Data Engineer INTERN at HACKVEDA LIMITED

AUTHOR : BANDANA PRAKASH
TASK 2 : STOCK PREDICTION
PURPOSE : TO PREDICT THE STOCK PRICE OF A COMPANY USING LSTM.

ABOUT DATASET

Google Stock Prediction This dataset contains historical data of Google's stock prices and related attributes. It consists of 14 columns and a smaller subset of 1257 rows. Each column represents a specific attribute, and each row contains the corresponding values for that attribute.

The columns in the dataset are as follows:

Symbol: The name of the company, which is GOOG in this case.
Date: The year and date of the stock data.
Close: The closing price of Google's stock on a particular day.
High: The highest value reached by Google's stock on the given day.
Low: The lowest value reached by Google's stock on the given day.
Open: The opening value of Google's stock on the given day.
Volume: The trading volume of Google's stock on the given day, i.e., the number of shares traded.
adjClose: The adjusted closing price of Google's stock, considering factors such as dividends and stock splits.
adjHigh: The adjusted highest value reached by Google's stock on the given day.
adjLow: The adjusted lowest value reached by Google's stock on the given day.
adjOpen: The adjusted opening value of Google's stock on the given day.
adjVolume: The adjusted trading volume of Google's stock on the given day, accounting for factors such as stock splits.
divCash: The amount of cash dividend paid out to shareholders on the given day.
splitFactor: The split factor, if any, applied to Google's stock on the given day. A split factor of 1 indicates no split.

STEPS INVOLVED :

1. IMPORTING LIBRARIES AND DATA TO BE USED
2. GATHERING INSIGHTS
3. DATA PRE-PROCESSING
4. CREATING LSTM MODEL
5. VISUALIZING ACTUAL VS PREDICTED DATA
6. PREDICTING UPCOMING 15 DAYS

STEP 1 : IMPORTING LIBRARIES AND DATA TO BE USED

```
#importing libraries to be used
import numpy as np # for linear algebra
import pandas as pd # data preprocessing
import matplotlib.pyplot as plt # data visualization library
import seaborn as sns # data visualization library
%matplotlib inline
import warnings
warnings.filterwarnings('ignore') # ignore warnings

from sklearn.preprocessing import MinMaxScaler # for normalization
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, Bidirectional
```

```
# Raw URL of the CSV file
url = 'https://raw.githubusercontent.com/bandanaprakash/finalYearProject/main/MarketMinder%7C%20Real-Time%20Market%20Analysis%20and%20Fraud%20Defense/Task2_STOCK%20PREDICTION/

# Read the CSV file
df = pd.read_csv(url)

# Display the first 10 rows of the dataset
print(df.head(10))
```

```
  symbol                       date   close    high       low    open  \
0   GOOG  2016-06-14 00:00:00+00:00  718.27  722.47  713.1200  716.48
1   GOOG  2016-06-15 00:00:00+00:00  718.92  722.98  717.3100  719.00
2   GOOG  2016-06-16 00:00:00+00:00  710.36  716.65  703.2600  714.91
3   GOOG  2016-06-17 00:00:00+00:00  691.72  708.82  688.4515  708.65
4   GOOG  2016-06-20 00:00:00+00:00  693.71  702.48  693.4100  698.77
5   GOOG  2016-06-21 00:00:00+00:00  695.94  702.77  692.0100  698.40
6   GOOG  2016-06-22 00:00:00+00:00  697.46  700.86  693.0819  699.06
7   GOOG  2016-06-23 00:00:00+00:00  701.87  701.95  687.0000  697.45
8   GOOG  2016-06-24 00:00:00+00:00  675.22  689.40  673.4500  675.17
9   GOOG  2016-06-27 00:00:00+00:00  668.26  672.30  663.2840  671.00

    volume  adjClose  adjHigh    adjLow  adjOpen  adjVolume  divCash  \
0  1306065    718.27   722.47  713.1200   716.48    1306065      0.0
1  1214517    718.92   722.98  717.3100   719.00    1214517      0.0
2  1982471    710.36   716.65  703.2600   714.91    1982471      0.0
3  3402357    691.72   708.82  688.4515   708.65    3402357      0.0
4  2082538    693.71   702.48  693.4100   698.77    2082538      0.0
5  1465634    695.94   702.77  692.0100   698.40    1465634      0.0
6  1184318    697.46   700.86  693.0819   699.06    1184318      0.0
7  2171415    701.87   701.95  687.0000   697.45    2171415      0.0
8  4449022    675.22   689.40  673.4500   675.17    4449022      0.0
9  2641085    668.26   672.30  663.2840   671.00    2641085      0.0

   splitFactor
0          1.0
1          1.0
2          1.0
3          1.0
4          1.0
5          1.0
6          1.0
7          1.0
8          1.0
9          1.0
```

STEP 2 : GATHERING INSIGHTS

```
# shape of data
print("Shape of data:",df.shape)
```

```
Shape of data: (1258, 14)
```

```
# statistical description of data
df.describe()
```

	close	high	low	open	volume	adjClose	adjHigh	adjLow	adjOpen	adjVolume	divCash	splitFactor
count	1258.000000	1258.000000	1258.000000	1258.000000	1.258000e+03	1258.000000	1258.000000	1258.000000	1258.000000	1.258000e+03	1258.0	1258.0
mean	1216.317067	1227.430934	1204.176430	1215.260779	1.601590e+06	1216.317067	1227.430936	1204.176436	1215.260779	1.601590e+06	0.0	1.0
std	383.333358	387.570872	378.777094	382.446995	6.960172e+05	383.333358	387.570873	378.777099	382.446995	6.960172e+05	0.0	0.0
min	668.260000	672.300000	663.284000	671.000000	3.467530e+05	668.260000	672.300000	663.284000	671.000000	3.467530e+05	0.0	1.0
25%	960.802500	968.757500	952.182500	959.005000	1.173522e+06	960.802500	968.757500	952.182500	959.005000	1.173522e+06	0.0	1.0
50%	1132.460000	1143.935000	1117.915000	1131.150000	1.412588e+06	1132.460000	1143.935000	1117.915000	1131.150000	1.412588e+06	0.0	1.0
75%	1360.595000	1374.345000	1348.557500	1361.075000	1.812156e+06	1360.595000	1374.345000	1348.557500	1361.075000	1.812156e+06	0.0	1.0
max	2521.600000	2526.990000	2498.290000	2524.920000	6.207027e+06	2521.600000	2526.990000	2498.290000	2524.920000	6.207027e+06	0.0	1.0

```
# summary of data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1258 entries, 0 to 1257
Data columns (total 14 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   symbol       1258 non-null   object
 1   date         1258 non-null   object
 2   close        1258 non-null   float64
 3   high         1258 non-null   float64
 4   low          1258 non-null   float64
 5   open         1258 non-null   float64
 6   volume       1258 non-null   int64
 7   adjClose     1258 non-null   float64
 8   adjHigh      1258 non-null   float64
 9   adjLow       1258 non-null   float64
 10  adjOpen      1258 non-null   float64
 11  adjVolume    1258 non-null   int64
 12  divCash      1258 non-null   float64
 13  splitFactor  1258 non-null   float64
dtypes: float64(10), int64(2), object(2)
memory usage: 137.7+ KB
```

```
# checking null values
df.isnull().sum()
```

	0
symbol	0
date	0
close	0
high	0
low	0
open	0
volume	0
adjClose	0
adjHigh	0
adjLow	0
adjOpen	0
adjVolume	0
divCash	0
splitFactor	0

dtype: int64

There are no null values in the dataset

```
df = df[['date','open','close']] # Extracting required columns
df['date'] = pd.to_datetime(df['date'].apply(lambda x: x.split()[0])) # converting object dtype of date column to datetime dtype
df.set_index('date',drop=True,inplace=True) # Setting date column as index
df.head(10)
```

date	open	close
2016-06-14	716.48	718.27
2016-06-15	719.00	718.92
2016-06-16	714.91	710.36
2016-06-17	708.65	691.72
2016-06-20	698.77	693.71
2016-06-21	698.40	695.94
2016-06-22	699.06	697.46
2016-06-23	697.45	701.87
2016-06-24	675.17	675.22
2016-06-27	671.00	668.26

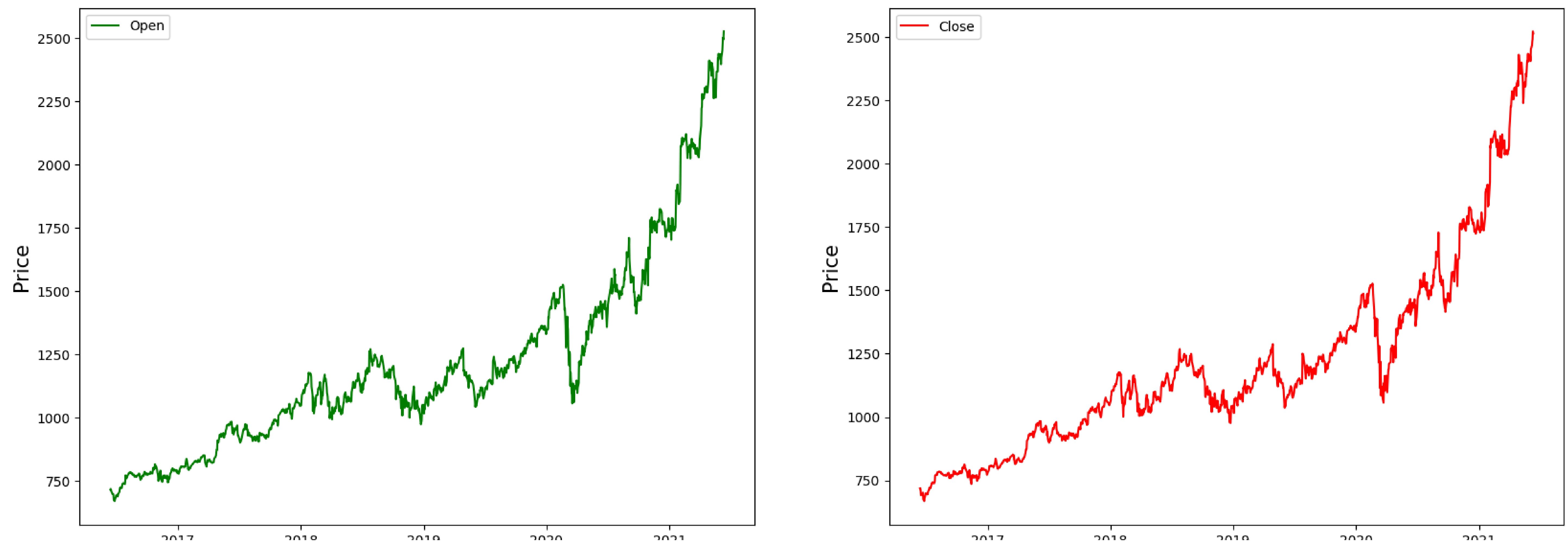
Next steps: Generate code with df | View recommended plots | New interactive sheet

```
# plotting open and closing price on date index
fig, ax =plt.subplots(1,2,figsize=(20,7))
```

```
ax[0].plot(df['open'],label='Open',color='green')
ax[0].set_xlabel('Date',size=15)
ax[0].set_ylabel('Price',size=15)
ax[0].legend()

ax[1].plot(df['close'],label='Close',color='red')
ax[1].set_xlabel('Date',size=15)
ax[1].set_ylabel('Price',size=15)
ax[1].legend()

fig.show()
```



STEP 3 : DATA PRE-PROCESSING

```
# normalizing all the values of all columns using MinMaxScaler
MMS = MinMaxScaler()
df[df.columns] = MMS.fit_transform(df)
df.head(10)
```

date	open	close
2016-06-14	0.024532	0.026984
2016-06-15	0.025891	0.027334
2016-06-16	0.023685	0.022716
2016-06-17	0.020308	0.012658
2016-06-20	0.014979	0.013732
2016-06-21	0.014779	0.014935
2016-06-22	0.015135	0.015755
2016-06-23	0.014267	0.018135
2016-06-24	0.002249	0.003755
2016-06-27	0.000000	0.000000

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
# splitting the data into training and test set
training_size = round(len(df) * 0.75) # Selecting 75 % for training and 25 % for testing
training_size
```

```
944
```

```
train_data = df[:training_size]
test_data  = df[training_size:]

train_data.shape, test_data.shape
```

```
((944, 2), (314, 2))
```

```
# Function to create sequence of data for training and testing

def create_sequence(dataset):
  sequences = []
  labels = []

  start_idx = 0

  for stop_idx in range(50,len(dataset)): # Selecting 50 rows at a time
    sequences.append(dataset.iloc[start_idx:stop_idx])
    labels.append(dataset.iloc[stop_idx])
    start_idx += 1
  return (np.array(sequences),np.array(labels))


train_seq, train_label = create_sequence(train_data)
test_seq, test_label = create_sequence(test_data)
train_seq.shape, train_label.shape, test_seq.shape, test_label.shape
```

```
((894, 50, 2), (894, 2), (264, 50, 2), (264, 2))
```

STEP 4 : CREATING LSTM MODEL

```
# imported Sequential from keras.models
model = Sequential()
```

```
# importing Dense, Dropout, LSTM, Bidirectional from keras.layers
model.add(LSTM(units=50, return_sequences=True, input_shape = (train_seq.shape[1], train_seq.shape[2])))

model.add(Dropout(0.1))
model.add(LSTM(units=50))

model.add(Dense(2))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_absolute_error'])

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 50, 50)	10,600
dropout (Dropout)	(None, 50, 50)	0
lstm_1 (LSTM)	(None, 50)	20,200
dense (Dense)	(None, 2)	102

Total params: 30,902 (120.71 KB)
Trainable params: 30,902 (120.71 KB)
Non-trainable params: 0 (0.00 B)

```
# fitting the model by iterating the dataset over 100 times(100 epochs)
model.fit(train_seq, train_label, epochs=100,validation_data=(test_seq, test_label), verbose=1)
```

```
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 57ms/step - loss: 1.5238e-04 - mean_absolute_error: 0.0089 - val_loss: 0.0030 - val_mean_absolute_error: 0.0448
Epoch 73/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 65ms/step - loss: 1.5071e-04 - mean_absolute_error: 0.0089 - val_loss: 0.0029 - val_mean_absolute_error: 0.0427
Epoch 74/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 1.5754e-04 - mean_absolute_error: 0.0093 - val_loss: 0.0037 - val_mean_absolute_error: 0.0498
Epoch 75/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 53ms/step - loss: 1.2796e-04 - mean_absolute_error: 0.0084 - val_loss: 0.0030 - val_mean_absolute_error: 0.0449
Epoch 76/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 1.4075e-04 - mean_absolute_error: 0.0087 - val_loss: 0.0029 - val_mean_absolute_error: 0.0425
Epoch 77/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 1.5298e-04 - mean_absolute_error: 0.0092 - val_loss: 0.0016 - val_mean_absolute_error: 0.0302
Epoch 78/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 84ms/step - loss: 1.4958e-04 - mean_absolute_error: 0.0088 - val_loss: 0.0017 - val_mean_absolute_error: 0.0318
Epoch 79/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 58ms/step - loss: 1.4603e-04 - mean_absolute_error: 0.0087 - val_loss: 0.0018 - val_mean_absolute_error: 0.0334
Epoch 80/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.4834e-04 - mean_absolute_error: 0.0090 - val_loss: 0.0012 - val_mean_absolute_error: 0.0270
Epoch 81/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 1.3472e-04 - mean_absolute_error: 0.0083 - val_loss: 0.0050 - val_mean_absolute_error: 0.0594
Epoch 82/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 53ms/step - loss: 1.3095e-04 - mean_absolute_error: 0.0084 - val_loss: 0.0054 - val_mean_absolute_error: 0.0618
Epoch 83/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 1.4732e-04 - mean_absolute_error: 0.0091 - val_loss: 0.0054 - val_mean_absolute_error: 0.0614
Epoch 84/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 4s 87ms/step - loss: 1.5452e-04 - mean_absolute_error: 0.0088 - val_loss: 0.0036 - val_mean_absolute_error: 0.0494
Epoch 85/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.2542e-04 - mean_absolute_error: 0.0080 - val_loss: 0.0034 - val_mean_absolute_error: 0.0474
Epoch 86/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 1.3994e-04 - mean_absolute_error: 0.0085 - val_loss: 0.0039 - val_mean_absolute_error: 0.0509
Epoch 87/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.4788e-04 - mean_absolute_error: 0.0086 - val_loss: 0.0014 - val_mean_absolute_error: 0.0284
Epoch 88/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.3139e-04 - mean_absolute_error: 0.0082 - val_loss: 0.0012 - val_mean_absolute_error: 0.0258
Epoch 89/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 1.4381e-04 - mean_absolute_error: 0.0083 - val_loss: 0.0011 - val_mean_absolute_error: 0.0260
Epoch 90/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 75ms/step - loss: 1.2231e-04 - mean_absolute_error: 0.0079 - val_loss: 0.0027 - val_mean_absolute_error: 0.0405
Epoch 91/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 69ms/step - loss: 1.2541e-04 - mean_absolute_error: 0.0082 - val_loss: 0.0028 - val_mean_absolute_error: 0.0427
Epoch 92/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.1055e-04 - mean_absolute_error: 0.0077 - val_loss: 0.0014 - val_mean_absolute_error: 0.0289
Epoch 93/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 67ms/step - loss: 1.3745e-04 - mean_absolute_error: 0.0085 - val_loss: 0.0019 - val_mean_absolute_error: 0.0348
Epoch 94/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 4s 115ms/step - loss: 1.2206e-04 - mean_absolute_error: 0.0080 - val_loss: 0.0012 - val_mean_absolute_error: 0.0263
Epoch 95/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - loss: 1.4256e-04 - mean_absolute_error: 0.0090 - val_loss: 9.3829e-04 - val_mean_absolute_error: 0.0232
Epoch 96/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 2s 59ms/step - loss: 1.1209e-04 - mean_absolute_error: 0.0075 - val_loss: 7.4422e-04 - val_mean_absolute_error: 0.0206
Epoch 97/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 1.2002e-04 - mean_absolute_error: 0.0078 - val_loss: 0.0021 - val_mean_absolute_error: 0.0376
Epoch 98/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 1.0345e-04 - mean_absolute_error: 0.0075 - val_loss: 0.0015 - val_mean_absolute_error: 0.0305
Epoch 99/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 1.1212e-04 - mean_absolute_error: 0.0078 - val_loss: 5.8592e-04 - val_mean_absolute_error: 0.0183
Epoch 100/100
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 1.0687e-04 - mean_absolute_error: 0.0074 - val_loss: 0.0019 - val_mean_absolute_error: 0.0335
<keras.src.callbacks.history.History at 0x7a68db5db9d0>
```

```
# predicting the values after running the model
test_predicted = model.predict(test_seq)
test_predicted[:5]
```

```
9/9 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step
array([[0.40194753, 0.40702906],
       [0.40166208, 0.4065613 ],
       [0.39763585, 0.40240565],
       [0.4002833 , 0.4054701 ],
       [0.40377948, 0.40909466]], dtype=float32)
```

```
# Inversing normalization/scaling on predicted data
test_inverse_predicted = MMS.inverse_transform(test_predicted)
test_inverse_predicted[:5]
```

```
array([[1416.1785, 1422.6233],
       [1415.6493, 1421.7562],
       [1408.185 , 1414.0544],
       [1413.0931, 1419.734 ],
       [1419.5748, 1426.4515]], dtype=float32)
```

STEP 5 : VISUALIZING ACTUAL VS PREDICTED DATA

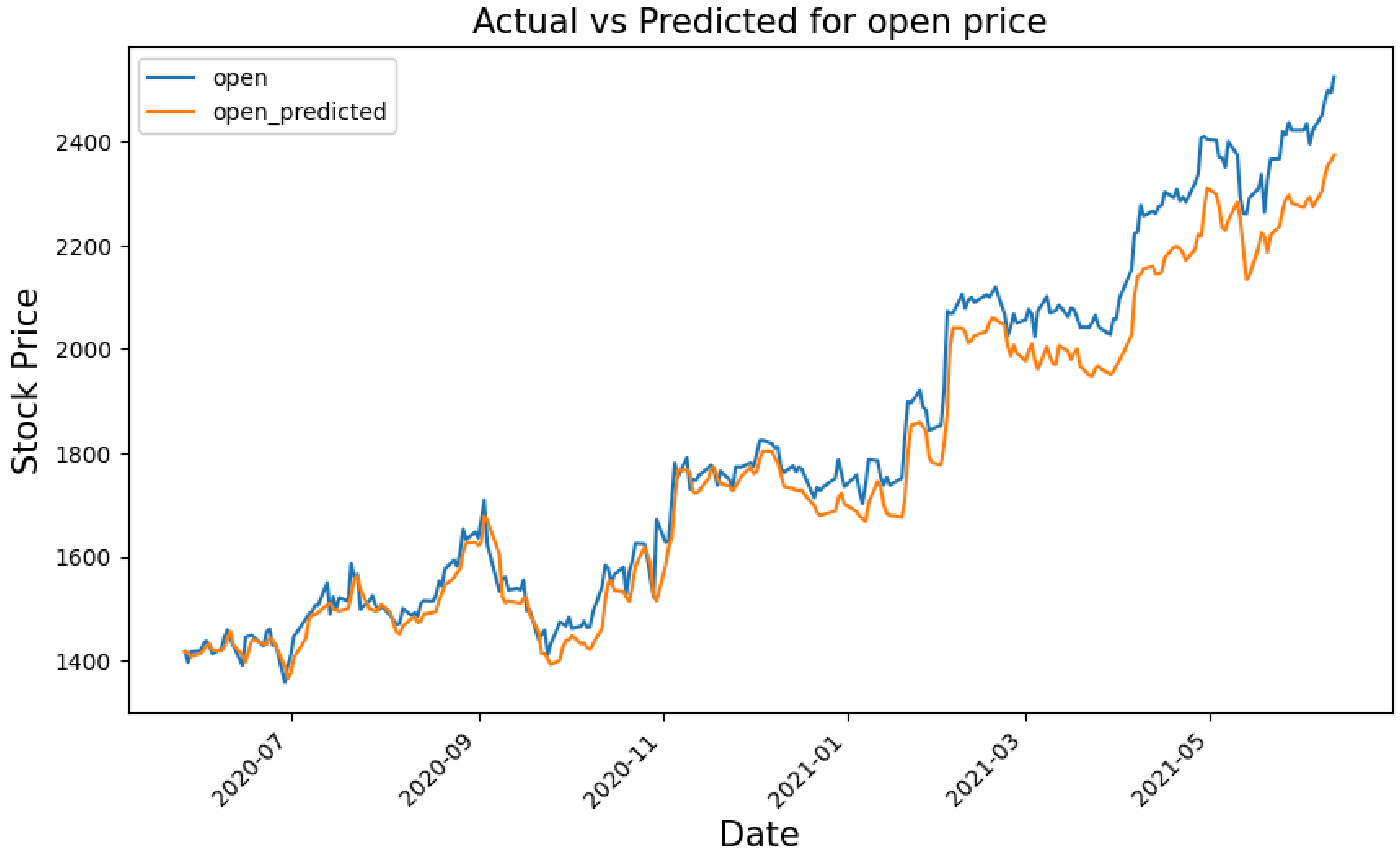
```
# Merging actual and predicted data for better visualization
df_merge = pd.concat([df.iloc[-264:].copy(),
                      pd.DataFrame(test_inverse_predicted,columns=['open_predicted','close_predicted'],
                                   index=df.iloc[-264:].index)], axis=1)
```

```
# Inversing normalization/scaling
df_merge[['open','close']] = MMS.inverse_transform(df_merge[['open','close']])
df_merge.head()
```

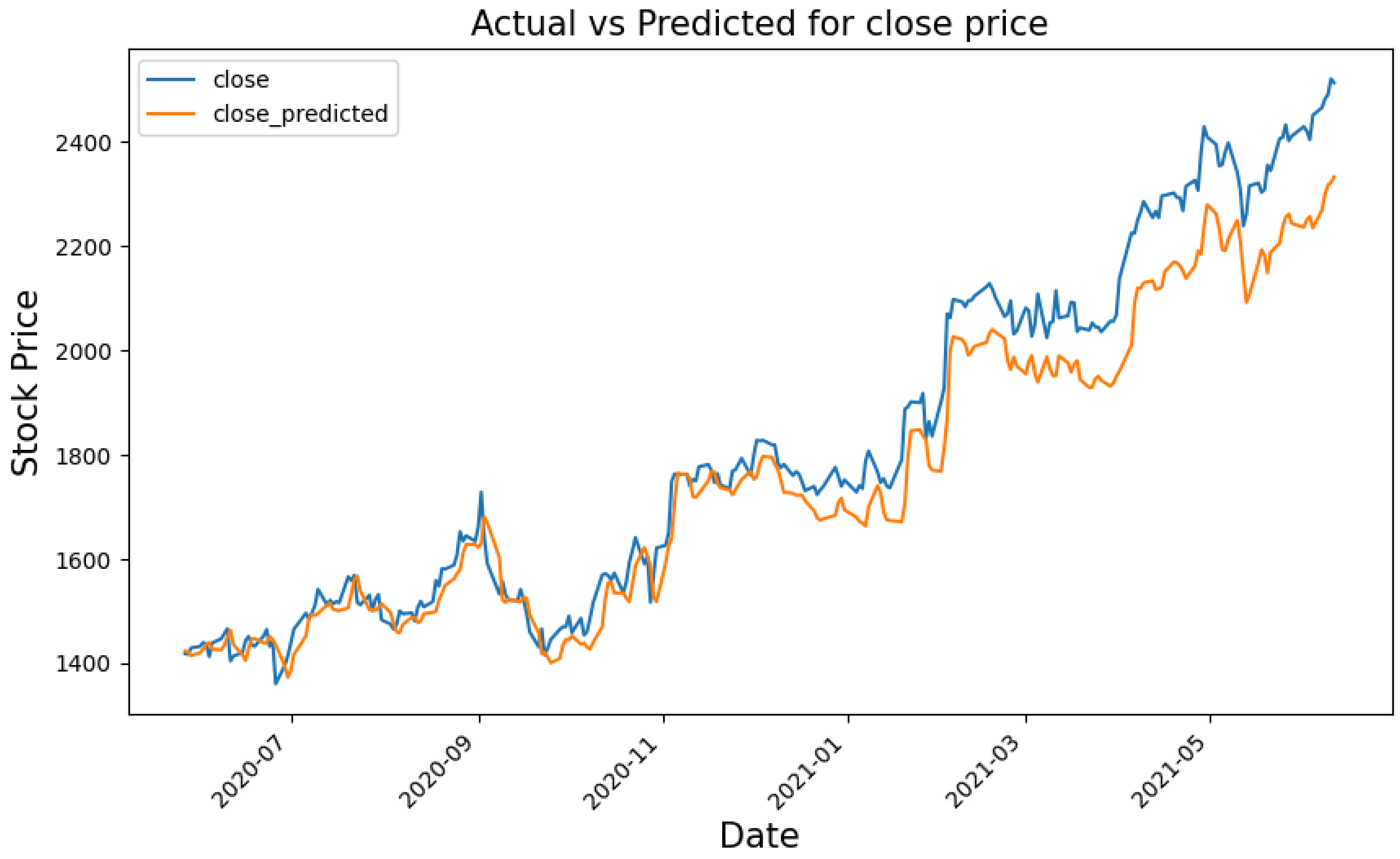
date	open	close	open_predicted	close_predicted
2020-05-27	1417.25	1417.84	1416.178467	1422.623291
2020-05-28	1396.86	1416.73	1415.649292	1421.756226
2020-05-29	1416.94	1428.92	1408.185059	1414.054443
2020-06-01	1418.39	1431.82	1413.093140	1419.734009
2020-06-02	1430.55	1439.22	1419.574829	1426.451538

Next steps: Generate code with df_merge | View recommended plots | New interactive sheet

```
# plotting the actual open and predicted open prices on date index
df_merge[['open','open_predicted']].plot(figsize=(10,6))
plt.xticks(rotation=45)
plt.xlabel('Date',size=15)
plt.ylabel('Stock Price',size=15)
plt.title('Actual vs Predicted for open price',size=15)
plt.show()
```




```
# plotting the actual close and predicted close prices on date index
df_merge[['close','close_predicted']].plot(figsize=(10,6))
plt.xticks(rotation=45)
plt.xlabel('Date',size=15)
plt.ylabel('Stock Price',size=15)
plt.title('Actual vs Predicted for close price',size=15)
plt.show()
```



STEP 6. PREDICTING UPCOMING 10 DAYS

```
new_index = pd.date_range(start=df_merge.index[-1] + pd.Timedelta(days=1), periods=10, freq='D')
new_rows = pd.DataFrame(columns=df_merge.columns, index=new_index)
df_merge = pd.concat([df_merge, new_rows])
```

```
df_merge.loc['2021-06-09':'2021-06-16']
```

	open	close	open_predicted	close_predicted
2021-06-09	2499.50	2491.40	2355.756104	2317.685059
2021-06-10	2494.01	2521.60	2362.784912	2322.546387
2021-06-11	2524.92	2513.93	2374.123047	2333.229492
2021-06-12	NaN	NaN	NaN	NaN
2021-06-13	NaN	NaN	NaN	NaN
2021-06-14	NaN	NaN	NaN	NaN
2021-06-15	NaN	NaN	NaN	NaN
2021-06-16	NaN	NaN	NaN	NaN

```
# creating a DataFrame and filling values of open and close column
upcoming_prediction = pd.DataFrame(columns=['open','close'],index=df_merge.index)
upcoming_prediction.index=pd.to_datetime(upcoming_prediction.index)
```

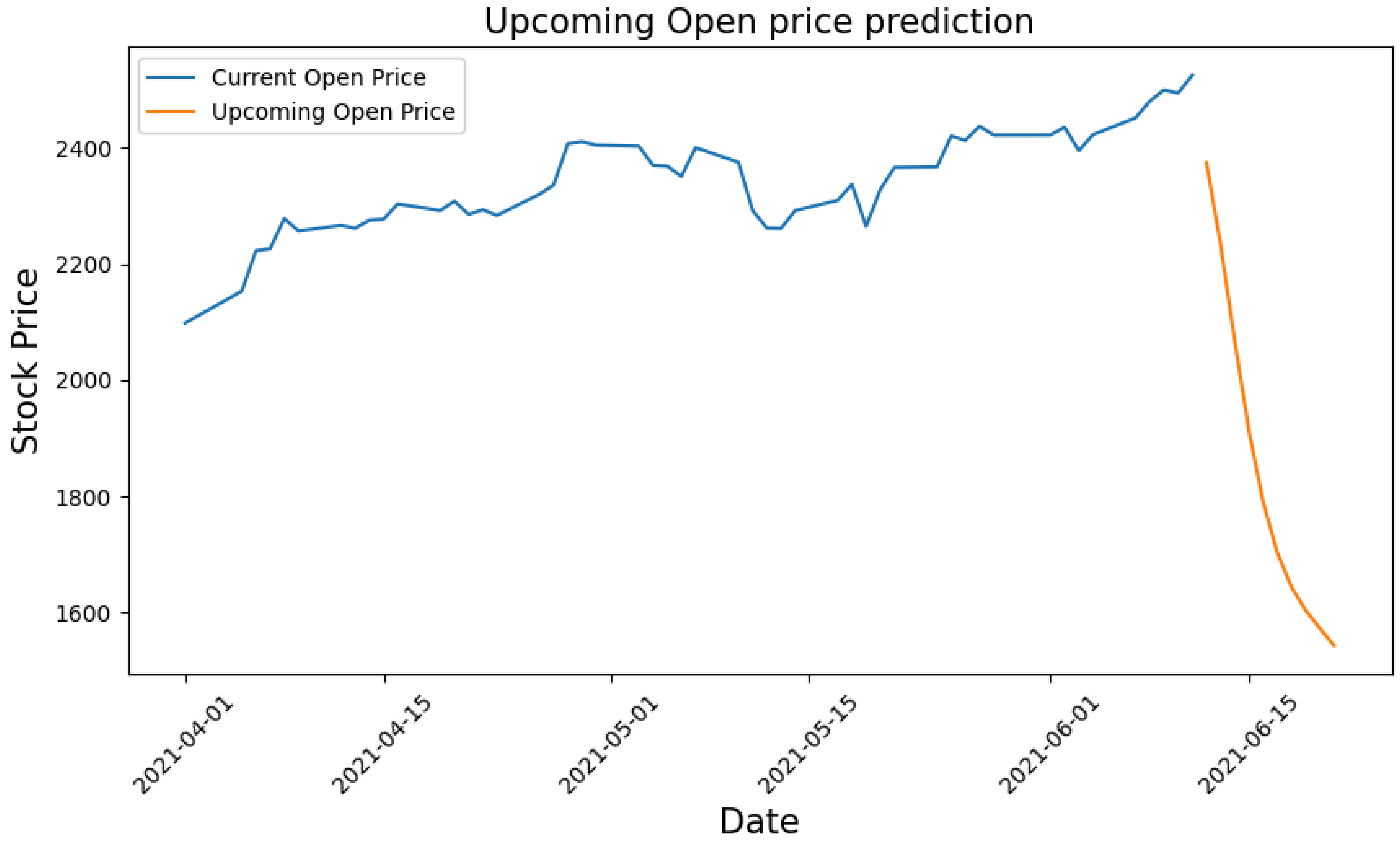
```
curr_seq = test_seq[-1:]

for i in range(-10,0):
  up_pred = model.predict(curr_seq)
  upcoming_prediction.iloc[i] = up_pred
  curr_seq = np.append(curr_seq[0][1:],up_pred,axis=0)
  curr_seq = curr_seq.reshape(test_seq[-1:].shape)
```

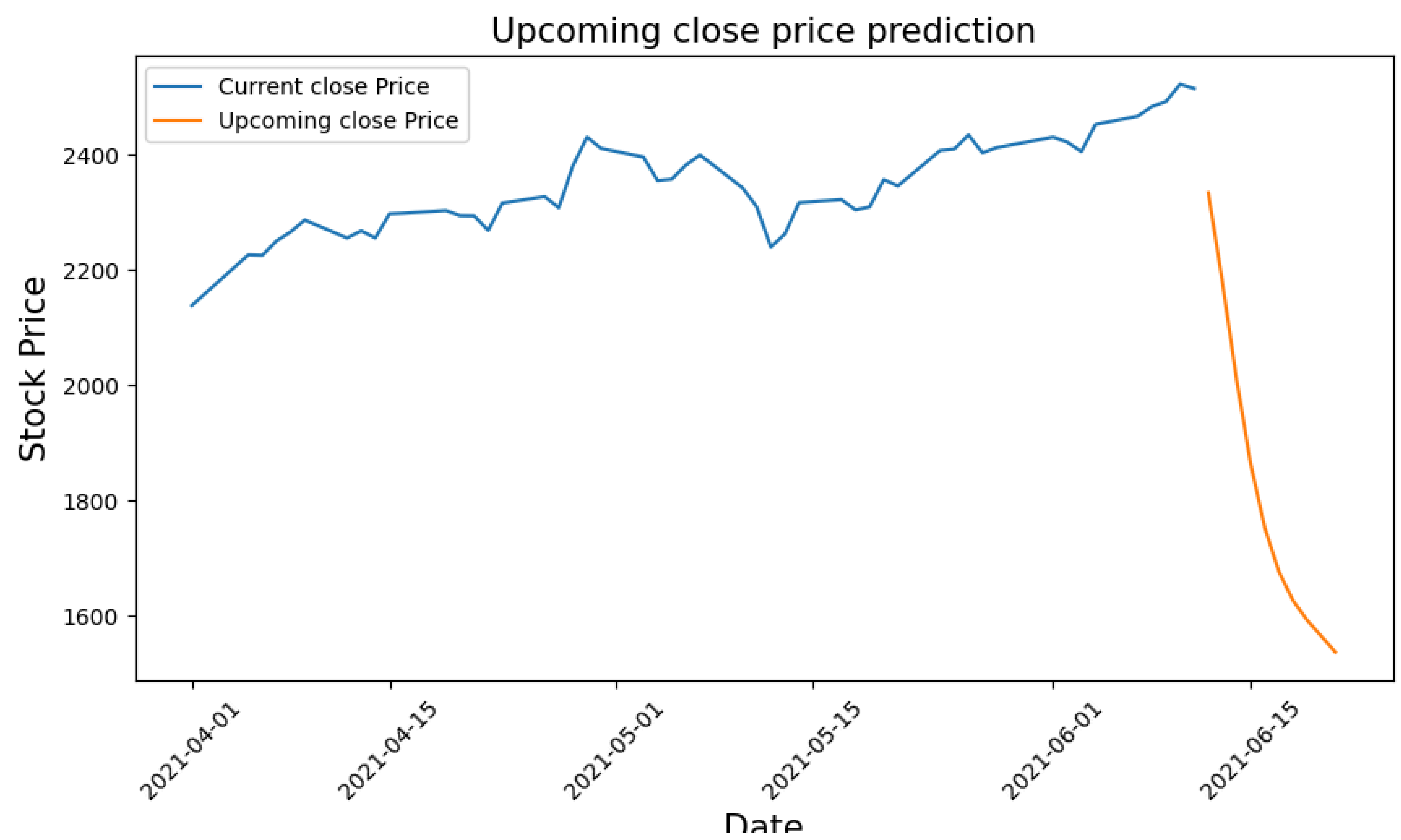
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
```

```
# inversing Normalization/scaling
upcoming_prediction[['open','close']] = MMS.inverse_transform(upcoming_prediction[['open','close']])
```

```
# plotting Upcoming Open price on date index
fig,ax=plt.subplots(figsize=(10,5))
ax.plot(df_merge.loc['2021-04-01':,'open'],label='Current Open Price')
ax.plot(upcoming_prediction.loc['2021-04-01':,'open'],label='Upcoming Open Price')
plt.setp(ax.xaxis.get_majorticklabels(), rotation=45)
ax.set_xlabel('Date',size=15)
ax.set_ylabel('Stock Price',size=15)
ax.set_title('Upcoming Open price prediction',size=15)
ax.legend()
fig.show()
```




```
# plotting Upcoming Close price on date index
fig,ax=plt.subplots(figsize=(10,5))
ax.plot(df_merge.loc['2021-04-01':,'close'],label='Current close Price')
ax.plot(upcoming_prediction.loc['2021-04-01':,'close'],label='Upcoming close Price')
plt.setp(ax.xaxis.get_majorticklabels(), rotation=45)
ax.set_xlabel('Date',size=15)
ax.set_ylabel('Stock Price',size=15)
ax.set_title('Upcoming close price prediction',size=15)
ax.legend()
fig.show()
```



Upcoming close price prediction
Current close Price
Upcoming close Price
Stock Price
2400
2200
2000
1800
1600
2021-04-01
2021-04-15
2021-05-01
2021-05-15
2021-06-01
2021-06-15
Date