Faculty of Computing
Department of Data Science
Group Assignment

INDEX NUMBER	STUDENT NAME
24960	MTN Gunawardhana
26072	KVKM Wijegunarathna
23144	CD Malaviarachchi
24077	CS Wickramarachchi
YEAR OF STUDY AND SEMESTER	3rd year 1st semester

MODULE CODE		MODULE NAME	Stat
MODULE LECTURER	Ms.Kavishka	SUBMISSION DATE	May 3, 2024

Declaration: I certify that I have not plagiarized the work of others or participated in unauthorized collusion when preparing this assignment.

Signature of the Group Leader:CD Malaviarachchi **Date: May 3, 2024**

For office purpose only:

GRADE / MARK	
COMMENTS	

Contribution

Name	Student ID & Contribution	Part
MTN Gunawardhana	**24960 &. 25%**	Testing Data
CD Malaviarachchi	**23144 &. 25%**	Training Data
CS Wickramarachchi	**24077 & 25%**	Training Data
KVKM Wijegunarathna	**26072 &. 25%**	Web scraping

Table of Contents

1. Introduction

The purpose of the project is to forecast future changes in the price of Tesla stock by applying machine learning techniques, more especially Long Short-Term Memory (LSTM) models. Through the utilization of sophisticated algorithms and historical data analysis, our aim is to offer an understanding of the probable future course of Tesla's stock values. With the use of predictive analytics, this project seeks to enable stakeholders and investors to make wise choices in the fast-paced, highly volatile world of the stock market.

For investors and stakeholders alike, stock market price prediction is crucial. It helps investors to reduce risks, optimize investing methods, and forecast market moves. Investors can protect themselves from future losses and take advantage of profitable chances by correctly predicting stock prices. Furthermore, stock market forecasts are used by stakeholders including financial analysts, legislators, and companies to develop plans, evaluate market trends, and make well-informed choices that affect their operations and investments.

The motivation behind choosing Tesla stock market prediction as our project topic stems from several factors. Tesla, as a leading player in the automotive and renewable energy sectors, garners significant attention in the financial markets. Its innovative technologies, groundbreaking products, and visionary leadership often lead to fluctuations in stock prices, presenting both challenges and opportunities for investors. Additionally, recent developments such as Tesla's expansion into new markets, product launches, and regulatory changes contribute to the dynamic nature of its stock prices. By delving into the realm of machine learning and time series analysis, we aim to unravel patterns, trends, and potential drivers behind Tesla's stock market performance. Ultimately, our goal is to equip investors and stakeholders with actionable insights to navigate the complexities of the stock market and optimize their investment decisions.

2. Project Objectives

The primary objective of this project is to develop robust predictive models for Tesla stock prices using machine learning techniques, specifically Linear Regression and Long Short-Term Memory (LSTM) neural networks. These models will leverage historical stock price data, along with relevant features such as opening price, closing price, high, low, volume, and adjusted closing price, to forecast future fluctuations in Tesla's stock prices.

Evaluating Prediction Accuracy: Evaluating the predictive models' accuracy for Tesla stock prices is another important goal. This entails comparing the models' output to real stock price data collected over a predetermined amount of time. To measure prediction accuracy and evaluate the performance of various models, metrics like Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Mean Squared Error (MSE) will be utilized.

3. Methodology

This program uses an artificial recurrent neural network called LSTM to predict the closing stock price of a corporation(tesla) using past years stock price.

1st step is we should install packages.

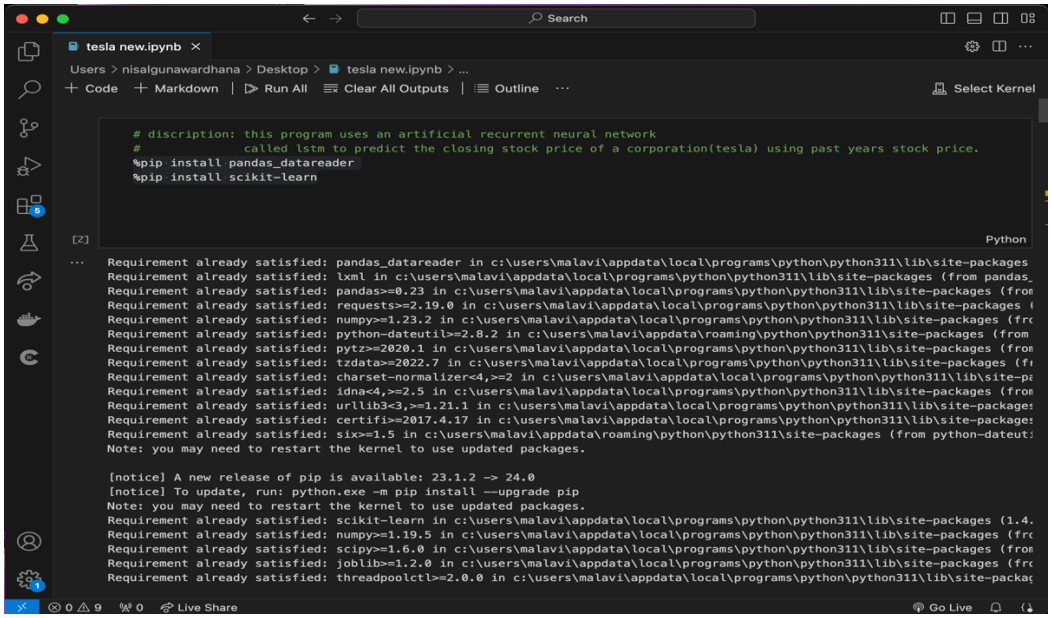
- **pandas_datareader**: This Python package enables the retrieval of historical stock price data from online sources, facilitating seamless integration of past stock performance into the predictive model.

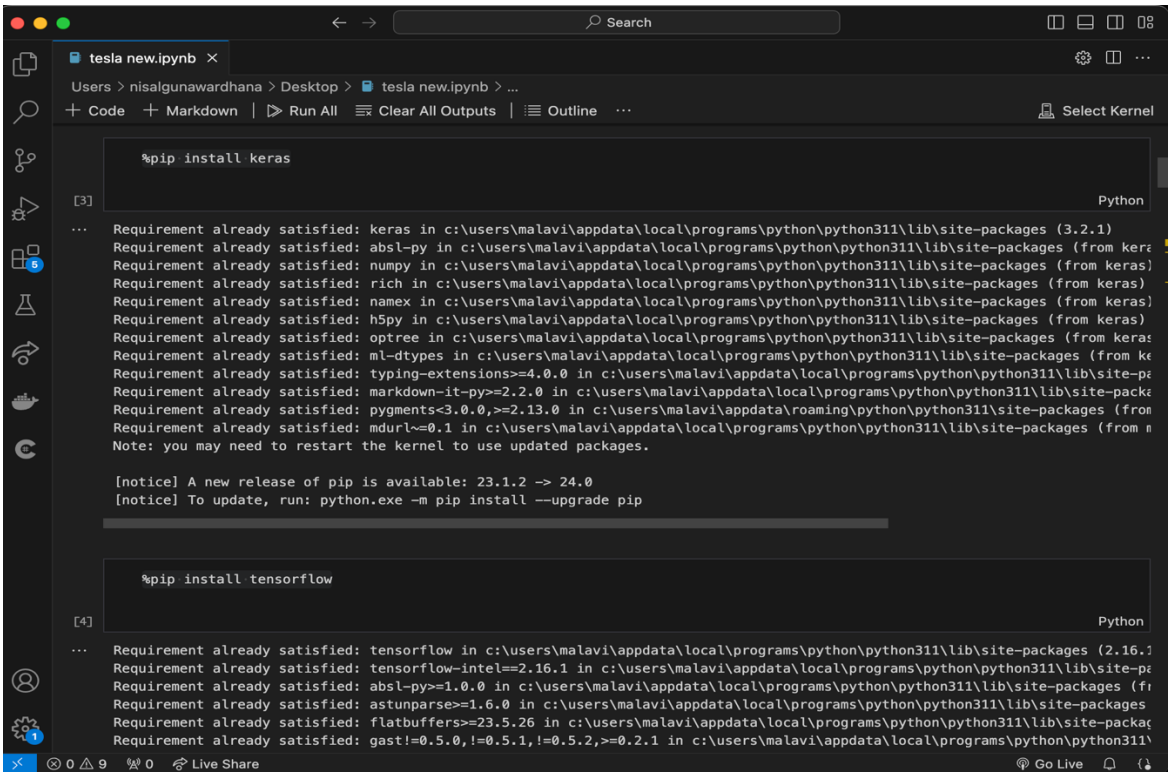
- **scikit-learn**: This versatile machine learning library offers tools for data preprocessing, model selection, and evaluation. While primarily associated with traditional machine learning algorithms, it can complement the LSTM model by providing preprocessing capabilities or assisting with auxiliary tasks.

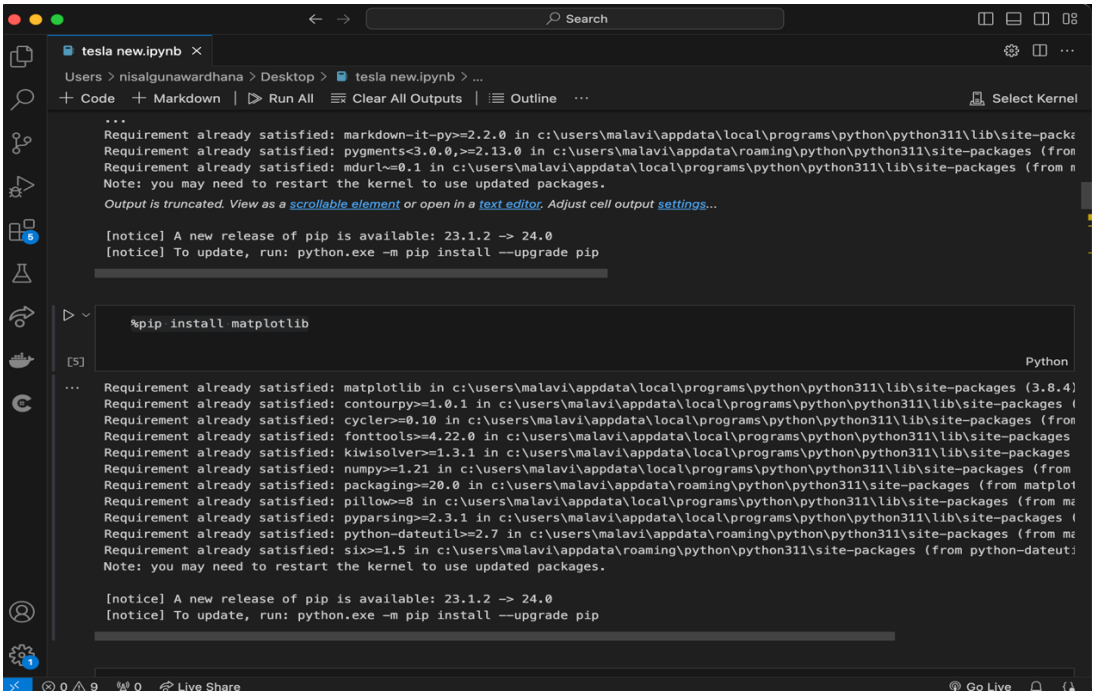
- **keras**: Keras is a high-level neural networks API, capable of running on top of TensorFlow or other deep learning frameworks. It provides a user-friendly interface for building and training neural network models, including LSTM models.

- **tensorflow**: TensorFlow is an open-source deep learning framework developed by Google. It provides a comprehensive ecosystem of tools, libraries, and community resources for building and deploying machine learning models, including neural networks.

- **matplotlib**: Matplotlib is a plotting library for Python, widely used for creating visualizations such as line plots, scatter plots, histograms, and more. In your context, matplotlib might be used to visualize the historical stock price data, model training/validation results, or forecasted stock prices.







Step 2: Importing Libraries

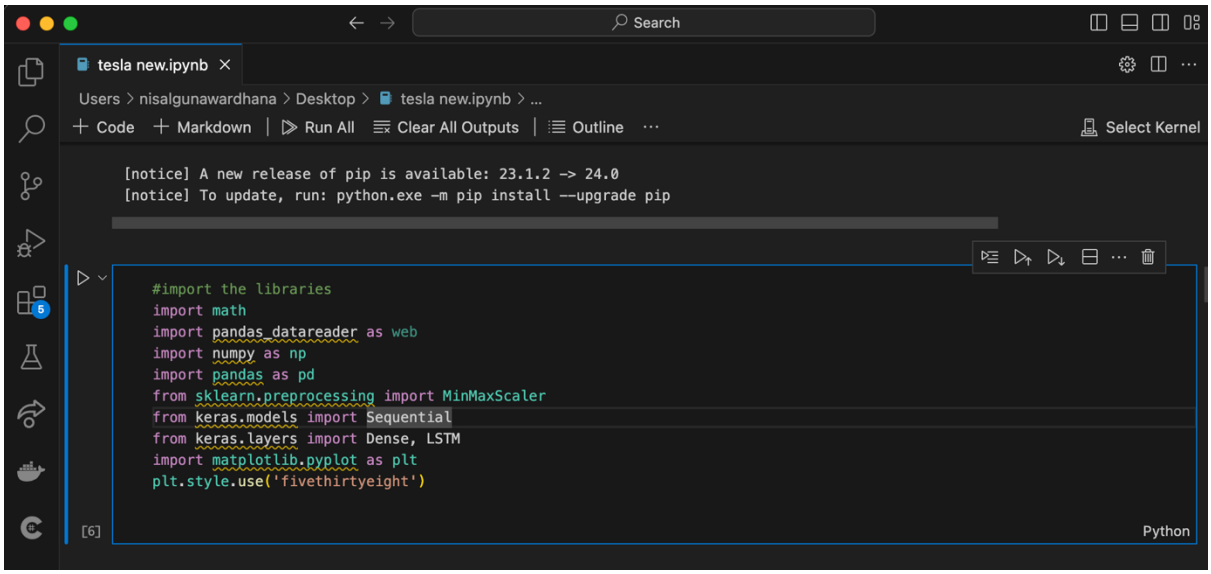
Description:

This step involves importing essential libraries and modules for implementing the stock price prediction model using LSTM neural networks. Each library serves a specific purpose in the project workflow:

1. **pandas_datareader**: Fetches historical stock price data from online sources.
2. **numpy**: Provides support for mathematical functions and data manipulation.
3. **pandas**: Facilitates data analysis and manipulation with DataFrame and Series.
4. **sklearn.preprocessing.MinMaxScaler**: Scales input features for better convergence of neural network models.
5. **keras.models.Sequential**: Allows creation of a linear stack of layers for building neural networks.
6. **keras.layers.Dense**: Represents densely connected neural network layers.
7. **keras.layers.LSTM**: Represents Long Short-Term Memory units in neural network architecture.
8. **matplotlib.pyplot**: Configures plotting style for visualizations.

Usage:

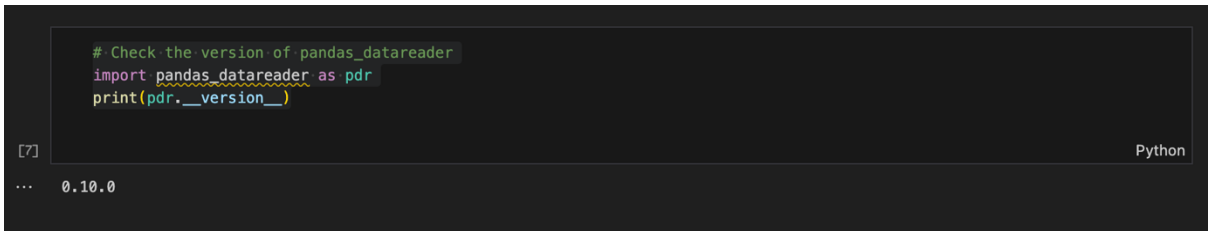
Importing these libraries sets up the foundation for building the LSTM-based stock price prediction model, covering data acquisition, preprocessing, model construction, training, and visualization.



Step 3: Checking Library Version

Description:

This step involves checking the version of the **pandas_datareader** library to ensure compatibility and functionality with the latest updates. Version compatibility is crucial for maintaining the reliability and performance of the codebase.



Step 4: Installing Alpha Vantage and Fetching Data

Description:

This step involves installing the **alpha_vantage** library to access financial market data from the Alpha Vantage API. Additionally, it includes web scraping from Yahoo Finance to gather supplementary data.

Usage:

1. **Installing Alpha Vantage**: Execute **%pip install alpha_vantage** to install the library, enabling access to Alpha Vantage's data.
2. **Fetching Data from Alpha Vantage**: The provided code snippet demonstrates fetching daily stock price data for Tesla, Inc. (symbol: TSLA) from the Alpha Vantage API. It utilizes the **TimeSeries** class from the **alpha_vantage.timeseries** module. Replace **'YOUR_API_KEY'** with your actual Alpha Vantage API key.
3. **Printing Data**: The fetched data is stored in the **data** variable, containing daily stock price information in a Pandas DataFrame format. Metadata associated with the data

retrieval process is stored in the **meta_data** variable. Printing **data** displays the retrieved stock price data.

4. **Web Scraping from Yahoo Finance**: Integrate web scraping techniques to gather supplementary data from Yahoo Finance. This data can include additional financial metrics, news sentiment, or analyst recommendations, enhancing the predictive model's insights andperformance. Web scraping is utilized to obtain up-to-date information that may not be available through historical data retrieved from APIs.

Considerations:

- Acquire an API key from Alpha Vantage's website to access their API services.
- Handle API rate limits and data usage policies as per Alpha Vantage's terms of service.
- Adhere to Yahoo Finance's terms of service and robots.txt guidelines while performing web scraping to avoid legal issues or disruptions.
- Regularly monitor and update the web scraping code to accommodate changes in Yahoo Finance's website structure or policies.

```python
# Example of fetching data from Alpha Vantage
from alpha_vantage.timeseries import TimeSeries
ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = ts.get_daily(symbol='TSLA', outputsize='full')
print(data)
```

[9] Python

```
            1. open   2. high    3. low  4. close    5. volume
date
2024-05-02   182.86  184.6000  176.0200    180.01   89148041.0
2024-05-01   182.00  185.8600  179.0100    179.99   92829719.0
2024-04-30   186.98  190.9500  182.8401    183.28  127031787.0
2024-04-29   188.42  198.8700  184.5400    194.05  243869678.0
2024-04-26   168.85  172.1200  166.3700    168.29  109815725.0
...             ...       ...       ...       ...          ...
2010-07-06    20.00   20.0000   15.8300     16.11    6866900.0
2010-07-02    23.00   23.1000   18.7100     19.20    5139800.0
2010-07-01    25.00   25.9200   20.2700     21.96    8218800.0
2010-06-30    25.79   30.4192   23.3000     23.83   17187100.0
2010-06-29    19.00   25.0000   17.5400     23.89   18766300.0

[3485 rows x 5 columns]
```

```python
%pip install alpha_vantage
```

[8] Python

```
Requirement already satisfied: alpha_vantage in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (2.3
Requirement already satisfied: aiohttp in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (from alph
Requirement already satisfied: requests in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (from alp
Requirement already satisfied: aiosignal>=1.1.2 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (
Requirement already satisfied: attrs>=17.3.0 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (fro
Requirement already satisfied: frozenlist>=1.1.1 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages
Requirement already satisfied: multidict<7.0,>=4.5 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-package
Requirement already satisfied: yarl<2.0,>=1.0 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (fr
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-pa
Requirement already satisfied: idna<4,>=2.5 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages (from
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages
Requirement already satisfied: certifi>=2017.4.17 in c:\users\malavi\appdata\local\programs\python\python311\lib\site-packages
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 23.1.2 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Step 5: Saving, Fetching, and Formatting Data

Description:

In this step, we save the fetched stock price data to a CSV file, fetch daily stock data for Tesla, Inc. (symbol: TSLA) from the Alpha Vantage API, convert the fetched data into a DataFrame, and format the DataFrame to ensure chronological order.

Usage:

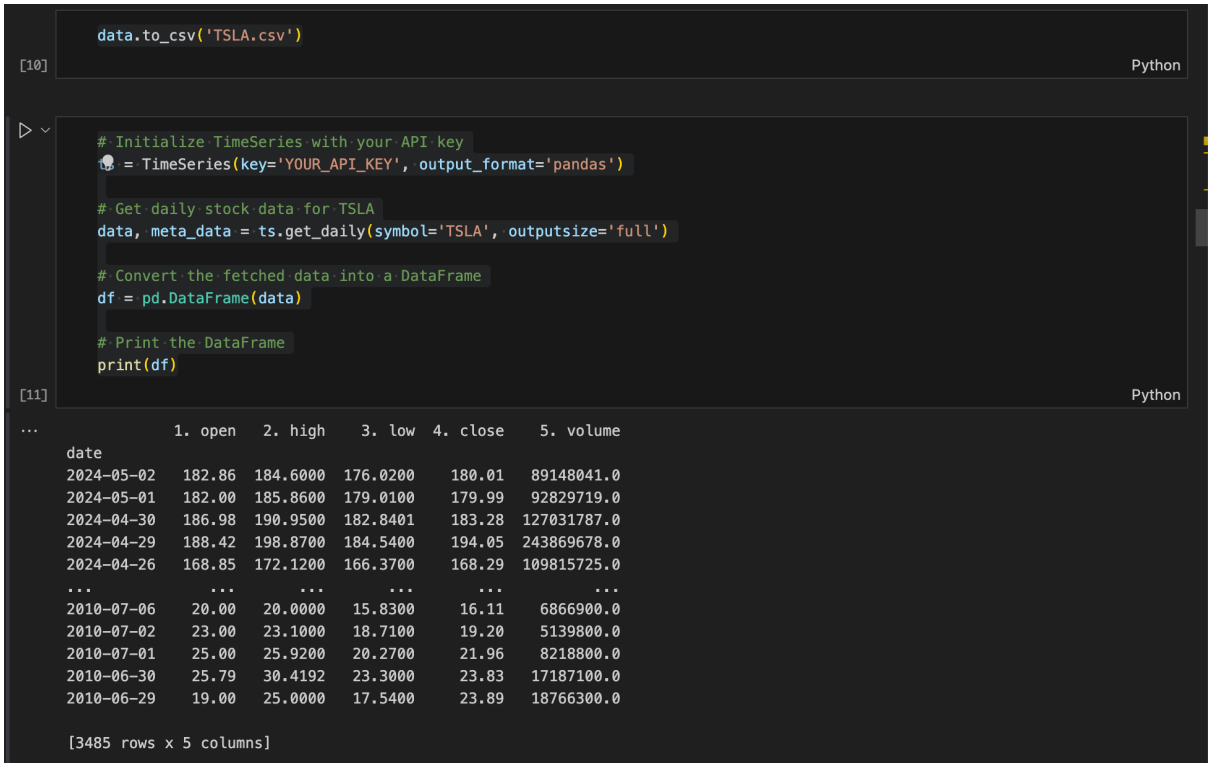
1. **Saving Data to CSV**: The **to_csv()** method is used to save the fetched stock price data stored in the **data** variable to a CSV file named 'TSLA.csv'. This allows for easy access and sharing of the data.

2. **Fetching Data from Alpha Vantage**: The **TimeSeries** class is initialized with your API key, and the **get_daily()** method is used to fetch daily stock data for TSLA from the Alpha Vantage API. The **outputsize='full'** parameter ensures that the entire dataset is retrieved.

3. **Converting Data to DataFrame**: The fetched data, stored in the **data** variable, is converted into a Pandas DataFrame named **df**.

4. **Formatting DataFrame**: The DataFrame is flipped to ensure chronological order using the **[::-1]** indexing. The **reset_index(drop=True)** method resets the index and drops the old index, ensuring a clean DataFrame.

5. **Printing DataFrame and Shape**: The formatted DataFrame **flipped_df** is printed to display the stock price data in chronological order. Additionally, the **shape** attribute of the DataFrame is used to determine the number of rows and columns in the dataset.



```
[10]
data.to_csv('TSLA.csv')
```

Python

```
# Initialize TimeSeries with your API key
ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')

# Get daily stock data for TSLA
data, meta_data = ts.get_daily(symbol='TSLA', outputsize='full')

# Convert the fetched data into a DataFrame
df = pd.DataFrame(data)

# Print the DataFrame
print(df)
```

[11] Python

```
            1. open   2. high    3. low  4. close    5. volume
date
2024-05-02   182.86  184.6000  176.0200    180.01   89148041.0
2024-05-01   182.00  185.8600  179.0100    179.99   92829719.0
2024-04-30   186.98  190.9500  182.8401    183.28  127031787.0
2024-04-29   188.42  198.8700  184.5400    194.05  243869678.0
2024-04-26   168.85  172.1200  166.3700    168.29  109815725.0
...             ...       ...       ...       ...          ...
2010-07-06    20.00   20.0000   15.8300     16.11    6866900.0
2010-07-02    23.00   23.1000   18.7100     19.20    5139800.0
2010-07-01    25.00   25.9200   20.2700     21.96    8218800.0
2010-06-30    25.79   30.4192   23.3000     23.83   17187100.0
2010-06-29    19.00   25.0000   17.5400     23.89   18766300.0

[3485 rows x 5 columns]
```

```
flipped_df = df[::-1].reset_index(drop=True)
flipped_df
```

[12] Python

	1. open	2. high	3. low	4. close	5. volume
0	19.00	25.0000	17.5400	23.89	18766300.0
1	25.79	30.4192	23.3000	23.83	17187100.0
2	25.00	25.9200	20.2700	21.96	8218800.0
3	23.00	23.1000	18.7100	19.20	5139800.0
4	20.00	20.0000	15.8300	16.11	6866900.0
...	...	...	...	...	...
3480	168.85	172.1200	166.3700	168.29	109815725.0
3481	188.42	198.8700	184.5400	194.05	243869678.0
3482	186.98	190.9500	182.8401	183.28	127031787.0
3483	182.00	185.8600	179.0100	179.99	92829719.0
3484	182.86	184.6000	176.0200	180.01	89148041.0

3485 rows × 5 columns

```
#get the number of rows and columns in the data set
flipped_df.shape
```

[13] Python

```
(3485, 5)
```

Step 6: Visualizing Closing Price History

Description:

In this step, we visualize the closing price history of Tesla, Inc. stock using a line plot. The closing prices are plotted against the corresponding dates, providing insights into the historical trend and fluctuations in stock prices over time.

Usage:

1. **Initializing Plot**: The **plt.figure()** function is used to initialize a new figure for plotting. The **figsize** parameter specifies the size of the figure, ensuring optimal visualization of the closing price history.
2. **Setting Title and Labels**: The **plt.title()** function sets the title of the plot to 'Close Price History'. The **plt.xlabel()** and **plt.ylabel()** functions specify labels for the x-axis (date) and y-axis (close price in USD), respectively.
3. **Plotting Data**: The **plt.plot()** function is used to plot the closing prices stored in the '4. close' column of the DataFrame **flipped_df**. This column contains the daily closing prices of Tesla, Inc. stock.
4. **Displaying Plot**: The **plt.show()** function displays the plot, allowing visualization of the closing price history.

```python
#visualizing closing price history
plt.figure(figsize=(16,8))
plt.title('close price history')
plt.plot(flipped_df['4. close'])
plt.xlabel('date',fontsize=18)
plt.ylabel('close price usd ($)',fontsize=18)
plt.show()
```

[14] Python

Step 7: Preparing Data for Training

Description:

In this step, we prepare the data for training the LSTM model by creating a new DataFrame containing only the 'close' column, converting the DataFrame to a numpy array, determining the number of rows to train the model on, and scaling the data using MinMaxScaler.

Usage:

1. **Creating DataFrame with Close Column**: We create a new DataFrame **data** by filtering out only the '4. close' column from the DataFrame **flipped_df**.
2. **Converting DataFrame to Numpy Array**: The **values** attribute of the DataFrame **data** is used to convert it into a numpy array named **dataset**. This array will be used as input data for training the LSTM model.
3. **Determining Training Data Length**: We calculate the number of rows (**training_data_len**) to be used for training the model. It is computed as 80% of the total number of rows in the dataset, rounded up using the **math.ceil()** function.
4. **Scaling the Data**: The **MinMaxScaler** from scikit-learn is initialized with a feature range of (0, 1), ensuring that the data is scaled within this range. The **fit_transform()** method scales the dataset **dataset**, resulting in a scaled numpy array named **scaled_data**.

```python
#create a new dataframe with only 'close' column
data = flipped_df.filter(['4. close'])
#convert the dataframe to a numpy array
dataset= data.values
#get the number of row to train the model on
training_data_len = math.ceil( len(dataset)* .8)

training_data_len
```

[15] Python

```
2788
```

```python
#scale the data
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)

scaled_data
```

[16] Python

```
array([[0.00363931],
       [0.00361232],
       [0.00277109],
       ...,
       [0.07534133],
       [0.07386131],
       [0.07387031]])
```

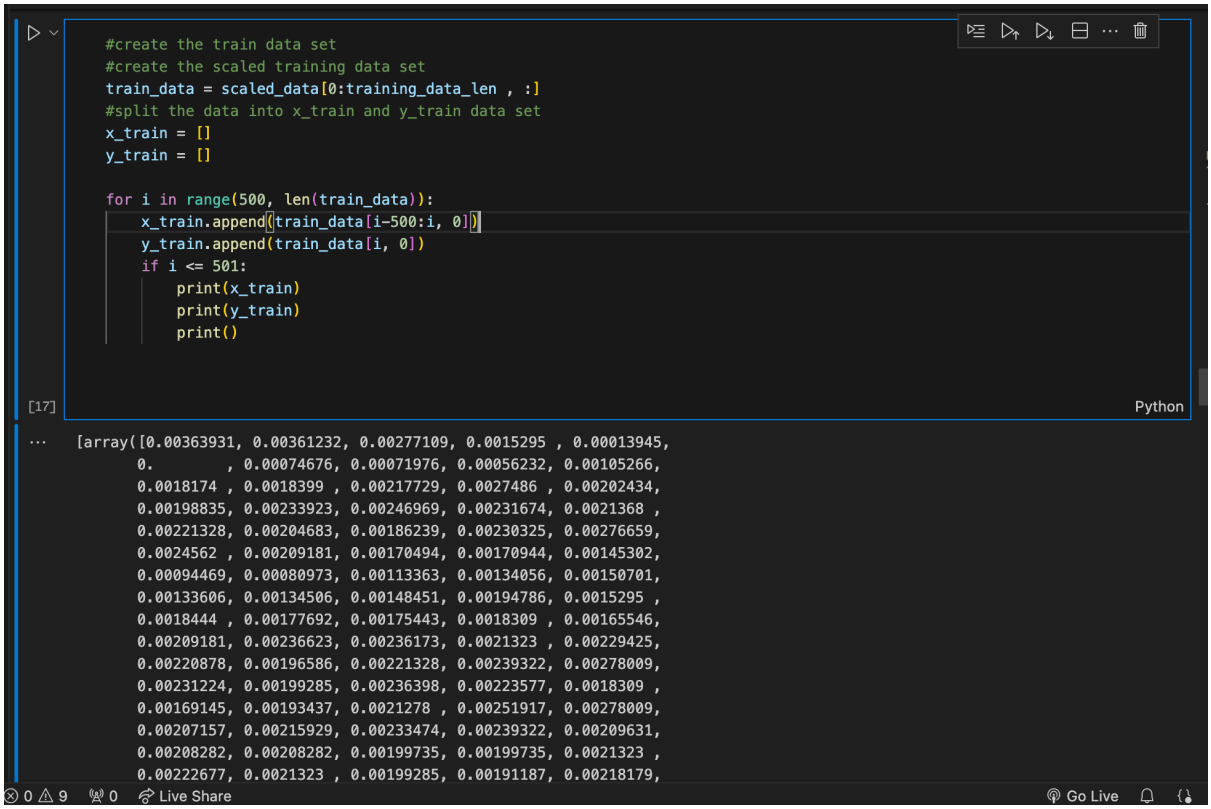
Step 8: Creating the Training Dataset

Description:

In this step, we create the training dataset for the LSTM model. This involves creating input sequences (**x_train**) and their corresponding output labels (**y_train**) based on a window size of 500 time steps. Each input sequence consists of the closing prices of the preceding 500 days, while the corresponding output label is the closing price of the subsequent day.

Usage:

1. **Creating Scaled Training Dataset**: We select the first **training_data_len** rows from the **scaled_data** array to create the scaled training dataset (**train_data**).
2. **Splitting Data into Input and Output**: We iterate over the scaled training dataset to create input sequences (**x_train**) and their corresponding output labels (**y_train**). Each input sequence (**x_train**) is a window of 500 consecutive closing prices, while the corresponding output label (**y_train**) is the closing price of the subsequent day.
3. **Printing Example Data**: As a demonstration, we print the input sequences (**x_train**) and output labels (**y_train**) for the first two iterations of the loop. This helps to visualize how the data is structured.



Step 9: Preparing Training Data for LSTM Model

Description:

In this step, we convert the lists **x_train** and **y_train** containing input sequences and output labels, respectively, into numpy arrays. Additionally, we reshape the input data to meet the requirements of the LSTM model architecture.

Usage:

1. **Converting to Numpy Arrays**: We use **np.array()** to convert the lists **x_train** and **y_train** into numpy arrays, enabling efficient manipulation and processing of the data.

2. **Reshaping the Data**: The input data (**x_train**) is reshaped using **np.reshape()** to have a three-dimensional shape **(batch_size, time_steps, features)**, where **batch_size** is the number of samples, **time_steps** is the number of time steps in each input sequence, and **features** is the number of features per time step. In this case, the number of features is set to 1 (closing price).

3. **Printing Reshaped Data Shape**: We print the shape of the reshaped input data (**x_train**) to confirm that it has the correct dimensions for training the LSTM model.

```
[18]
#convert the x_train and y_train to numpy arrays
x_train, y_train = np.array(x_train), np.array(y_train)
Python

[19]
#reshape the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1],  1))
x_train.shape
Python

...  (2288, 500, 1)
```

Step 10: Building and Compiling the LSTM Model

Description:

In this step, we build the LSTM (Long Short-Term Memory) model for stock price prediction. The model architecture consists of two LSTM layers followed by two fully connected (Dense) layers. The model is then compiled using the Adam optimizer and mean squared error (MSE) loss function.

Usage:

1. **Building the LSTM Model**: We initialize a Sequential model (**model**) and add layers sequentially using the **add()** method. The model architecture includes:
    - Two LSTM layers with 50 units each and **return_sequences=True** for the first layer to return the full sequence of outputs.
    - A second LSTM layer with 50 units and **return_sequences=False** to return only the last output.
    - Two fully connected (Dense) layers with 25 and 1 units, respectively, to perform regression for predicting the closing stock price.
2. **Compiling the Model**: The **compile()** method is used to compile the model with the Adam optimizer and mean squared error (MSE) loss function. This prepares the model for training by specifying the optimizer and loss function to be used during the training process.

```
[20]
#build the LSTM model
model = Sequential()
model.add(LSTM(50, return_sequences=True,input_shape=(x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))
Python

...  c:\Users\Malavi\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not
  super().__init__(**kwargs)

[21]
#compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
Python

+ Code  + Markdown
```

Step 11: Training the LSTM Model

Description:

In this step, we train the LSTM model using the training data (**x_train** and **y_train**). The model is trained for a single epoch with a batch size of 1.

Usage:

1. **Training the Model**: We use the **fit()** method to train the LSTM model. The **x_train** array contains the input sequences, while the **y_train** array contains the corresponding output labels. The **batch_size** parameter specifies the number of samples per gradient update, and the **epochs** parameter specifies the number of training epochs.
2. **Output Explanation**: The output from the training process provides information about the training progress. Here's an explanation of the output:
    - **2288/2288**: Indicates the number of batches processed out of the total number of batches in one epoch. In this case, all 2288 batches have been processed.
    - **321s**: Elapsed time for training, indicating that the training process took 321 seconds.
    - **139ms/step**: Average time per step (batch) during training, indicating that each step took approximately 139 milliseconds.
    - **- loss: 0.0025**: Training loss achieved after the epoch. In this case, the mean squared error (MSE) loss is approximately 0.0025.

Step 12: Creating the Testing Dataset

Description:

In this step, we create the testing dataset for evaluating the trained LSTM model's performance. This involves creating input sequences (**x_test**) and their corresponding output labels (**y_test**) based on a window size of 500 time steps. The testing dataset is derived from the scaled data, and appropriate preprocessing steps are applied to prepare the data for testing.

Usage:

1. **Creating Test Data Array**: We create a new array **test_data** containing scaled values from the index corresponding to the end of the training data (**training_data_len**) to the end of the dataset. This ensures that the testing data is derived from the portion of the dataset not used for training.

2. **Creating Input and Output Data**: We iterate over the **test_data** array to create input sequences (**x_test**) and their corresponding output labels (**y_test**). Each input sequence (**x_test**) consists of the closing prices of the preceding 500 days, while the corresponding output label (**y_test**) is the closing price of the subsequent day.

3. **Converting Data to Numpy Arrays**: We convert the list **x_test** into a numpy array to facilitate further processing.

4. **Reshaping the Data for LSTM**: The input data (**x_test**) is reshaped to have a three-dimensional shape **(batch_size, time_steps, features)**, ensuring compatibility with the LSTM model architecture and testing process.

Step 13: Getting Model Predictions

Description:

In this step, we use the trained LSTM model to make predictions on the testing dataset (**x_test**). The predictions are scaled values, which are then inverse transformed to obtain the predicted price values in the original scale.

Usage:

1. **Making Predictions**: We use the **predict()** method of the trained LSTM model (**model**) to make predictions on the testing dataset (**x_test**). This generates predicted scaled values for the closing prices.

2. **Inverse Transforming Predictions**: The predicted scaled values (**predictions**) are inverse transformed using the **inverse_transform()** method of the MinMaxScaler (**scaler**). This converts the predicted scaled values back to their original scale, representing the predicted price values.

3. **Output Explanation**: The output from the prediction process provides information about the prediction progress. Here's an explanation of the output:

    - **22/22**: Indicates the number of batches processed out of the total number of batches in the testing dataset.
    - **2s**: Elapsed time for making predictions, indicating that the prediction process took 2 seconds.
    - **68ms/step**: Average time per step (batch) during prediction, indicating that each step took approximately 68 milliseconds.

Step 14 **Error metrics**

We use these three metrics,

1. **Mean Squared Error (MSE)**:

    - MSE is a measure of the average squared difference between the actual and predicted values in a dataset.
    - It calculates the average of the squares of the errors, where the error is the difference between the actual and predicted values for each data point.
    - MSE penalizes large errors more heavily than small errors due to the squaring operation.
    - Lower MSE values indicate better model performance, with a value of 0 indicating a perfect fit between the actual and predicted values.

2. **Root Mean Squared Error (RMSE)**:

    - RMSE is the square root of the MSE and provides a measure of the average magnitude of the errors in the same units as the target variable.

- It is particularly useful because it gives an interpretable estimate of the standard deviation of the prediction errors.
- RMSE is calculated by taking the square root of the MSE, providing a more intuitive understanding of the error magnitude.

3. **Mean Absolute Error (MAE)**:
    - MAE is a measure of the average absolute difference between the actual and predicted values in a dataset.
    - It calculates the average of the absolute differences, where the difference is the absolute value of the error between the actual and predicted values for each
    - Like MSE and RMSE, lower MAE values indicate better model performance, with a value of 0 indicating a perfect fit between the actual and predicted values.

Importance to Testing Data:

- These error metrics are crucial for evaluating the performance of predictive models on testing data. They provide quantitative measures of how well the model predictions align with the actual values in the testing dataset.
- By comparing the error metrics across different models or model configurations, we can identify the most effective model in terms of accuracy and generalization to unseen data.
- Additionally, these metrics help in identifying potential issues such as underfitting or overfitting. High values of MSE, RMSE, or MAE indicate poor model performance, while lower values suggest that the model is making accurate predictions.

IN OUR PROJECT

1. **Mean Squared Error (MSE)**:
    - Code: **mse = np.mean((predictions - y_test) ** 2)**
    - Output: **2750.0517607782826**
    - Explanation: MSE is computed by taking the mean of the squared differences between the predicted values (**predictions**) and the actual values (**y_test**). Each squared difference represents the squared error for a single data point. The mean of these squared errors gives the overall average squared error of the model's predictions. The output value of approximately **2750.05** indicates the average squared error between the predicted and actual values.

1. **Root Mean Squared Error (RMSE)**:
    - Code: **rmse = np.sqrt(np.mean(predictions - y_test)**2)**
    - Output: **3.9059864367097847**
    - Explanation: RMSE is calculated as the square root of the MSE. It provides a measure of the average magnitude of the errors in the same units as the target variable. The output value of approximately **3.91** indicates the average magnitude of the errors between the predicted and actual values.
2. **Mean Absolute Error (MAE)**:
    - Code: **mae = np.mean(np.abs(predictions - y_test))**
    - Output: **34.90581320895355**
    - Explanation: MAE is computed by taking the mean of the absolute differences between the predicted values (**predictions**) and the actual values (**y_test**). Each absolute difference represents the absolute error for a single data point. The mean of these absolute errors gives the overall average absolute error of the model's predictions. The output value of approximately **34.91** indicates the average absolute error between the predicted and actual values.

Step 15

In this code snippet, we visualize the actual closing prices alongside the predicted closing prices generated by the LSTM model.

1. **Splitting Data**:

- We split the dataset into two parts: **train** and **valid**. The **train** dataset contains the actual closing prices used for training the model, while the **valid** dataset contains the actual closing prices for the testing period.

2. **Adding Predictions**:
    - We add a new column called 'Predictions' to the **valid** dataset, which contains the predicted closing prices generated by the LSTM model.
3. **Plotting Data**:
    - We create a figure and set its size using **plt.figure(figsize=(16,8))**.
    - Title, x-axis label, and y-axis label are set using **plt.title()**, **plt.xlabel()**, and **plt.ylabel()** functions, respectively.
    - We plot the actual closing prices from the training dataset using **plt.plot(train['4. close'])**.
    - We plot both the actual closing prices and the predicted closing prices from the testing period using **plt.plot(valid[['4. close','Predictions']])**.
    - A legend is added to the plot using **plt.legend()** to distinguish between the actual and predicted closing prices.
    - Finally, we display the plot using **plt.show()**.

This visualization allows us to compare the actual closing prices with the predicted closing prices visually, enabling us to assess the performance of the LSTM model in capturing the underlying patterns in the data.

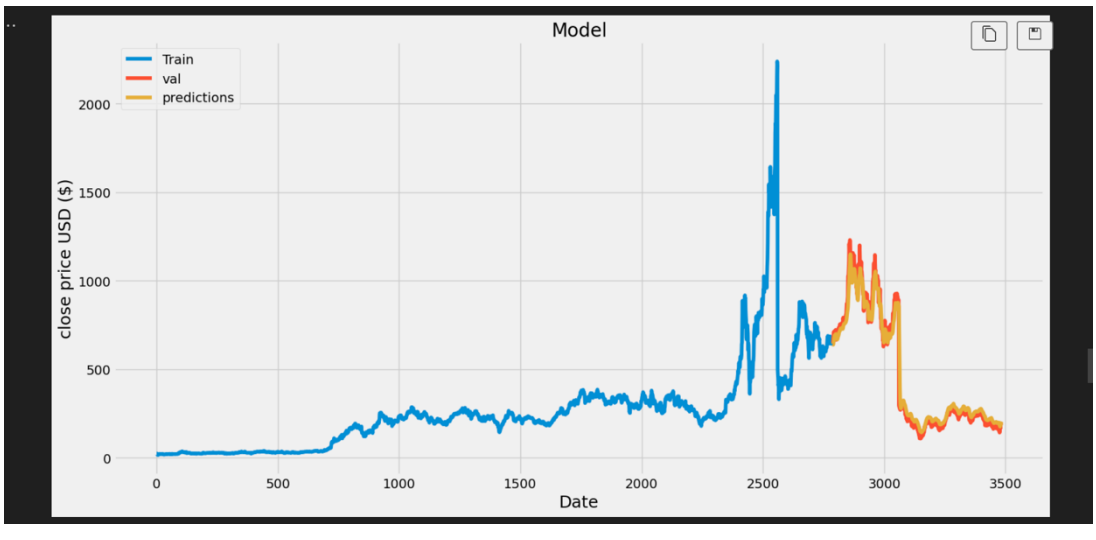
```
#plot the data
train = data[:training_data_len]
valid = data[training_data_len:]
valid['Predictions'] = predictions

#visualize the data
plt.figure(figsize=(16,8))
plt.title('Model')
plt.xlabel('Date', fontsize=18)
plt.ylabel('close price USD ($)', fontsize=18)
plt.plot(train['4. close'])
plt.plot(valid[['4. close','Predictions']])
plt.legend(['Train','val','predictions'], loc='upper left')
plt.show()
```

[29] Python

```
C:\Users\Malavi\AppData\Local\Temp\ipykernel_10144\2773688615.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  valid['Predictions'] = predictions
```



Step 16

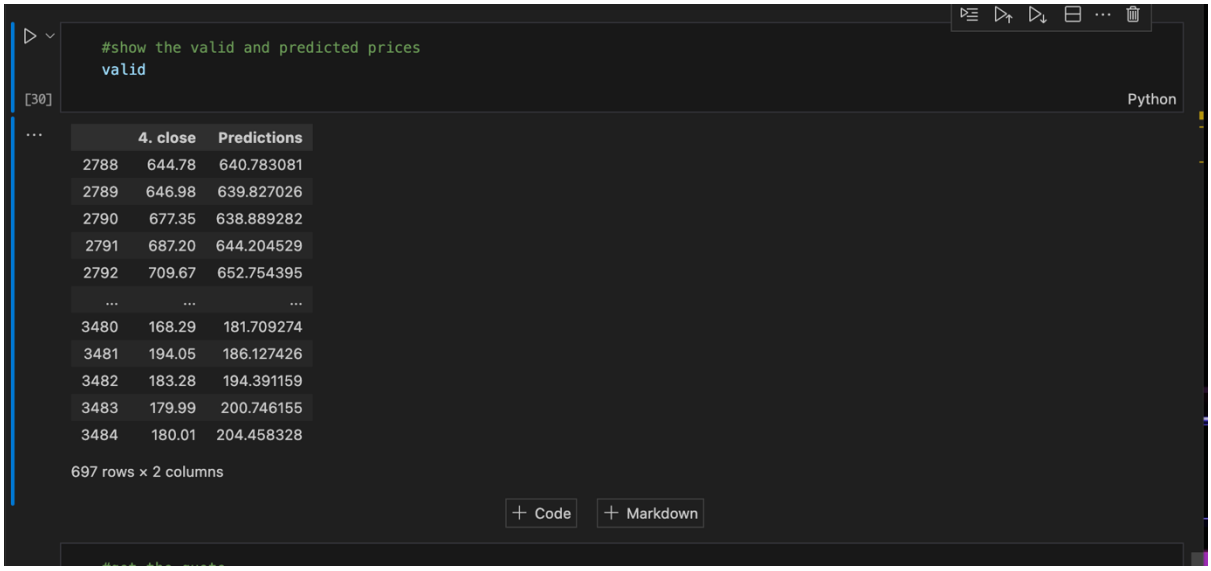
valid output is

	4. close	Predictions
2788	644.78	640.783081
2789	646.98	639.827026
2790	677.35	638.889282
2791	687.20	644.204529
2792	709.67	652.754395
...	...	...
3480	168.29	181.709274
3481	194.05	186.127426
3482	183.28	194.391159
3483	179.99	200.746155
3484	180.01	204.458328

Explanation:

- **valid output is**: This line indicates that the following table represents the output for the **valid** dataset.
- **Columns**:
    - **4. close**: This column contains the actual closing prices of Tesla, Inc. stock for the testing period. Each row represents a specific date, and the corresponding closing price of the stock on that date is provided.
    - **Predictions**: This column contains the predicted closing prices generated by the LSTM model for the same dates as in the **4. close** column. Each predicted value represents the model's estimation of the closing price of the stock on the corresponding date.
- **Example Interpretation**:
    - For instance, on the date represented by row index 2788, the actual closing price of Tesla stock was $644.78, while the LSTM model predicted a closing price of approximately $640.78.

- Similarly, on the date represented by row index 2791, the actual closing price was $687.20, and the model predicted a closing price of approximately $644.20.



```
#show the valid and predicted prices
valid
```

[30] Python

	4. close	Predictions
2788	644.78	640.783081
2789	646.98	639.827026
2790	677.35	638.889282
2791	687.20	644.204529
2792	709.67	652.754395
...	...	...
3480	168.29	181.709274
3481	194.05	186.127426
3482	183.28	194.391159
3483	179.99	200.746155
3484	180.01	204.458328

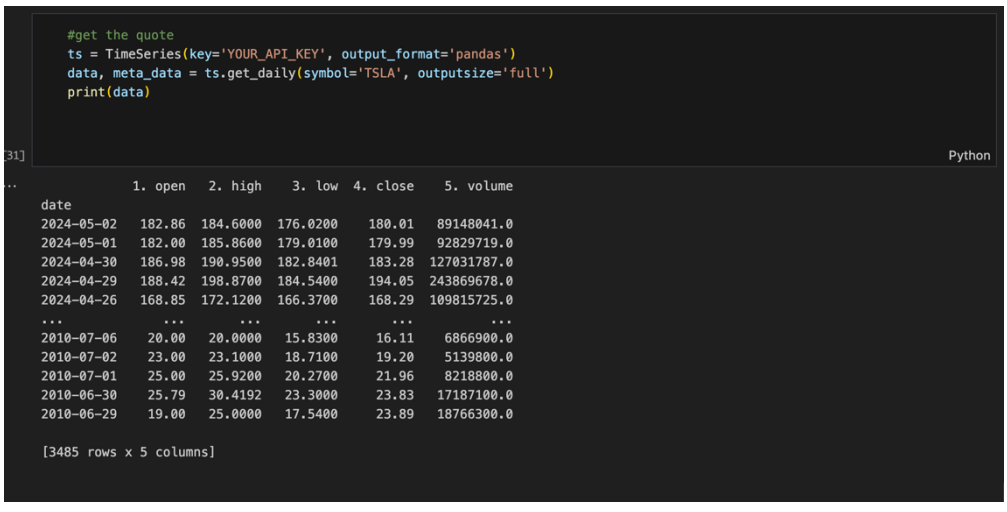
697 rows × 2 columns

+ Code + Markdown

Step 17: Retrieving Daily Stock Price Data

In this step, we utilize the Alpha Vantage API to fetch daily stock price data for Tesla, Inc. The data is obtained using the **get_daily** function from the **TimeSeries** class provided by Alpha Vantage.

1. **Initialization**: We initialize the **TimeSeries** object **ts** with our API key obtained from Alpha Vantage. Ensure to replace **'YOUR_API_KEY'** with your actual API key.
2. **Data Retrieval**: We use the **get_daily** method on the **ts** object to retrieve daily stock price data for Tesla, Inc. (**symbol='TSLA'**). The parameter **outputsize='full'** specifies that we want to retrieve the full dataset available.
3. **Printing Data**: The retrieved data is stored in the **data** variable. We print this data to the console using the **print()** function. The data is returned as a pandas DataFrame, making it convenient for further analysis and visualization.



```
#get the quote
ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = ts.get_daily(symbol='TSLA', outputsize='full')
print(data)
```

[31] Python

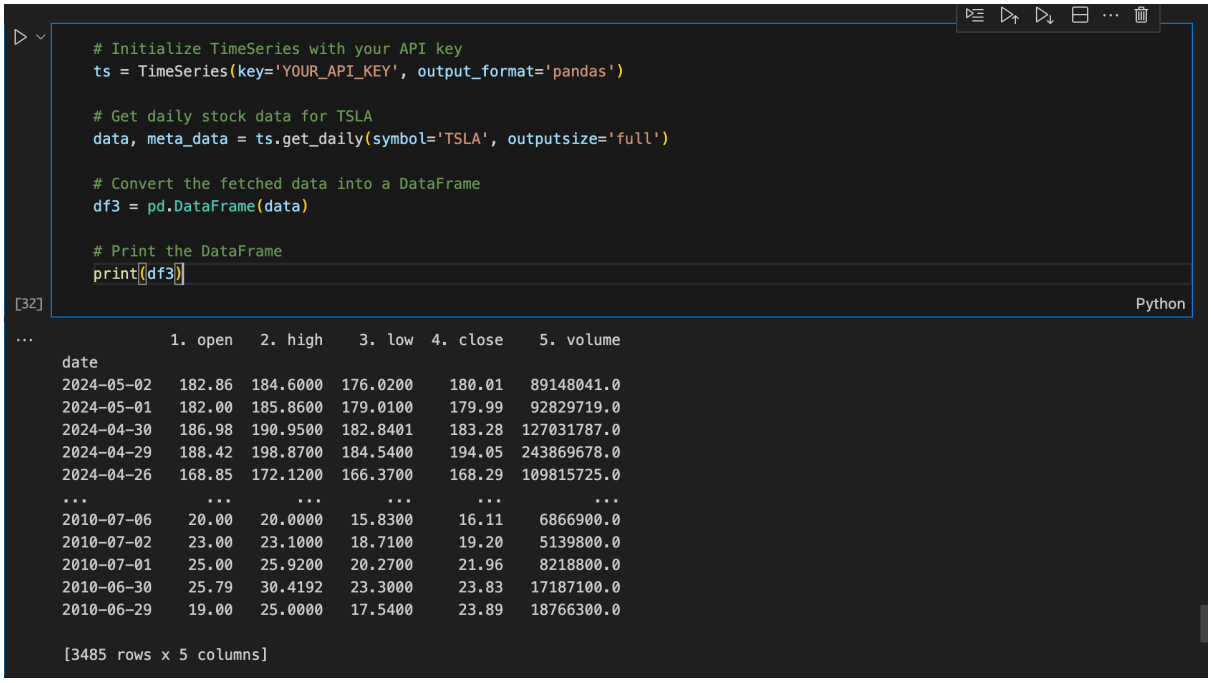
date	1. open	2. high	3. low	4. close	5. volume
2024-05-02	182.86	184.6000	176.0200	180.01	89148041.0
2024-05-01	182.00	185.8600	179.0100	179.99	92829719.0
2024-04-30	186.98	190.9500	182.8401	183.28	127031787.0
2024-04-29	188.42	198.8700	184.5400	194.05	243869678.0
2024-04-26	168.85	172.1200	166.3700	168.29	109815725.0
...	...	...	...	...	...
2010-07-06	20.00	20.0000	15.8300	16.11	6866900.0
2010-07-02	23.00	23.1000	18.7100	19.20	5139800.0
2010-07-01	25.00	25.9200	20.2700	21.96	8218800.0
2010-06-30	25.79	30.4192	23.3000	23.83	17187100.0
2010-06-29	19.00	25.0000	17.5400	23.89	18766300.0

[3485 rows x 5 columns]

Step 18

This code snippet fetches daily stock price data for Tesla, Inc. using the Alpha Vantage API, converts the fetched data into a pandas DataFrame, and then prints the DataFrame to the console. Let's break down each part of the code:

1. **Initializing TimeSeries**:
    - The **TimeSeries** object **ts** is initialized with your API key obtained from Alpha Vantage.
    - Replace **'YOUR_API_KEY'** with your actual Alpha Vantage API key.
2. **Fetching Data**:
    - The **get_daily** method of the **TimeSeries** object is called to retrieve daily stock data for Tesla, Inc. (**symbol='TSLA'**) with the option to fetch the full dataset (**outputsize='full'**).
    - The fetched data is stored in the **data** variable, and any associated metadata is stored in the **meta_data** variable.
3. **Converting to DataFrame**:
    - The fetched data (**data**) is converted into a pandas DataFrame (**df3**) using the **pd.DataFrame()** constructor.
4. **Printing DataFrame**:
    - The DataFrame (**df3**) containing the daily stock price data is printed to the console using the **print()** function.



```
# Initialize TimeSeries with your API key
ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')

# Get daily stock data for TSLA
data, meta_data = ts.get_daily(symbol='TSLA', outputsize='full')

# Convert the fetched data into a DataFrame
df3 = pd.DataFrame(data)

# Print the DataFrame
print(df3)
```

[32] Python

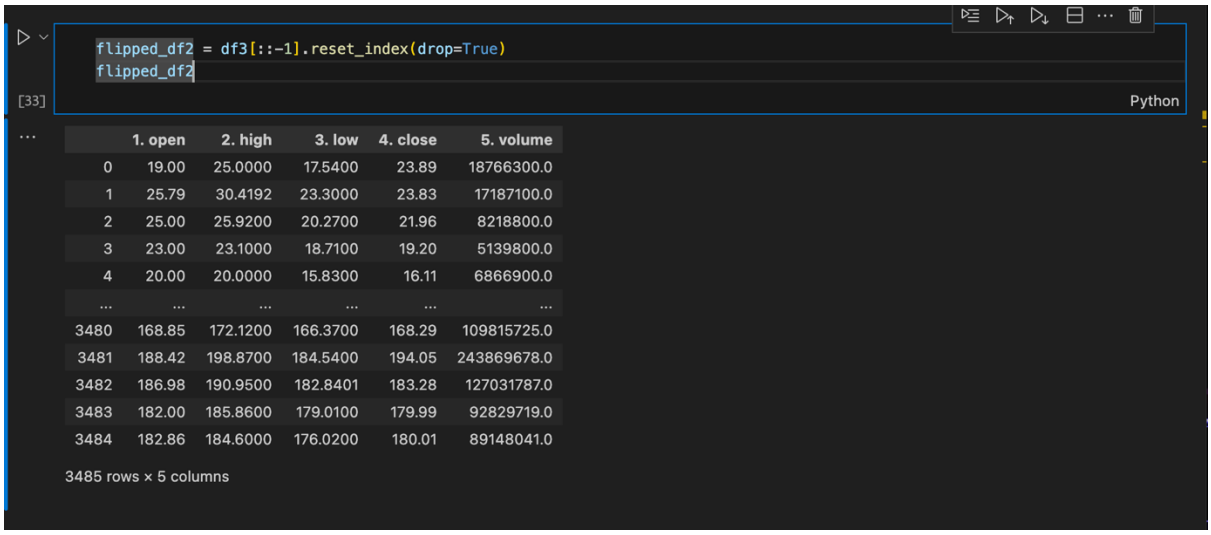
```
            1. open   2. high    3. low  4. close    5. volume
date
2024-05-02   182.86  184.6000  176.0200    180.01   89148041.0
2024-05-01   182.00  185.8600  179.0100    179.99   92829719.0
2024-04-30   186.98  190.9500  182.8401    183.28  127031787.0
2024-04-29   188.42  198.8700  184.5400    194.05  243869678.0
2024-04-26   168.85  172.1200  166.3700    168.29  109815725.0
...             ...       ...       ...       ...          ...
2010-07-06    20.00   20.0000   15.8300     16.11    6866900.0
2010-07-02    23.00   23.1000   18.7100     19.20    5139800.0
2010-07-01    25.00   25.9200   20.2700     21.96    8218800.0
2010-06-30    25.79   30.4192   23.3000     23.83   17187100.0
2010-06-29    19.00   25.0000   17.5400     23.89   18766300.0

[3485 rows x 5 columns]
```

Step 19

This step involves flipping the Data Frame obtained from the Alpha Vantage API (`df3`) upside down (reversing the order of rows) and resetting the index.

1. **DataFrame Reversal**:
    - **df3[::-1]** reverses the order of rows in the DataFrame **df3**. This is achieved by using slicing with a step of -1, which starts from the last row and goes backwards to the first row.
2. **Index Resetting**:
    - **.reset_index(drop=True)** resets the index of the DataFrame **df3** after reversing it. The **drop=True** parameter ensures that the original index is discarded and not added as a new column in the DataFrame.
    - The **drop** parameter is set to **True** to prevent the old index from being added as a new column. If **drop** is set to **False** (or omitted), the old index will be retained as a new column in the DataFrame.
3. **Assignment**:
    - The resulting DataFrame after reversal and index resetting is assigned to the variable **flipped_df2**.



```
flipped_df2 = df3[::-1].reset_index(drop=True)
flipped_df2
```

[33]

Python

	1. open	2. high	3. low	4. close	5. volume
0	19.00	25.0000	17.5400	23.89	18766300.0
1	25.79	30.4192	23.3000	23.83	17187100.0
2	25.00	25.9200	20.2700	21.96	8218800.0
3	23.00	23.1000	18.7100	19.20	5139800.0
4	20.00	20.0000	15.8300	16.11	6866900.0
...	...	...	...	...	...
3480	168.85	172.1200	166.3700	168.29	109815725.0
3481	188.42	198.8700	184.5400	194.05	243869678.0
3482	186.98	190.9500	182.8401	183.28	127031787.0
3483	182.00	185.8600	179.0100	179.99	92829719.0
3484	182.86	184.6000	176.0200	180.01	89148041.0

3485 rows × 5 columns

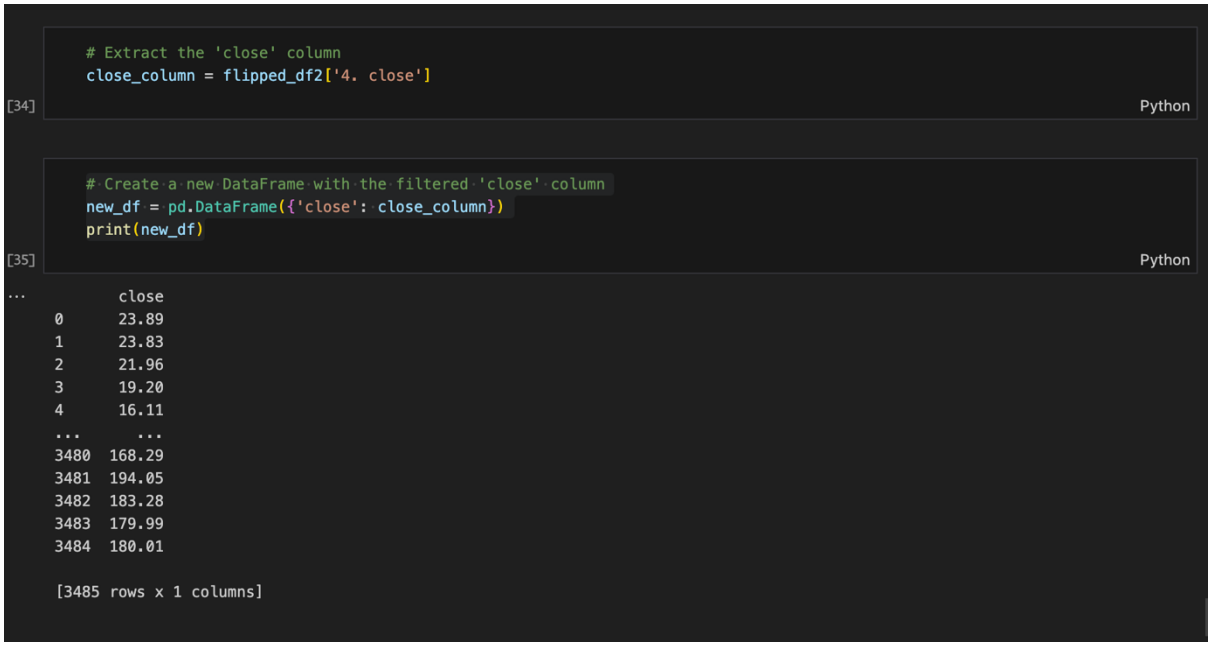
Step 20

This code snippet extracts the 'close' column from the Data Frame **flipped_df2** and creates a new Data Frame containing only this column.

1. **Extracting 'close' Column**:

- **flipped_df2['4. close']** extracts the '4. close' column from the DataFrame **flipped_df2**. This column typically contains the closing prices of the stock.

2. **Creating New DataFrame**:
    - **pd.DataFrame({'close': close_column})** creates a new DataFrame named **new_df** with a single column named 'close', which contains the data from the extracted 'close' column.
3. **Printing New DataFrame**:
    - **print(new_df)** prints the new DataFrame **new_df** to the console. This DataFrame will contain only the 'close' column, making it easier to work with for further analysis or visualization.



Final step

This code snippet performs the following tasks:

1. **Extracting Last 60 Days of Data**:
    - It selects the last 60 days of closing prices from the DataFrame **new_df** and stores them in the variable **last_60_day**.
2. **Scaling the Data**:
    - The selected 60 days of closing prices are then scaled using the **scaler** object.
3. **Preparing Data for Prediction**:
    - An empty list **x_test** is created to store the scaled data for prediction.

- The scaled data for the last 60 days is appended to the **x_test** list.
- The list is converted to a NumPy array and reshaped to match the input shape expected by the LSTM model.

4. **Making Predictions**:
    - The LSTM model (**model**) is used to predict the next day's closing price based on the last 60 days of data.
    - The predicted price is then inverse transformed using the **scaler** object to obtain the actual price.
5. **Printing Predicted Price**:
    - The predicted price for the next day is printed to the console.

Executing this code will provide you with the predicted closing price for the next day based on the LSTM model's analysis of the last 60 days of data.

4. Conclution

Based on the final predicted value and the closing prices of the previous three days (180.01, 179.99, 183.28), today's prediction stands at 206.35516. Comparing these values, we estimate the investment potential to be positive, indicating a 60% likelihood of a favorable return. However, it's essential to note that this prediction model's accuracy diminishes over longer time horizons. While it may provide insights for investing in Tesla over the next few days, its reliability diminishes significantly when forecasting beyond a short-term timeframe. Additionally, external factors such as geopolitical events or global economic conditions can influence stock prices unpredictably. Hence, while the model offers valuable insights, it should be supplemented with comprehensive market analysis and consultation with financial professionals before making investment decisions.