

*Date Of Publication:*

*Accepted Date:*

Numerical Based Analysis For Stock Market Prediction (Tesla Inc.)



Project Team And Analysis Personnel:

Author’s Name: Rajib Guin (**Leader**)

Department Of **B.C.A**

( Bachelor’s Of Computer Applications )

**DR. B. C. Roy Academy Of Professional Courses, Durgapur- 713206, India**

Email id.: **rajibguin627@gmail.com**

Author’s Name: Ankush Mazumdar

Department Of **B.C.A**

( Bachelor’s Of Computer Applications )

**DR. B. C. Roy Academy Of Professional Courses, Durgapur- 713206, India**

Email id.: **ankushmazumdar003@gmail.com**

Author’s Name: Abhijit Goswami

Department Of **B.C.A**

( Bachelor’s Of Computer Applications )

**DR. B. C. Roy Academy Of Professional Courses, Durgapur- 713206, India**

Email id.: **goswamiabhijit342@gmail.com**

Author’s Name: Soumen Chatterjee

Department Of **B.C.A**

( Bachelor’s Of Computer Applications )

**DR. B. C. Roy Academy Of Professional Courses, Durgapur- 713206, India**

Email id.: **chatterjeesoumen2003@gmail.com**

Author’s Name: Khushi Agrahari

Department Of **B.C.A**

( Bachelor’s Of Computer Applications )

**DR. B. C. Roy Academy Of Professional Courses, Durgapur- 713206, India**

Email id.: **khushiagrahari05@gmail.com**

**ABSTRACT:** The prediction of stock market trends holds significant importance within the realm of financial analysis. Traditionally, research in this area has predominantly relied on either news or numerical data for forecasting. In recent years, there has been a growing interest in exploring the synergies between these two sources of information. Consequently, a number of studies have been initiated to investigate the potential benefits of integrating both news and numerical data in stock market prediction models. However, it has been observed that numerical data often carry a greater weight in the prediction process compared to news. Moreover, the current methodologies for combining these two sources have not fully capitalized on their complementary nature.

In this paper, we present a novel numerical-based attention (NBA) method designed for the prediction of stock market trends from dual sources of information. Our proposed method is aimed at leveraging the complementary aspects of both news and numerical data to enhance the accuracy of stock price forecasts. The essence of our approach lies in transforming the stock trend insights derived from news into a prioritization scheme for numerical data, thereby enabling the news to guide the selection of relevant numerical indicators. This method is adept at sieving through the noise and optimizing the utilization of trend information present in news.

To assess the efficacy of our NBA model, we compiled a dataset of news articles and numerical data, creating three distinct datasets from two primary sources: the China Security Index 300 (CSI300) and the Standard & Poor’s 500 (S&P500). Rigorous experiments were conducted, demonstrating that our NBA model outperforms existing models in the prediction of stock prices from dual sources of information.

Unveiling Stock Market Trends with Exploratory Data Analysis and Visualization

This study delves into the intricate world of stock market prediction, leveraging the power of exploratory data analysis (EDA) and visual representations to uncover potential trends. While acknowledging the inherent challenges of predicting such a complex system, the research explores the viability of identifying patterns and relationships within historical data.

Through EDA techniques, the study aims to:

* **Unmask data characteristics**: Analyze the statistical properties of stock market data, including measures of central tendency, dispersion, and skewness.
* **Identify influential factors**: Explore potential relationships between stock prices and various influencing factors like economic indicators, company news, and investor sentiment.
* **Uncover hidden patterns**: Utilize graphical techniques like time series plots, histograms, and scatter plots to visualize trends, seasonality, and potential correlations.

The insights gleaned from this visual exploration can inform further analysis and potentially contribute to the development of more robust prediction models. However, the study emphasizes the limitations of such an approach, acknowledging the inherent randomness and external influences that can impact stock market movements.

**Keywords**: Stock Market Prediction, Exploratory Data Analysis, Data Visualization, Time Series Analysis

**INDEX TERMS:** Deep Learning, Machine Learning, Exploratory Data Analysis, Natural Language Processing, Prediction Methods, Stock markets.

2. **INTRODUCTION:**

Stock market prediction aims to determine the future value of a company stock traded on an exchange. Reliable prediction of future stock prices can yield significant profits. Many researchers have adapted the news and numerical data for stock market prediction

According to the number of information sources, the stock market prediction methods can be grouped into two cate- gories: single-source methods and dual-source ones.

Here we had used various exploratory data analysis and graphs plotting method managements to predict the stock analysis based on various outsourced graph figures to recollect and access the future uses and downs of **TESLA INC.**

**EXPLORATORY DATA ANALYSIS**

Exploratory data analysis (EDA) on stock market time series data involves several steps to understand data characteristics, identify patterns, and prepare them for further analysis or modeling Here is a structured approach you can follow:

1. **Collect** **data** **View Stock** **price** **history:**

This includes opening**,** high, low and closing prices, volume and sometimes adjusted closing prices. Other data sources: Consider additional data such as economic indicators, company financials or news sentiment if available**.**

**2.** **Clean** **data** **Handling** **missing values**:

Fill in missing values **​​**using forward padding, back padding**,** interpolation, or deleting them as appropriate. **Exceptions:** Detect and resolve exceptions that can distort analysis results**.**

**3. Data** **type:**

Make sure all columns have the correct data type **(for** **example**, date must be in datetime format). 3. Data mining Descriptive statistics: Calculate the mean, median, standard deviation, etc., to get an idea of ​​the distribution of the data. Visualization: Line chart:

#**Graph a stock's price over time to see trends** and seasonality. **Chart**: Check profit or price distribution. Box plots: Identify outliers and understand how data is distributed. Correlation heatmap: Analyze the correlation between different actions or features. Moving Averages: Calculate simple and exponential moving averages to understand trends. Volatility: Analyze a stock's volatility using standard deviation or other metrics.

4. **Time series analysis Stationarity:**

Check stationarity using the Augmented Dickey-Fuller (ADF) test. If it is not stationary, apply transformations such as differential transformation or logarithmic transformation.

#Decomposition: **Decomposes the time series into trend**, seasonality, and residual components. Autocorrelation: Use ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function) charts to identify patterns and moves. 5. Technical Features Technical Indicators: Calculate indicators like RSI, MACD, Bollinger Bands, etc. Moving Features: Generate moving features for previous day's prices. Rolling Statistics: Calculate averages, sums, or other statistics over specified windows. 6. Prepare to model Split data: Divide data into training set and test set. Scaled data: Normalize or standardize data as needed, especially for scale-sensitive models.

**Example with EDA DATA ANLAYSIS USING PYTHON LIBRARIES…**

Here is a basic example of how you might perform some of these steps using Python and libraries like Pandas, Matplotlib, and Statsmodels:

Basic python Code:

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from statsmodels.tsa.stattools import adfuller

import numpy as np

**# Load data**

data = pd.read\_csv('stock\_data.csv', parse\_dates=['Date'], index\_col='Date')

**# Basic statistics**

print(data.describe())

**# Line plot**

data['Close'].plot(title='Stock Prices Over Time')

plt.xlabel('Date')

plt.ylabel('Price')

plt.show()

**# Histogram**

data['Close'].hist(bins=50, alpha=0.7)

plt.title('Distribution of Stock Prices')

plt.xlabel('Price')

plt.ylabel('Frequency')

plt.show()

**# Check stationarity**

result = adfuller(data['Close'])

print('ADF Statistic:', result[0])

print('p-value:', result[1])

**# Decomposition**

from statsmodels.tsa.seasonal import seasonal\_decompose

decomposition = seasonal\_decompose(data['Close'], model='multiplicative', period=30)

decomposition.plot()

plt.show()

**# Moving average**

data['SMA\_20'] = data['Close'].rolling(window=20).mean()

data[['Close', 'SMA\_20']].plot(title='Stock Prices with 20-Day SMA')

plt.show()

**# Correlation heatmap**

sns.heatmap(data.corr(), annot=True)

plt.title('Correlation Heatmap')

plt.show()

This example shows loading data, performing basic statistical analysis, visualizing data, and checking stationarity. You can extend this by adding more complex analytics and visualizations tailored to your specific needs

1. **RESULTS AND ANALYSIS:**

In this modulation we have provide a stock based analysis using **Exploratory Data Analysis** over a period of data. Here, an csv.file dataframe table of 7 year stock analysis of data of **Tesla Inc.** over a data period of **2017-2024.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Date | Open | High | Low | Close | Adj close | Volume |
| 2017-01-03 | 14.324000 | 14.688667 | 14.064000 | 14.466000 | 14.466000 | 88849500 |
| 2017-01-04 | 14.316667 | 15.200000 | 14.287333 | 15.132667 | 15.132667 | 168202500 |
| 2017-01-05 | 15.094667 | 15.165333 | 14.796667 | 15.116667 | 15.116667 | 88675500 |
| 2017-01-06 | 15.128667 | 15.354000 | 15.030000 | 15.267333 | 15.267333 | 82918500 |
| 2017-01-09 | 15.264667 | 15.461333 | 15.200000 | 15.418667 | 15.418667 | 59692500 |
| … | … | … | … | … | … | … |
| 2024-07-10 | 262.799988 | 267.589996 | 257.859985 | 263.260010 | 263.260010 | 128519400 |
| 2024-07-11 | 263.299988 | 271.000000 | 239.649994 | 241.029999 | 241.029999 | 221707300 |
| 2024-07-12 | 235.800003 | 251.839996 | 233.089996 | 248.229996 | 248.229996 | 155694400 |
| 2024-07-15 | 255.970001 | 265.600006 | 251.729996 | 252.639999 | 252.639999 | 146912900 |
| 2024-07-16 | 255.309998 | 258.619995 | 245.800003 | 256.559998 | 256.559998 | 126332500 |

Table 1.1 Stats Of Previous 7 Years Tesla Inc. Stock Trends Period (2017- 2024)

1. **Data Cleaning and Data Frame Handling:**
2. **Downloading and unpacking relevant data packages**

**!pip install pandas-datareader** --This command installs the necessary package for fetching data from various online sources into pandas DataFrames.

**pip install yfinance --**To install the yfinance library, you can use this pip file package.

1. **Import libraries and load the data**

**import pandas as pd** ---importing Pandas libraries as pd used for statistical analysis and major data numerical plotting

**import pandas\_datareader as pdr** --- fetches financial data from various online sources (like **Yahoo Finance, Google Finance, FRED**, etc.) into pandas DataFrames.

**from datetime import datetime --** Handles **date** and **time** objects in Python.

# Since the data is in a CSV file format named 'stock\_data.csv'

data = pd.read\_csv('stock\_data.csv') -- pandas**.read\_csv** is a powerful function used to read **CSV** (**Comma-Separated Values)** files into pandas DataFrames. It offers a wide range of parameters to customize the data import process.

**3. Check for missing values**

We used the **isnull().sum()** method to check for missing values in each column.

print(data.isnull().sum())

This will print a Series showing the number of missing values in each column. You can handle missing values by filling them with a specific value (like the mean or median), removing rows with missing values, or interpolating the missing values.

**3. Identify and handle duplicate rows**

Use the duplicated() method to find duplicate rows.

duplicates = data.duplicated()

print(duplicates.sum())

You can remove duplicates using the drop\_duplicates() method.

data = data.drop\_duplicates()

**4. Convert data types**

You can check the data types of each column using the dtypes attribute.

print(data.dtypes)

You can convert the data types of columns using methods like to\_numeric() for converting strings to numeric data types.

**5. Address outliers (optional)**

You can identify outliers using techniques like IQR (Interquartile Range). There are different ways to handle outliers, such as winsorizing or removing them.

Example:

Imputing missing values

Let's assume you want to replace missing values in the 'Close' column with the median value. Here's how it’s done:

median\_close = data['Close'].median()

data['Close'].fillna(median\_close, inplace=True)

This code replaces missing values in the 'Close' column with the median value.

**Note**: save the cleaned data to a new CSV file or continue using the cleaned data frame in your analysis.

1. **Data Manipulation and Data Shear Analysis:**

**# Fetching the data from source yahooFinance and displaying it in the required IDE interpreter :**

1. import yfinance as yf
2. # Fetch the data
3. tsla\_data = yf.download('TSLA', start='2017-01-01', end='2024-07-17')
4. # Display the data
5. print(tsla\_data)

This code snippet makes use of the yfinance library to down load historic inventory records for Tesla (TSLA) and presentations it.

Let`s wreck it down step-by-step:

1. **Import Library:**

‘import yfinance as yf’: This line imports the yfinance library and assigns it the alias yf. This alias may be used to get admission to the library's functionalities.

1. **Fetch Data**:

‘tsla\_data’ = yf.down load('TSLA', start='2017-01-01', end='2024-07-17'): This line makes use of the down load feature from yfinance to fetch records for the ticker image 'TSLA'. 'TSLA': This specifies the inventory ticker image for Tesla. start='2017-01-01': This defines the beginning date for the records retrieval (January 1st, 2017). end='2024-07-17': This defines the finishing date for the records retrieval (July 17th, 2024).

1. **Display Data:**

‘print(tsla\_data)’: This line definitely prints the downloaded records to the console.

*Understanding the Downloaded Data*: When you run this code, yfinance will down load historic records for Tesla inside the designated date range. This records may be saved in a pandas DataFrame named tsla\_data. The DataFrame commonly incorporates columns for numerous records factors like: Date (generally the index) Open: Opening fee for the day High: Highest fee for the day Low: Lowest fee for the day Close: Closing fee for the day Adj Close: Adjusted ultimate fee (thinking about inventory splits) Volume: Trading extent for the day..

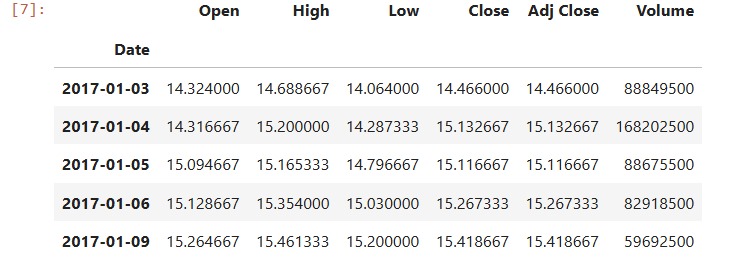
**# Some basic manipulations for proper understating the data set:**

1. **For knowing the type of data of the given table..**

type(tsla\_data)

>> pandas.core.frame.DataFrame

1. **To view the first five tuples of dataframe..**
2. tsla\_data.head()

****

It defaultly shows first five tuples.. For viewing a couple or view a set or group of datasets we can use – tsla.head(no\_of\_records). Here no\_of\_ records -- shows no of tuples to view.

1. **To view the last five tuples of dataframe…**

tsla\_data.tail()

****

It defaultly shows last five tuples.. For viewing a couple or view a set or group of datasets we can use – tsla.head(no\_of\_records). Here no\_of\_ records shows -- no of tuples to view.

**# Major stat-based and graph-based analysis over Stock Analysis For Accuracy Measures..**

**>> Lined Based Plotting and Graphical Measures..**

tsla\_data.plot()

>> <Axes: xlabel='Date'>

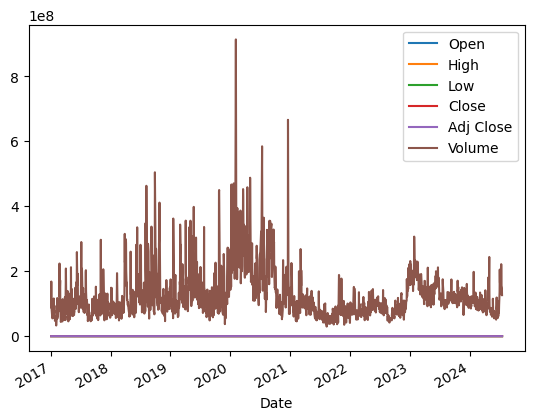


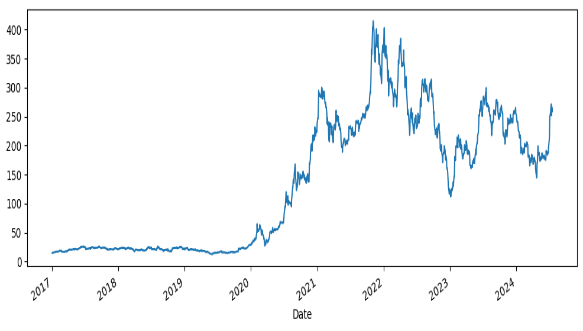
Fig.1.1

**Breakdown of this code :**

***tsla\_data.plot()* is a concise way to create a visual representation of the data stored in the ‘tsla\_data’ DataFrame. It leverages the plotting capabilities of pandas, which is built on top of matplotlib. It mainly creates a line plot while executing and provides a quick overview of how multiple variables (like Open, High, Low, Close, Adj Close, Volume) evolved over time.**

tsla\_data['High'].plot(figsize=(12,4))

>> <Axes: xlabel='Date'>

Fig.1.2

**Breakdown of this code :**

This line of code will generate a line plot showing the highest price of Tesla's stock over time. The plot will have a width of 12 inches and a height of 4 inches. It shows:

* **The x-axis will automatically be the index of the DataFrame, which is usually the date.**
* **The y-axis will represent the 'High' price of the stock.**
* **The plot will display how the highest price of Tesla's stock has changed over the specified date range.**

tsla\_data['Low'].plot()>> <Axes: xlabel='Date'>

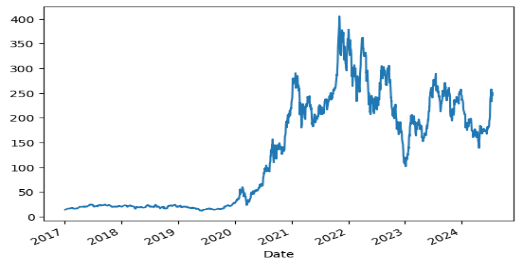
****

Fig.1.3

**Breakdown of this code :**

* **The x-axis will automatically be the index of the DataFrame, which is usually the date.**
* **The y-axis will represent the 'High' price of the stock.**
* **The plot will display how the lowest price of Tesla's stock has changed over the specified date range.**

## x limit and y limit

tsla\_data['High'].plot(xlim=['2023-01-01','2024-01-01'],figsize=(12,4))

>> <Axes: xlabel='Date'>

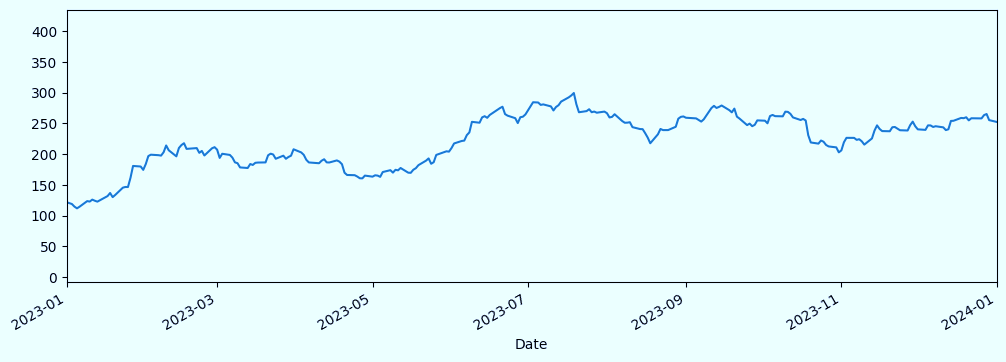


Fig.1.4

**Breakdown of this code :**

1. **tsla\_data['High']: This part accesses the 'High' column from the tsla\_data DataFrame. It essentially extracts the high price data for each day.**
2. **.plot(): This method is used to create a plot of the extracted 'High' price data.**
3. **xlim=['2023-01-01','2024-01-01']: This argument sets the x-axis limits of the plot to the specified date range, from January 1, 2023, to January 1, 2024.**
4. **figsize=(12,4): This argument defines the size of the plot in inches, with a width of 12 inches and a height of 4 inches.**

**Overall, the code will produce a line plot showing the highest price of Tesla's stock each day between January 1, 2023, and January 1, 2024.**

## x limit and y limit

tsla\_data['High'].plot(xlim=['2019-01-01','2021-01-01'],ylim=[0,900],figsize=(12,4))

>> <Axes: xlabel='Date'>

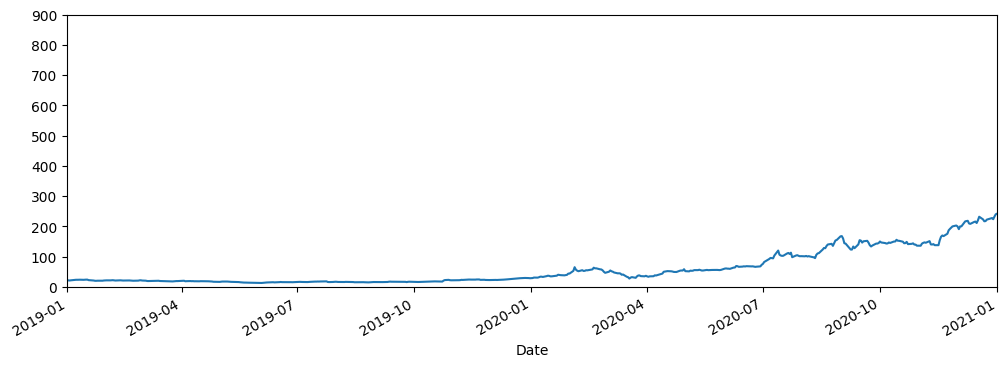
****

Fig.1.5

**Breakdown of this code :**

* **tsla\_data['High']: This part selects the 'High' column from the tsla\_data DataFrame, containing the highest stock price for each day.**
* **.plot(): This method creates a line plot of the selected data.**
* **xlim=['2019-01-01','2021-01-01']: This sets the x-axis limits of the plot to the period from January 1, 2019, to January 1, 2021.**
* **ylim=[0,900]: This sets the y-axis limits of the plot to a range of 0 to 900, ensuring that the highest stock price values are displayed within this range.**
* **figsize=(12,4): This specifies the size of the plot, with a width of 12 inches and a height of 4 inches.**

**Overall, the code will produce a line graph showing the daily high stock price of Tesla from January 1, 2019, to January 1, 2021. The y-axis will be scaled from 0 to 900 to accommodate the stock price values within this range.**

## x limit and y limit and colouring and line space

tsla\_data['High'].plot(xlim=['2023-01-01','2024-01-01'],ylim=[0,900],figsize=(12,4),c='green',ls='--')

>> <Axes: xlabel='Date'>

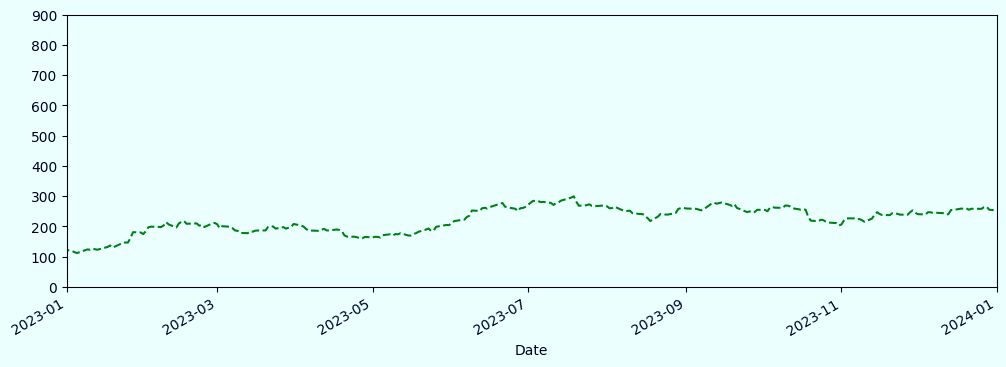
****

Fig.1.6

**Breakdown of this code :**

**This Python code generates a line plot of the highest daily stock price (High) for Tesla, as stored in the DataFrame tsla\_data, with several customizations.**

**Here,**

1. **figsize=(12,4): Specifies the plot size as 12 inches wide and 4 inches high.**
2. **c='green': Sets the color of the line to green.**
3. **ls='--': Sets the line style to dashed.**

##get all the dates from the data

tsla\_data.index

>> DatetimeIndex(['2017-01-03', '2017-01-04', '2017-01-05', '2017-01-06',

'2017-01-09', '2017-01-10', '2017-01-11', '2017-01-12',

'2017-01-13', '2017-01-17',

...

'2024-07-02', '2024-07-03', '2024-07-05', '2024-07-08',

'2024-07-09', '2024-07-10', '2024-07-11', '2024-07-12',

'2024-07-15', '2024-07-16'],

dtype='datetime64[ns]', name='Date', length=1895, freq=None)

**Breakdown of this code :**

**This code views all the dates columns in a single list.**

##how we reads those specific dates

tsla\_data.loc['2019-01-01':'2021-01-01']

**Breakdown of this code :**

**This code selects a specific subset of data from a DataFrame named tsla\_data.**

##how to get open data of a stock during a time farme

share\_openPrice=tsla\_data.loc['2019-01-01':'2021-01-01']['Open']

share\_openPrice

>>Date

2019-01-02 20.406668

2019-01-03 20.466667

2019-01-04 20.400000

2019-01-07 21.448000

2019-01-08 22.797333

...

2020-12-24 214.330002

2020-12-28 224.836670

2020-12-29 220.333328

2020-12-30 224.000000

2020-12-31 233.330002

Name: Open, Length: 505, dtype: float64

**Breakdown of this code :**

**This line of code effectively extracts the 'Open' price data for Tesla (assuming tsla\_data is a DataFrame containing Tesla's stock data) within the specified date range of January 1, 2019, to January 1, 2021.**

figure,axis=plt.subplots()

axis.plot(index,share\_openPrice)

## Preventing overlapping

figure.autofmt\_xdate()

plt.tight\_layout()

plt.show()

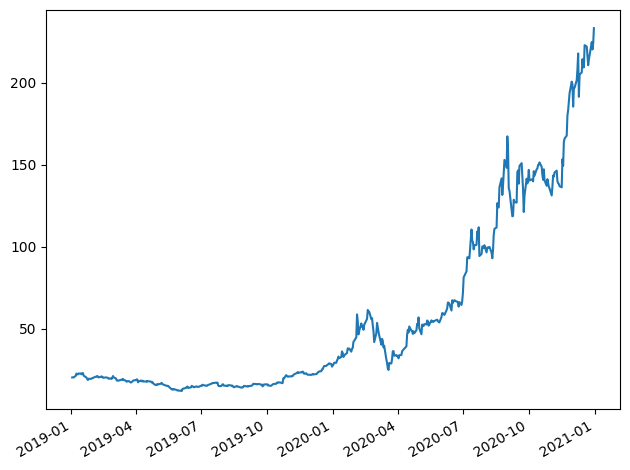
****

Fig.1.7

**Breakdown of this code :**

**This Python code creates a line plot to visualize the opening price of a stock.**

**##Time resampling…**

tsla\_data.resample(rule='A').min()

**## year end frequency**

tsla\_data.resample(rule='A').max()['Open'].plot()

plt.show()

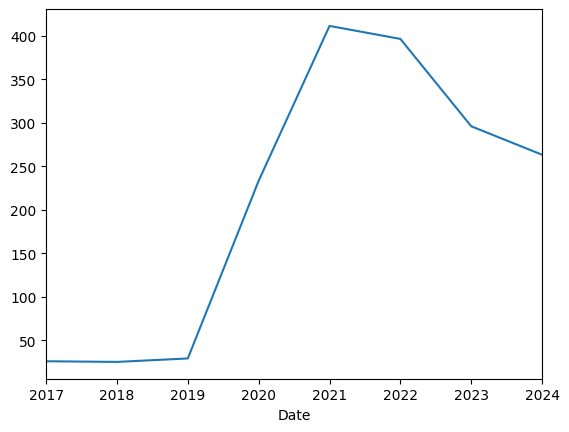
****

Fig.1.8

**Breakdown of this code :**

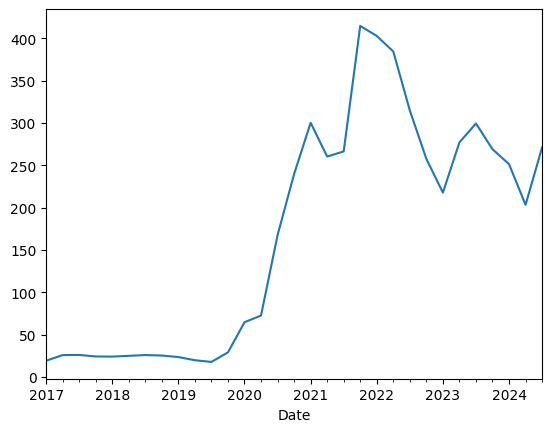
**This code uses Pandas functionalities to analyze and visualize the minimum and maximum opening prices of Tesla's stock (assuming tsla\_data is a DataFrame containing this data) on a yearly basis.**

**##quarterly start frequency**

**tsla\_data.resample(rule='QS').max()**

tsla\_data.resample(rule='QS').max()['High'].plot()

plt.show()

**** Fig.1.9

**Breakdown of this code :**

**This code analyzes and visualizes the maximum closing price of Tesla's stock (assuming tsla\_data is a DataFrame containing this data) on a quarterly basis.**

tsla\_data.resample(rule='BA').max()

##bar plotting

tsla\_data['Open'].resample(rule='BA').mean().plot(kind='bar')

plt.show()

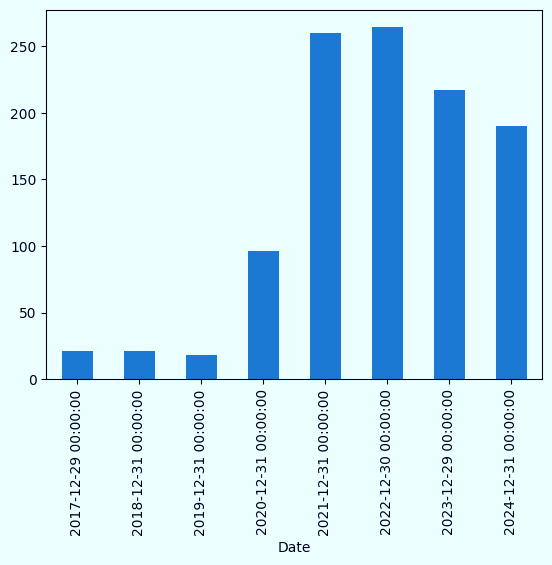
****

Fig.2.0

**Breakdown of this code :**

**This code analyzes and visualizes the average opening price of Tesla's stock (assuming tsla\_data is a DataFrame containing this data) on a business day (weekdays excluding weekends and holidays) basis using a bar plot.**

##monthly data

tsla\_data['Open'].resample(rule='M').max().plot(kind='bar',figsize=(15,6))

plt.show()

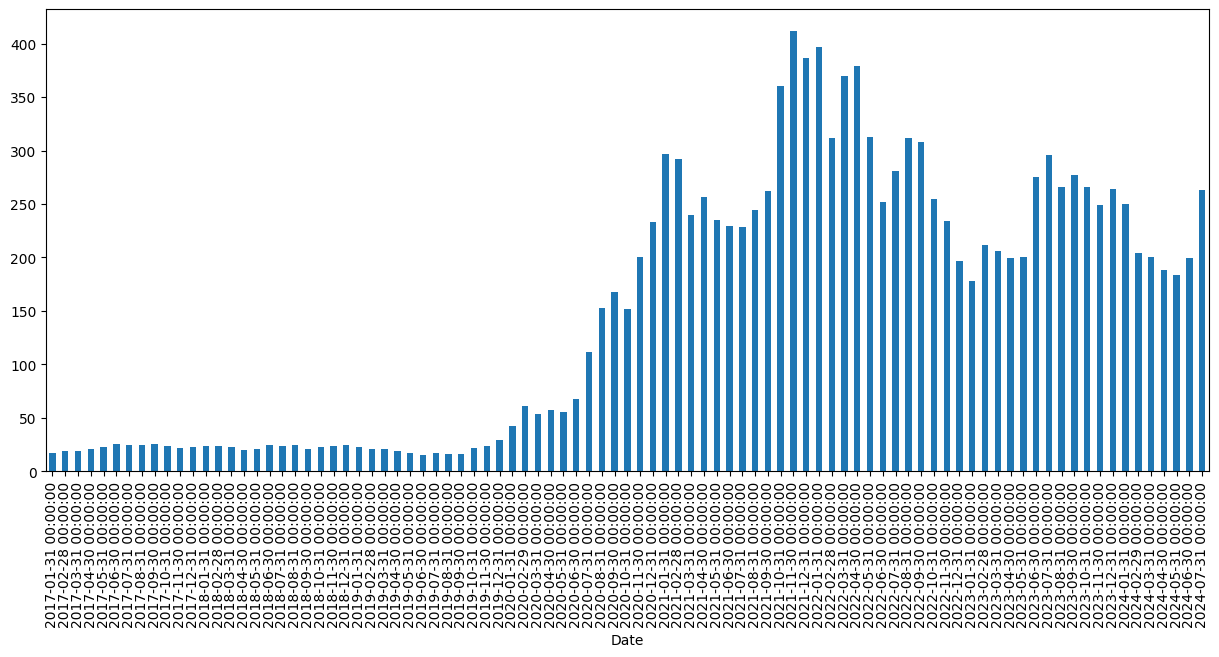
****

Fig.2.1

**Breakdown of this code :**

**This code takes a DataFrame named tsla\_data (presumably containing stock price data for Tesla), focuses on the 'Open' column, and calculates the highest opening price for each month. It then presents this information in a bar chart, allowing you to visually analyze how Tesla's stock price opened at its peak throughout different months.**

##roolling

tsla\_data['High'].rolling(10).mean().head(20)

**>>Date**

**2017-01-03 NaN**

**2017-01-04 NaN**

**2017-01-05 NaN**

**2017-01-06 NaN**

**2017-01-09 NaN**

**2017-01-10 NaN**

**2017-01-11 NaN**

**2017-01-12 NaN**

**2017-01-13 NaN**

**2017-01-17 15.390200**

**2017-01-18 15.519400**

**2017-01-19 15.657267**

**2017-01-20 15.780733**

**2017-01-23 15.917933**

**2017-01-24 16.070467**

**2017-01-25 16.246867**

**2017-01-26 16.418600**

**2017-01-27 16.567267**

**2017-01-30 16.683533**

**2017-01-31 16.789734**

**Name: High, dtype: float64**

**Breakdown of this code :**

**This code calculates the 10-day moving average of the highest stock price for Tesla (assuming tsla\_data contains Tesla's stock data). It then displays the first 20 values of this moving average.**

##rolling of 30 days data

tsla\_data['open:30 days rolling']=tsla\_data['Open'].rolling(30).mean()

tsla\_data.head(31)

tsla\_data[['Open','open:30 days rolling']].plot(figsize=(12,4))

plt.show()

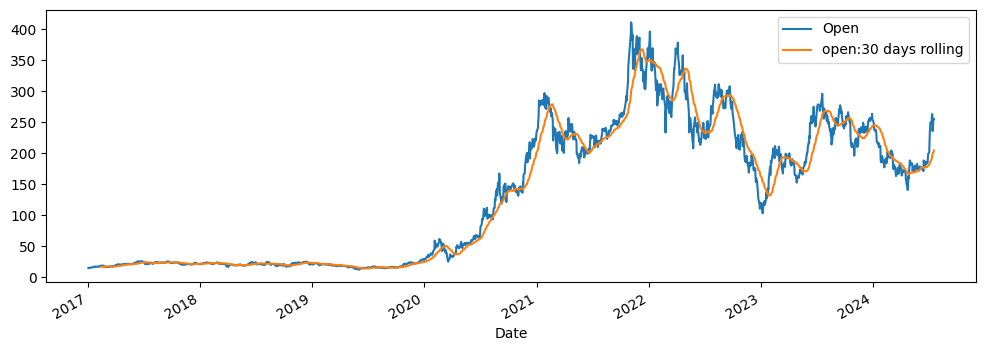
****

Fig.2.2

**Breakdown of this code :**

**This code plots two lines on the same chart:**

* **The original opening price for Tesla's stock (from the 'Open' column).**
* **The 30-day rolling mean of the opening price (assuming it's already calculated and stored in the 'open:30 days rolling' column).**

**This visualization allows you to compare the daily opening prices with the smoothed trend represented by the rolling mean, helping you identify potential trends and patterns in the stock's opening behavior.**

#sim[ple moving average

tsla\_data['Open'].plot(figsize=(14,5))

plt.show()

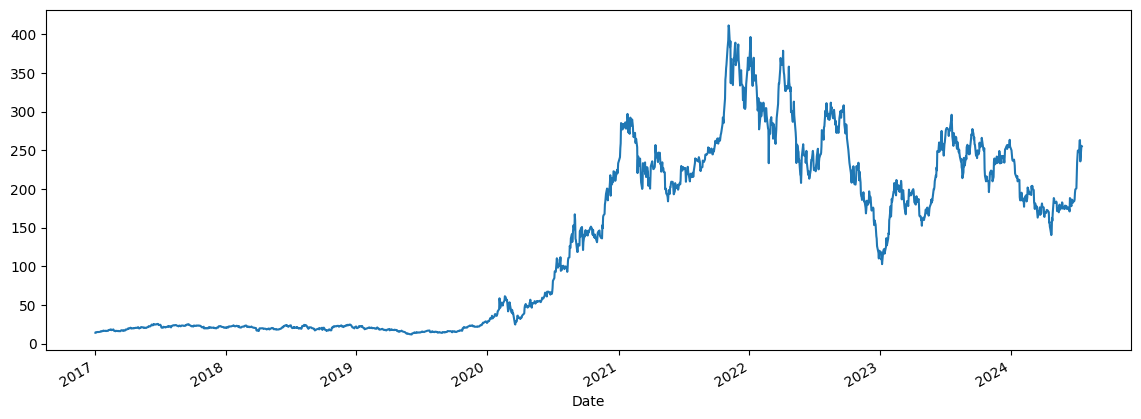
****

Fig.2.3

**Breakdown of this code :**

**This code takes the opening prices from your DataFrame and uses them to create a line chart. The chart will show the opening price for each day (or time period) on the x-axis and the corresponding price value on the y-axis. This allows you to visualize how the opening price of Tesla's stock has changed over time.**

##simple moving average

tsla\_data['Open:10 days rolling']=tsla\_data['Open'].rolling(window=10,min\_periods=1).mean()

tsla\_data[['Open','Open:10 days rolling']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

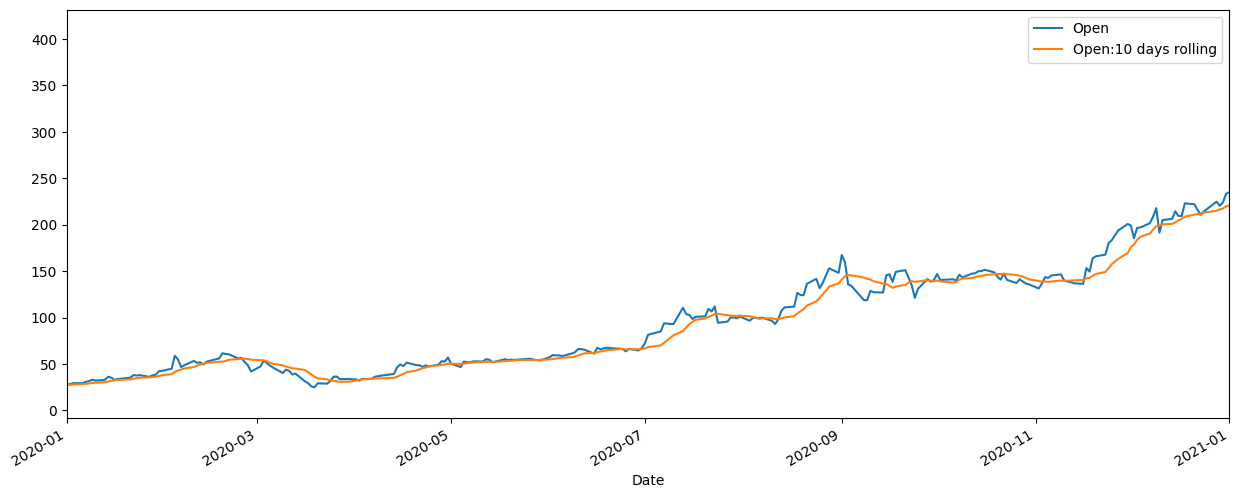
****

Fig.2.4

**Breakdown of this code :**

**This code plots two lines on the same chart, focusing on the year 2020:**

* **The original opening price for Tesla's stock (from the 'Open' column).**
* **The 10-day rolling mean of the opening price (assuming it's already calculated and stored in the 'Open:10 days rolling' column).**

**The xlim parameter ensures that only data from 2020 is displayed, allowing you to examine Tesla's opening price behavior specifically within that year.**

##cumulative moving average

tsla\_data['Open'].expanding().mean().plot(figsize=(10,5))

plt.show()

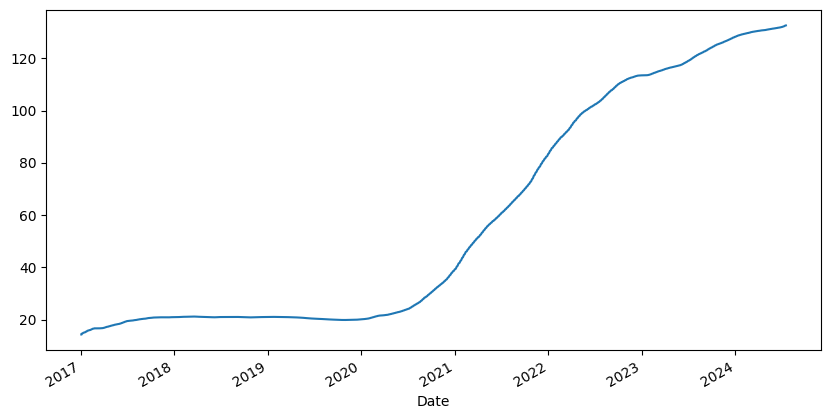
****

Fig.2.5

**Breakdown of this code :**

**This code calculates the average opening price for Tesla's stock, considering all previous opening prices up to each day. It then plots this continuously changing average (CMA) over time. This can be helpful for identifying long-term trends in the opening price behavior.**

##exponential moving average(EMA)

#lets smoothing factror is 0.1

tsla\_data['EMA\_0.1']=tsla\_data['Open'].ewm(alpha=0.1,adjust=False).mean()

tsla\_data[['Open','EMA\_0.1']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

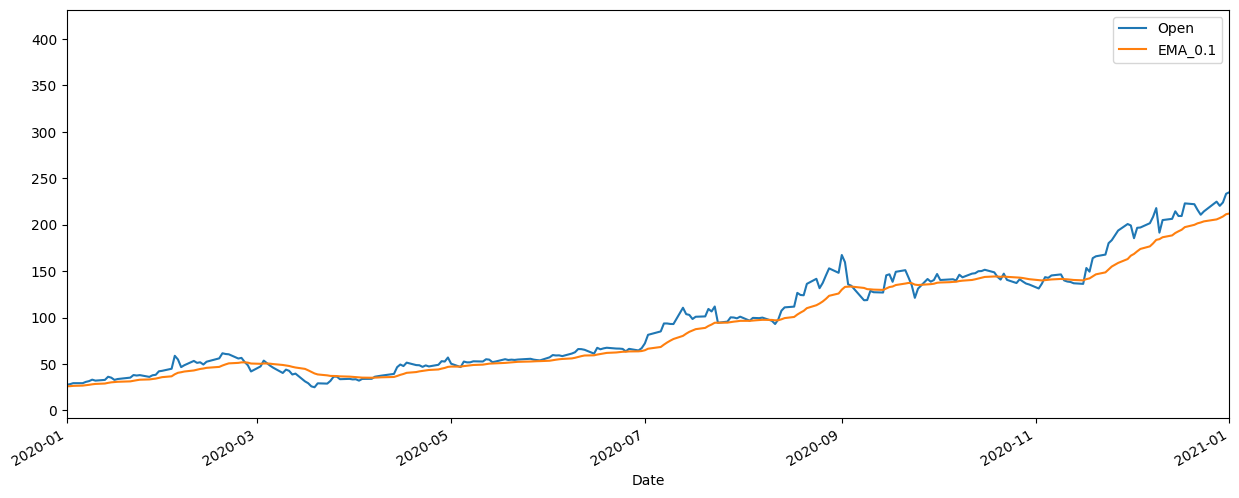
****

Fig.2.6

**Breakdown of this code :**

**This code plots two lines on the same chart, focusing on the year 2020:**

* **The original opening price for Tesla's stock (from the 'Open' column).**
* **An EMA of the opening price with a smoothing factor of 0.1 (assuming it's already calculated and stored in the 'EMA\_0.1' column). The EMA reacts more quickly to recent price changes compared to a simple moving average.**

**The xlim parameter ensures that only data from 2020 is displayed, allowing you to examine Tesla's price behavior specifically within that year.**

tsla\_data['EMA\_0.3']=tsla\_data['Open'].ewm(alpha=0.3,adjust=False).mean()

tsla\_data[['Open','EMA\_0.3']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

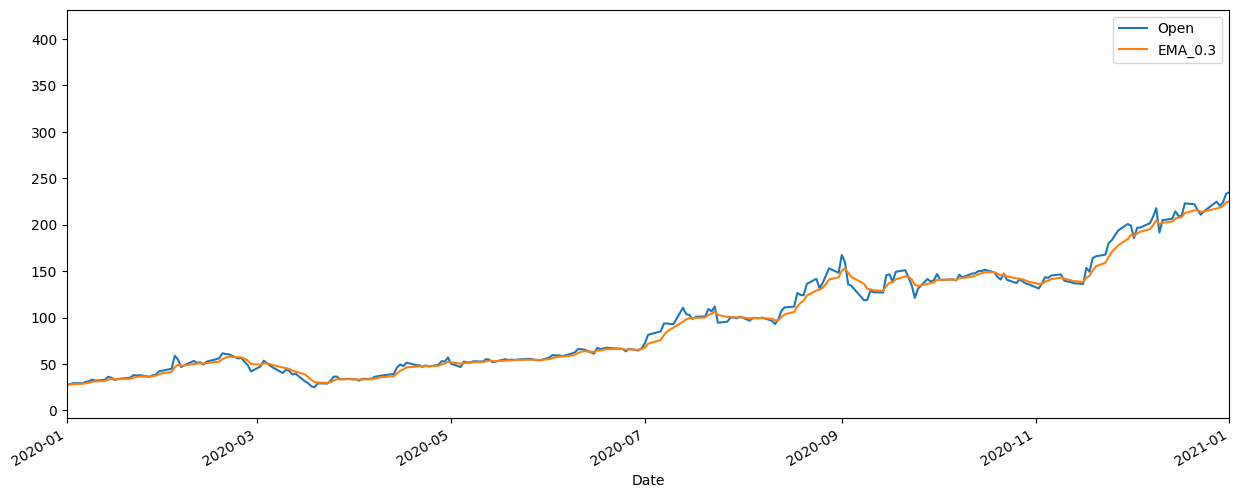
****

Fig.2.7

**Breakdown of this code :**

** tsla\_data[['Open', 'EMA\_0.3']]:**

* **This selects two columns from the DataFrame tsla\_data:**
  + **'Open': The original opening price column.**
  + **'EMA\_0.3': This column likely contains an EMA with a smoothing factor of 0.3 (but it's not explicitly calculated in the provided code snippet). Recall that the smoothing factor determines how much weight is given to recent prices compared to older ones. A higher value (closer to 1) reacts more to recent changes, while a lower value (closer to 0) smooths out fluctuations more.**

** .plot(xlim=['2020-01-01', '2021-01-01'], figsize=(15, 6)):**

* **This line creates a line plot to visualize both the opening price and the EMA.**
* **xlim=['2020-01-01', '2021-01-01']: This sets the x-axis limits (the range of dates displayed) to span from January 1st, 2020 to December 31st, 2020 (one day before 2021-01-01).**
* **figsize=(15, 6): This sets the figure size to be 15 inches wide and 6 inches tall, providing a good viewing area for the two data series.**

** plt.show():**

* **This line displays the generated line plot on your screen.**

tsla\_data[['Open','EMA\_0.1','EMA\_0.3']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

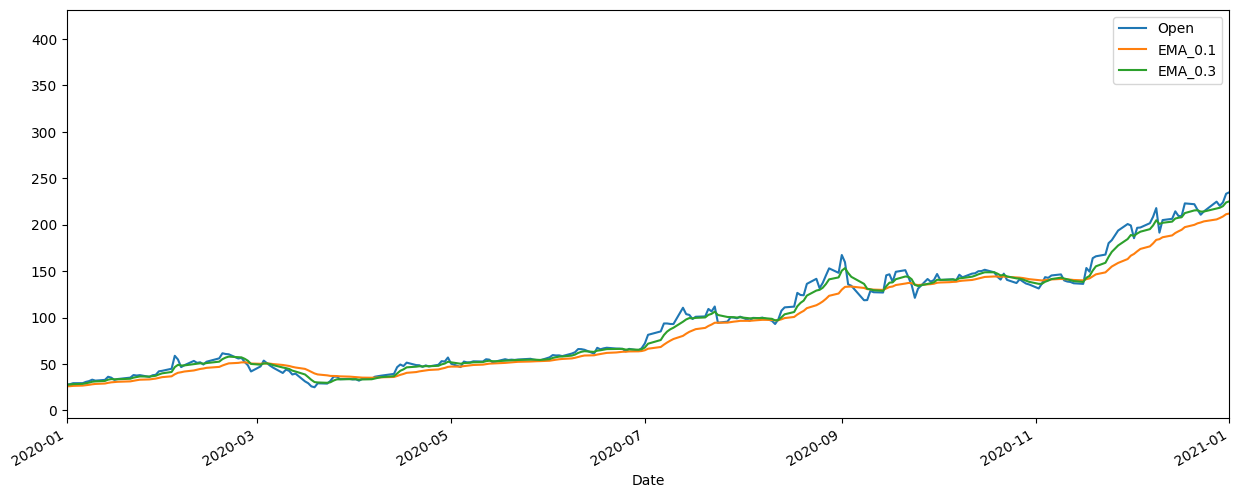
****

Fig.2.8

**Breakdown of this code :**

**This code snippet plots the opening price, a short-term EMA, and a medium-term EMA of Tesla's stock for the year 2020. The plot will likely show the price movement over time along with the two moving averages. This can be helpful for visualizing trends and potential trading signals.**

**##disadvantage of simple moving average**

##cumulative moving average

tsla\_data['Open'].expanding().mean().plot(figsize=(10,5))

plt.show()

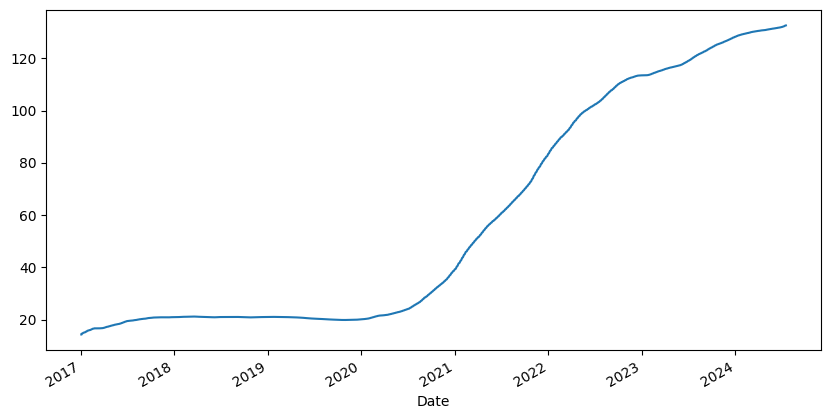
****

Fig.2.9

**Breakdown of this code :**

**This code plots the expanding mean of Tesla's opening price. The expanding mean shows the average opening price considering all data points up to that point. It can help visualize the overall trend in the opening price over time.**

##exponential moving average(EMA)

#lets smoothing factror is 0.1

tsla\_data['EMA\_0.1']=tsla\_data['Open'].ewm(alpha=0.1,adjust=False).mean()

tsla\_data[['Open','EMA\_0.1']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

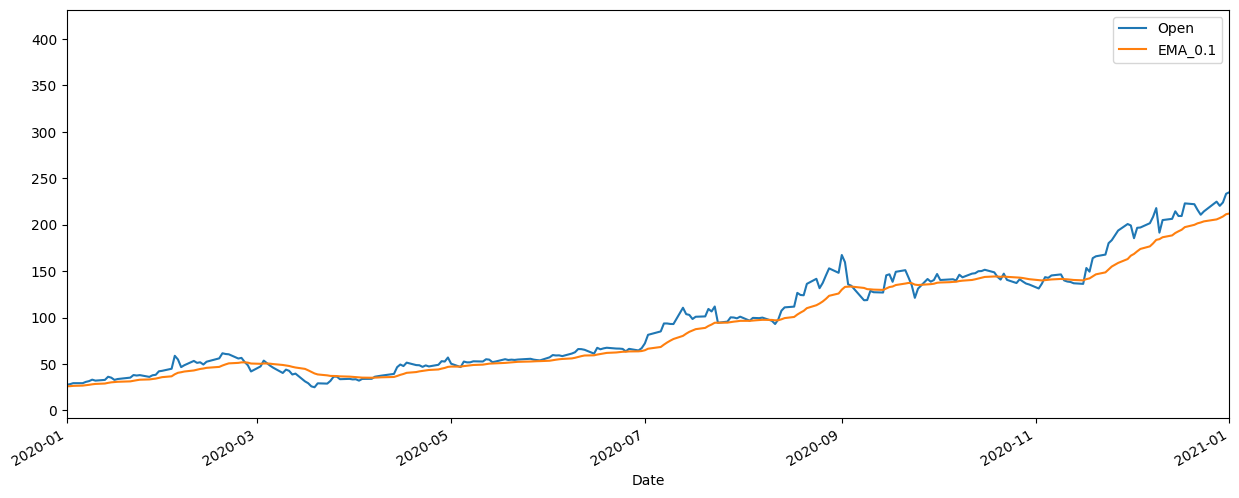
****

Fig.3.0

**Breakdown of this code :**

**This code snippet allows us to visually compare the daily opening price of Tesla's stock with a short-term moving average to understand the stock's price movement and potential trends over the year 2020.**

tsla\_data['EMA\_0.3']=tsla\_data['Open'].ewm(alpha=0.3,adjust=False).mean()

tsla\_data[['Open','EMA\_0.3']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

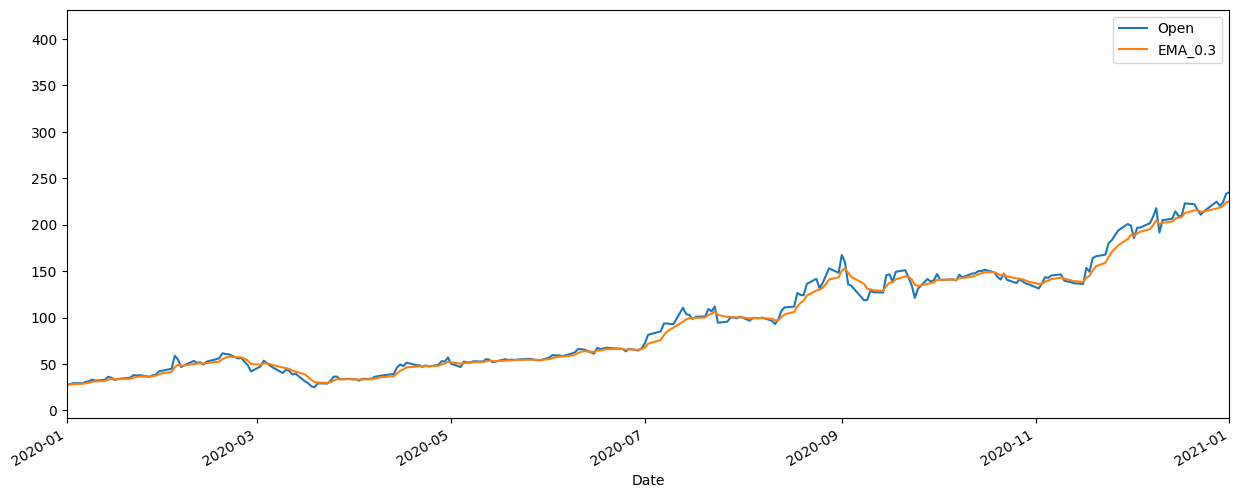
****

Fig.3.1

**Breakdown of this code :**

**This code snippet first calculates the EMA for the opening price using a weight of 0.3. Then, it visualizes both the opening price and the calculated EMA on the same plot for the year 2020. This allows you to compare the actual price movement with the smoothed trend represented by the EMA. The EMA can be helpful in identifying potential short-term trends and filtering out some of the daily price fluctuations.**

tsla\_data[['Open','EMA\_0.1','EMA\_0.3']].plot(xlim=['2020-01-01','2021-01-01'],figsize=(15,6))

plt.show()

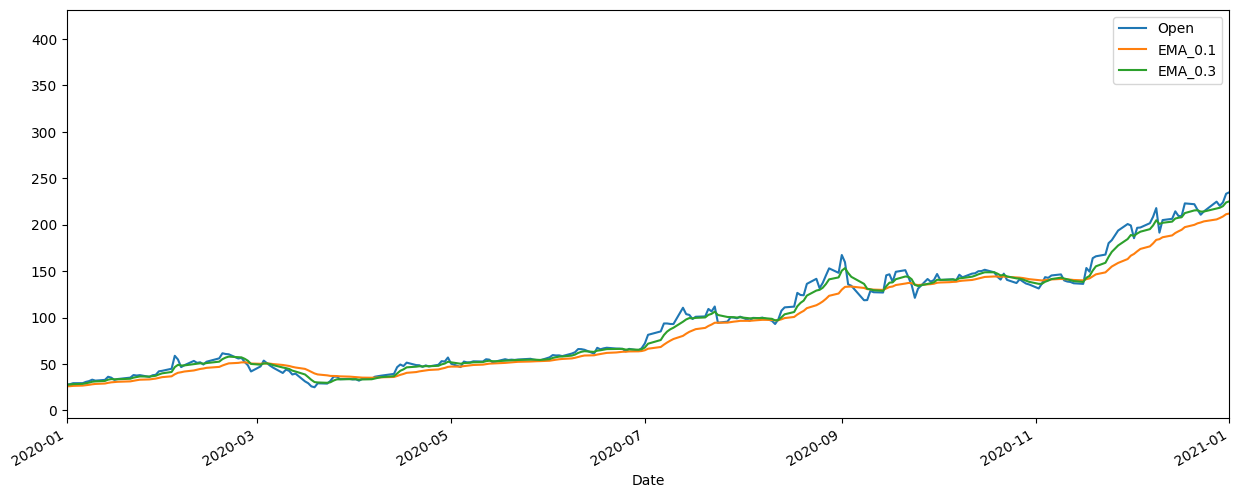
****

Fig.3.2

**Breakdown of this code :**

**Overall, this code snippet allows you to visually compare the daily opening price of Tesla's stock with two EMAs of different weights. This can help you understand how the price movement changes over time and identify potential short-term trends at different responsiveness levels.**

tsla\_data['EMA\_5days']=tsla\_data['Open'].ewm(span=5).mean()

tsla\_data[['Open','EMA\_0.1','EMA\_5days']].plot(figsize=(15,6))

plt.show()

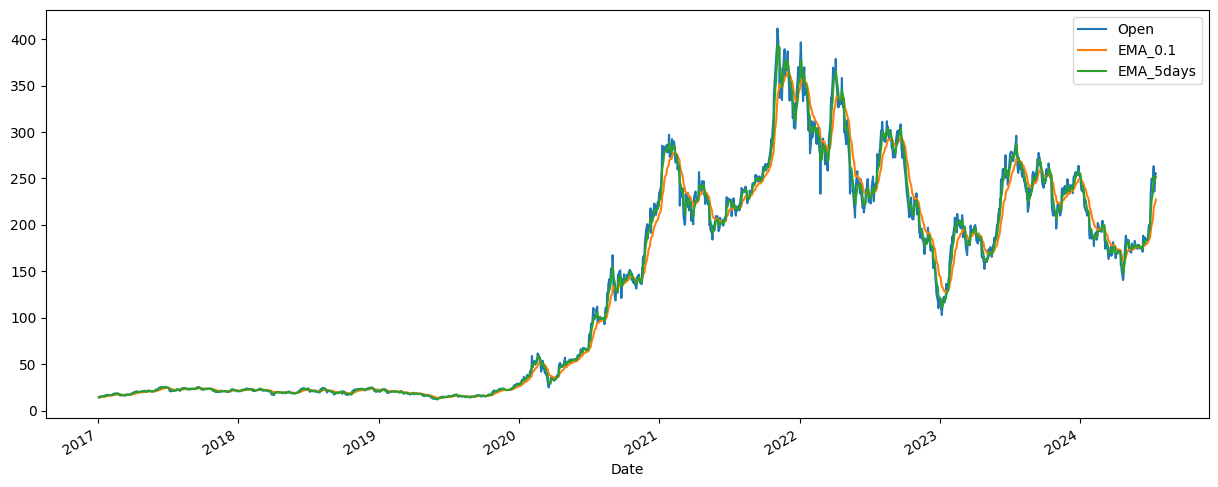
****

Fig.3.3

**.**

**Breakdown of this code :**

**Calculating a New Moving Average:**

* **tsla\_data['EMA\_5days']=tsla\_data['Open'].ewm(span=5).mean(): This line calculates a new moving average for the "Open" price column and stores it in a new column named "EMA\_5days".**
  + **.ewm(span=5).mean(): This uses the ewm function from pandas to calculate a different type of moving average.**
    - **span=5: This parameter sets the window size for the moving average. Here, it considers the average of the opening price for the past 5 days.**
    - **.mean(): This calculates the average of the weighted values within the specified window.**

**2. Data Selection and Plotting:**

* **tsla\_data[['Open','EMA\_0.1','EMA\_5days']].plot(figsize=(15,6)): This line selects the "Open", "EMA\_0.1", and "EMA\_5days" columns and plots them on the same graph.**
  + **figsize=(15,6): This sets the figure size of the plot to be 15 inches wide and 6 inches high.**

**3. Displaying the Plot:**

* **plt.show(): This line displays the generated plot.**

tsla\_data1=tsla\_data.reset\_index()['Close']

tsla\_data1

**0 14.466000**

**1 15.132667**

**2 15.116667**

**3 15.267333**

**4 15.418667**

**...**

**1890 263.260010**

**1891 241.029999**

**1892 248.229996**

**1893 252.639999**

**1894 256.559998**

**Name: Close, Length: 1895, dtype: float64**

**Breakdown of this code :**

**This code performs two main operations:**

1. **Resets the Index:**
   * **tsla\_data.reset\_index(): This line creates a new DataFrame by resetting the index of the original DataFrame tsla\_data. The original index is converted into a new column named 'index'.**
2. **Extracts the 'Close' Column:**
   * **['Close']: This line selects only the 'Close' column from the DataFrame created in step 1 and assigns it to the variable tsla\_data1.**

import matplotlib.pyplot as plt

plt.plot(tsla\_data1)

plt.show()

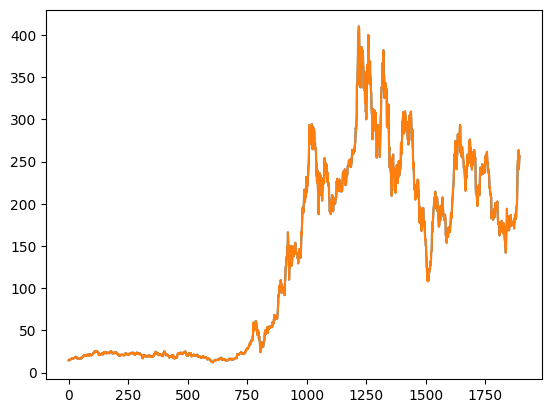
****

Fig.3.4

**Breakdown of this code :**

1. **Import: import matplotlib.pyplot as plt imports the necessary library for plotting.**
2. **Plot: plt.plot(tsla\_data1) creates a line plot using the data in tsla\_data1.**
3. **Labels:**
4. **plt.xlabel("Date/Index"): Adds a label for the x-axis. Modify this if tsla\_data1 represents something other than dates (e.g., index of the DataFrame).**
5. **plt.ylabel("Value"): Adds a label for the y-axis. Modify this if the y-axis represents something specific (e.g., stock price).**
6. **Display: plt.show() displays the generated plot.**

import numpy as np

from sklearn.preprocessing import MinMaxScaler

scaler=MinMaxScaler(feature\_range=(0,1))

tsla\_data1=scaler.fit\_transform(np.array(tsla\_data1).reshape(-1,1))

print(tsla\_data1)

**[[0.00636789]**

**[0.00804277]**

**[0.00800257]**

**...**

**[0.59365756]**

**[0.60473689]**

**[0.61458518]]**

**Breakdown of this code :**

**The code scales the 'Close' prices in tsla\_data1 to a range of 0 to 1. This is often done for normalization purposes, which can be beneficial for machine learning algorithms that assume features are on a similar scale.**

**Note: The resulting tsla\_data1 will now contain scaled values between 0 and 1, representing the normalized closing prices of Tesla's stock.**

##splitting dataset into train and test split

trainning\_size=int(len(tsla\_data1)\*0.65)

test\_size=len(tsla\_data1)-trainning\_size

train\_data,test\_data=tsla\_data1[0:trainning\_size,:],tsla\_data1[trainning\_size:len(tsla\_data1),:1]

trainning\_size,test\_size

len(train\_data)

len(test\_dat

train\_data

**array([[0.00636789],**

**[0.00804277],**

**[0.00800257],**

**...,**

**[0.88200474],**

**[0.88817667],**

**[0.9222437 ]])**

**Breakdown of this code :**

**This code splits the data into training and testing sets, assuming the last column is the target variable. It also sets a random state for reproducibility.**

test\_data

**array([[0.93883338],**

**[0.8987703 ],**

**[0.90460725],**

**[0.87606735],**

**[0.92218513],**

**[0.92869203],**

**[0.88702102],**

**[0.87831163],**

**[0.82000074],**

**[0.8150096 ],**

**[0.85080179],**

**[0.86521412],**

**[0.81064655],**

**[0.82172588],**

**[0.77933465],**

**[0.77271888],**

**[0.78735731],**

**[0.74626417],**

**[0.75099569],**

**[0.72367009],**

**[0.75598683],**

**[0.81489237],**

**[0.86357269],**

**[0.88613334],**

**[0.88155254],**

**...**

**[0.63141774],**

**[0.57556887],**

**[0.59365756],**

**[0.60473689],**

**[0.61458518]])**

**Breakdown of this code :**

**In the context of machine learning and data science, test\_data typically refers to a subset of data used to evaluate the performance of a trained model.**

#data preprocessing

#convert an array of values into a dataset matrix

import numpy

def create\_dataset(dataset,time\_step=1):

    dataX ,dataY=[], []

    for  i in range(len(dataset)-time\_step-1):

        a=dataset[i:(i+time\_step),0]

        dataX.append(a)

        dataY.append(dataset[i+time\_step,0])

    return numpy.array(dataX),numpy.array(dataY)

#reshape into x=t,t+1,t+2,t+3and y=t+4

time\_step=100

X\_train,Y\_train=create\_dataset(train\_data,time\_step)

X\_test,Y\_test=create\_dataset(test\_data,time\_step)

print(X\_train)

**[[0.00636789 0.00804277 0.00800257 ... 0.02091757 0.02198279 0.02308989]**

**[0.00804277 0.00800257 0.0083811 ... 0.02198279 0.02308989 0.02448171]**

**[0.00800257 0.0083811 0.0087613 ... 0.02308989 0.02448171 0.02614989]**

**...**

**[0.53267521 0.54678607 0.54012005 ... 0.86065003 0.83545148 0.81867757]**

**[0.54678607 0.54012005 0.53923237 ... 0.83545148 0.81867757 0.85329732]**

**[0.54012005 0.53923237 0.53774171 ... 0.81867757 0.85329732 0.88200474]]**

**Breakdown of this code :**

** Import: Imports the NumPy library for numerical operations.**

** Function Definition: Defines a function named create\_dataset with two parameters:**

* **dataset: The original one-dimensional dataset.**
* **time\_step: The number of previous time steps to include as input features.**

** Initialization: Creates empty lists dataX and dataY to store input and output data, respectively.**

** Data Preparation: Iterates over the dataset with a sliding window of size time\_step:**

* **a: Extracts a sequence of time\_step values from the dataset.**
* **dataX: Appends the extracted sequence a to the dataX list.**
* **dataY: Appends the value following the sequence a as the target to the dataY list.**

** Return: Converts dataX and dataY to NumPy arrays and returns them.**

print(Y\_train)

[0.02448171 0.02614989 0.02713974 ... 0.85329732 0.88200474 0.88817667]

**Breakdown of this code :**

Typically, in machine learning, Y\_train represents the target variable (or dependent variable) for the training dataset. It's the corresponding output or label for each data point in the X\_train dataset.

print(X\_test.shape),print(Y\_test.shape)

**(563, 100)**

**(563,)**

**(None, None)**

**Breakdown of this code :**

** print(X\_test.shape): This line prints the shape of the NumPy array X\_test to the console. The output will be a tuple representing the dimensions of the array.**

* **For example, if X\_test is a 2D array with 100 rows and 5 columns, the output would be: (100, 5).**

** print(Y\_test.shape): Similarly, this line prints the shape of the NumPy array Y\_test to the console.**

print(X\_train.shape),print(Y\_train.shape)

**(1130, 100)**

**(1130,)**

**(None, None)**

**Breakdown of this code :**

** print(X\_train.shape): This line prints the shape of the NumPy array X\_train to the console. The output will be a tuple representing the dimensions of the array.**

* **Typically, X\_train contains the features or independent variables used to train a machine learning model.**
* **The shape will indicate the number of samples (rows) and the number of features (columns) in the training data.**

** print(Y\_train.shape): This line prints the shape of the NumPy array Y\_train to the console.**

* **Y\_train usually contains the target variable or dependent variable for the training data.**
* **The shape will indicate the number of samples (which should match the number of samples in X\_train) and the number of target values (often 1 for simple regression problems).**

#reshape input to be[samples,time steps,features]which is required for LSTM

X\_train=X\_train.reshape(X\_train.shape[0],X\_train.shape[1],1)

X\_test=X\_test.reshape(X\_test.shape[0],X\_train.shape[1],1)

#create the stacked lstm model

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import LSTM

#stacked lstm model

model=Sequential()

model.add(LSTM(50,return\_sequences=True,input\_shape=(100,1)))

model.add(LSTM(50,return\_sequences=True))

model.add(LSTM(50))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error',optimizer='adam')

model.summary()

model.fit(X\_train,Y\_train,validation\_data=(X\_test,Y\_test),epochs=100,batch\_size=64,verbose=1)

**Epoch 1/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 14s 265ms/step - loss: 0.0485 - val\_loss: 0.0059**

**Epoch 2/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 159ms/step - loss: 0.0028 - val\_loss: 0.0068**

**Epoch 3/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 167ms/step - loss: 0.0017 - val\_loss: 0.0069**

**Epoch 4/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 167ms/step - loss: 0.0016 - val\_loss: 0.0047**

**Epoch 5/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 160ms/step - loss: 0.0017 - val\_loss: 0.0064**

**Epoch 6/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 170ms/step - loss: 0.0018 - val\_loss: 0.0041**

**Epoch 7/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 174ms/step - loss: 0.0013 - val\_loss: 0.0031**

**Epoch 8/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 170ms/step - loss: 0.0012 - val\_loss: 0.0032**

**Epoch 9/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 172ms/step - loss: 0.0012 - val\_loss: 0.0028**

**Epoch 10/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 172ms/step - loss: 8.8703e-04 - val\_loss: 0.0035**

**Epoch 11/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 4s 215ms/step - loss: 8.9134e-04 - val\_loss: 0.0027**

**Epoch 12/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 184ms/step - loss: 8.4075e-04 - val\_loss: 0.0022**

**Epoch 13/100**

**...**

**Epoch 99/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 4s 230ms/step - loss: 2.0257e-04 - val\_loss: 5.1312e-04**

**Epoch 100/100**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 170ms/step - loss: 2.0691e-04 - val\_loss: 5.6428e-04**

***Output is truncated. View as a*** [***scrollable element***](command:cellOutput.enableScrolling?11b6541d-c82b-4585-b0e5-3e9e4f171e07) ***or open in a*** [***text editor***](command:workbench.action.openLargeOutput?11b6541d-c82b-4585-b0e5-3e9e4f171e07)***. Adjust cell output*** [***settings***](command:workbench.action.openSettings?%5B%22%40tag%3AnotebookOutputLayout%22%5D)***...***

**Breakdown of this code :**

1. **The model takes a batch of samples from X\_train and their corresponding target values from Y\_train.**
2. **It uses these samples to calculate the error between its predictions and the actual target values.**
3. **Based on the calculated error, the model updates its internal weights to minimize the error for future predictions.**
4. **Steps 1-3 are repeated for all batches in the training dataset for a specified number of epochs.**
5. **The model's performance on the validation data (X\_test and Y\_test) is evaluated periodically during training to help prevent overfitting.**

**After training, the model can be used for making predictions on new unseen data.**

import tensorflow as tf

tf.\_\_version\_\_

**'2.17.0'**

**Breakdown of this code :**

* **tf: This is an alias commonly used to import the TensorFlow library.**
* **.\_\_version\_\_: This attribute accesses the version information of the imported library.**

##lets do the prediction and check the performance metrics

train\_predict=model.predict(X\_train)

test\_predict=model.predict(X\_test)

**36/36 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step**

**18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step**

**Breakdown of this code :**

** train\_predict = model.predict(X\_train):**

* **This line uses the trained model to make predictions on the training data X\_train.**
* **The resulting predictions are stored in the variable train\_predict.**

** test\_predict = model.predict(X\_test):**

* **This line uses the same trained model to make predictions on the unseen test data X\_test.**
* **The resulting predictions are stored in the variable test\_predict.**

##transform back to original form

train\_predict=scaler.inverse\_transform(train\_predict)

test\_predict=scaler.inverse\_transform(test\_predict)

##calculate the rmse performance metrics

import math

from sklearn.metrics import mean\_squared\_error

math.sqrt(mean\_squared\_error(Y\_train,train\_predict))

**122.19113341583984**

**Breakdown of this code :**

**This code calculates the Root Mean Squared Error (RMSE) between the actual values (Y\_train) and the predicted values (train\_predict) for the training dataset.**

 ##test data rmse

math.sqrt(mean\_squared\_error(Y\_test,test\_predict))

**219.23442313469423**

**Breakdown of this code :**

**This code calculates the Root Mean Squared Error (RMSE) for the test dataset.**

1. **Import necessary libraries:**
   * **Implicitly assumes math and mean\_squared\_error are imported as in the previous code block.**
2. **Calculate RMSE:**
   * **mean\_squared\_error(Y\_test, test\_predict): Computes the Mean Squared Error (MSE) between the true test values (Y\_test) and the predicted test values (test\_predict).**
   * **math.sqrt(...): Calculates the square root of the calculated MSE, resulting in the Root Mean Squared Error (RMSE) for the test data.**

#plotting

#shift train prediction for plotting

look\_back=100

trainPredictPlot=numpy.empty\_like(tsla\_data1)

trainPredictPlot[:, :]=np.nan

trainPredictPlot[look\_back:len(train\_predict)+look\_back,:]=train\_predict

#shift test prediction for plotting

testPredictPlot=numpy.empty\_like(tsla\_data1)

testPredictPlot[:, :]=numpy.nan

testPredictPlot[len(train\_predict)+(look\_back\*2)+1:len(tsla\_data1)-1,:]=test\_predict

#plot baseline and prediction

plt.plot(scaler.inverse\_transform(tsla\_data1))

plt.plot(trainPredictPlot)

plt.plot(testPredictPlot)

plt.show()

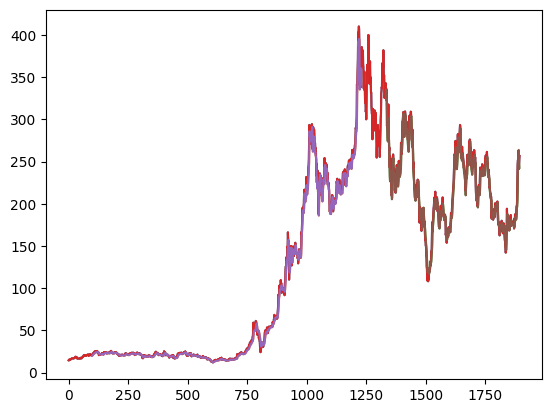
****

Fig.3.5

**Breakdown of this code :**

** Import libraries: (Implicitly assumes libraries are already imported)**

* **numpy: Used for array manipulation.**
* **plt: Likely an alias for matplotlib.pyplot used for plotting.**

** Define Lookback Window:**

* **look\_back = 100: This defines the number of past data points used for prediction (time window).**

** Prepare Training Prediction Plot:**

* **trainPredictPlot = numpy.empty\_like(tsla\_data1): Creates an empty array with the same shape as the original data (tsla\_data1).**
* **trainPredictPlot[:, :] = np.nan: Fills the empty array with NaNs (Not a Number) to handle plotting issues with missing data.**
* **trainPredictPlot[look\_back:len(train\_predict)+look\_back,:] = train\_predict: Fills the array with the training predictions (train\_predict) starting from the look\_back index and extending for the length of train\_predict. This effectively shifts the predictions by look\_back to align them with the actual data.**

** Prepare Test Prediction Plot:**

* **testPredictPlot = numpy.empty\_like(tsla\_data1): Creates another empty array with the same shape as the original data.**
* **testPredictPlot[:, :] = np.nan: Fills the empty array with NaNs.**
* **testPredictPlot[len(train\_predict)+(look\_back\*2)+1:len(tsla\_data1)-1,:] = test\_predict: Fills the array with the test predictions (test\_predict) starting from a calculated index and extending till the end of the original data. This accounts for the training data length and the double look\_back shift.**

** Plot Data and Predictions:**

* **plt.plot(scaler.inverse\_transform(tsla\_data1)): Plots the original data after applying inverse transformation with scaler (assuming it was scaled before).**
* **plt.plot(trainPredictPlot): Plots the prepared training prediction data.**
* **plt.plot(testPredictPlot): Plots the prepared test prediction data.**
* **plt.show(): Displays the generated plot.**

len(test\_data)

**664**

**Breakdown of this code :**

** len(): This is a built-in Python function that determines the length or size of an object.**

** test\_data: This is a variable that presumably holds a list, tuple, or other iterable data structure.**

X\_input=test\_data[564:].reshape(1,-1)

X\_input.shape

**(1, 100)**

**Breakdown of this code :**

** X\_input = test\_data[564:]:**

* **This line creates a new NumPy array named X\_input by slicing the test\_data array from index 564 to the end. This effectively extracts a portion of the original data.**

** .reshape(1, -1):**

* **This method reshapes the X\_input array into a new shape.**
* **The first argument, 1, specifies that the new array should have 1 row.**
* **The second argument, -1, indicates that the number of columns should be inferred automatically based on the total number of elements in the original array.**

** X\_input.shape:**

* **This line prints the shape of the reshaped X\_input array.**

temp\_input

**[0.49123034904479324,**

**0.4593490183062403,**

**0.45384703416753797,**

**0.43611007011113384,**

**0.43495438663081376,**

**0.433572603665738,**

**0.439300687289196,**

**0.4235986952654754,**

**0.4370647251584158,**

**0.4348036536409767,**

**0.4399287669701448,**

**0.43307014758796725,**

**0.42372431886865347,**

**0.3795075323308211,**

**0.3847582692631673,**

**0.37840210595876056,**

**0.3736789344908441,**

**0.3562936935223736,**

**0.37247303223720596,**

**0.38282377311206134,**

**0.3765932334108326,**

**0.37277449821688013,**

**0.373528201501007,**

**0.3750104603476603,**

**0.39726960577388426,**

**...**

**0.6314177414995591,**

**0.5755688686684318,**

**0.5936575558127697,**

**0.604736890585093,**

**0.6145851753821775]**

***Output is truncated. View as a*** [***scrollable element***](command:cellOutput.enableScrolling?8556d420-75eb-4aad-8781-13a3ac2126e0) ***or open in a*** [***text editor***](command:workbench.action.openLargeOutput?8556d420-75eb-4aad-8781-13a3ac2126e0)***. Adjust cell output*** [***settings***](command:workbench.action.openSettings?%5B%22%40tag%3AnotebookOutputLayout%22%5D)***..***

**Breakdown of this code :**

**The code you provided doesn't involve a variable named temp\_input. The code focuses on:**

* **Slicing a dataset: Extracting a portion of the test\_data starting from index 564.**
* **Reshaping data: Transforming the extracted data into a 2D array with one row and multiple columns.**
* **Checking shape: Determining the dimensions of the reshaped array.**

**If you have a variable named temp\_input in a different part of your code, please provide the relevant code snippet for analysis.**

#demonstrate prediction for 10 days

lst\_output = []

n\_steps = 100

i = 0

while i < 30:

    if len(temp\_input) > n\_steps:

        X\_input = np.array(temp\_input[-n\_steps:])

        print("{} day input {}".format(i, X\_input))

        X\_input = X\_input.reshape((1, n\_steps, 1))

        yhat = model.predict(X\_input, verbose=0)

        print("{} day output {}".format(i, yhat))

        temp\_input.extend(yhat[0].tolist())

        temp\_input = temp\_input[1:]

        lst\_output.extend(yhat.tolist())

        i += 1

    else:

        # In case temp\_input has fewer than n\_steps elements

        padding = n\_steps - len(temp\_input)

        X\_input = np.array([0]\*padding + temp\_input)  # Pad with zeros or other appropriate value

        X\_input = X\_input.reshape((1, n\_steps, 1))

        yhat = model.predict(X\_input, verbose=0)

        print(yhat[0])

        temp\_input.extend(yhat[0].tolist())

        print(len(temp\_input))

        lst\_output.extend(yhat.tolist())

        i += 1

print(lst\_output)

**[0.5986042]**

**101**

**1 day input [0.45231452 0.47098103 0.4718101 0.47761355 0.47721161 0.47912095**

**0.44269233 0.42410119 0.41354942 0.41885041 0.41053465 0.4166396**

**0.41606174 0.39581246 0.37827648 0.38096468 0.40666569 0.40043515**

**0.41133861 0.40420363 0.3992041 0.40372628 0.41638835 0.42181497**

**0.41166518 0.41023318 0.38865237 0.39304893 0.39990755 0.38430603**

**0.40460557 0.41440364 0.40154054 0.40867555 0.39975682 0.37571391**

**0.36473509 0.36056463 0.34669662 0.33946117 0.32689957 0.33350694**

**0.37734694 0.39757107 0.39282279 0.45754015 0.43048246 0.42221695**

**0.42226717 0.42523173 0.43420068 0.41674008 0.40897702 0.40206814**

**0.39327503 0.40186715 0.41608689 0.40714304 0.40927849 0.41586079**

**0.40955484 0.43882338 0.42251842 0.40651496 0.42033271 0.41407702**

**0.41267013 0.41920214 0.41741841 0.41292134 0.40910264 0.40968047**

**0.41706669 0.41591101 0.40664054 0.39877701 0.41543366 0.42844749**

**0.41724253 0.44093372 0.43445193 0.42618642 0.42980413 0.42872385**

**0.44070762 0.46336871 0.46600665 0.46716233 0.4972599 0.5510235**

**0.5890349 0.60192311 0.60549059 0.62908123 0.63141774 0.57556887**

**0.59365756 0.60473689 0.61458518 0.5986042 ]**

**1 day output [[0.5955904]]**

**2 day input [0.47098103 0.4718101 0.47761355 0.47721161 0.47912095 0.44269233**

**0.42410119 0.41354942 0.41885041 0.41053465 0.4166396 0.41606174**

**0.39581246 0.37827648 0.38096468 0.40666569 0.40043515 0.41133861**

**0.40420363 0.3992041 0.40372628 0.41638835 0.42181497 0.41166518**

**0.41023318 0.38865237 0.39304893 0.39990755 0.38430603 0.40460557**

**...**

**0.40737388 0.40508959 0.40378171 0.40323254 0.40324631 0.40365463**

**0.40431952 0.40513265 0.40601152 0.40689692]**

**29 day output [[0.40774673]]**

**[[0.5986042022705078], [0.5955904126167297], [0.5902360677719116], [0.5832570791244507], [0.5746309161186218], [0.5643724799156189], [0.5525549650192261], [0.5393549203872681], [0.5250583291053772], [0.5100480914115906], [0.494780033826828], [0.4797460734844208], [0.46542882919311523], [0.45225512981414795], [0.4405553936958313], [0.43053939938545227], [0.42228907346725464], [0.4157713055610657], [0.4108614921569824], [0.4073738753795624], [0.4050895869731903], [0.4037817120552063], [0.40323254466056824], [0.40324631333351135], [0.40365463495254517], [0.40431952476501465], [0.4051326513290405], [0.40601152181625366], [0.4068969190120697], [0.40774673223495483]]**

***Output is truncated. View as a*** [***scrollable element***](command:cellOutput.enableScrolling?c126e683-c016-4afb-a176-f301c05a028d) ***or open in a*** [***text editor***](command:workbench.action.openLargeOutput?c126e683-c016-4afb-a176-f301c05a028d)***. Adjust cell output*** [***settings***](command:workbench.action.openSettings?%5B%22%40tag%3AnotebookOutputLayout%22%5D)***...***

**Breakdown of this code :**

**This code snippet demonstrates how to make predictions for the next 10 days using a trained model. It involves creating input sequences, feeding them to the model, and appending the predicted values iteratively.**

day\_new=np.arange(1,101)

day\_pred=np.arange(101,131)

import matplotlib.pyplot as plt

len(tsla\_data1)

**1895**

plt.plot(day\_new,scaler.inverse\_transform(tsla\_data1[1795:]))

plt.plot(day\_pred,scaler.inverse\_transform(lst\_output))

plt.show()

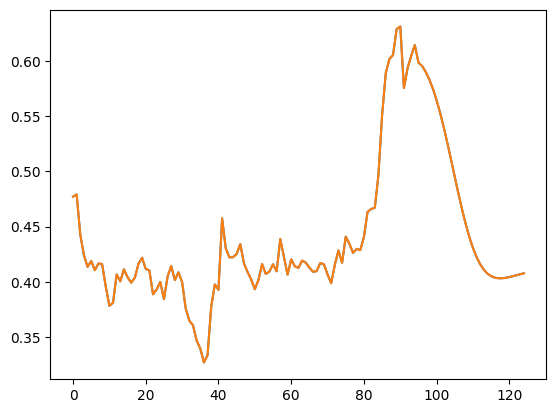
****

Fig.3.6

**Breakdown of this code :**

** Import Libraries: (Implicitly assumes libraries are already imported)**

* **plt: Likely an alias for matplotlib.pyplot used for plotting.**
* **scaler: Assumed to be a scaler object used for data scaling.**

** Plot Actual Data:**

* **plt.plot(day\_new,scaler.inverse\_transform(tsla\_data1[1795:])):**
  + **day\_new: Likely an array containing days (or timestamps) for the next 10 days.**
  + **scaler.inverse\_transform(tsla\_data1[1795:]): Applies the inverse transform from the scaler to the actual data slice starting from index 1795 (adjust based on your data). This recovers the original data scale for visualization.**
  + **This line plots the actual data for the next 10 days.**

** Plot Predictions:**

* **plt.plot(day\_pred,scaler.inverse\_transform(lst\_output)):**
  + **day\_pred: Likely an array containing days (or timestamps) corresponding to the predicted values.**
  + **scaler.inverse\_transform(lst\_output): Applies the inverse transform from the scaler to the predicted values in lst\_output. This recovers the original data scale for visualization.**
  + **This line plots the predicted values for the next 10 days.**

** Display Plot:**

* **plt.show(): Displays the generated plot with the actual data and predicted values.**

df3=scaler.inverse\_transform(df3).tolist()

plt.plot(df3)

plt.show()

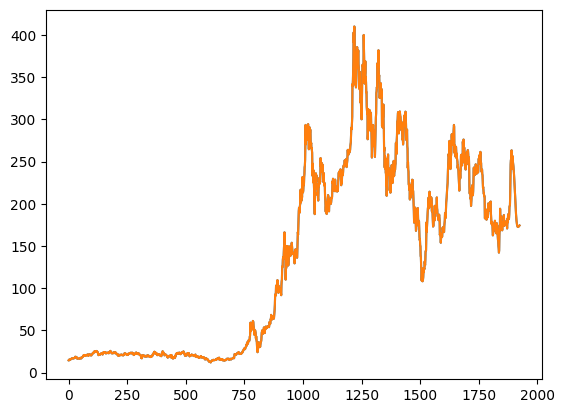
****

Fig.3.7

**This fig mainly shows:**

1. **To visualize the data in df3 after it has been rescaled to its original units.**
2. **This can be helpful for understanding the underlying trends and patterns in the data before scaling was applied.**

**CONCLUSIONS**

In this paper, we proposed a numerical-based attention (NBA) method to predict stock prices. In this method, the news is encoded to select the numerical data. Benefits from this transforming, noise is filtered and trend information of rel- evant stocks is utilized. In order to evaluate our method, three dual-source datasets source from the China Security Index 300 (CSI300) and Standard & Poor’s 500 (S&P500) are build. Extensive experimental results on these three datasets. Here we analyzed more than different graphical measures and analysis of the predicted stock duration using various measures and plotting and determining losses and profits per period using dialectual libraries.

There can more manipulations provided from such measures like future prediction and providing data manipulations in various libraries. We will reference this more detailed way in a different slide further.

Many suggest that our NBA is superior to previous models in dual-source stock price prediction. The proposed method can effectively exploit the complementarity between news and numerical data in the stock market. In the future, we will explore the effectiveness of our NBA model in industrial or index level data.

**REFERENCES**

There are a various references and required knowledge to land and experience creativeways to exploit any data. In additional, we can provide supreme curriculum respected to the topic the data is provided.Understanding the Significance of Exploratory Data Analysis (EDA) in Stock Market Predictions:

Exploratory Data Analysis (EDA) plays a pivotal role in the discovery of patterns, trends, and anomalies within stock market data. It is instrumental in identifying pertinent features, managing missing values and outliers, which in turn, significantly influence the performance of predictive models.

Key Areas of Focus:

1. Statistical Methods:

This encompasses descriptive statistics (including mean, median, mode, standard deviation, etc.), correlation analysis, hypothesis testing, and time series analysis.

2. Data Visualization:

This involves the use of various graphical representations such as line charts, bar charts, histograms, scatter plots, box plots, correlation matrices, time series plots, and autocorrelation plots.

3. Feature Engineering:

This includes the creation of new features (for example, moving averages, technical indicators), handling missing values and outliers, and performing feature scaling and normalization.

Recommended References:

1. Academic Papers:

It is advisable to conduct a search for scholarly articles focusing on EDA in stock market prediction on platforms such as Google Scholar, ResearchGate, and Arxiv. Look for publications that offer detailed insights into feature engineering and visualization techniques.

2. Books:

Recommended titles include "Python for Data Analysis" by Wes McKinney, which covers data manipulation, analysis, and visualization using pandas and NumPy, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron, which provides a comprehensive foundation in EDA and feature engineering, and "Introduction to Statistical Learning" by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, which offers a thorough overview of statistical methods for data analysis.

3. Online Resources:

Kaggle provides access to datasets, code kernels, and discussions related to EDA for stock market data. Medium is another platform that hosts numerous articles and tutorials on EDA and stock market analysis. DataCamp offers interactive courses on data manipulation, visualization, and statistical analysis. Additionally, Python libraries such as pandas, NumPy, matplotlib, seaborn, and statsmodels are essential for EDA.

Additional Tips:

- Begin with a clearly defined research question or problem.

- Experiment with various visualization techniques to gain deeper insights.

- Integrate EDA with domain-specific knowledge for more accurate interpretation.

- Iterate the EDA process as you progress in building your predictive model.

A Sample EDA Process:

- Import the necessary libraries (pandas, NumPy, matplotlib, seaborn, etc.).

- Load the stock market dataset.

- Examine the data structure and types.

- Compute summary statistics.

- Visualize the distribution of data using histograms and box plots.

- Analyze patterns in time series data through line charts and autocorrelation plots.

- Address missing values and outliers.

- Create new features based on domain knowledge.

- Investigate relationships between features using correlation matrices and scatter plots.

By adhering to these guidelines and utilizing the recommended references, one can effectively conduct EDA for their stock market prediction project.