]:
```
len(y_test[0])
```

Out[38]:
```
4
```

In [39]:
```
y_test[0]
```

Out[39]:
```
array([0.08354184, 0.08332146, 0.07726576, 0.08959118])
```

Converting x_train, y_train, x_test, y_test to numpy arrays

- This step is necesaary as currently x_train, y_train, x_test and y_test are all python lists while LSTMs require Numpy Arrays as input

In [40]:
```
x_train, y_train = np.array(x_train), np.array(y_train)
x_test, y_test = np.array(x_test), np.array(y_test)
```

In [41]:
```
x_train[0][0]
```

Out[41]:
```
array([0.83937825, 0.87905499, 0.86355787, 0.8597123 ])
```

In [42]:
```
x_train.shape
```

Out[42]:
```
(2284, 100, 4)
```

In [43]:
```
y_train[0]
```

Out[43]:
```
array([0.84628671, 0.74788436, 0.84628671, 0.77697848])
```

In [44]:
```
y_train.shape
```

Out[44]:
```
(2284, 4)
```

In [45]:
```
x_test[0][0]
```

Out[45]:
```
array([0.15373922, 0.16820965, 0.1598218 , 0.16599013])
```

In [46]:
```
x_test.shape
```

Out[46]:
```
(496, 100, 4)
```

In [47]:
```
y_test[0]
```

Out[47]:
```
array([0.08354184, 0.08332146, 0.07726576, 0.08959118])
```

In [48]:
```
y_test.shape
```

Out[48]:
```
(496, 4)
```

Creating the Machine Learning Model using stacked LSTMs via keras

- To train our model to predict stock prices, we used a few different models to test which performs better.
- We validated our model on the test data created above = LAST 20% of the total stock price data. The word LAST is important here as this is a sequential time-series data and we cannot randomly split train and test data.
- We used two broad models as our base for the predictions, ARIMA and Stacked LSTMs
- After tuning our hyperparameters and recording every observation in an excel sheet, we went ahead with the stacked LSTMs model.
- Final model:
  - 4 LSTM layers with increasing nodes in each layer and the ReLU activtion function in each layer;
  - Dropout layers with an increasing dropout rate between each LSTM layer to ensure uniformity of the predictions;
  - Followed by a dense layer that gives 4 outputs as a single prediction value:
    - Opening price, Closing Price, Daily High, Daily Low.
    - Note: The dense layer does not have any activation function since this is not a classification problem.

Importing necessary libraries

In [49]:
```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout, LSTM
from keras.models import Sequential
from tensorflow.keras.models import load_model
from tensorflow.keras.callbacks import EarlyStopping
```

In [50]:
```
model = Sequential()
model.add(LSTM(units=60,
               activation='relu',
               return_sequences=True,
              input_shape=x_train[0].shape))
model.add(Dropout(0.2))

model.add(LSTM(units=80,
               activation='relu',
               return_sequences=True))
model.add(Dropout(0.3))

model.add(LSTM(units=100,
               activation='relu',
               return_sequences=True))
model.add(Dropout(0.4))

model.add(LSTM(units=140,
               activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(units=4))
```

In [51]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100, 60)           15600

 dropout (Dropout)           (None, 100, 60)           0

 lstm_1 (LSTM)               (None, 100, 80)           45120

 dropout_1 (Dropout)         (None, 100, 80)           0

 lstm_2 (LSTM)               (None, 100, 100)          72400

 dropout_2 (Dropout)         (None, 100, 100)          0

 lstm_3 (LSTM)               (None, 140)               134960

 dropout_3 (Dropout)         (None, 140)               0

 dense (Dense)               (None, 4)                 564

=================================================================
Total params: 268,644
Trainable params: 268,644
Non-trainable params: 0
_________________________________________________________________
```

In [52]:
```
model.compile(optimizer='adam', loss='mse')
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=6, restore_best_weights=True)
r = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=80, callbacks=[es])
```

```
Epoch 1/80
72/72 [==============================] - 17s 196ms/step - loss: 0.0329 - val_loss: 0.0054
Epoch 2/80
72/72 [==============================] - 13s 188ms/step - loss: 0.0119 - val_loss: 0.0061
Epoch 3/80
72/72 [==============================] - 14s 192ms/step - loss: 0.0097 - val_loss: 0.0039
Epoch 4/80
72/72 [==============================] - 17s 237ms/step - loss: 0.0083 - val_loss: 0.0032
Epoch 5/80
72/72 [==============================] - 15s 212ms/step - loss: 0.0084 - val_loss: 0.0032
Epoch 6/80
72/72 [==============================] - 19s 264ms/step - loss: 0.0066 - val_loss: 0.0039
Epoch 7/80
72/72 [==============================] - 15s 205ms/step - loss: 0.0064 - val_loss: 0.0044
Epoch 8/80
72/72 [==============================] - 13s 183ms/step - loss: 0.0057 - val_loss: 0.0024
Epoch 9/80
72/72 [==============================] - 13s 177ms/step - loss: 0.0057 - val_loss: 0.0028
Epoch 10/80
```

```
72/72 [==============================] - 13s 175ms/step - loss: 0.0052 - val_loss: 0.0036
Epoch 11/80
72/72 [==============================] - 12s 166ms/step - loss: 0.0049 - val_loss: 0.0034
Epoch 12/80
72/72 [==============================] - 11s 160ms/step - loss: 0.0048 - val_loss: 0.0018
Epoch 13/80
72/72 [==============================] - 11s 159ms/step - loss: 0.0047 - val_loss: 0.0028
Epoch 14/80
72/72 [==============================] - 12s 173ms/step - loss: 0.0041 - val_loss: 0.0028
Epoch 15/80
72/72 [==============================] - 12s 170ms/step - loss: 0.0039 - val_loss: 0.0026
Epoch 16/80
72/72 [==============================] - 11s 157ms/step - loss: 0.0036 - val_loss: 0.0041
Epoch 17/80
72/72 [==============================] - 13s 177ms/step - loss: 0.0039 - val_loss: 0.0017
Epoch 18/80
72/72 [==============================] - 15s 203ms/step - loss: 0.0036 - val_loss: 0.0059
Epoch 19/80
72/72 [==============================] - 12s 160ms/step - loss: 0.0039 - val_loss: 0.0023
Epoch 20/80
72/72 [==============================] - 12s 161ms/step - loss: 0.0033 - val_loss: 0.0027
Epoch 21/80
72/72 [==============================] - 13s 175ms/step - loss: 0.0035 - val_loss: 0.0015
Epoch 22/80
72/72 [==============================] - 11s 155ms/step - loss: 0.0033 - val_loss: 0.0019
Epoch 23/80
72/72 [==============================] - 11s 156ms/step - loss: 0.0031 - val_loss: 0.0019
Epoch 24/80
72/72 [==============================] - 12s 167ms/step - loss: 0.0030 - val_loss: 0.0024
Epoch 25/80
72/72 [==============================] - 14s 196ms/step - loss: 0.0030 - val_loss: 0.0022
Epoch 26/80
72/72 [==============================] - 12s 160ms/step - loss: 0.0029 - val_loss: 0.0011
Epoch 27/80
72/72 [==============================] - 11s 159ms/step - loss: 0.0028 - val_loss: 0.0023
Epoch 28/80
72/72 [==============================] - 12s 161ms/step - loss: 0.0031 - val_loss: 0.0016
Epoch 29/80
72/72 [==============================] - 12s 170ms/step - loss: 0.0027 - val_loss: 0.0021
Epoch 30/80
72/72 [==============================] - 13s 188ms/step - loss: 0.0028 - val_loss: 0.0018
Epoch 31/80
72/72 [==============================] - 13s 187ms/step - loss: 0.0028 - val_loss: 0.0022
Epoch 32/80
72/72 [==============================] - ETA: 0s - loss: 0.0027Restoring model weights from the end of the best e
poch: 26.
72/72 [==============================] - 12s 169ms/step - loss: 0.0027 - val_loss: 0.0020
Epoch 00032: early stopping
```

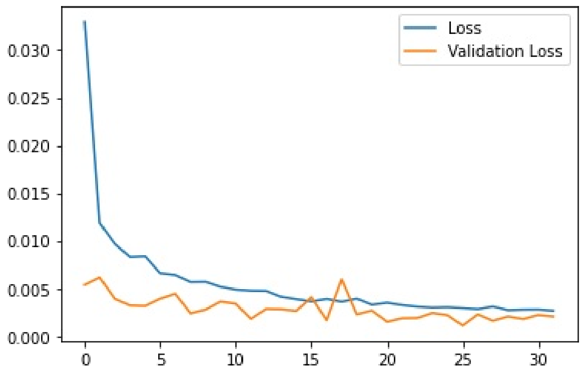
Plotting Loss Graph

- This graph is the most important one in the python notebook as it compares our loss on the training set(labelled 'Loss') to the loss on the validation set(labelled 'Validation Loss').
- The convergence of this graph tells us that we were heading towards overfitting but were stopped using EarlyStopping.

In [53]:

```
plt.plot(r.history['loss'], label='Loss')
plt.plot(r.history['val_loss'], label='Validation Loss')
plt.legend()
```

Out[53]: <matplotlib.legend.Legend at 0x28421b59f70>



Saving our model for future use in the web application

In [54]:
```
model.save('Algorithm Explanation File Model.h5')
```

Testing the model on Test Data

- We need to use the data from the previous 100 days from index 2013 for each row of the test set.
- For example, for index 2384 of test set, we need to take previous 100 days data from the train set

In [55]:
```
test_set.head()
```

Out[55]:

	Open	Close	High	Low
2384	84.599998	88.349998	89.0	84.449997
2385	85.599998	88.800003	89.0	85.599998
2386	88.000000	86.800003	89.0	85.599998
2387	86.800003	85.599998	89.0	85.000000
2388	88.750000	88.849998	89.0	85.000000

In [56]:
```
# This will give the previous 100 days
train_set.tail(100)
```

Out[56]:

	Open	Close	High	Low
2284	143.949997	144.649994	148.500000	143.949997
2285	143.000000	146.000000	146.300003	143.000000
2286	146.199997	146.550003	147.699997	144.449997
2287	147.949997	147.100006	149.149994	143.750000
2288	148.050003	146.800003	149.899994	145.550003
...	...	...	...	...
2379	94.000000	90.349998	97.699997	89.550003
2380	89.750000	90.000000	93.099998	88.000000
2381	90.000000	89.750000	94.250000	89.400002
2382	88.000000	84.949997	91.000000	81.949997
2383	84.000000	84.300003	85.849998	83.099998

100 rows × 4 columns

In [57]:
```
last_input = train_set.tail(100)
last_input.head()
```

Out[57]:

	Open	Close	High	Low
2284	143.949997	144.649994	148.500000	143.949997
2285	143.000000	146.000000	146.300003	143.000000
2286	146.199997	146.550003	147.699997	144.449997
2287	147.949997	147.100006	149.149994	143.750000
2288	148.050003	146.800003	149.899994	145.550003

In [58]:
```
test_set.head()
```

Out[58]:

	Open	Close	High	Low
2384	84.599998	88.349998	89.0	84.449997
2385	85.599998	88.800003	89.0	85.599998
2386	88.000000	86.800003	89.0	85.599998
2387	86.800003	85.599998	89.0	85.000000
2388	88.750000	88.849998	89.0	85.000000

Appedning test data to the last 100 days data

In [59]:

```
final_df = last_input.append(test_set, ignore_index=True)
final_df
```

Out[59]:

	Open	Close	High	Low
0	143.949997	144.649994	148.500000	143.949997
1	143.000000	146.000000	146.300003	143.000000
2	146.199997	146.550003	147.699997	144.449997
3	147.949997	147.100006	149.149994	143.750000
4	148.050003	146.800003	149.899994	145.550003
...	...	...	...	...
691	361.649994	344.649994	361.750000	344.649994
692	335.000000	327.450012	336.000000	327.450012
693	311.100006	334.000000	338.799988	311.100006
694	322.250000	337.799988	350.700012	320.000000
695	338.000000	329.899994	347.899994	325.000000

696 rows × 4 columns

Scaling down the final test dataframe

In [60]:

```
final_test_data = scaler.fit_transform(final_df)
final_test_data
```

Out[60]:

```
array([[0.31873783, 0.32858565, 0.32549074, 0.3385039 ],
       [0.31609675, 0.33243127, 0.31936518, 0.33574949],
       [0.32499304, 0.33399801, 0.32326325, 0.33995359],
       ...,
       [0.78343065, 0.86796751, 0.85535288, 0.82313714],
       [0.81442869, 0.87879215, 0.88848674, 0.84894171],
       [0.85821518, 0.85628825, 0.8806905 , 0.86343866]])
```

In [61]:

```
final_test_data.shape
```

Out[61]:

```
(696, 4)
```

Splitting final_test_data into x_test and y_test

In [62]:

```
x_test = []
y_test = []

for i in range(100, final_test_data.shape[0]):
    x_test.append(final_test_data[i-100: i])
    y_test.append(final_test_data[i])
```

In [63]:

```
x_test, y_test = np.array(x_test), np.array(y_test)
print(x_test.shape)
print(y_test.shape)
```

```
(596, 100, 4)
(596, 4)
```

In [64]:

```
x_test[0][0]
```

Out[64]:

```
array([0.31873783, 0.32858565, 0.32549074, 0.3385039 ])
```

In [65]:

```
y_test[0]
```

```
array([0.15373922, 0.16820965, 0.1598218 , 0.16599013])
```

Making our predictions

- To make predictions for a company, it is not feasible to train the entire model each time.
- Hence, we keep the same weights and only call model.predict() on the test data of the new company.

In [66]:
```
y_predicted = model.predict(x_test)
```

In [67]:
```
y_predicted.shape
```

Out[67]:
```
(596, 4)
```

In [68]:
```
y_predicted[0]
```

Out[68]:
```
array([0.18424352, 0.18373334, 0.19113111, 0.19536152], dtype=float32)
```

Now, we have to scale the predicted values back up to their original prices

In [69]:
```
scale_factor = scaler.scale_
scale_factor
```

Out[69]:
```
array([0.00278009, 0.0028486 , 0.00278435, 0.00289939])
```

In [70]:
```
scale_factor[0], scale_factor[1], scale_factor[2], scale_factor[3] = 1.0/scale_factor[0], 1.0/scale_factor[1], 1.
scale_factor
```

Out[70]:
```
array([359.70000076, 351.05000687, 359.14999962, 344.90000534])
```

In [71]:
```
y_predicted = y_predicted * scale_factor
y_test = y_test * scale_factor
```

In [72]:
```
y_predicted[0]
```

Out[72]:
```
array([66.27239253, 64.4995917 , 68.64473986, 67.38019095])
```

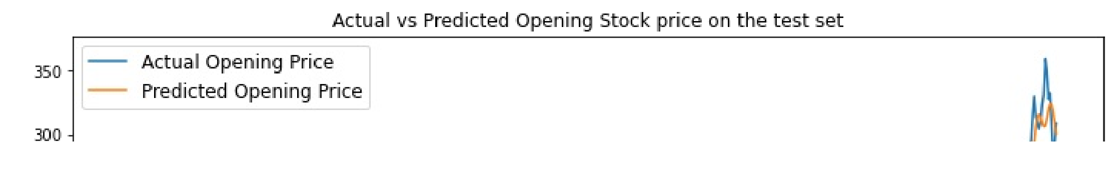
In [73]:
```
y_test[0]
```

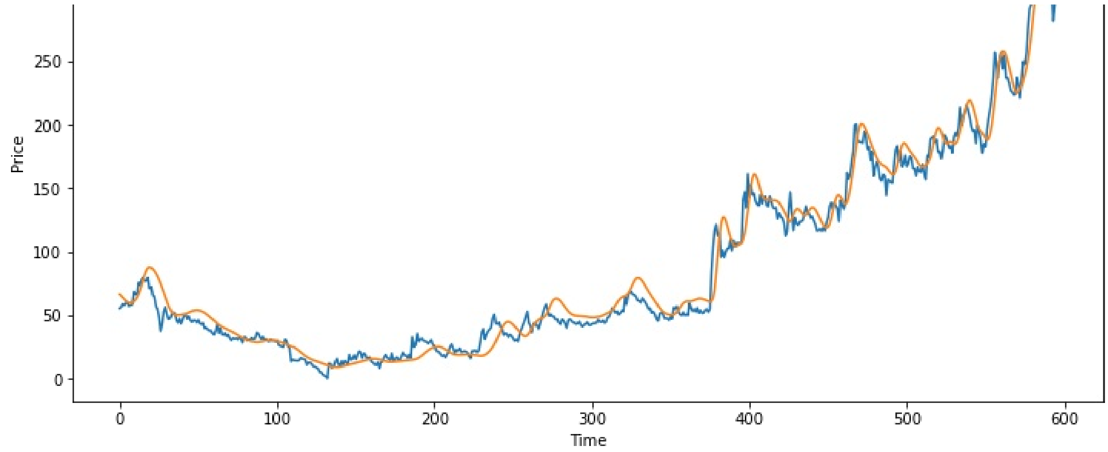
Out[73]:
```
array([55.29999924, 59.04999924, 57.39999962, 57.24999619])
```

Plotting actual vs predicted opening stock prices

- As we can see, the predicted prices form a smooth curve as it cannot keep up with the frequent fluctuations of stock prices daily

In [74]:
```
plt.figure(figsize=(12, 6))
plt.plot(y_test[:,0], label='Actual Opening Price')
plt.plot(y_predicted[:,0], label='Predicted Opening Price')
plt.xlabel('Time')
plt.ylabel('Price')
plt.title('Actual vs Predicted Opening Stock price on the test set')
plt.legend(fontsize=12)
plt.show()
```



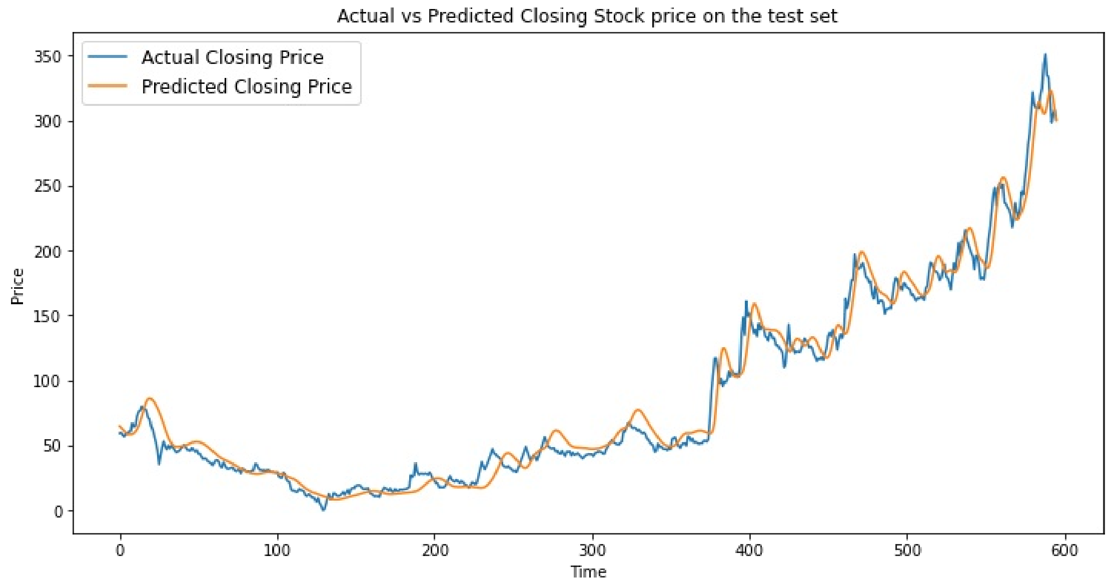


Plotting actual vs predicted closing stock prices

- As we can see, the predicted prices form a smooth curve as it cannot keep up with the frequent fluctuations of stock prices daily

In [75]:

```
plt.figure(figsize=(12, 6))
plt.plot(y_test[:,1], label='Actual Closing Price')
plt.plot(y_predicted[:,1], label='Predicted Closing Price')
plt.xlabel('Time')
plt.ylabel('Price')
plt.title('Actual vs Predicted Closing Stock price on the test set')
plt.legend(fontsize=12)
plt.show()
```



In [76]:

```
len(y_predicted)
```

Out[76]: 596

In [77]:

```
predicted_df = pd.DataFrame(y_predicted, columns=['Open', 'Close', 'High', 'Low'])
predicted_df.head()
```

Out[77]:

	Open	Close	High	Low
0	66.272393	64.499592	68.644740	67.380191
1	65.004341	63.261543	67.385119	66.172527
2	63.642733	61.936915	66.038271	64.875391
3	62.332038	60.667553	64.744297	63.627779
4	61.181627	59.559835	63.609373	62.533233

In [78]:

```
# Create a list containing dates from the starting date to the present date
import datetime
date_28 = '2022-01-28'
date_28 = datetime.datetime.strptime(date_28, '%Y-%m-%d')
date_28
```

Out[78]:

```
datetime.datetime(2022, 1, 28, 0, 0)
```

In [79]:

```
days = len(predicted_df.Close)
days
```

Out[79]:

```
596
```

In [80]:

```
d = datetime.timedelta(days=days-1)
start_date_predicted = date_28 - d
start_date_predicted
```

Out[80]:

```
datetime.datetime(2020, 6, 12, 0, 0)
```

In [81]:

```
predicted_df['Date'] = pd.date_range(start=start_date_predicted, end=date_28)
predicted_df.head()
```

Out[81]:

	Open	Close	High	Low	Date
0	66.272393	64.499592	68.644740	67.380191	2020-06-12
1	65.004341	63.261543	67.385119	66.172527	2020-06-13
2	63.642733	61.936915	66.038271	64.875391	2020-06-14
3	62.332038	60.667553	64.744297	63.627779	2020-06-15
4	61.181627	59.559835	63.609373	62.533233	2020-06-16

In [91]:

```
y_test_df = pd.DataFrame(y_test, columns=['Open', 'Close', 'High', 'Low'])
y_test_df['Date'] = pd.date_range(start=start_date_predicted, end=date_28)
y_test_df.head()
```

Out[91]:

	Open	Close	High	Low	Date
0	55.299999	59.049999	57.4	57.249996	2020-06-12
1	56.299999	59.500004	57.4	58.399998	2020-06-13
2	58.700001	57.500004	57.4	58.399998	2020-06-14
3	57.500004	56.299999	57.4	57.799999	2020-06-15
4	59.450001	59.549999	57.4	57.799999	2020-06-16

In [126…

```
candlestick_fig = go.Figure(data=[go.Candlestick(x=predicted_df.index[-30:], open=predicted_df['Open'], close=pre

candlestick_fig.show()
```