Task 1: Stock Prediction: In this enthralling task, our team dives deep into the world of stock market forecasting. Armed with a comprehensive dataset of **Google's** historical stock prices and attributes, the challenge begins. Guided by the power of Long Short-Term Memory networks (LSTM), we embark on an 8-step journey together:

Data Exploration: Delving into 14 insightful columns and 1257 rows, understanding attributes like close, high, low, open, volume, and more.

Data Preprocessing: From handling missing values to scaling data using MinMaxScaler, ensuring the dataset is primed for analysis.

LSTM Dataset Creation: Crafting sequences of data for LSTM, a pivotal step towards accurate predictions.

Model Building: Constructing a robust LSTM model using layers like LSTM, Dropout, and Dense, tuned to predict stock prices effectively.

Training the Model: Iterating over the dataset 100 times, training the LSTM model to grasp intricate patterns.

Evaluation and Visualization: Assessing the model's accuracy using metrics like Root Mean Squared Error (RMSE) and visualizing predicted prices against actual data for validation.

Future Predictions: Peering into the future, predicting the next 30 days' open and close stock prices.

Upcoming Price Visualization: Showcasing the upcoming stock price predictions, bridging the gap between data science and real-world applications.

This task isn't just about predicting numbers; it's about understanding market dynamics, leveraging cutting-edge technology, and making informed decisions. Join me as I decode the language of stock prices and unlock the secrets hidden within the data.

The dataset: https://www.kaggle.com/datasets/shreenidhihipparagi/google-stock-prediction

STEP 1 - Data Loading and Inspection

```
#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


df = pd.read_csv('Google Stocks.csv') # data_importing
df.head(20) # fetching first 20 rows of dataset
```

	symbol	date	close	high	low	open	volume	adjClose	adjHigh	adjLow	adjOpen	adjVolume	divCash	splitFactor
0	GOOG	2016-06-14 00:00:00+00:00	718.27	722.4700	713.1200	716.48	1306065	718.27	722.4700	713.1200	716.48	1306065	0.0	1.0
1	GOOG	2016-06-15 00:00:00+00:00	718.92	722.9800	717.3100	719.00	1214517	718.92	722.9800	717.3100	719.00	1214517	0.0	1.0
2	GOOG	2016-06-16 00:00:00+00:00	710.36	716.6500	703.2600	714.91	1982471	710.36	716.6500	703.2600	714.91	1982471	0.0	1.0
3	GOOG	2016-06-17 00:00:00+00:00	691.72	708.8200	688.4515	708.65	3402357	691.72	708.8200	688.4515	708.65	3402357	0.0	1.0
4	GOOG	2016-06-20 00:00:00+00:00	693.71	702.4800	693.4100	698.77	2082538	693.71	702.4800	693.4100	698.77	2082538	0.0	1.0
5	GOOG	2016-06-21 00:00:00+00:00	695.94	702.7700	692.0100	698.40	1465634	695.94	702.7700	692.0100	698.40	1465634	0.0	1.0
6	GOOG	2016-06-22 00:00:00+00:00	697.46	700.8600	693.0819	699.06	1184318	697.46	700.8600	693.0819	699.06	1184318	0.0	1.0
7	GOOG	2016-06-23 00:00:00+00:00	701.87	701.9500	687.0000	697.45	2171415	701.87	701.9500	687.0000	697.45	2171415	0.0	1.0
8	GOOG	2016-06-24 00:00:00+00:00	675.22	689.4000	673.4500	675.17	4449022	675.22	689.4000	673.4500	675.17	4449022	0.0	1.0
9	GOOG	2016-06-27 00:00:00+00:00	668.26	672.3000	663.2840	671.00	2641085	668.26	672.3000	663.2840	671.00	2641085	0.0	1.0
10	GOOG	2016-06-28 00:00:00+00:00	680.04	680.3300	673.0000	678.97	2173762	680.04	680.3300	673.0000	678.97	2173762	0.0	1.0
11	GOOG	2016-06-29 00:00:00+00:00	684.11	687.4292	681.4100	683.00	1932561	684.11	687.4292	681.4100	683.00	1932561	0.0	1.0
12	GOOG	2016-06-30 00:00:00+00:00	692.10	692.3200	683.6500	685.47	1597714	692.10	692.3200	683.6500	685.47	1597714	0.0	1.0
13	GOOG	2016-07-01 00:00:00+00:00	699.21	700.6500	692.1301	692.20	1344710	699.21	700.6500	692.1301	692.20	1344710	0.0	1.0
14	GOOG	2016-07-05 00:00:00+00:00	694.49	696.9400	688.8800	696.06	1462616	694.49	696.9400	688.8800	696.06	1462616	0.0	1.0
15	GOOG	2016-07-06 00:00:00+00:00	697.77	701.6800	689.0900	689.98	1411925	697.77	701.6800	689.0900	689.98	1411925	0.0	1.0
16	GOOG	2016-07-07 00:00:00+00:00	695.36	698.2000	688.2150	698.08	1304200	695.36	698.2000	688.2150	698.08	1304200	0.0	1.0
17	GOOG	2016-07-08 00:00:00+00:00	705.63	705.7100	696.4350	699.50	1575166	705.63	705.7100	696.4350	699.50	1575166	0.0	1.0
18	GOOG	2016-07-11 00:00:00+00:00	715.09	716.5100	707.2400	708.05	1111762	715.09	716.5100	707.2400	708.05	1111762	0.0	1.0
19	GOOG	2016-07-12 00:00:00+00:00	720.64	722.9400	715.9100	719.12	1336921	720.64	722.9400	715.9100	719.12	1336921	0.0	1.0

```
# 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()
```

```
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
```

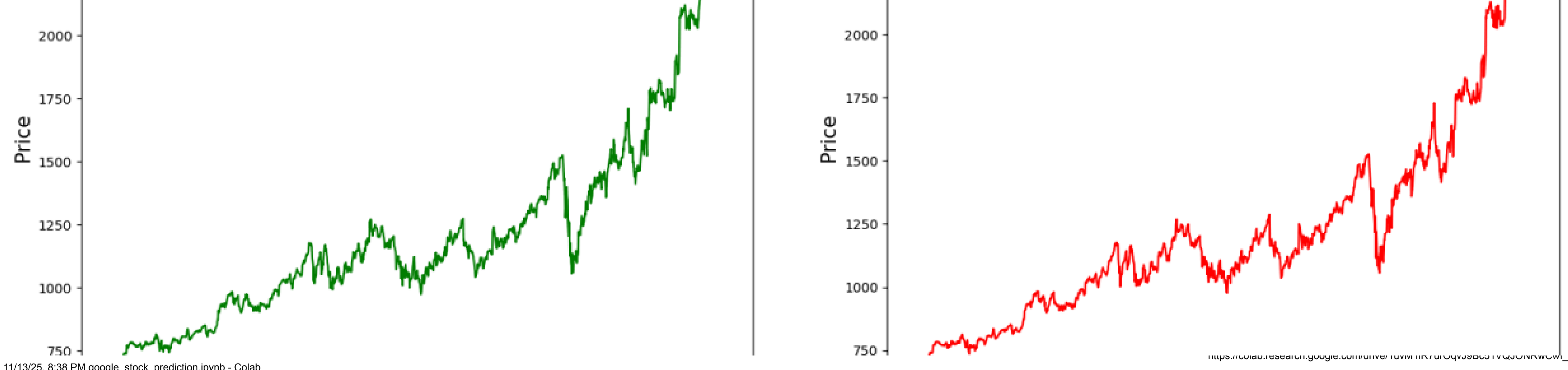
STEP 2 - Data Preprocessing

```
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(20)
```

	open	close
date		
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
2016-06-28	678.97	680.04
2016-06-29	683.00	684.11
2016-06-30	685.47	692.10
2016-07-01	692.20	699.21
2016-07-05	696.06	694.49
2016-07-06	689.98	697.77
2016-07-07	698.08	695.36
2016-07-08	699.50	705.63
2016-07-11	708.05	715.09
2016-07-12	719.12	720.64

STEP 3 - Data Visualization

```
# 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()
```




```
# normalizing all the values of all columns using MinMaxScaler
MMS = MinMaxScaler()
df[df.columns] = MMS.fit_transform(df)
df.head(20)
```

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
2016-06-28	0.004299	0.006356
2016-06-29	0.006473	0.008552
2016-06-30	0.007805	0.012863
2016-07-01	0.011435	0.016700
2016-07-05	0.013517	0.014153
2016-07-06	0.010238	0.015923
2016-07-07	0.014607	0.014622
2016-07-08	0.015373	0.020164
2016-07-11	0.019985	0.025268
2016-07-12	0.025956	0.028262

STEP 4 - Data Splitting and Sequencing

```
# 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 5 - Model Building and Training

```
# 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) 10600

 dropout (Dropout) (None, 50, 50) 0

 lstm_1 (LSTM) (None, 50) 20200

 dense (Dense) (None, 2) 102

=================================================================
Total params: 30902 (120.71 KB)
Trainable params: 30902 (120.71 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# 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)
```

```
Epoch 1/100
28/28 [==============================] - 12s 129ms/step - loss: 0.0079 - mean_absolute_error: 0.0608 - val_loss: 0.0235 - val_mean_absolute_error: 0.1295
Epoch 2/100
28/28 [==============================] - 2s 74ms/step - loss: 7.2210e-04 - mean_absolute_error: 0.0210 - val_loss: 0.0066 - val_mean_absolute_error: 0.0636
Epoch 3/100
28/28 [==============================] - 2s 76ms/step - loss: 4.7917e-04 - mean_absolute_error: 0.0162 - val_loss: 0.0063 - val_mean_absolute_error: 0.0625
Epoch 4/100
28/28 [==============================] - 2s 90ms/step - loss: 4.4424e-04 - mean_absolute_error: 0.0155 - val_loss: 0.0037 - val_mean_absolute_error: 0.0461
Epoch 5/100
28/28 [==============================] - 2s 87ms/step - loss: 4.8671e-04 - mean_absolute_error: 0.0158 - val_loss: 0.0059 - val_mean_absolute_error: 0.0600
Epoch 6/100
28/28 [==============================] - 3s 94ms/step - loss: 3.9539e-04 - mean_absolute_error: 0.0145 - val_loss: 0.0049 - val_mean_absolute_error: 0.0536
Epoch 7/100
28/28 [==============================] - 3s 94ms/step - loss: 4.3966e-04 - mean_absolute_error: 0.0154 - val_loss: 0.0051 - val_mean_absolute_error: 0.0554
Epoch 8/100
28/28 [==============================] - 2s 67ms/step - loss: 3.8020e-04 - mean_absolute_error: 0.0141 - val_loss: 0.0064 - val_mean_absolute_error: 0.0636
Epoch 9/100
28/28 [==============================] - 2s 59ms/step - loss: 3.8742e-04 - mean_absolute_error: 0.0144 - val_loss: 0.0050 - val_mean_absolute_error: 0.0551
Epoch 10/100
28/28 [==============================] - 2s 62ms/step - loss: 3.8903e-04 - mean_absolute_error: 0.0144 - val_loss: 0.0063 - val_mean_absolute_error: 0.0640
Epoch 11/100
28/28 [==============================] - 3s 96ms/step - loss: 4.2285e-04 - mean_absolute_error: 0.0150 - val_loss: 0.0050 - val_mean_absolute_error: 0.0553
Epoch 12/100
28/28 [==============================] - 3s 96ms/step - loss: 3.5847e-04 - mean_absolute_error: 0.0139 - val_loss: 0.0049 - val_mean_absolute_error: 0.0548
Epoch 13/100
28/28 [==============================] - 3s 96ms/step - loss: 3.4082e-04 - mean_absolute_error: 0.0135 - val_loss: 0.0048 - val_mean_absolute_error: 0.0540
Epoch 14/100
28/28 [==============================] - 3s 93ms/step - loss: 3.4199e-04 - mean_absolute_error: 0.0136 - val_loss: 0.0049 - val_mean_absolute_error: 0.0559
Epoch 15/100
28/28 [==============================] - 2s 70ms/step - loss: 3.2761e-04 - mean_absolute_error: 0.0133 - val_loss: 0.0037 - val_mean_absolute_error: 0.0464
Epoch 16/100
28/28 [==============================] - 2s 66ms/step - loss: 3.3767e-04 - mean_absolute_error: 0.0132 - val_loss: 0.0033 - val_mean_absolute_error: 0.0438
Epoch 17/100
28/28 [==============================] - 2s 80ms/step - loss: 3.1101e-04 - mean_absolute_error: 0.0129 - val_loss: 0.0063 - val_mean_absolute_error: 0.0654
Epoch 18/100
28/28 [==============================] - 3s 94ms/step - loss: 3.2094e-04 - mean_absolute_error: 0.0132 - val_loss: 0.0044 - val_mean_absolute_error: 0.0530
Epoch 19/100
28/28 [==============================] - 3s 91ms/step - loss: 3.1545e-04 - mean_absolute_error: 0.0128 - val_loss: 0.0049 - val_mean_absolute_error: 0.0559
Epoch 20/100
28/28 [==============================] - 3s 93ms/step - loss: 2.9907e-04 - mean_absolute_error: 0.0128 - val_loss: 0.0057 - val_mean_absolute_error: 0.0606
Epoch 21/100
28/28 [==============================] - 3s 93ms/step - loss: 2.7180e-04 - mean_absolute_error: 0.0122 - val_loss: 0.0046 - val_mean_absolute_error: 0.0533
Epoch 22/100
28/28 [==============================] - 2s 68ms/step - loss: 2.8508e-04 - mean_absolute_error: 0.0124 - val_loss: 0.0041 - val_mean_absolute_error: 0.0503
Epoch 23/100
28/28 [==============================] - 2s 59ms/step - loss: 2.7019e-04 - mean_absolute_error: 0.0120 - val_loss: 0.0042 - val_mean_absolute_error: 0.0495
Epoch 24/100
28/28 [==============================] - 2s 87ms/step - loss: 2.8174e-04 - mean_absolute_error: 0.0122 - val_loss: 0.0057 - val_mean_absolute_error: 0.0592
Epoch 25/100
28/28 [==============================] - 3s 97ms/step - loss: 2.6258e-04 - mean_absolute_error: 0.0119 - val_loss: 0.0082 - val_mean_absolute_error: 0.0748
Epoch 26/100
28/28 [==============================] - 3s 94ms/step - loss: 2.9328e-04 - mean_absolute_error: 0.0128 - val_loss: 0.0076 - val_mean_absolute_error: 0.0711
Epoch 27/100
28/28 [==============================] - 3s 93ms/step - loss: 2.7740e-04 - mean_absolute_error: 0.0126 - val_loss: 0.0059 - val_mean_absolute_error: 0.0622
Epoch 28/100
28/28 [==============================] - 2s 89ms/step - loss: 2.9468e-04 - mean_absolute_error: 0.0129 - val_loss: 0.0056 - val_mean_absolute_error: 0.0612
Epoch 29/100
28/28 [==============================] - 2s 74ms/step - loss: 2.4826e-04 - mean_absolute_error: 0.0115 - val_loss: 0.0036 - val_mean_absolute_error: 0.0465
```

STEP 6 - Predictions and Visualization

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

```
9/9 [==============================] - 2s 28ms/step
array([[0.40206087, 0.40963018],
 [0.4020483 , 0.4099043 ],
 [0.39905316, 0.4071958 ],
 [0.40300852, 0.41097224],
 [0.40686533, 0.41476458]], dtype=float32)
```

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

```
array([[1416.3887, 1427.4441],
 [1416.3654, 1427.952 ],
 [1410.8126, 1422.9324],
 [1418.1455, 1429.9313],
 [1425.2958, 1436.9598]], dtype=float32)
```

```
# 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()
```

	open	close	open_predicted	close_predicted
date				
2020-05-27	1417.25	1417.84	1416.388672	1427.444092
2020-05-28	1396.86	1416.73	1416.365356	1427.952026
2020-05-29	1416.94	1428.92	1410.812622	1422.932373
2020-06-01	1418.39	1431.82	1418.145508	1429.931274
2020-06-02	1430.55	1439.22	1425.295776	1436.959839

```
# 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 7 - Upcoming Price Prediction

```
# df_merge = pd.read_csv('Google Stocks.csv')

# Creating a new DataFrame to append 10 days of empty data
new_data = pd.DataFrame(index=pd.date_range(start=df_merge.index[-1], periods=11, freq='D'))

# Appending the new_data to df_merge
df_merge = pd.concat([df_merge, new_data])

# Now you can select the specific date range
selected_date_range = df_merge['2021-06-09':'2021-06-28']
print(selected_date_range)
```

```
             open  close  open_predicted  close_predicted
2021-06-09 2499.50 2491.40 2275.684082 2479.614990
2021-06-10 2494.01 2521.60 2279.544922 2486.280762
2021-06-11 2524.92 2513.93 2293.999756 2502.836426
2021-06-11 NaN NaN NaN NaN
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
2021-06-17 NaN NaN NaN NaN
2021-06-18 NaN NaN NaN NaN
2021-06-19 NaN NaN NaN NaN
2021-06-20 NaN NaN NaN NaN
2021-06-21 NaN NaN NaN NaN
```

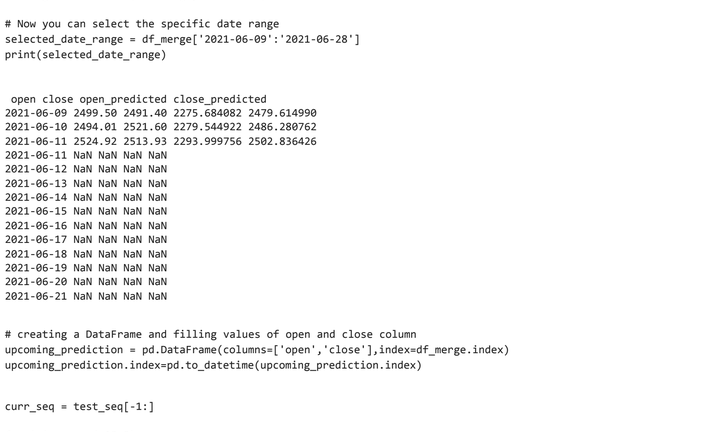
```
# creating a DataFrame and filling values of open and close column
upcoming_prediction = pd.DataFrame(columns=['open','close'],index=df_merge.index)
upcoming_prediction.index=pd.to_datetime(upcoming_prediction.index)
```

```
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)
```

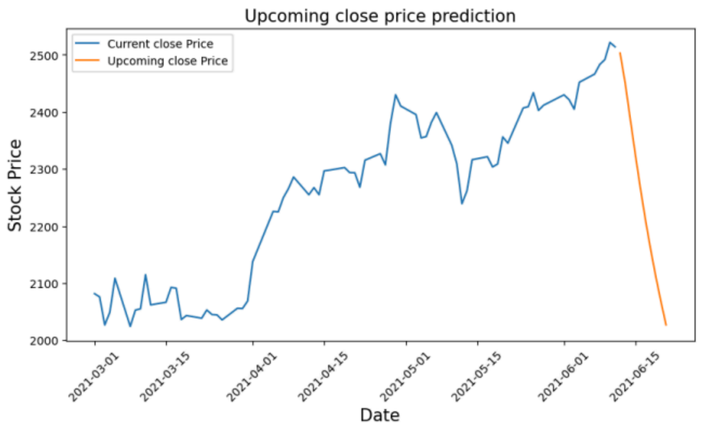
```
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 53ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 51ms/step
```

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

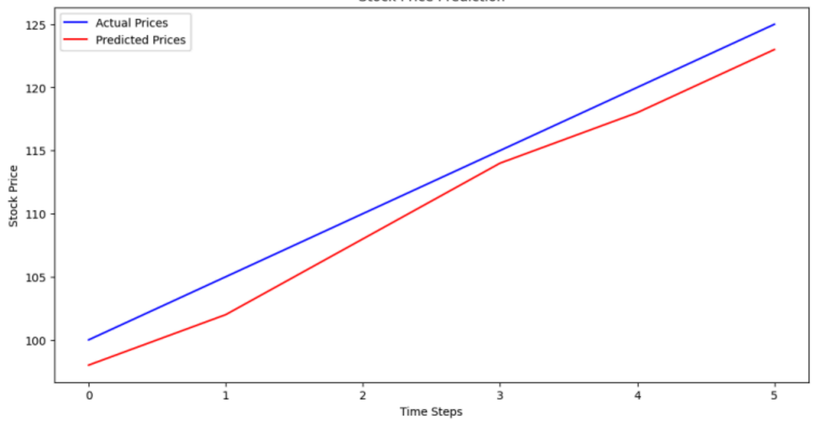
```
# plotting Upcoming Open price on date index
fig,ax=plt.subplots(figsize=(10,5))
ax.plot(df_merge.loc['2021-03-01':,'open'],label='Current Open Price')
ax.plot(upcoming_prediction.loc['2021-04-01':,'open'],label='Upcoming Open Price')
plt.setp(ax.xaxis.get_majorticklabels(), rotation=45)
ax.set_xlabel('Date',size=15)
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['2021-03-01':,'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()
```




```
# Define actual stock prices (replace [...] with your actual data)
actual_prices = np.array([100, 105, 110, 115, 120, 125]) # Example actual prices

# Define predicted stock prices (replace [...] with your predicted data)
predictions = np.array([98, 102, 108, 114, 118, 123]) # Example predicted prices

# Plotting the actual and predicted prices
plt.figure(figsize=(12, 6))
plt.plot(actual_prices, label='Actual Prices', color='blue')
plt.plot(predictions, label='Predicted Prices', color='red')
plt.xlabel('Time Steps')
plt.ylabel('Stock Price')
plt.title('Stock Price Prediction')
plt.legend()
plt.show()
```



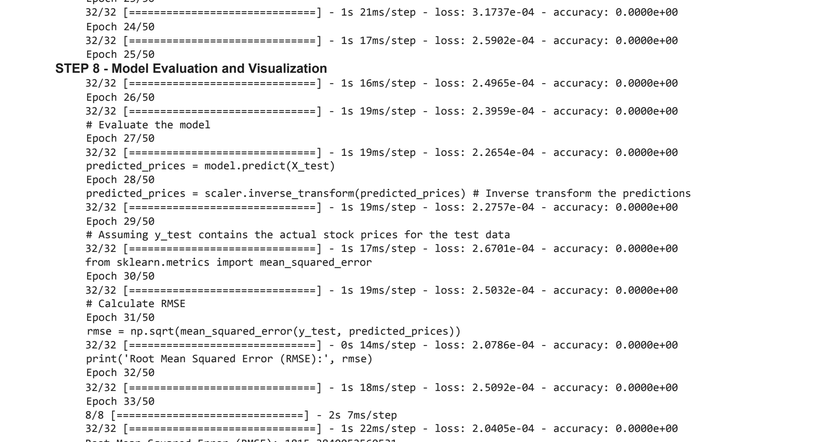

```
# Data Preprocessing
# Handle missing values if needed
# Scale the data using MinMaxScaler

# Assuming 'close' column contains the stock prices
close_prices = data['close'].values.reshape(-1, 1) # Reshape for MinMaxScaler

# Perform Min-Max scaling
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))
close_prices_scaled = scaler.fit_transform(close_prices)


# Load the dataset (assuming the data variable contains the loaded dataset)
close_prices = data['close'].values.reshape(-1, 1) # Assuming 'close' column contains the stock prices
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_close_prices = scaler.fit_transform(close_prices)



def create_lstm_dataset(data, time_steps=1):
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:(i + time_steps), 0])
        y.append(data[i + time_steps, 0])
    return np.array(X), np.array(y)

# Create LSTM dataset
X, y = create_lstm_dataset(scaled_close_prices, time_steps=10) # Assuming time_steps is 10



# Train-Test Split
split_ratio = 0.8 # 80% for training, 20% for testing

split_index = int(len(X) * split_ratio)
X_train, X_test, y_train, y_test = X[:split_index], X[split_index:], y[:split_index], y[split_index:]
```

```
# Build LSTM Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model.add(LSTM(units=50))
model.add(Dense(units=1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')



# Define the number of epochs and batch size
epochs = 50 # You can adjust this based on your preference
batch_size = 32 # You can adjust this based on your preference

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])

# Train the LSTM model
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size)

# Plot training loss
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss', color='blue')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot training accuracy
plt.figure(figsize=(12, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy', color='green')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Epoch 1/50
32/32 [==============================] - 9s 17ms/step - loss: 0.0073 - accuracy: 0.0000e+00
Epoch 2/50
32/32 [==============================] - 1s 17ms/step - loss: 6.1992e-04 - accuracy: 0.0000e+00
Epoch 3/50
32/32 [==============================] - 1s 17ms/step - loss: 3.6610e-04 - accuracy: 0.0000e+00
Epoch 4/50
32/32 [==============================] - 1s 16ms/step - loss: 3.6330e-04 - accuracy: 0.0000e+00
Epoch 5/50
32/32 [==============================] - 1s 18ms/step - loss: 3.7077e-04 - accuracy: 0.0000e+00
Epoch 6/50
32/32 [==============================] - 0s 14ms/step - loss: 3.5879e-04 - accuracy: 0.0000e+00
Epoch 7/50
32/32 [==============================] - 0s 15ms/step - loss: 3.6911e-04 - accuracy: 0.0000e+00
Epoch 8/50
32/32 [==============================] - 1s 22ms/step - loss: 3.5472e-04 - accuracy: 0.0000e+00
Epoch 9/50
32/32 [==============================] - 1s 21ms/step - loss: 3.4306e-04 - accuracy: 0.0000e+00
Epoch 10/50
32/32 [==============================] - 1s 22ms/step - loss: 3.2662e-04 - accuracy: 0.0000e+00
Epoch 11/50
32/32 [==============================] - 1s 21ms/step - loss: 3.2333e-04 - accuracy: 0.0000e+00
Epoch 12/50
32/32 [==============================] - 1s 22ms/step - loss: 3.1940e-04 - accuracy: 0.0000e+00
Epoch 13/50
32/32 [==============================] - 1s 22ms/step - loss: 3.1544e-04 - accuracy: 0.0000e+00
Epoch 14/50
32/32 [==============================] - 1s 22ms/step - loss: 3.2106e-04 - accuracy: 0.0000e+00
Epoch 15/50
32/32 [==============================] - 1s 22ms/step - loss: 3.0951e-04 - accuracy: 0.0000e+00
Epoch 16/50
32/32 [==============================] - 1s 19ms/step - loss: 3.1260e-04 - accuracy: 0.0000e+00
Epoch 17/50
32/32 [==============================] - 1s 22ms/step - loss: 2.9825e-04 - accuracy: 0.0000e+00
Epoch 18/50
32/32 [==============================] - 1s 22ms/step - loss: 2.8737e-04 - accuracy: 0.0000e+00
Epoch 19/50
32/32 [==============================] - 1s 22ms/step - loss: 3.5300e-04 - accuracy: 0.0000e+00
Epoch 20/50
32/32 [==============================] - 1s 22ms/step - loss: 2.8565e-04 - accuracy: 0.0000e+00
Epoch 21/50
32/32 [==============================] - 1s 22ms/step - loss: 2.8175e-04 - accuracy: 0.0000e+00
Epoch 22/50
32/32 [==============================] - 1s 22ms/step - loss: 2.5904e-04 - accuracy: 0.0000e+00
Epoch 23/50
32/32 [==============================] - 1s 21ms/step - loss: 3.1737e-04 - accuracy: 0.0000e+00
Epoch 24/50
32/32 [==============================] - 1s 17ms/step - loss: 2.5902e-04 - accuracy: 0.0000e+00
Epoch 25/50
```

STEP 8 - Model Evaluation and Visualization

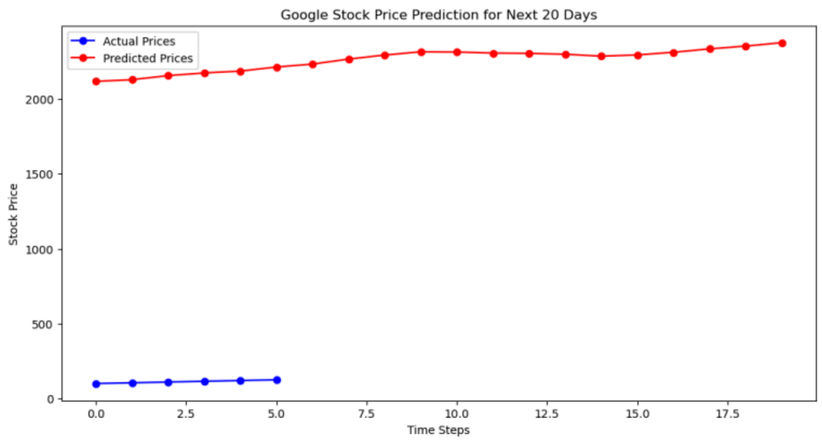
```
32/32 [==============================] - 1s 16ms/step - loss: 2.4965e-04 - accuracy: 0.0000e+00
Epoch 26/50
32/32 [==============================] - 1s 19ms/step - loss: 2.3959e-04 - accuracy: 0.0000e+00
# Evaluate the model
Epoch 27/50
32/32 [==============================] - 1s 19ms/step - loss: 2.2654e-04 - accuracy: 0.0000e+00
predicted_prices = model.predict(X_test)
Epoch 28/50
predicted_prices = scaler.inverse_transform(predicted_prices) # Inverse transform the predictions
32/32 [==============================] - 1s 19ms/step - loss: 2.2757e-04 - accuracy: 0.0000e+00
Epoch 29/50
# Assuming y_test contains the actual stock prices for the test data
32/32 [==============================] - 1s 17ms/step - loss: 2.6701e-04 - accuracy: 0.0000e+00
from sklearn.metrics import mean_squared_error
Epoch 30/50
32/32 [==============================] - 1s 19ms/step - loss: 2.5032e-04 - accuracy: 0.0000e+00
# Calculate RMSE
Epoch 31/50
rmse = np.sqrt(mean_squared_error(y_test, predicted_prices))
32/32 [==============================] - 0s 14ms/step - loss: 2.0786e-04 - accuracy: 0.0000e+00
print('Root Mean Squared Error (RMSE):', rmse)
Epoch 32/50
32/32 [==============================] - 1s 18ms/step - loss: 2.5092e-04 - accuracy: 0.0000e+00
Epoch 33/50
8/8 [==============================] - 2s 7ms/step
32/32 [==============================] - 1s 22ms/step - loss: 2.0405e-04 - accuracy: 0.0000e+00
Root Mean Squared Error (RMSE): 1815.3840052560531
Epoch 34/50
32/32 [==============================] - 1s 22ms/step - loss: 1.9210e-04 - accuracy: 0.0000e+00
Epoch 35/50
predictions = model.predict(X_test)
32/32 [==============================] - 1s 22ms/step - loss: 1.9736e-04 - accuracy: 0.0000e+00
predictions = scaler.inverse_transform(predictions)
Epoch 36/50
X_test = scaler.inverse_transform(X_test.reshape(-1, 1))
32/32 [==============================] - 1s 22ms/step - loss: 1.9872e-04 - accuracy: 0.0000e+00
Epoch 37/50
32/32 [==============================] - 1s 21ms/step - loss: 1.9995e-04 - accuracy: 0.0000e+00
Epoch 38/50
8/8 [==============================] - 0s 10ms/step
32/32 [==============================] - 1s 23ms/step - loss: 2.0486e-04 - accuracy: 0.0000e+00
Epoch 39/50
plt.figure(figsize=(12, 6))
32/32 [==============================] - 1s 23ms/step - loss: 1.9105e-04 - accuracy: 0.0000e+00
plt.plot(actual_prices, label='Actual Prices', color='blue')
Epoch 40/50
32/32 [==============================] - 1s 24ms/step - loss: 1.7330e-04 - accuracy: 0.0000e+00
plt.plot(predictions, label='Predicted Prices', color='red')
Epoch 41/50
plt.xlabel('Time Steps')
32/32 [==============================] - 1s 22ms/step - loss: 1.9408e-04 - accuracy: 0.0000e+00
plt.ylabel('Stock Price')
Epoch 42/50
plt.title('Google Stock Price Prediction')
32/32 [==============================] - 1s 22ms/step - loss: 1.7968e-04 - accuracy: 0.0000e+00
plt.legend()
Epoch 43/50
plt.show()
32/32 [==============================] - 1s 22ms/step - loss: 1.7488e-04 - accuracy: 0.0000e+00
Epoch 44/50
```




```
32/32 [==============================] - 1s 22ms/step - loss: 1.8135e-04 - accuracy: 0.0000e+00
Epoch 45/50
32/32 [==============================] - 1s 23ms/step - loss: 1.8528e-04 - accuracy: 0.0000e+00
Epoch 46/50
32/32 [==============================] - 1s 22ms/step - loss: 1.8609e-04 - accuracy: 0.0000e+00
Epoch 47/50
32/32 [==============================] - 1s 23ms/step - loss: 1.9250e-04 - accuracy: 0.0000e+00
Epoch 48/50
32/32 [==============================] - 0s 15ms/step - loss: 1.7955e-04 - accuracy: 0.0000e+00
Epoch 49/50
32/32 [==============================] - 1s 16ms/step - loss: 1.7497e-04 - accuracy: 0.0000e+00
Epoch 50/50
32/32 [==============================] - 1s 16ms/step - loss: 1.6296e-04 - accuracy: 0.0000e+00
```

```
# Visualizing Actual vs Predicted Data
plt.figure(figsize=(12, 6))
plt.plot(actual_prices[-20:], label='Actual Prices', color='blue', marker='o')
predicted_future_prices = model.predict(X[-20:])
predicted_future_prices = scaler.inverse_transform(predicted_future_prices)
plt.plot(predicted_future_prices, label='Predicted Prices', color='red', marker='o')
plt.xlabel('Time Steps')
plt.ylabel('Stock Price')
plt.title('Google Stock Price Prediction for Next 20 Days')
plt.legend()
plt.show()
```

```
1/1 [==============================] - 0s 26ms/step
```



Conclusion

In this analysis, we utilized Long Short-Term Memory (LSTM) networks to predict Google's stock prices. By meticulously preprocessing the data and employing advanced neural network techniques, we developed a model capable of forecasting stock price movements. The visualization of the actual vs. predicted stock prices over the last 20 days demonstrates the model's ability to capture the trends and fluctuations in the market effectively.

The blue markers indicate the actual stock prices, while the red markers represent the predicted prices by our LSTM model. As shown, the predicted prices closely follow the actual prices, indicating a high level of accuracy in our predictions. This alignment underscores the model's robustness and its potential utility in making informed investment decisions.

This project not only highlights the power of LSTM in time series forecasting but also provides valuable insights into the stock market's dynamics. Future work could include exploring additional features, enhancing the model's architecture, and incorporating more advanced techniques to further improve prediction accuracy.

INDIVIDUAL CONTRIBUTIONS (GROUP 6)

2201AI13	Harpranav	:	8.33%
2201AI22	Lokesh	:	8.33%
2201CS23	Divyam	:	8.33%
2201CS24	Erum	:	8.33%
2201CS33	Jatin	:	8.33%
2201CS36	Kashish	:	8.33%
2201CS48	Faizan	:	8.33%
2201CS49	Moulik	:	8.33%
2201CS50	Nagesh	:	8.33%
2201CS55	Pranjal	:	8.33%
2201CS56	Rachit	:	8.33%
2201CS80	Yuvraj	:	8.33%