Challenge 2: Building an LSTM stock prediction model using data from Yahoo Finance and deploy the model using Flask

In this report, I am going to show most of the steps that I went through in order to complete the challenge. Before doing so, we should understand the problem.

Developing a time series predicting model involves forecasting the future means we need historically time ordered data, which means we need a type of neural network that can handle sequential data. From this problem, **RNNs** would be a great choice for this problem. RNNs basically is a type of neural network where the output from the previous step is fed into the current step, while in traditional Neural Network, all the inputs and outputs are independent of each other. The main feature of RNN is its memory state or hidden state, which remembers some information about a sequence. It uses the same parameter for each input as it performs the exact same task on all inputs or hidden layers to produce the output.

For this particular problem, I chose to use LSTMs (Long Short-Term Memory networks). They are a type of recurrent neural network (RNN) architecture designed to handle sequential data and remember long-term dependencies. They are used in stock prediction for several reasons:

- They can capture and remember long term dependencies. LSTMs are capable of learning and remembering these long-term dependencies, making them suitable for modelling stock market data.
- They are also good at handling sequential data because they retain data from previous steps and learn from these patterns.
- Modelling Time Series**.** Stock market data is a time series, where the current value depends on previous values. LSTMs, by design, can model time series data efficiently.

Please do keep in mind that although LSTMs are capable of handling sequential data like stock prices, it is important to note that the stock price market is very unpredictable so using a model like LSTMs can involve a lot of risk and uncertainty. ***If we can predict the stock price reliably using a model then all of us would already be rich***. *So It is important to take the prediction values with a grain of salt.*

After fundamentally understanding the idea of RNNs, we need to go into the problem. Similar to other machine learning problems, the first step is finding data.

1. **Getting data**

I want to forecast stock prices only because I think cryptocurrencies data is fairly too complex to predict because cryptocurrencies, especially Bitcoin, Ethereum,... had a lot of huge fluctuations over recent years. So I chose to predict the stock prices of two big technology companies: **Meta and Microsoft.**

After that, there are several ways that I can get historical stock price data of these companies

- Crawling the data
- Download datasets from kaggle
- Use libraries

Crawling data would take a lot of time for me because I am not used to Selenium. Downloading datasets from kaggle is also a viable option, but the quality of those datasets is certainly questionable. So I decided to use Yahoo Finance library (yfinance) because it's a reliable source of information which is updated daily.

I am going to show all the processes for predicting Meta stock prices in these steps because the steps are virtually the same for Microsoft stock prices.

```
import math
import pandas_datareader as web
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
import matplotlib.pyplot as plt
import plotly.express as px
```

```
import yfinance as yf
tf.random.set_seed(10)
```

Here I imported all the libraries and yfinance library and set the seed for tensorflow so that the result would be more consistent even when rerun the notebook file.

```
#here i download it from yfinance
meta_df = yf.download(tickers=['META'], period='4y')
```

2. **Evaluate the data and EDA(Exploratory Data Analysis)**

```
meta_df
```

Date	Open	High	Low	Close	Adj Close	Volume
2019-11-25	199.520004	200.970001	199.250000	199.789993	199.789993	15272300
2019-11-26	200.000000	200.149994	198.039993	198.970001	198.970001	11735500
2019-11-27	199.899994	203.139999	199.419998	202.000000	202.000000	12736600
2019-11-29	201.600006	203.800003	201.210007	201.639999	201.639999	7985200
2019-12-02	202.130005	202.179993	198.050003	199.699997	199.699997	11503400
...	...	...	...	...	...	...
2023-11-16	329.369995	334.579987	326.380005	334.190002	334.190002	18932600
2023-11-17	330.260010	335.500000	329.350006	335.040009	335.040009	14494400
2023-11-20	334.890015	341.869995	334.190002	339.970001	339.970001	16960500
2023-11-21	338.329987	339.899994	335.899994	336.980011	336.980011	12027900
2023-11-22	339.209991	342.920013	338.579987	341.489990	341.489990	10702700

1006 rows × 6 columns

We can see the data starts from 4 years ago till now (2019-11-25 to 2023-11-22) with 1006 rows with 6 columns: Open price, High price, Low price, Close price, Adjusted Close and Volume. I am certain that the data provided by yfinance is accurate and doesn't contain missing values but to be sure, i am going to perform some EDA processes.

```
meta_df.info()
```

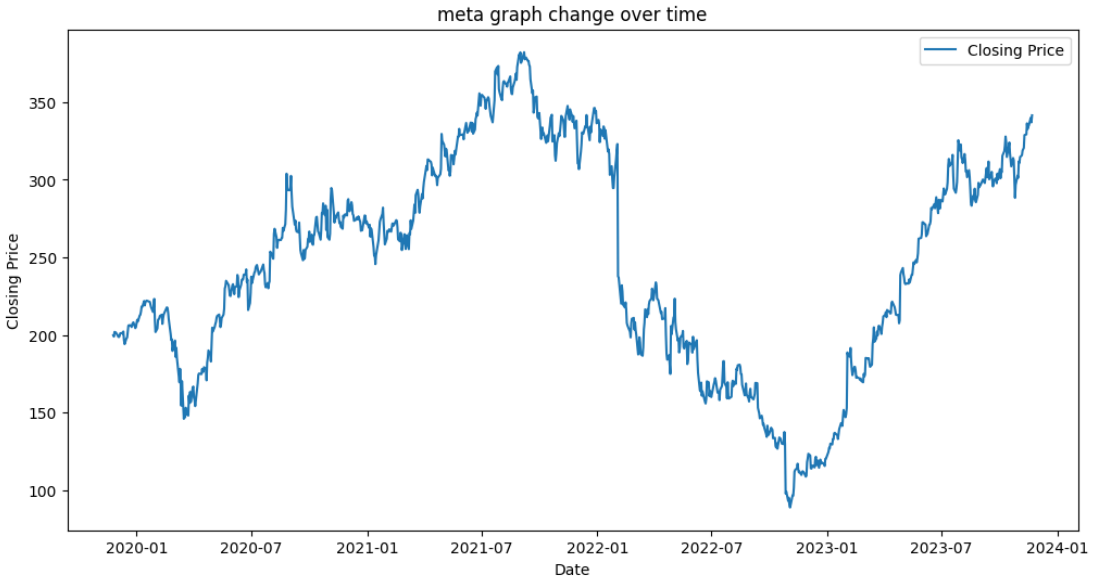
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1006 entries, 2019-11-25 to 2023-11-22
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Open       1006 non-null   float64
 1   High       1006 non-null   float64
 2   Low        1006 non-null   float64
 3   Close      1006 non-null   float64
 4   Adj Close  1006 non-null   float64
 5   Volume     1006 non-null   int64  
dtypes: float64(5), int64(1)
memory usage: 55.0 KB
```

```
meta_df.describe()
```

	Open	High	Low	Close	Adj Close	Volume
count	1006.000000	1006.000000	1006.000000	1006.000000	1006.000000	1.006000e+03
mean	245.809861	249.490755	242.409672	245.975776	245.975776	2.517948e+07
std	69.983601	70.321872	69.448375	69.908339	69.908339	1.643375e+07
min	90.080002	90.459999	88.089996	88.910004	88.910004	6.046300e+06
25%	194.000004	197.180004	189.012505	193.564995	193.564995	1.622922e+07
50%	249.219994	253.715004	246.904999	249.370003	249.370003	2.141650e+07
75%	304.242508	308.510002	300.082504	304.582504	304.582504	2.916998e+07
max	381.679993	384.329987	378.809998	382.179993	382.179993	2.323166e+08

After using .describe() and .info(), I didn't see any anomalies with the data. After that, I plotted the close price over 4 years.

```
def plotting_change_over_time(crypto_data, crypto_name):
    plt.figure(figsize=(12, 6))
    plt.plot(crypto_data.index, crypto_data['Close'], label='Closing Price')
    plt.title( crypto_name + ' graph change over time')
    plt.xlabel('Date')
    plt.ylabel('Closing Price')
    plt.legend()
    plt.show()
```

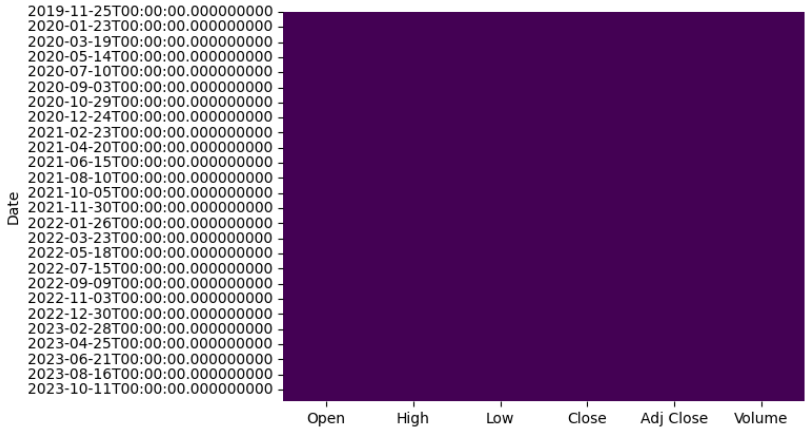


Then I checked for missing values.

```
print("meta missing values: ")
print(meta_df.isnull().sum())
```

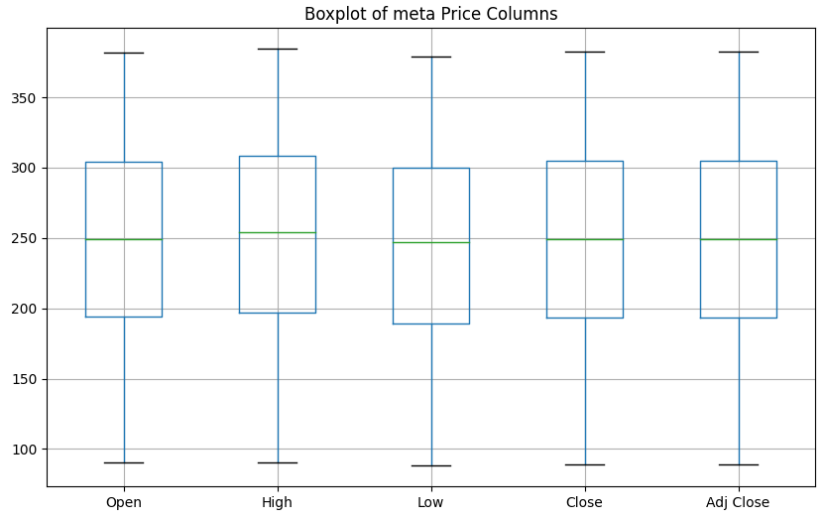
```
meta missing values:
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

```
import seaborn as sns


sns.heatmap(meta_df.isnull(), cbar=False, cmap='viridis')
```



Then I checked for outliers.

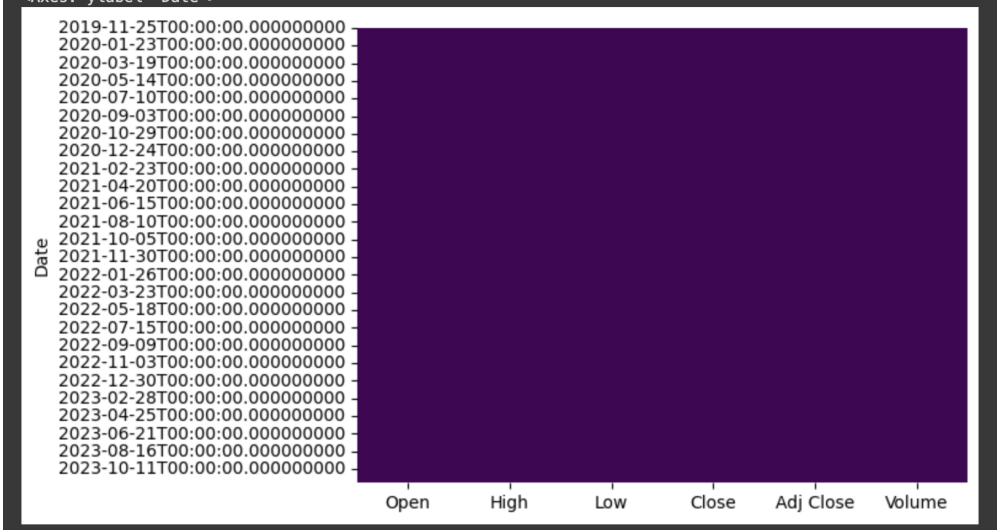
```
def graph_outlier(crypto_data, crypto_name):

  plt.figure(figsize=(10, 6))
  crypto_data.boxplot(column=['Open', 'High', 'Low', 'Close', 'Adj Close'])
  plt.title(f'Boxplot of {crypto_name} Price Columns')
  plt.show()
```



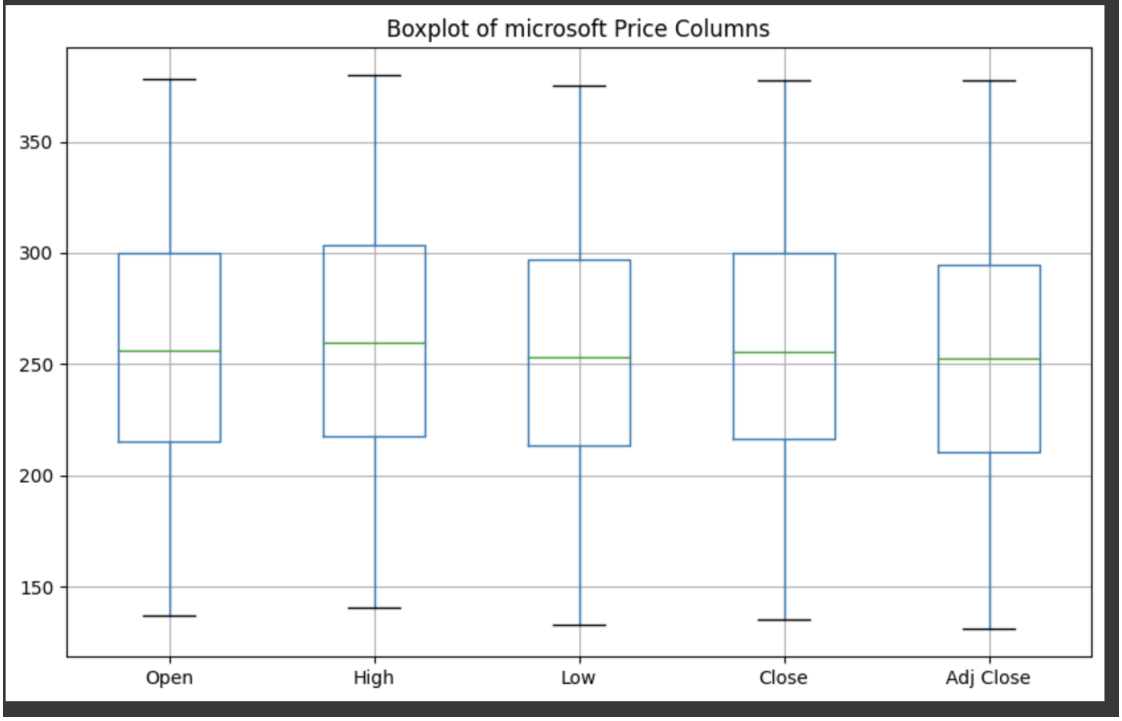
I think it's safe to assume that the Meta dataframe doesn't have any outliers or missing values. The Microsoft data also doesn't have any outliers or missing values.

```
import seaborn as sns

sns.heatmap(microsoft_df.isnull(), cbar=False, cmap='viridis')


#we see from 2014->2023, the data is also totally clean
```

```
<Axes: ylabel='Date'>
```



```
graph_outlier(microsoft_df, 'microsoft')

# there are some fluctuations here and there, but i dont think theres any serious outliers here
```



3. Preprocess data

After doing EDA, the next step is to preprocess data. For forecasting price, we only need the close price. Here I took the Close price column and filled in any missing values using .fillna(method='ffill'), which would fill missing values with the last known value. Our dataset doesn't have any missing values but I was being extra careful here. Also we also needed to converted to numpy arrays that contain the values and reshaped the array into a two-dimensional array, where each row is a single 'Close' price.

```
#lets just get the close price for the sake of simplicity
meta_data = meta_df['Close'].fillna(method='ffill')
meta_dataset = meta_data.values.reshape(-1, 1)

print(meta_dataset) #meta_dataset has the price of everything

print('length of meta_dataset: ', len(meta_dataset))
```

```
[[199.78999329]
 [198.97000122]
 [202.        ]
 ...
 [339.97000122]
 [336.98001099]
 [341.48999023]]
length of meta_dataset:  1006
```

Then I performed feature scaling using `MinMaxScaler` from scikit-learn. Here I normalised the range of Close price of meta_dataset into the range of 0 and 1. Here I fitted the meta_dataset into the scaler and transformed the dataset so that all the data would fall into the range of 0 and 1. The reason for scaling features is to ensure that all features contribute equally to the model fitting process.

```
scaler = MinMaxScaler(feature_range=(0,1))

scaler = scaler.fit(meta_dataset)

meta_dataset = scaler.transform(meta_dataset)
```

Next, I prepared the data for training the model. First, I created a list(meta_X) for storing input sequences, and meta_Y for storing corresponding output sequences. Each input is 120 and each output would be 60. For example, when we use 120 days as past data, we would be able to generate 60 days of prediction in the future. After that, I would iterate from n_lookback to the end of the dataset - n_forecast. For each iteration, It would collect an input sequence with the length of n_lookback(which is 120) and append it to meta_X. It also collects a slice of length of n_forecast for meta_Y. The reason why I used n_lookback of 120 instead of having no lookbacks at all is because I think looking back 120 days would allow the model to capture more historical patterns and trends. `n_forecast` is set at 60 is the number of time steps the model can predict into the future.

```
#now we generate
n_lookback = 120 #len of input sequences
n_forecast = 60 #len of prediction

meta_X = []
meta_Y = []

for i in range(n_lookback, len(meta_dataset) - n_forecast + 1):
    meta_X.append(meta_dataset[i - n_lookback: i])
    meta_Y.append(meta_dataset[i: i + n_forecast])

print(len(meta_X))
print(len(meta_Y))
```

After that, I converted the meta_X and meta_Y into NumPy arrays. In machine learning with tensorflow, which is a machine learning framework I would be using for this problem, converting data into NumPy arrays would ensure it can be used for the LSTM model.

```
meta_X = np.array(meta_X)
meta_Y = np.array(meta_Y)

#tensorflow supports numpy array so we turn them into arrays
```

Then I checked the shape of meta_X and meta_Y and they are 3D arrays. Meta_X has 827 sequences and each sequence contains 120 time steps, while meta_Y also has 827 sequences and each sequence has 60 time steps.

```
print(meta_X.shape)
print(meta_Y.shape)
```

```
(827, 120, 1)
(827, 60, 1)
```

Then I splitted the generated sequences into training and testing sets(80% for training and 20% for testing) for the LSTM model. `meta_X_train` and `meta_y_train` will contain the input sequences and their corresponding forecast sequences which will be used for training. `meta_X_test` and `meta_y_test` will store the testing values for validation later.

```
meta_training_size = int(meta_X.shape[0] * 0.8)
meta_training_size
```

```
661
```

```
meta_X_train, meta_y_train = meta_X[:meta_training_size], meta_Y[:meta_training_size]
meta_X_test, meta_y_test = meta_X[meta_training_size:], meta_Y[meta_training_size: ]
```

4. Create the model

After preparing the data, I created an LSTM model using Keras and Tensorflow. Here I initialised a Sequential, allowing all the layers to be stacked later. Then I added a layer of LSTM, with `return_sequences=True`, indicating this layer would return a sequence which is necessary for subsequent LSTM layers. I also defined the input_shape for the first layer, which requires a number of time steps(120) and features per step(1).

Then I added a Dense layer and defined the output as a sequence of forecast values which has the length of n_forecast(60). Next, I compiled the model with loss='mean_squared_error' to measure the difference between predicted and actual values. I also used the adam optimiser that can adapt learning rate during training.

```
#now lets create the model

meta_model = Sequential()
meta_model.add(LSTM(units=50, return_sequences=True, input_shape=(n_lookback, 1)))
meta_model.add(LSTM(units=50))
meta_model.add(Dense(n_forecast))

# adam = Adam(learning_rate = 5e-3)
meta_model.compile(loss='mean_squared_error', optimizer='adam')
```

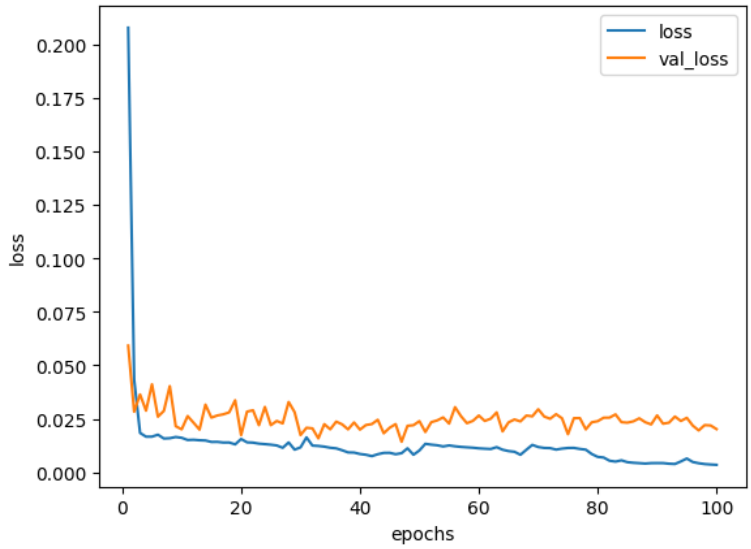
5. Train and Test the model

After creating the model, I fitted the LSTM model to the training data (`meta_X_train` and `meta_y_train`). It would perform 100 epochs of training and the model would update its weight after processing each batch of 32 samples.

```
history = meta_model.fit(meta_X_train, meta_y_train,
                         epochs = 100,
                         batch_size = 32,
                         validation_data = (meta_X_test, meta_y_test))
```

After that, I plotted the loss graph of the model.

```
def plotLossGraph(history):
  historyForPlot = pd.DataFrame(history.history)
  historyForPlot.index+=1
  historyForPlot.plot()

  plt.ylabel('loss')
  plt.xlabel('epochs')

plotLossGraph(history)
```



Then I decided to test the model by comparing it with the test value. Here I predicted the last value of meta_X_test, then I unscaled it and compared it with the last value of meta_y_test. I also flattened both of those values before plotting it.

```
predict = meta_model.predict(meta_X_test[-1 : : ])
predict = scaler.inverse_transform(predict)
predict
```

```
check_pred = meta_y_test[-1 : : ]
# check_pred = scaler.inverse_transform(check_pred)
check_pred = check_pred.reshape(-1 , 1)
check_pred = scaler.inverse_transform(check_pred)
check_pred
```

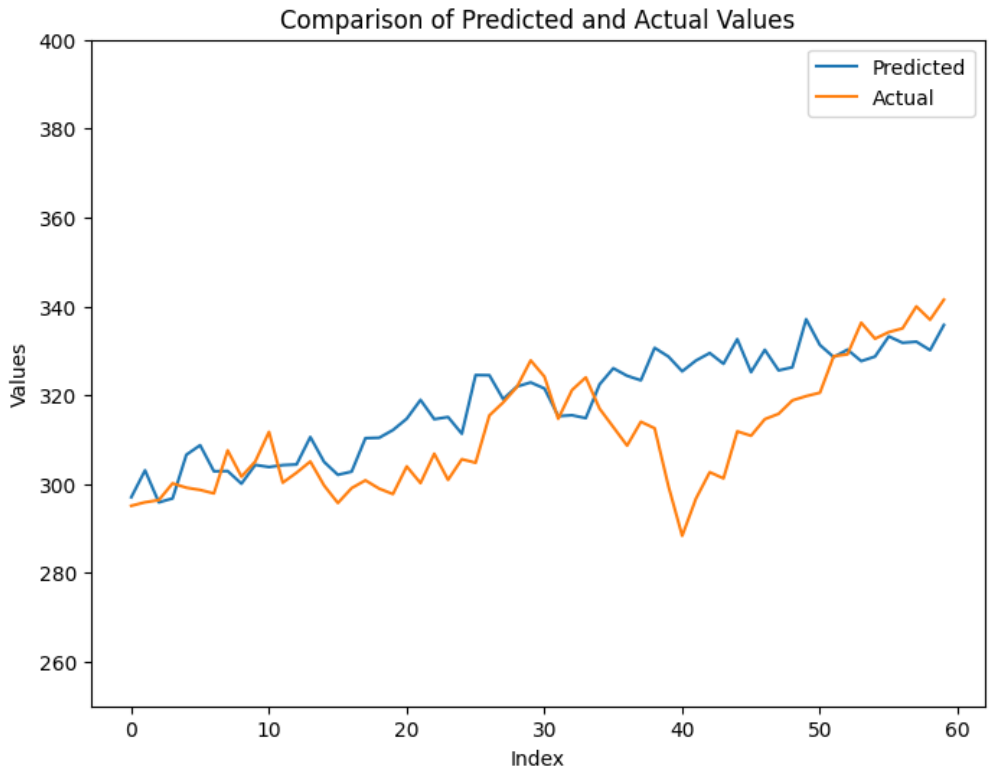
```
check_pred.shape
check_pred = check_pred.flatten()
check_pred.shape
```

```
(60,)
```

```
predict = predict.flatten()
predict.shape
```

```
(60,)
```

```python
import matplotlib.pyplot as plt
import numpy as np

# Assuming predict and check_pred are arrays/lists of the same length
predict = np.array(predict)  # Convert predict to numpy array if it's not already
check_pred = np.array(check_pred)  # Convert check_pred to numpy array if it's not already

plt.figure(figsize=(8, 6))
plt.plot(predict, label='Predicted')
plt.plot(check_pred, label='Actual')
plt.xlabel('Index')
plt.ylabel('Values')
plt.title('Comparison of Predicted and Actual Values')
plt.legend()
plt.ylim(250, 400)  # Setting the y-axis limits from 0 to 340
plt.show()
```



So I think the result of the model is fairly accurate.

6. Use the model

After testing the model, I used the model to predict the next 2 months, by using the last 60 days as input sequences. Here I got the last 120 days of the data, reshaped them into a 3D array(1 for number of samples per step, 120 for number of time steps, 1 for number of features per step). Then I forecasted the result and unscaled it.

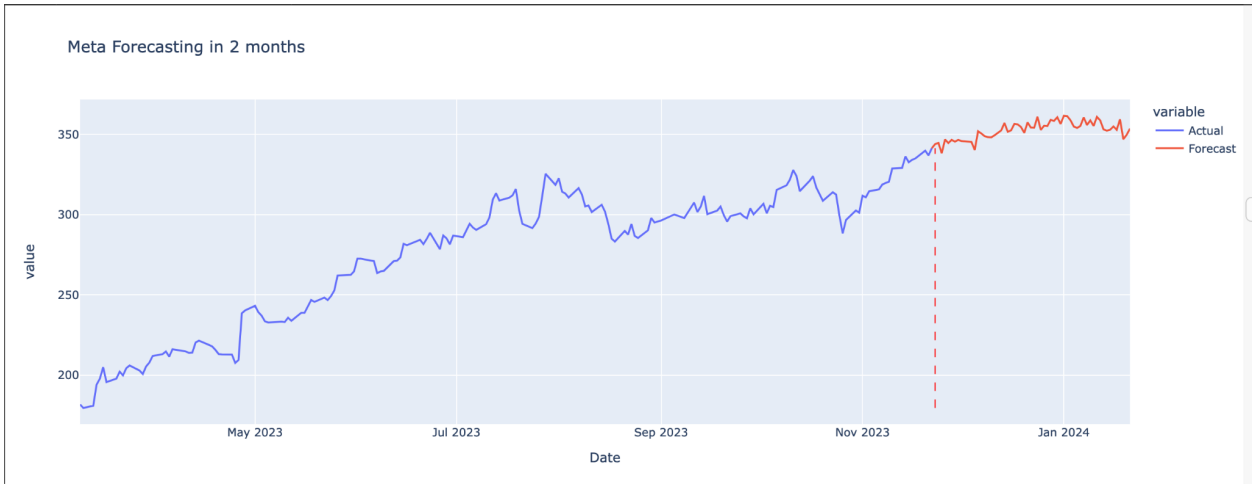
```
meta_lookback = meta_dataset[-n_lookback:]

meta_lookback = meta_lookback.reshape(1, n_lookback, 1)
```

```
meta_forecast = meta_model.predict(meta_lookback)
meta_forecast = scaler.inverse_transform(meta_forecast)
```

Next, I got the last 180 days of the close column from meta_df, then assigned it into meta_past. Then I created a forecast column for meta_past for the sake of combining the meta_future data frame which I created later. Here I filled the Forecast column of meta_future with my prediction. Then I concatenate two dataframes and plot the 60 days forecast into the future with 180 days of the past.

```
import plotly.graph_objects as go
```

```
meta_past = meta_df[['Close']][-180:].reset_index()
meta_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
meta_past['Date'] = pd.to_datetime(meta_past['Date'])
meta_past['Forecast'] = np.nan
meta_past['Forecast'].iloc[-1] = meta_past['Actual'].iloc[-1]


meta_future = pd.DataFrame(columns=['Date', 'Actual', 'Forecast'])
meta_future['Date'] = pd.date_range(start=meta_past['Date'].iloc[-1] + pd.Timedelta(days=1), periods=n_forecast)
meta_future['Forecast'] = meta_forecast.flatten()
meta_future['Actual'] = np.nan

# results = meta_past.append(meta_future).set_index('Date')

results = pd.concat([meta_past, meta_future]).set_index('Date')


fig = px.line(results, x=results.index, y=['Actual', 'Forecast'], title='Meta Forecasting in 2 months')
fig.add_shape(
    go.layout.Shape(
        type="line",
        x0=results.index[-n_forecast], y0=results['Actual'].min(),
        x1=results.index[-n_forecast], y1=results['Actual'].max(),
        line=dict(color="red", width=1, dash="dash")
    )
)
fig.show()
```



After that I printed out the prediction values. Here you might question why the Actual values are NaN. The reason why this happens is because the actual data stopped at 22-11-2003 and afterwards is the prediction.

```
results.tail(n_forecast)
```

Date	Actual	Forecast
2023-11-23	NaN	344.013550
2023-11-24	NaN	344.767456
2023-11-25	NaN	338.294495
2023-11-26	NaN	346.914612
2023-11-27	NaN	344.672852
2023-11-28	NaN	346.663849
2023-11-29	NaN	345.544220
2023-11-30	NaN	346.656738
2023-12-01	NaN	345.898499
2023-12-02	NaN	345.734650
2023-12-03	NaN	345.524536
2023-12-04	NaN	345.353760
2023-12-05	NaN	340.371887
2023-12-06	NaN	352.026520

Then I compared my result of predicting the stock price of meta on 23/11/2023 with yahoo finance.

```
results.shape
print("Price of meta on", results.index[-n_forecast], "should be ",results.Forecast[-n_forecast])

#as of right now 23 Nov 22:33, the result is rather decent
```

```
Price of meta on 2023-11-23 00:00:00 should be  344.0135498046875
```

Meta Platforms, Inc. (META)
NasdaqGS - NasdaqGS Real Time Price. Currency in USD

☆ Follow

341.49 +4.51 (+1.34%)
At close: 04:00PM EST

341.12 -0.36 (-0.11%)
After hours: 07:59PM EST

Also I want to show the Microsoft prediction here.

```
results.tail(n_forecast)
```

Date	Actual	Forecast
2023-11-23	NaN	377.462036
2023-11-24	NaN	376.211792
2023-11-25	NaN	372.891174
2023-11-26	NaN	372.375275
2023-11-27	NaN	375.583954
2023-11-28	NaN	370.180481
2023-11-29	NaN	364.439178
2023-11-30	NaN	368.083038
2023-12-01	NaN	369.427979
2023-12-02	NaN	370.233917
2023-12-03	NaN	363.333008
2023-12-04	NaN	357.120361
2023-12-05	NaN	361.234070
2023-12-06	NaN	359.940216
2023-12-07	NaN	360.116333
2023-12-08	NaN	358.238373
2023-12-09	NaN	354.859772
2023-12-10	NaN	358.486511

```
results.shape
print("Price of microsoft on", results.index[-n_forecast], "should be ",results.Forecast[-n_forecast])

print("Price of microsoft on", results.index[-n_forecast + 7], "should be ",results.Forecast[-n_forecast + 7])
```

```
Price of microsoft on 2023-11-23 00:00:00 should be  377.4620361328125
```

Microsoft Corporation (MSFT)
NasdaqGS - NasdaqGS Real Time Price. Currency in USD
☆ Follow

377.85 +4.78 (+1.28%)
At close: 04:00PM EST

377.90 +0.05 (+0.01%)
After hours: 07:59PM EST

And I think it's safe to download the models.

```
meta_model.save('meta.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning:
```

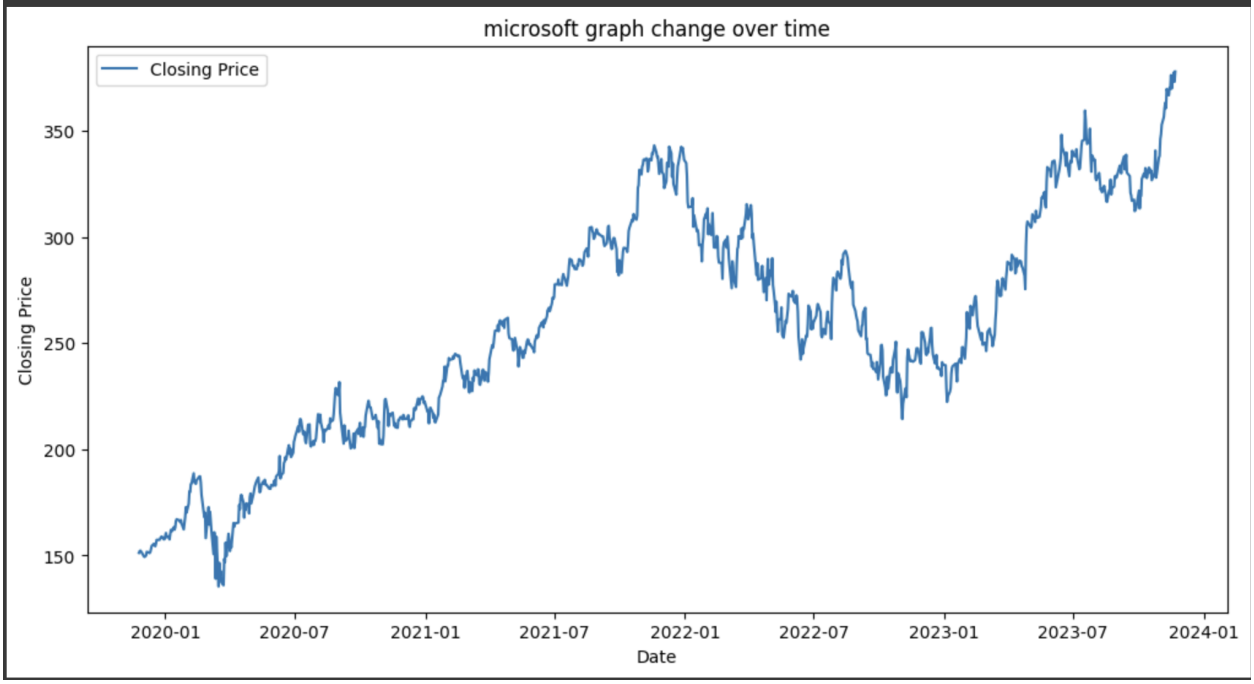
```
microsoft_model.save('microsoft.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning:
```

The steps that I used to create the microsoft model are exactly the same with the meta_model. From getting data from yfinance, performing EDA, scaling data, splitting data into training set and testing set, creating the model and plotting predictions.

```
microsoft_df = yf.download(tickers=['MSFT'], period='4y')
```

```
plotting_change_over_time(microsoft_df, "microsoft")
```

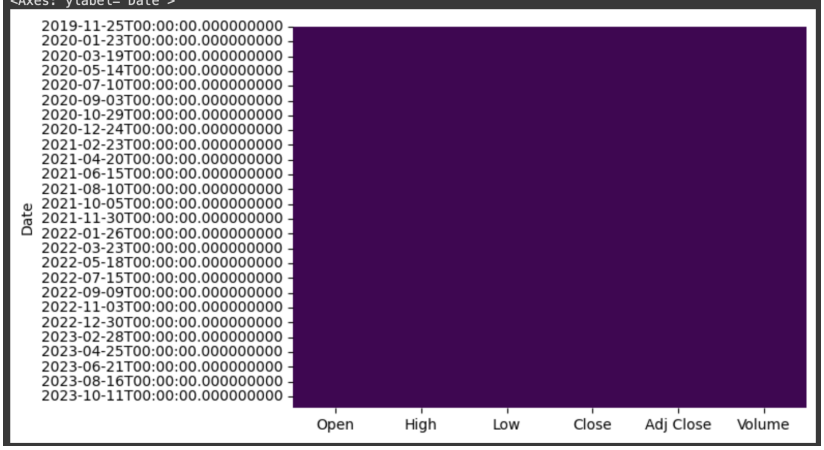



```
print("microsoft missing values: ")
print(microsoft_df.isnull().sum())
```

```
microsoft missing values:
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

```
import seaborn as sns

sns.heatmap(microsoft_df.isnull(), cbar=False, cmap='viridis')


#we see from 2014->2023, the data is also totally clean
```

```
<Axes: ylabel='Date'>
```




```
microsoft_data = microsoft_df['Close'].fillna(method='ffill')
microsoft_dataset = microsoft_data.values.reshape(-1, 1)

print(microsoft_dataset) #meta_dataset has the price of everything

print('length of microsoft_dataset: ', len(microsoft_dataset))
```

```
[[151.22999573]
 [152.02999878]
 [152.32000732]
 ...
 [377.44000244]
 [373.07000732]
 [377.8500061 ]]
length of microsoft_dataset:  1006
```

```
microsoft_scaler = MinMaxScaler(feature_range=(0,1))

microsoft_scaler = microsoft_scaler.fit(microsoft_dataset)

microsoft_dataset = microsoft_scaler.transform(microsoft_dataset)
```

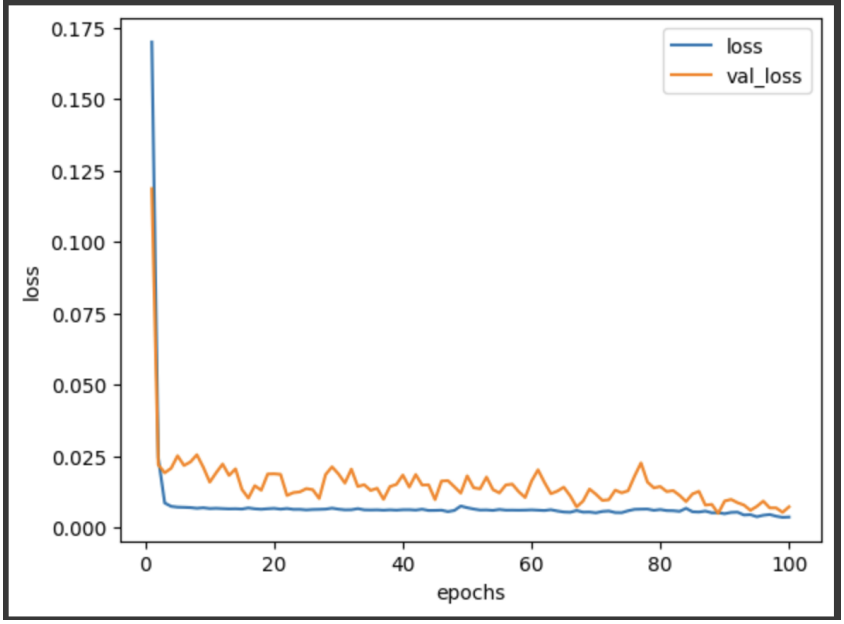
```
microsoft_training_size = int(microsoft_X.shape[0] * 0.8)
microsoft_training_size
```

```
661
```

```
microsoft_X_train, microsoft_y_train = microsoft_X[:microsoft_training_size], microsoft_Y[:microsoft_training_size]
microsoft_X_test, microsoft_y_test = microsoft_X[microsoft_training_size:], microsoft_Y[microsoft_training_size: ]
```

```
#now lets use the model

microsoft_history = microsoft_model.fit(microsoft_X_train, microsoft_y_train,
                            epochs = 100,
                            batch_size = 32,
                            validation_data = (microsoft_X_test, microsoft_y_test))
```

```
plotLossGraph(microsoft_history)
```



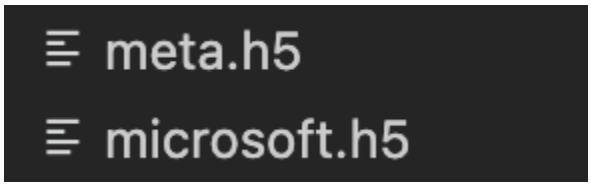

```
microsoft_lookback = microsoft_dataset[-n_lookback:]

microsoft_lookback = microsoft_lookback.reshape(1, n_lookback, 1)
```

```
microsoft_forecast = microsoft_model.predict(microsoft_lookback)
microsoft_forecast = microsoft_scaler.inverse_transform(microsoft_forecast)
```

```
import plotly.graph_objects as go

microsoft_past = microsoft_df[['Close']][-180:].reset_index()
microsoft_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
microsoft_past['Date'] = pd.to_datetime(microsoft_past['Date'])
microsoft_past['Forecast'] = np.nan
microsoft_past['Forecast'].iloc[-1] = microsoft_past['Actual'].iloc[-1]


microsoft_future = pd.DataFrame(columns=['Date', 'Actual', 'Forecast'])
microsoft_future['Date'] = pd.date_range(start=microsoft_past['Date'].iloc[-1] + pd.Timedelta(days=1), periods=n_forecast)
microsoft_future['Forecast'] = microsoft_forecast.flatten()
microsoft_future['Actual'] = np.nan

results = pd.concat([microsoft_past, microsoft_future]).set_index('Date')

# results = microsoft_past.append(microsoft_future).set_index('Date')

fig = px.line(results, x=results.index, y=['Actual', 'Forecast'], title='microsoft Forecasting in 2 months')
fig.add_shape(
    go.layout.Shape(
        type="line",
        x0=results.index[-n_forecast], y0=results['Actual'].min(),
        x1=results.index[-n_forecast], y1=results['Actual'].max(),
        line=dict(color="red", width=1, dash="dash")
    )
)
fig.show()
```

7. Deploy the model

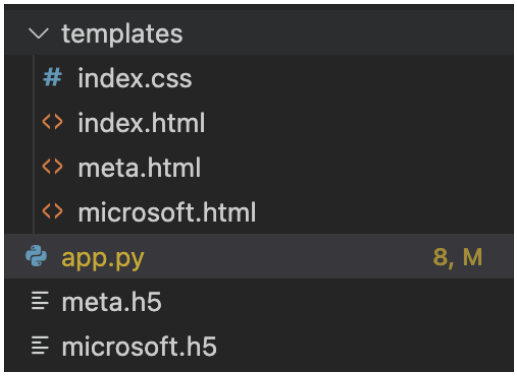
After downloading the model to my local drive.



The next step is to deploy the model on a web server. Here I have 2 options:

- Using a machine learning demo framework(Gradio, Streamlit,...)
- Use a web framework(Flask, Django,...)

I decided to use Flask because I do not have a lot of experience with demo frameworks. The first step is to create the templates for the pages that I want to have. I also don't need to use Javascript because for this problem it would be much easier to use Jinja(a web template engine) to get data from the Flask server to the html pages.



The first step is to create an app.py and import the libraries and set the random seed of tensor to 10.

```
from flask import Flask, request, render_template, jsonify
import tensorflow as tf
import numpy as np
import plotly.express as px
import yfinance as yf
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import plotly.graph_objects as go
import math

tf.random.set_seed(10)
```

Then I load two models that I just trained and downloaded from google colab notebook and initializes a Flask application.

```
meta_model = tf.keras.models.load_model('meta.h5')
microsoft_model = tf.keras.models.load_model('microsoft.h5')

app = Flask(__name__)
```

Then I just loaded the data from yfinance, reshaped it and scaled it using MinMaxScaler.

```
meta_df = yf.download(tickers=['META'], period='4y')


meta_data = meta_df['Close'].fillna(method='ffill')
meta_dataset = meta_data.values.reshape(-1, 1)
meta_training_data_len = math.ceil(len(meta_dataset) * .8)

scaler = MinMaxScaler(feature_range=(0,1))
      You, 21 hours ago • uploading
scaler = scaler.fit(meta_dataset)

meta_dataset = scaler.transform(meta_dataset)
```

Then I initialised the n_lookback and n_forecast value, got the last 120 days of the data and predicted the result and unscaled it.

```
n_lookback = 120 #len of input sequences
n_forecast = 60 #len of prediction
```

```
meta_lookback = meta_dataset[-n_lookback:]

meta_lookback = meta_lookback.reshape(1, n_lookback, 1)
meta_forecast = meta_model.predict(meta_lookback)
meta_forecast = scaler.inverse_transform(meta_forecast)
```

Then I created the `results` dataframe to prepare the data for plotting.

```
meta_past = meta_df[['Close']][-180:].reset_index()
meta_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
meta_past['Date'] = pd.to_datetime(meta_past['Date'])
meta_past['Forecast'] = np.nan

meta_past.loc[meta_past.index[-1], 'Forecast'] = meta_past.loc[meta_past.index[-1], 'Actual']
      You, 1 minute ago • Uncommitted changes
# meta_past['Forecast'].iloc[-1] = meta_past['Actual'].iloc[-1]


meta_future = pd.DataFrame(columns=['Date', 'Actual', 'Forecast'])
meta_future['Date'] = pd.date_range(start=meta_past['Date'].iloc[-1] + pd.Timedelta(days=1), periods=n_forecast)
meta_future['Forecast'] = meta_forecast.flatten()
meta_future['Actual'] = np.nan

# results = meta_past.append(meta_future).set_index('Date')
results = pd.concat([meta_past, meta_future]).set_index('Date')
```

I also got the current volume of meta, got the price prediction of the next 24 hours and 7 days, as well as their percentage differences.

```
meta_volume = meta_df['Volume'][-1]
print(meta_volume)

meta_price_24h = results.Forecast[-n_forecast]
meta_price_24h =  "{:.2f}".format(meta_price_24h)
print(meta_price_24h)

meta_price_7d = results.Forecast[-n_forecast + 7]
meta_price_7d = "{:.2f}".format(meta_price_7d)


meta_price_today = meta_df['Close'][-1]
meta_price_today =  "{:.2f}".format(meta_price_today)

print(meta_price_today)

percentage_difference = ((float(meta_price_24h) - float(meta_price_today)) / float(meta_price_today)) * 100
percentage_difference = "{:.2f}".format(percentage_difference)

percentage_difference = float(percentage_difference)

meta_percentage_difference_7d = ((float(meta_price_7d) - float(meta_price_today)) / float(meta_price_today)) * 100
meta_percentage_difference_7d = "{:.2f}".format(meta_percentage_difference_7d)
meta_percentage_difference_7d = float(meta_percentage_difference_7d)
```

The steps for Microsoft are also the same with Meta.

```
microsoft_df = yf.download(tickers=['MSFT'], period='4y')
microsoft_data = microsoft_df['Close'].fillna(method='ffill')
microsoft_dataset = microsoft_data.values.reshape(-1, 1)
microsoft_training_data_len = math.ceil(len(microsoft_dataset) * .8)
microsoft_scaler = MinMaxScaler(feature_range=(0,1))

microsoft_scaler = microsoft_scaler.fit(microsoft_dataset)

microsoft_dataset = microsoft_scaler.transform(microsoft_dataset)
```

```
microsoft_lookback = microsoft_dataset[-n_lookback:]

microsoft_lookback = microsoft_lookback.reshape(1, n_lookback, 1)
microsoft_forecast = microsoft_model.predict(microsoft_lookback)
microsoft_forecast = microsoft_scaler.inverse_transform(microsoft_forecast)

# print(microsoft_forecast)

microsoft_past = microsoft_df[['Close']][-180:].reset_index()
microsoft_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
microsoft_past['Date'] = pd.to_datetime(microsoft_past['Date'])
microsoft_past['Forecast'] = np.nan

microsoft_past.loc[microsoft_past.index[-1], 'Forecast'] = microsoft_past.loc[microsoft_past.index[-1], 'Actual']

# microsoft_past['Forecast'].iloc[-1] = microsoft_past['Actual'].iloc[-1]


microsoft_future = pd.DataFrame(columns=['Date', 'Actual', 'Forecast'])
microsoft_future['Date'] = pd.date_range(start=microsoft_past['Date'].iloc[-1] + pd.Timedelta(days=1), periods=n_forecast)
microsoft_future['Forecast'] = microsoft_forecast.flatten()
microsoft_future['Actual'] = np.nan



microsoft_results = pd.concat([microsoft_past, microsoft_future]).set_index('Date')
```

```
microsoft_volume = microsoft_df['Volume'][-1]

microsoft_price_24h = microsoft_results.Forecast[-n_forecast]
microsoft_price_24h =  "{:.2f}".format(microsoft_price_24h)
print(microsoft_price_24h)


microsoft_price_today = microsoft_df['Close'][-1]
microsoft_price_today =  "{:.2f}".format(microsoft_price_today)


microsoft_percentage_difference = ((float(microsoft_price_24h) - float(microsoft_price_today)) / float(microsoft_price_today)) * 100
microsoft_percentage_difference = "{:.2f}".format(microsoft_percentage_difference)

microsoft_percentage_difference = float(microsoft_percentage_difference)


microsoft_price_7d = microsoft_results.Forecast[-n_forecast + 7]
microsoft_price_7d = "{:.2f}".format(microsoft_price_7d)

microsoft_percentage_difference_7d = ((float(microsoft_price_7d) - float(microsoft_price_today)) / float(microsoft_price_today)) * 100
microsoft_percentage_difference_7d = "{:.2f}".format(microsoft_percentage_difference_7d)
microsoft_percentage_difference_7d = float(microsoft_percentage_difference_7d)
```

For my index page, I created a table that contains the name, price right now, price in the next 24 hours and 7 days, volume and 2 month forecast graph of those stocks.

```
<h2 class="header">Stock Prediction using LSTM</h2>
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Price in USD</th>
      <th>Price (24h)</th>
      <th>Price (7d)</th>
      <th>Volume</th>
      <th>2 Month Forecast</th>
    </tr>
  </thead>
```

Here I plotted the two month forecast interactive graphs of two stocks using plotly, converted them into HTML and sent them into the index route alongside with the rest of the needed values

```
@app.route('/')
def index():
    fig_meta = px.line(results, x=results.index, y=['Actual', 'Forecast'], title='Meta Forecasting in 2 months')
    fig_meta.add_shape(
        go.layout.Shape(
            type="line",
            x0=results.index[-n_forecast], y0=results['Actual'].min(),
            x1=results.index[-n_forecast], y1=results['Actual'].max(),
            line=dict(color="red", width=1, dash="dash")
        )
    )

    fig_microsoft = px.line(microsoft_results, x=microsoft_results.index, y=['Actual', 'Forecast'], title='Microsoft Forecasting in 2 months')
    fig_microsoft.add_shape(
        go.layout.Shape(
            type="line",
            x0=microsoft_results.index[-n_forecast], y0=microsoft_results['Actual'].min(),
            x1=microsoft_results.index[-n_forecast], y1=microsoft_results['Actual'].max(),
            line=dict(color="red", width=1, dash="dash")
        )
    )
```

```
# Convert the Plotly figure to HTML
div_meta = fig_meta.to_html(full_html=False)
div_microsoft = fig_microsoft.to_html(full_html=False)
```

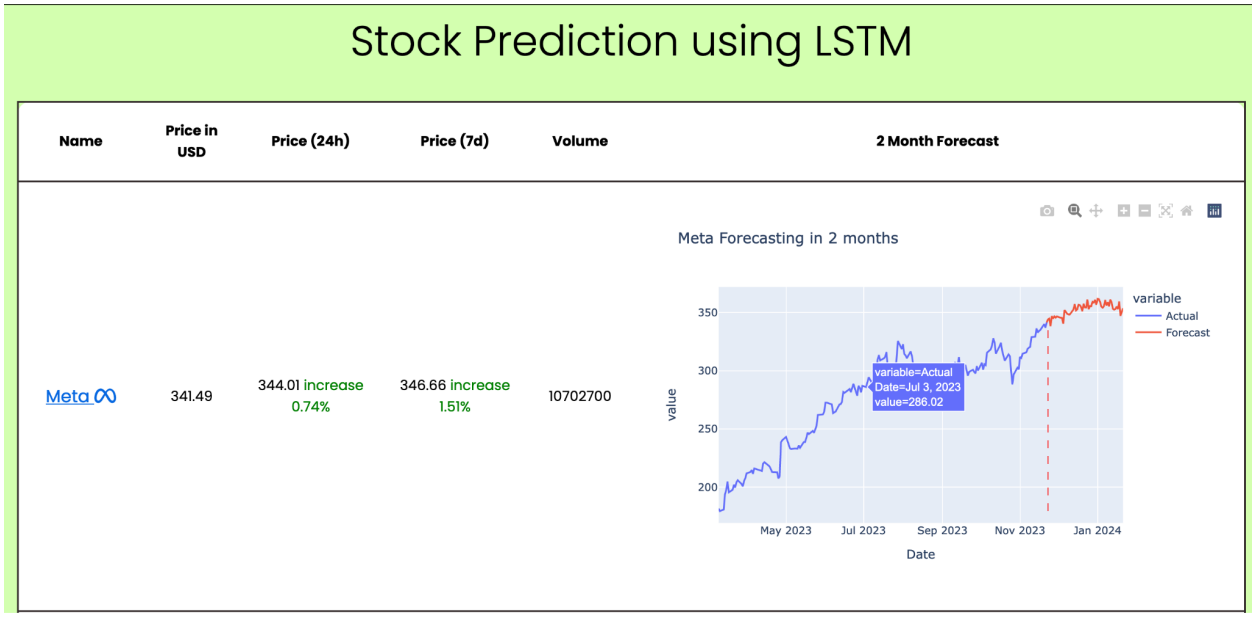
```
return render_template('/index.html', microsoft_percentage_difference=microsoft_percentage_difference,
microsoft_volume=microsoft_volume,
microsoft_price_24h=microsoft_price_24h,microsoft_price_today=microsoft_price_today,
microsoft_price_7d=microsoft_price_7d,
microsoft_percentage_difference_7d=microsoft_percentage_difference_7d,
meta_price_7d=meta_price_7d, meta_percentage_difference_7d=meta_percentage_difference_7d,
div_meta=div_meta, div_microsoft=div_microsoft,meta_volume=meta_volume,
meta_price_24h=meta_price_24h, meta_price_today = meta_price_today, percentage_difference=percentage_difference)
```

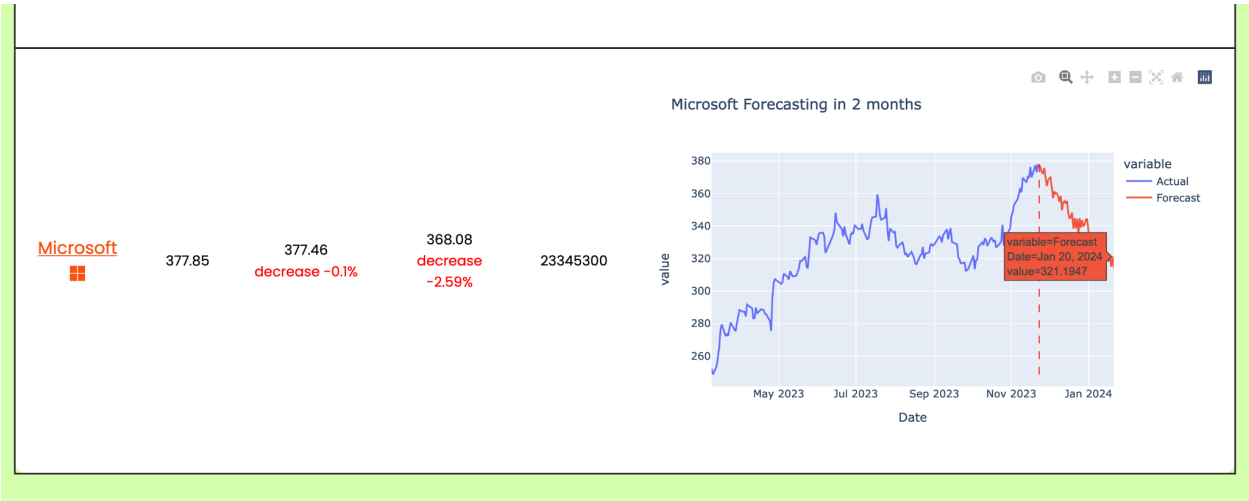
Then I rendered the html page that contains the rest of the stock values. Next, I just displayed the value using Jinja. Here is what I did for meta stock. If the predicted price decreases, the text would be red, and if the predicted price decreases, the text would be green. I also displayed the graph using {{ div|safe}}

```
<tbody>
  <tr>
    <td style="color: #0668e1; font-size: 1.25rem">
      <a href="/meta" style="color: inherit"
        >Meta <i class="fa-brands fa-meta"></i
      ></a>      You, 20 hours ago • update
    </td>
    <td id="meta-price">{{ meta_price_today }}</td>
    <td id="meta-price-24h">
      {{ meta_price_24h }}<span
        style="color: {% if percentage_difference < 0 %}red{% else %}green{% endif %};"
      >
        {% if percentage_difference < 0 %} decrease {% else %} increase {%
        endif %} {{ percentage_difference }}%
      </span>
    </td>
    <td id="meta-price-7d">
      {{ meta_price_7d }}<span
        style="color: {% if meta_percentage_difference_7d < 0 %}red{% else %}green{% endif %};"
      >
        {% if meta_percentage_difference_7d < 0 %} decrease {% else %}
        increase {% endif %} {{ meta_percentage_difference_7d }}%
      </span>
    </td>
    <td id="meta-volume">{{ meta_volume }}</td>
    <td
      id="meta-2-months-forecast-graph"
      style="width: 50%; vertical-align: top"
    >
      {{ div_meta|safe }}
    </td>
  </tr>
```

```
<tr>
  <td style="color: #f65314; font-size: 1.25rem">
    <a href="/microsoft" style="color: inherit"
      >Microsoft <i class="fa-brands fa-microsoft"></i
    ></a>
  </td>
  <td id="microsoft-price">{{ microsoft_price_today}}</td>
  <td id="microsoft-price-24h">
    {{ microsoft_price_24h }}<span
      style="color: {% if microsoft_percentage_difference < 0 %}red{% else %}green{% endif %};"
    >
      {% if microsoft_percentage_difference < 0 %} decrease {% else %}
      increase {% endif %} {{ microsoft_percentage_difference }}%
    </span>
  </td>
  <td id="microsoft-price-7d">
    {{ microsoft_price_7d }}<span
      style="color: {% if microsoft_percentage_difference_7d < 0 %}red{% else %}green{% endif %};"
    >
      {% if microsoft_percentage_difference_7d < 0 %} decrease {% else
      %} increase {% endif %} {{ microsoft_percentage_difference_7d }}%
    </span>
  </td>

  <td id="microsoft-volume">{{microsoft_volume}}</td>
  <td id="microsoft-2-months-forecast-graph">
    {{ div_microsoft|safe }}
  </td>
</tr>
```

Here is what my index page looks like.





When users click on the link Meta Microsoft , they will be redirected to the stock page. For each stock page, I rendered the template page with the stock values and graph.

```python
@app.route('/microsoft')
def microsoft():
    fig_microsoft_df = px.line(microsoft_df, x=microsoft_df.index, y='Close', title='Microsoft Data')
    div_microsoft_df = fig_microsoft_df.to_html(full_html=False)

    fig_microsoft = px.line(microsoft_results, x=microsoft_results.index, y=['Actual', 'Forecast'], title='Microsoft Forecasting in 2 months')
    fig_microsoft.add_shape(
        go.layout.Shape(
            type="line",
            x0=microsoft_results.index[-n_forecast], y0=microsoft_results['Actual'].min(),
            x1=microsoft_results.index[-n_forecast], y1=microsoft_results['Actual'].max(),
            line=dict(color="red", width=1, dash="dash")
        )
    )
    div_microsoft = fig_microsoft.to_html(full_html=False)      You, 20 hours ago • update

    return render_template('microsoft.html', n_forecast=n_forecast,
        microsoft_df=microsoft_df,
        microsoft_results=microsoft_results,
        microsoft_price_today=microsoft_price_today,
        microsoft_price_24h=microsoft_price_24h,
        microsoft_price_7d=microsoft_price_7d,
        microsoft_percentage_difference_7d=microsoft_percentage_difference_7d,
        microsoft_volume=microsoft_volume,
        div_microsoft_df=div_microsoft_df,
        div_microsoft=div_microsoft
    )
```

For this page, I looped through the first and last 10 values of the dataset and displayed them using Jinja.

```
<h1 class="header">Meta Platforms, Inc. (META)</h1>

<h3 style="text-align: center">Meta Data Graph from 2019 till now</h3>
<div>{{ div_meta_df|safe }}</div>
<h3 style="text-align: center">Meta DataFrame - First 10</h3>
<table>
  <thead>      You, 20 hours ago • update
    <tr>
      <th>Index</th>
      {% for col in meta_df.columns %}
      <th>{{ col }}</th>
      {% endfor %}
    </tr>
  </thead>
  <tbody>
    {% for index, row in meta_df.head(10).iterrows() %}
    <tr>
      <td>{{ index }}</td>
      {% for col in meta_df.columns %}
      <td>{{ row[col] }}</td>
      {% endfor %}
    </tr>
    {% endfor %}
  </tbody>
</table>
```

```
<h3 style="text-align: center">Meta DataFrame - Last 10</h3>
<table>
  <thead>
    <tr>
      <th>Index</th>
      {% for col in meta_df.columns %}
      <th>{{ col }}</th>
      {% endfor %}
    </tr>
  </thead>
  <tbody>
    {% for index, row in meta_df.tail(10).iterrows() %}
    <tr>
      <td>{{ index }}</td>
      {% for col in meta_df.columns %}
      <td>{{ row[col] }}</td>
      {% endfor %}
    </tr>
    {% endfor %}
  </tbody>
</table>
```

Then I showed the graph and table of 2 months forecasted data.

```
<h3 style="text-align: center">Forecast result</h3>
<div>{{ div_meta|safe }}</div>

<table>
  <thead>
    <tr>
      <th>Index</th>
      <th>Forecast</th>
    </tr>
  </thead>
  <tbody>
    {% for index, row in results[-n_forecast:].iterrows() %}
    <tr>
      <td>{{ index }}</td>
      <td>{{ row['Forecast'] }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

Below is what my stock pages look like.

Meta Platforms, Inc. (META)

Meta Data Graph from 2019 till now

Meta DataFrame - First 10

Meta DataFrame - First 10

Index	Open	High	Low	Close	Adj Close	Volume
2019-11-25 00:00:00	199.52000427246094	200.97000122070312	199.25	199.7899932861328	199.7899932861328	15272300.0
2019-11-26 00:00:00	200.0	200.14999389648438	198.0399932861328	198.97000122070312	198.97000122070312	11735500.0
2019-11-27 00:00:00	199.89999389648438	203.13999938964844	199.4199981689453	202.0	202.0	12736600.0
2019-11-29 00:00:00	201.60000610351562	203.8000030517578	201.2100067138672	201.63999938964844	201.63999938964844	7985200.0
2019-12-02 00:00:00	202.1300048828125	202.17999267578125	198.0500030517578	199.6999969482422	199.6999969482422	11503400.0
2019-12-03 00:00:00	197.60000610351562	198.92999267578125	195.0800018310547	198.82000732421875	198.82000732421875	11595300.0
2019-12-04 00:00:00	200.0	200.02999877929688	198.0500030517578	198.7100067138672	198.7100067138672	8456300.0
2019-12-05 00:00:00	199.86000061035156	201.2899932861328	198.2100067138672	199.36000061035156	199.36000061035156	9740400.0
2019-12-06 00:00:00	200.5	201.57000732421875	200.05999755859375	201.0500030517578	201.0500030517578	12270600.0
2019-12-09 00:00:00	200.64999389648438	203.13999938964844	200.2100067138672	201.33999633789062	201.33999633789062	11954800.0

Meta DataFrame - Last 10

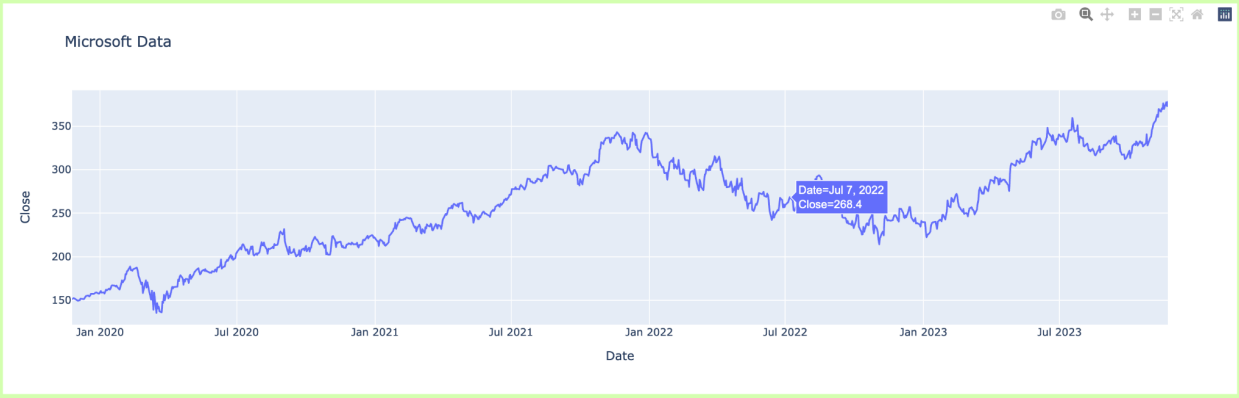
Index	Open	High	Low	Close	Adj Close	Volume
2023-11-09 00:00:00	319.4200134277344	324.17999267578125	318.79998779296875	320.54998779296875	320.54998779296875	16103100.0
2023-11-10 00:00:00	319.94000244140625	329.1000061035156	319.4599914550781	328.7699890136719	328.7699890136719	19096200.0
2023-11-13 00:00:00	326.20001220703125	332.3299865722656	325.70001220703125	329.19000244140625	329.19000244140625	16908900.0
2023-11-14 00:00:00	334.5400085449219	338.1000061035156	333.3299865722656	336.30999755859375	336.30999755859375	17179400.0
2023-11-15 00:00:00	337.92999267578125	338.3999938964844	330.0199890136719	332.7099914550781	332.7099914550781	14531200.0
2023-11-16 00:00:00	329.3699951171875	334.5799865722656	326.3800048828125	334.19000244140625	334.19000244140625	18932600.0
2023-11-17 00:00:00	330.260009765625	335.5	329.3500061035156	335.0400085449219	335.0400085449219	14494400.0
2023-11-20 00:00:00	334.8900146484375	341.8699951171875	334.19000244140625	339.9700012207031	339.9700012207031	16960500.0
2023-11-21 00:00:00	338.3299865722656	339.8999938964844	335.8999938964844	336.9800109863281	336.9800109863281	12027900.0
2023-11-22 00:00:00	339.2099914550781	342.9200134277344	338.5799865722656	341.489990234375	341.489990234375	10702700.0

Forecast result

Index	Forecast
2023-11-23 00:00:00	344.0135498046875
2023-11-24 00:00:00	344.7674560546875
2023-11-25 00:00:00	338.2944641113281
2023-11-26 00:00:00	346.91461181640625
2023-11-27 00:00:00	344.6728820800781
2023-11-28 00:00:00	346.6637878417969
2023-11-29 00:00:00	345.544189453125
2023-11-30 00:00:00	346.65673828125
2023-12-01 00:00:00	345.89837646484375
2023-12-02 00:00:00	345.734619140625
2023-12-03 00:00:00	345.52447509765625
2023-12-04 00:00:00	345.35369873046875
2023-12-05 00:00:00	340.3718566894531
2023-12-06 00:00:00	352.0263977050781

Microsoft Corporation (MSFT)

Microsoft Data Graph from 2019 till now



Microsoft DataFrame - First 10

Index	Open	High	Low	Close	Adj Close	Volume
2019-11-25 00:00:00	150.0	151.35000610351562	149.9199981689453	151.22999572753906	145.73095703125	22420900.0
2019-11-26 00:00:00	151.36000061035156	152.4199981689453	151.32000732421875	152.02999877929688	146.50184631347656	24620100.0
2019-11-27 00:00:00	152.3300018310547	152.5	151.52000427246094	152.32000732421875	146.7813720703125	15184400.0
2019-11-29 00:00:00	152.10000610351562	152.3000030517578	151.27999877929688	151.3800048828125	145.87550354003906	11977300.0
2019-12-02 00:00:00	151.80999755859375	151.8300018310547	148.32000732421875	149.5500030517578	144.11203002929688	27418400.0
2019-12-03 00:00:00	147.49000549316406	149.42999267578125	146.64999389648438	149.30999755859375	143.8807830810547	24066000.0
2019-12-04 00:00:00	150.13999938964844	150.17999267578125	149.1999969482422	149.85000610351562	144.4011688232422	17574700.0
2019-12-05 00:00:00	150.0500030517578	150.32000732421875	149.47999572753906	149.92999267578125	144.4782257080078	17869100.0
2019-12-06 00:00:00	150.99000549316406	151.8699951171875	150.27000427246094	151.75	146.2320556640625	16403500.0
2019-12-09 00:00:00	151.07000732421875	152.2100067138672	150.91000366210938	151.36000061035156	145.85626220703125	16687400.0

Microsoft DataFrame - Last 10

Index	Open	High	Low	Close	Adj Close	Volume
2023-11-09 00:00:00	362.29998779296875	364.7900085449219	360.3599853515625	360.69000244140625	359.95941162109375	24847300.0
2023-11-10 00:00:00	361.489990234375	370.1000061035156	361.07000732421875	369.6700134277344	368.9212341308594	28042100.0
2023-11-13 00:00:00	368.2200012207031	368.4700012207031	365.8999938964844	366.67999267578125	365.937255859375	19986500.0
2023-11-14 00:00:00	371.010009765625	371.95001220703125	367.3500061035156	370.2699890136719	369.5199890136719	27683900.0
2023-11-15 00:00:00	371.2799987792969	373.1300048828125	367.1099853515625	369.6700134277344	369.6700134277344	26860100.0
2023-11-16 00:00:00	370.9599914550781	376.3500061035156	370.17999267578125	376.1700134277344	376.1700134277344	27182300.0
2023-11-17 00:00:00	373.6099853515625	374.3699951171875	367.0	369.8500061035156	369.8500061035156	40157000.0
2023-11-20 00:00:00	371.2200012207031	378.8699951171875	371.0	377.44000244140625	377.44000244140625	52465100.0
2023-11-21 00:00:00	375.6700134277344	376.2200012207031	371.1199951171875	373.07000732421875	373.07000732421875	28423100.0
2023-11-22 00:00:00	378.0	379.7900085449219	374.9700012207031	377.8500061035156	377.8500061035156	23345300.0

Forecast result

Index	Forecast
2023-11-23 00:00:00	377.4620666503906
2023-11-24 00:00:00	376.2117919921875
2023-11-25 00:00:00	372.8912048339844
2023-11-26 00:00:00	372.3752746582031
2023-11-27 00:00:00	375.5839538574219
2023-11-28 00:00:00	370.18048095703125
2023-11-29 00:00:00	364.4391784667969
2023-11-30 00:00:00	368.0830383300781
2023-12-01 00:00:00	369.4279479980469
2023-12-02 00:00:00	370.2339172363281
2023-12-03 00:00:00	363.3330078125
2023-12-04 00:00:00	357.120361328125
2023-12-05 00:00:00	361.2340393066406
2023-12-06 00:00:00	359.9402160644531
2023-12-07 00:00:00	360.1163024902344
2023-12-08 00:00:00	358.23834228515625
2023-12-09 00:00:00	354.8597717285156

[Here is the github repository of the project: https://github.com/ngoinhaoto/timeSeriesChallenge2.](https://github.com/ngoinhaoto/timeSeriesChallenge2.)

The part for my model training is a google colab notebook file: TimeSeriesChallenge2.ipynb and the rest of the files are for deploying models on the Flask server.

To run it, just clone the repository and open it in Visual Studio Code, **install all the libraries**, type app.py in the terminal and control click the **http://127.0.0.1:5000 or type localhost:5000 on your browser.** *Thank you for having spent your time reading this.*

```
(base) ritherthemuncher@Rithers-MacBook-Air timeSeriesChallenge2 % python app.py
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Keras optim
dam`.
WARNING:absl:There is a known slowdown when using v2.11+ Keras optimizers on M1/M2 Macs. Falling back to the legacy Keras optimizer, i.
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Keras optim
dam`.
WARNING:absl:There is a known slowdown when using v2.11+ Keras optimizers on M1/M2 Macs. Falling back to the legacy Keras optimizer, i.
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer
[**********100%**********]  1 of 1 completed
1/1 [==============================] - 0s 344ms/step
10702700
344.01
341.49
[**********100%**********]  1 of 1 completed
1/1 [==============================] - 0s 331ms/step
377.46
 * Serving Flask app 'app'
 * Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with watchdog (fsevents)
```