Data Preprocessing:

Preprocessing the data for Time Series Analysis Modeling and Forecasting

```
# Importing required packages
import pandas as pd, numpy as np, matplotlib.pyplot as plt, seaborn as sns
import yfinance as yf
import datetime as dt
from datetime import timedelta, datetime
import plotly.graph_objects as go
import plotly.express as px
import warnings
warnings.filterwarnings("ignore")
```

```
# Define dates to fetch stock data
today = dt.date.today()
d1 = today.strftime("%Y-%m-%d")
enddate = today

d2 = today - timedelta(days=365*2)
d2 = d2.strftime("%Y-%m-%d")
startdate = d2

print("Start Date:", startdate)
print("End Date:", enddate)
```

```
Start Date: 2023-07-06
End Date: 2025-07-05
```

```
# Fetching stock data using yfinance
ticker = 'TATAMOTORS.NS'
df = yf.download(ticker, start=startdate, end=enddate, progress = False)
df.columns = df.columns.droplevel('Ticker')
df
```

Price Date	Close	High	Low	Open	Volume
2023-07-06	592.000793	596.090037	579.930204	582.147251	14356681
2023-07-07	609.145996	615.649310	583.526786	591.212537	21066726
2023-07-10	609.589355	625.502789	608.505430	614.860976	23802524
2023-07-11	619.295166	621.216619	613.038190	615.797170	12051173
2023-07-12	612.348389	621.561427	610.279109	620.723899	10785502
...	...	...	...	...	...
2025-06-30	688.000000	691.900024	685.000000	688.900024	6960104
2025-07-01	683.799988	693.849976	680.400024	691.099976	6866073
2025-07-02	688.549988	692.450012	680.650024	683.799988	8034013
2025-07-03	690.400024	696.950012	688.500000	693.849976	9668110
2025-07-04	689.049988	692.849976	686.349976	691.000000	4942900

493 rows × 5 columns

```
df.insert(0, 'Date', df.index)
df.reset_index(drop=True, inplace=True)

df.info()
```

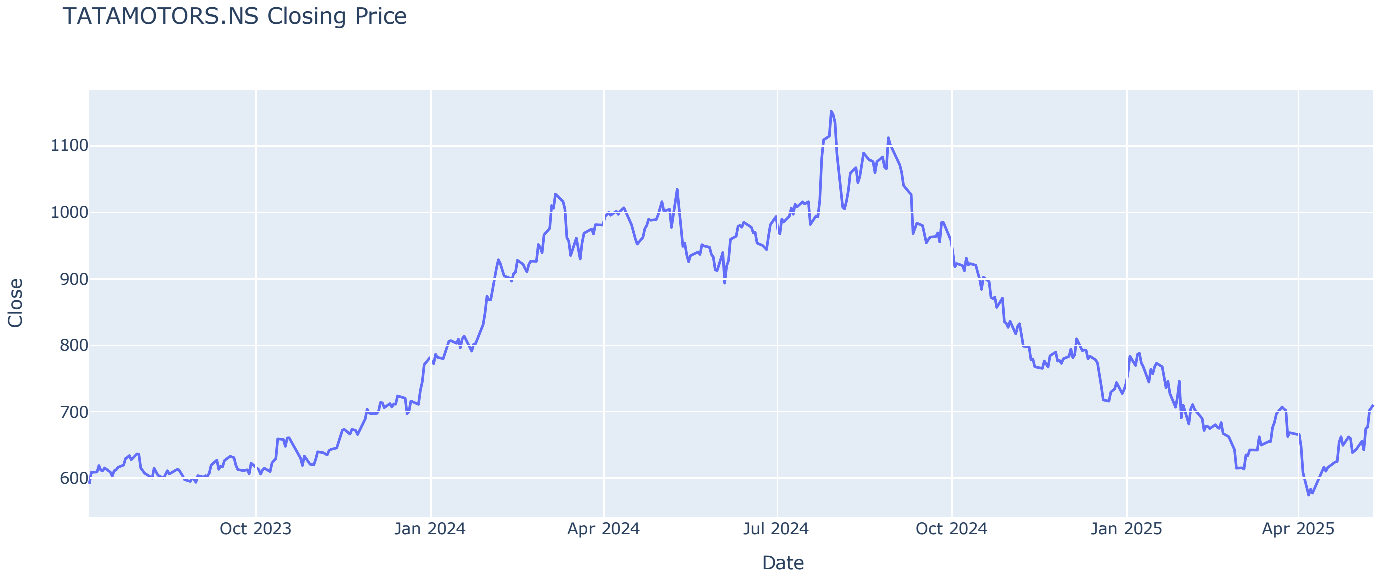
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 493 entries, 0 to 492
Data columns (total 6 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   Date     493 non-null     datetime64[ns]
 1   Close    493 non-null     float64
 2   High     493 non-null     float64
 3   Low      493 non-null     float64
 4   Open     493 non-null     float64
 5   Volume   493 non-null     int64
dtypes: datetime64[ns](1),
float64(4), int64(1) memory usage:
23.2 KB
```

```
df.describe()
```

Price	Date	Close	High	Low	Open	Volume
count	493	493.000000	493.000000	493.000000	493.000000	4.930000e+02
mean	2024-07-06 00:29:12.535496960	796.498569	806.847796	787.491285	798.469776	1.269633e+07
min	2023-07-06 00:00:00	574.807922	577.038742	531.183000	555.722019	0.000000e+00
25%	2024-01-04 00:00:00	662.652588	669.954600	655.414856	662.305592	8.324354e+06
50%	2024-07-09 00:00:00	765.319946	776.325311	751.935010	764.427580	1.087302e+07
75%	2025-01-06 00:00:00	949.431885	958.031132	937.719059	951.816447	1.428910e+07
max	2025-07-04 00:00:00	1151.945801	1168.949630	1135.536938	1157.051852	5.981103e+07
std	NaN	153.025373	155.207258	150.929272	153.714820	7.563426e+06

```
# Make a plot for all the columns
fig = px.line(df, x='Date', y=df.columns, title='Stock Data Overview',
width=1200, height=500) fig.show()
```

```
df1 = df[['Date', 'Close']]


# Plotting the closing price of the stock
fig = px.line(df1, x='Date', y='Close', title=f'{ticker} Closing Price',width=1200, height=500)
fig.show()
```



Stationarity Check:

Many time series models assume stationarity (constant statistical properties over time) for reliable forecasting. The ADF test is used to check for stationarity, with a low p-value (typically < 0.05) indicating that the series is stationary.

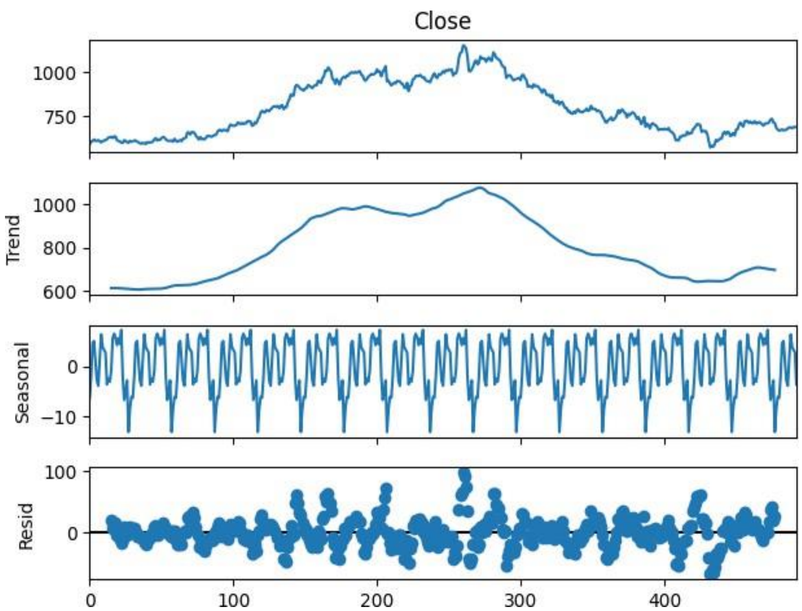
```
# Importing the required libraries for stationarity check
from statsmodels.tsa.stattools import adfuller

def stationarity_check(df1): """
   Perform the Augmented Dickey-Fuller test to check for stationarity. """
   result = adfuller(df1)
   print('ADF Statistic: %f' % result[0]) print('p-value: %f' % result[1])
   if result[1] <= 0.05:
      print("Rejects null hypothesis, Data is stationary") else:
      print("Fails to reject null hypothesis, Data is non-stationary")

# Stationarity Check
stationarity_check(df1['Close'])
```

```
ADF Statistic: -1.391721
p-value: 0.586239
Fails to reject null hypothesis, Data is non-stationary
```

Decompose Data:

Time series decomposition separates a series into its underlying components: trend (long-term movement), seasonality (recurring patterns), and residuals (random noise). This helps in understanding the data's structure and can improve forecasting by modeling each component individually.

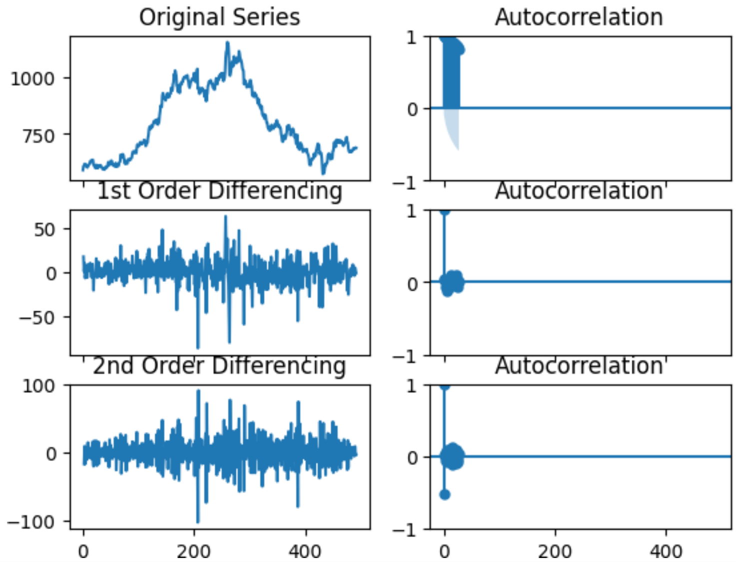
```
# Importing the seasonal decomposition library
from statsmodels.tsa.seasonal import seasonal_decompose

decompose = seasonal_decompose(df1['Close'], model='additive', period=30)
decompose.plot();
```



Auto Correlation check:

Autocorrelation analysis examines the correlation between a time series and its lagged versions, revealing repeating patterns (like seasonality) or trends. Significant autocorrelations indicate that past values influence current ones, which is crucial for selecting appropriate time series models (e.g., ARIMA).

```
# Importing ACF and PACF libraries
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Original series
fig, axes = plt.subplots(3, 2, sharex=True)
axes[0, 0].plot(df1['Close']); axes[0, 0].set_title('Original Series')
plot_acf(df1['Close'], ax=axes[0, 1])

# 1st differencing
axes[1, 0].plot(df1['Close'].diff()); axes[1, 0].set_title('\n1st Order Differencing')
plot_acf(df1['Close'].diff().dropna(), ax=axes[1, 1])

# 2nd differencing
axes[2, 0].plot(df1['Close'].diff().diff()); axes[2, 0].set_title('\n2nd Order Differencing')
plot_acf(df1['Close'].diff().diff().dropna(), ax=axes[2, 1])
plt.show()
```



Therefore, d = 1

Finding the value of p :

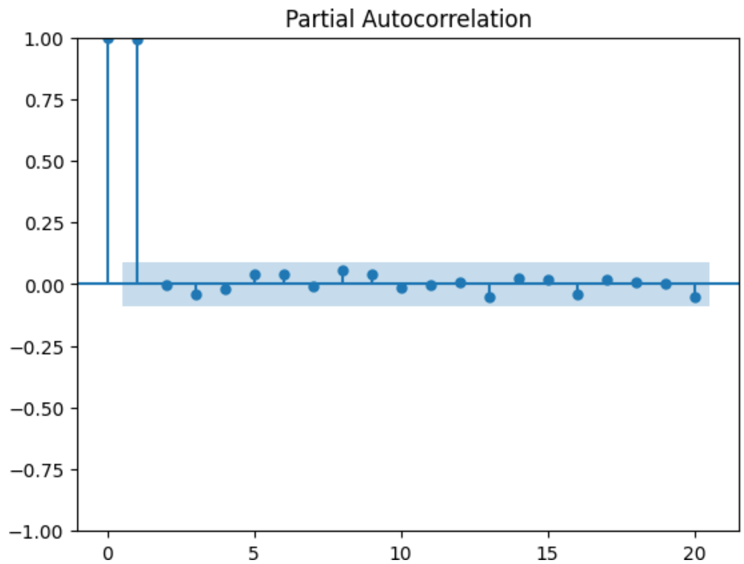
```
from statsmodels.tsa.stattools import acf, pacf
```

```
x_acf = pd.DataFrame(acf(df1['Close'], nlags=20))
print(x_acf)
```

```
          0
0   1.000000
1   0.992805
2   0.985615
3   0.977926
4   0.970016
5   0.962736
6   0.956064
7   0.949361
8   0.943554
9   0.938381
10  0.933088
11  0.927847
12  0.922660
13  0.916620
14  0.910957
15  0.905538
16  0.899523
17  0.893884
18  0.888389
19  0.882878
20  0.876813
```

Therefore, p = 6 **, as 95% confidence level is till 6th index value**

```
plot_pacf(df1['Close'], lags=20, alpha=0.05);
```



***Therefore, q = 2*, as 95% confidence level have only 2 spikes**

```
# Let’s define the values of p, d, q based on the above analysis

p = 6
d = 1
q = 2  # Assuming q=2 based on the ACF plot
```

ARIMA model:

Putting the value of p, d, q and fiting the model to ARIMA

```
# Importing Libraries for ARIMA modeling
from statsmodels.tsa.arima.model import ARIMA


p, d, q = 6,1,2
model = ARIMA(df1['Close'], order=(p, d, q))
model = model.fit()
print(model.summary())
```

```
                              SARIMAX Results
==============================================================================
Dep. Variable:          Close   No. Observations:          493
Model:           ARIMA(6, 1, 2)   Log Likelihood          -2029.527
Date:          Sat, 05 Jul 2025   AIC                      4077.054
Time:                15:08:56   BIC                      4114.840
Sample:               0 - 493   HQIC                     4091.892
Covariance Type:          opg
==============================================================================
                 coef   std err        z     P>|z|     [0.025    0.975]
------------------------------------------------------------------------------
ar.L1     0.0958      0.900    0.106    0.915    -1.669    1.861
ar.L2     0.4977      0.452    1.100    0.271    -0.389    1.384
ar.L3     0.0215      0.064    0.337    0.736    -0.104    0.147
ar.L4    -0.1007      0.056   -1.788    0.074    -0.211    0.010
ar.L5    -0.0924      0.085   -1.086    0.278    -0.259    0.074
ar.L6     0.0346      0.092    0.378    0.706    -0.145    0.214
ma.L1    -0.0720      0.901   -0.080    0.936    -1.838    1.694
ma.L2    -0.4491      0.446   -1.007    0.314    -1.323    0.425
sigma2  224.1150      8.403   26.669    0.000   207.644   240.586
==============================================================================
Ljung-Box (L1) (Q):            0.00  Jarque-Bera (JB):         506.03
Prob(Q):                       0.98  Prob(JB):                   0.00
Heteroskedasticity (H):        1.56  Skew:                      -0.70
Prob(H) (two-sided):           0.00  Kurtosis:                   7.76
==============================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

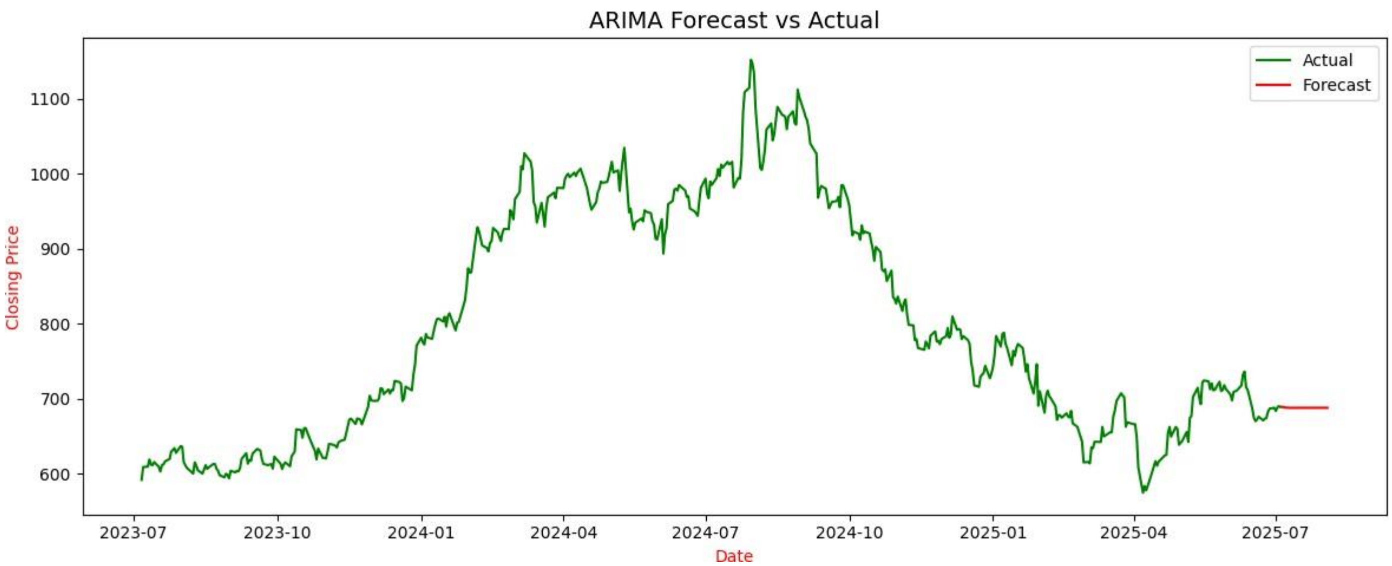
Generating a 30-Day forecast

```
# Generate forecast
n_forecast = 30

# Get forecast object
forecast_obj = model.get_forecast(steps=n_forecast)
forecast_values = forecast_obj.predicted_mean

last_date = df1.index[-1]
```

```
# Create forecast index
forecast_index = pd.date_range(start=last_date + pd.Timedelta(days=1), periods=n_forecast, freq='D')
forecast_series = pd.Series(forecast_values.values, index=forecast_index)
```

Ploting the Actual vs next 30-Day forecast data

```
# Plot actual and forecast with aligned datetime x-axis
plt.figure(figsize=(12, 5))
plt.plot(df1['Close'], label='Actual', color='green')
plt.plot(forecast_series, label='Forecast',
color='red') plt.title('ARIMA Forecast vs
Actual', fontsize=14)
plt.xlabel('Date', color='red')
plt.ylabel('Closing Price',
color='red') plt.legend()
plt.tight_layout()
plt.show()
```



SARIMA Modeling:

```
# Importing necessary libraries for SARIMA modeling
import statsmodels.api as sm
import sklearn.metrics as metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import warnings
warnings.filterwarnings("ignore")

# To ensure 'Date' is in datetime and set as
index df1['Date'] = pd.to_datetime(df1['Date'])
df1.set_index('Date', inplace=True)
```

Dividing and splitting the data into Train and Test sets and finally fitting it to the SARIMA model

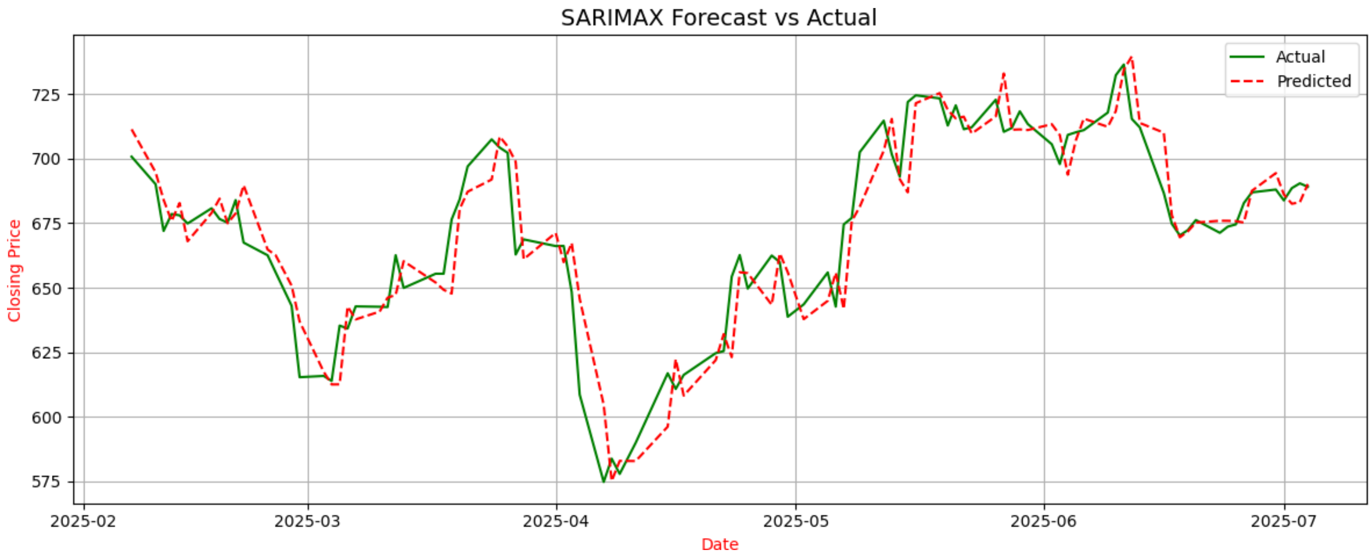
```
# Train-test split
train_size = int(len(df1) * 0.8)
train, test = df1['Close'][:train_size], df1['Close'][train_size:]


train.shape, test.shape
```

```
((394,), (99,))
```

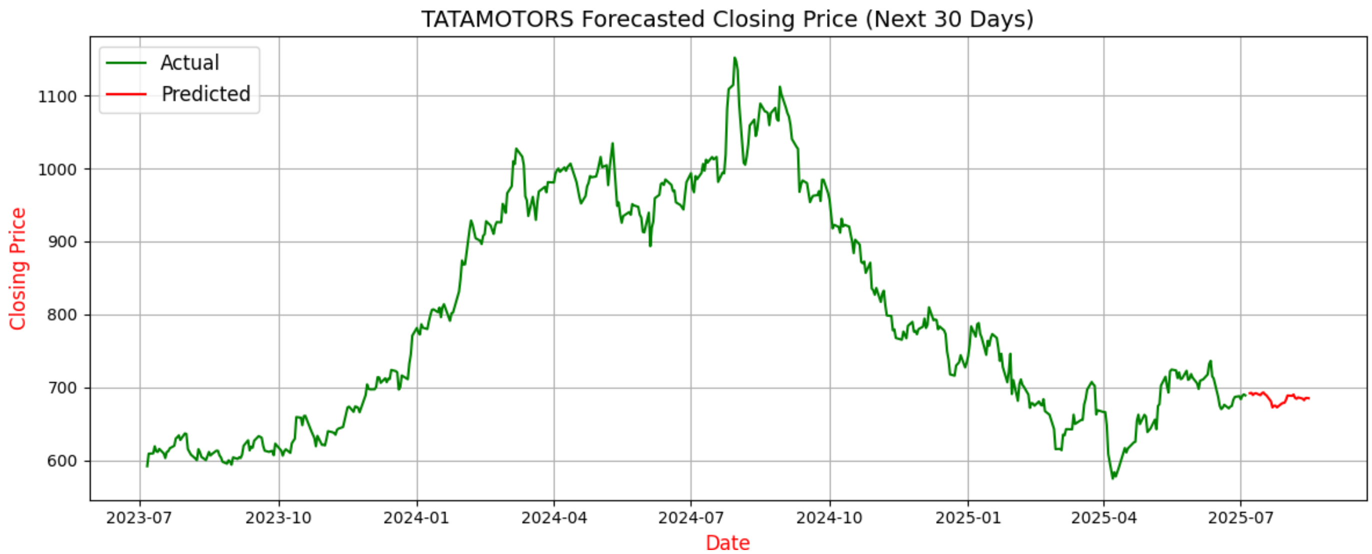
```
p, d, q = 6, 1, 2
model = sm.tsa.statespace.SARIMAX(df1['Close'], order=(p, d, q), seasonal_order=(p, d, q, 12))
model = model.fit()
print(model.summary())
```

```
                                SARIMAX Results
==========================================================================================
Dep. Variable:                    Close   No. Observations:          493
Model:         SARIMAX(6, 1, 2)x(6, 1, 2, 12)   Log Likelihood       -1996.243
Date:                  Sat, 05 Jul 2025   AIC                       4026.486
Time:                          15:34:45   BIC                       4097.440
Sample:                               0   HQIC                      4054.377
                                  - 493
Covariance Type:                    opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.3101      1.549      0.200      0.841      -2.727       3.347
ar.L2          0.4144      0.964      0.430      0.667      -1.474       2.303
ar.L3         -0.0023      0.096     -0.024      0.981      -0.190       0.186
ar.L4         -0.0937      0.061     -1.536      0.125      -0.213       0.026
ar.L5         -0.0588      0.120     -0.489      0.625      -0.295       0.177
ar.L6          0.0321      0.085      0.376      0.707      -0.135       0.199
ma.L1         -0.2985      1.547     -0.193      0.847      -3.330       2.734
ma.L2         -0.3603      0.955     -0.377      0.706      -2.231       1.511
ar.S.L12      -0.6835      0.311     -2.199      0.028      -1.293      -0.074
ar.S.L24       0.0677      0.076      0.888      0.374      -0.082       0.217
ar.S.L36       0.0195      0.065      0.299      0.765      -0.109       0.148
ar.S.L48      -0.0154      0.069     -0.222      0.824      -0.151       0.120
ar.S.L60      -0.0389      0.069     -0.562      0.574      -0.175       0.097
ar.S.L72       0.0408      0.066      0.615      0.538      -0.089       0.171
ma.S.L12      -0.2099      0.309     -0.680      0.496      -0.815       0.395
ma.S.L24      -0.7314      0.277     -2.642      0.008      -1.274      -0.189
sigma2       224.1421     11.904     18.829      0.000     200.810     247.474
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):               414.19
Prob(Q):                              0.98   Prob(JB):                         0.00
Heteroskedasticity (H):               1.40   Skew:                            -0.70
Prob(H) (two-sided):                  0.04   Kurtosis:                         7.33
===================================================================================
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

```
# Forecast the test set
forecast = model.predict(start=len(train), end=len(train)+len(test)-1)

# Plot actual vs forecast
plt.figure(figsize=(12, 5))
plt.plot(test.index, test, label='Actual', color='green')
plt.plot(test.index, forecast, label='Predicted', color='red', linestyle ='--')
plt.xlabel('Date', color='red')
plt.ylabel('Closing Price', color='red')
plt.title('SARIMAX Forecast vs Actual', fontsize = 14) plt.legend()
plt.grid(True)
plt.tight_layout() plt.show()
```




```
# Predict next 30 days

# Forecast next 30 days from last point
future_start = df1.index[-1] + pd.Timedelta(days=1)
future_dates = pd.date_range(start=future_start, periods=30, freq='B')  # 30 business days

future_forecast = model.forecast(steps=30)
future_forecast.index = future_dates


# Plot full history + next 30 days forecast plt.figure(figsize=(12, 5))
plt.plot(df1['Close'], color='green', label='Actual')
plt.plot(future_forecast.index, future_forecast, color='red', label='Predicted')
plt.xlabel('Date', color='red',fontsize=12)
plt.ylabel('Closing Price', color='red',fontsize=12)
plt.title('TATAMOTORS Forecasted Closing Price (Next 30 Days)',fontsize = 14)
plt.legend(loc='upper left', fontsize = 12)
plt.grid(True)
plt.tight_layout() plt.show()
```



Calculating the Model Performance

```
# Calculate evaluation metrics of SARIMAX

print("SARIMAX Model Performance Metrics:\n") # MSE
mse = mean_squared_error(test, forecast)
print(f"MSE: {mse:.2f}")

# RMSE
rmse = np.sqrt(mse)
print(f"RMSE: {rmse:.2f}")

# MAE
mae = mean_absolute_error(test, forecast)
print(f"MAE: {mae:.2f}")

# MAPE
mape = np.mean(np.abs((test - forecast) / test)) * 100
print(f"MAPE: {mape:.2f}%")

# R² Score
r2 = r2_score(test, forecast)
print(f"R² Score: {r2:.4f}")
```

```
SARIMAX Model Performance Metrics:
```

```
MSE: 166.55
RMSE: 12.91
MAE: 9.32
MAPE: 1.40%
R² Score: 0.8721
```

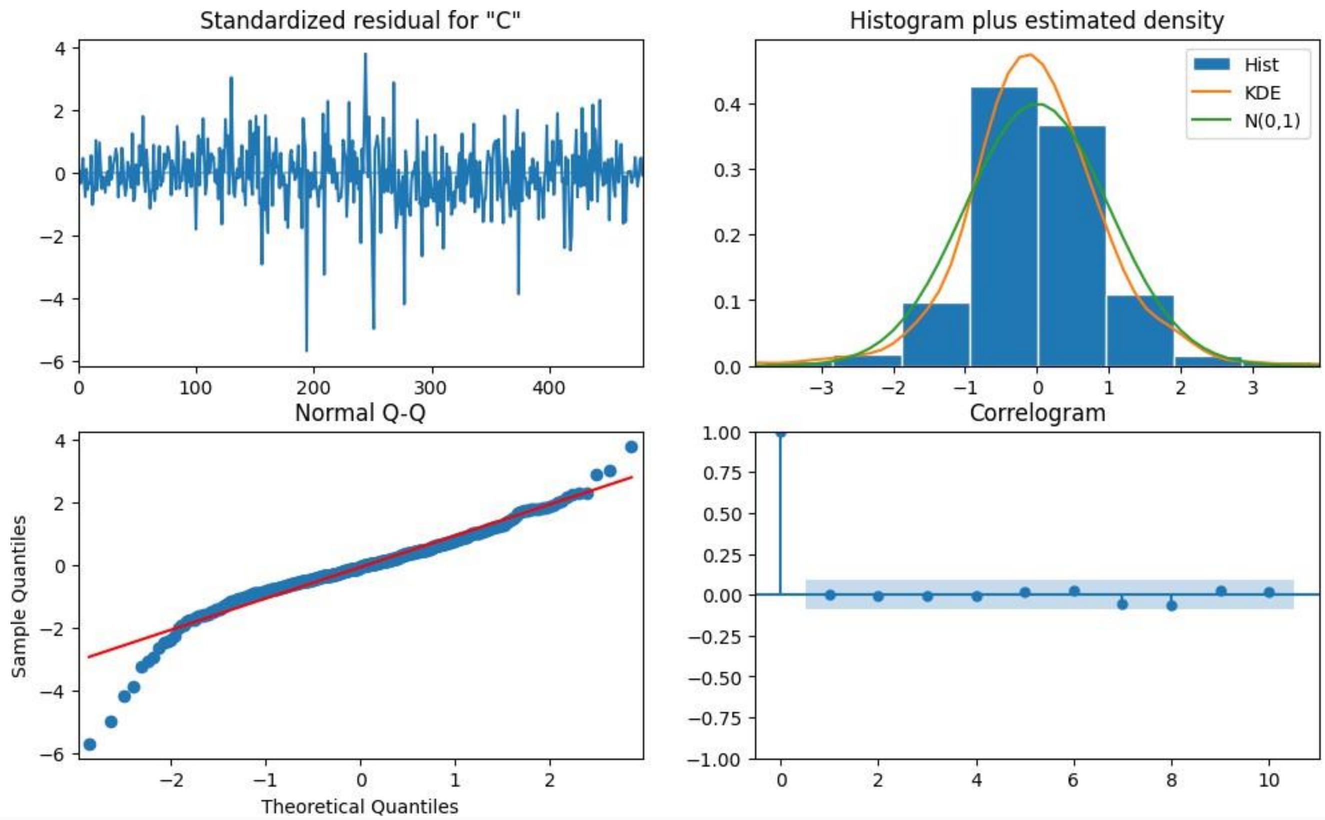
`Ljung Box test` : If lb p-value >= 0.05 ==> Fail to reject Null Hypothesis ==> No significant autocorrelation ==> **`Model is adequate`**

```
from statsmodels.stats.diagnostic import acorr_ljungbox
lb_test = acorr_ljungbox(model.resid, lags=[10], return_df=True)
lb_test
```

	lb_stat	lb_pvalue
10	0.618883	0.999982

The plots show no patterns in standardized residuals over time (indicating white noise), a normal distribution of residuals (histogram and Q-Q plot), and no significant autocorrelation in the residuals (correlogram).

```
model.plot_diagnostics(figsize=(12, 7))
plt.show()
```



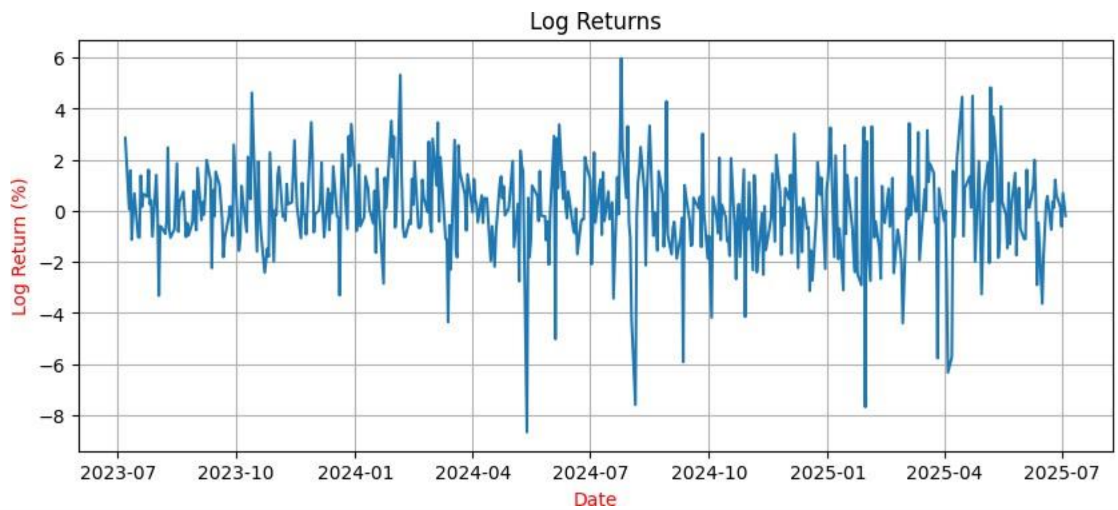
GARCH Model:

```
# Importing the required models for GARCH model from arch import arch_model
import warnings
warnings.filterwarnings("ignore")


# Compute Log Returns (in %)
returns = 100 * np.log(df1['Close'] / df1['Close'].shift(1)).dropna()
returns.head()
```

	Close
1	2.854999
2	0.072757
3	1.579646
4	-1.128062
5	-0.128802

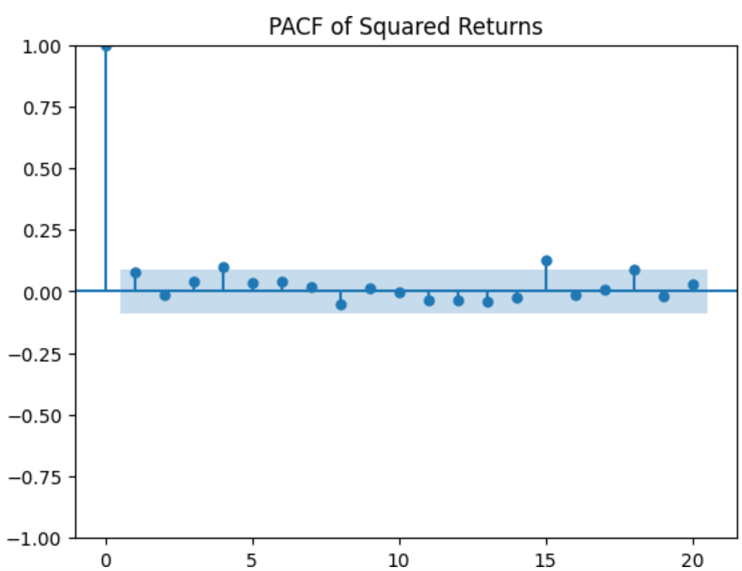
Plotting the Log Returns from the data

```
# Plot the Log Returns
plt.figure(figsize=(10, 4))
plt.plot(returns)
plt.title('Log Returns')
plt.ylabel('Log Return (%)', color='red')
plt.xlabel('Date', color='red')
plt.grid(True)
plt.show()
```



PACF for GARCH modeling:

```
# Plot PACF of Squared Returns to Check for Volatility Clustering
plot_pacf(returns**2, lags=20)
plt.title('PACF of Squared Returns')
plt.show()
```



Fitting the Model:

Initializing and fitting an EGARCH model with an order of p=3 and q=1, assuming a Student's t-distribution for the errors, to model the returns data.

```
model = arch_model(returns, vol='EGARCH', p=3, q=1, dist='t')
model_fit = model.fit(disp='on')  # You can set disp='off' to suppress output
print(model_fit.summary())
```

```
Iteration:      1,   Func. Count:      9,   Neg. LLF: 24699.861243430252
Iteration:      2,   Func. Count:     21,   Neg. LLF: 23057.816634130675
Iteration:      3,   Func. Count:     32,   Neg. LLF: 2116.3952794841152
Iteration:      4,   Func. Count:     42,   Neg. LLF: 1076.9977115294168
Iteration:      5,   Func. Count:     52,   Neg. LLF: 1110.2353207013402
Iteration:      6,   Func. Count:     61,   Neg. LLF: 9059.76517710912
Iteration:      7,   Func. Count:     71,   Neg. LLF: 979.142728544044
Iteration:      8,   Func. Count:     80,   Neg. LLF: 965.3902350311942
Iteration:      9,   Func. Count:     89,   Neg. LLF: 964.5125849011857
Iteration:     10,   Func. Count:     97,   Neg. LLF: 964.3905787414078

Iteration:     11,   Func. Count:    105,   Neg. LLF: 964.3167980565632
Iteration:     12,   Func. Count:    113,   Neg. LLF: 964.2969886731739
Iteration:     13,   Func. Count:    121,   Neg. LLF: 964.2903124568302
Iteration:     14,   Func. Count:    129,   Neg. LLF: 964.2853018855476
Iteration:     15,   Func. Count:    137,   Neg. LLF: 964.280269278292
Iteration:     16,   Func. Count:    145,   Neg. LLF: 964.2784535699975
Iteration:     17,   Func. Count:    153,   Neg. LLF: 964.2781910784017
Iteration:     18,   Func. Count:    161,   Neg. LLF: 964.278176256512
Iteration:     19,   Func. Count:    168,   Neg. LLF: 964.2781762565119
                     Constant Mean - EGARCH Model Results
==============================================================================
Dep. Variable:                  Close   R-squared:                       0.000
Mean Model:             Constant Mean   Adj. R-squared:                  0.000
Vol Model:                     EGARCH   Log-Likelihood:               -964.278
Distribution:  Standardized Student's t   AIC:                         1942.56
Method:            Maximum Likelihood   BIC:                           1971.95
                                        No. Observations:                  492
Date:                Sat, Jul 05 2025   Df Residuals:                      491
Time:                        15:15:15   Df Model:                            1
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0677  7.102e-02      0.953      0.341 [-7.153e-02,  0.207]
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.4034      0.134      3.006  2.650e-03    [  0.140,  0.666]
alpha[1]       0.0240      0.138      0.173      0.862    [ -0.247,  0.295]
alpha[2]       0.1061      0.106      1.002      0.316    [ -0.101,  0.314]
alpha[3]       0.2200      0.116      1.896  5.799e-02 [-7.450e-03,  0.447]
beta[1]        0.6759      0.111      6.104  1.036e-09    [  0.459,  0.893]
                              Distribution
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             4.8867      0.952      5.131  2.881e-07 [  3.020,  6.753]
========================================================================

Covariance estimator: robust
```

Testing the Ljung Box test for checking the autocorrelation of the model

```
lb_test = acorr_ljungbox(model_fit.std_resid.dropna(), lags=[10], return_df=True)
print("Ljung-Box Test Result:\n", lb_test)
```

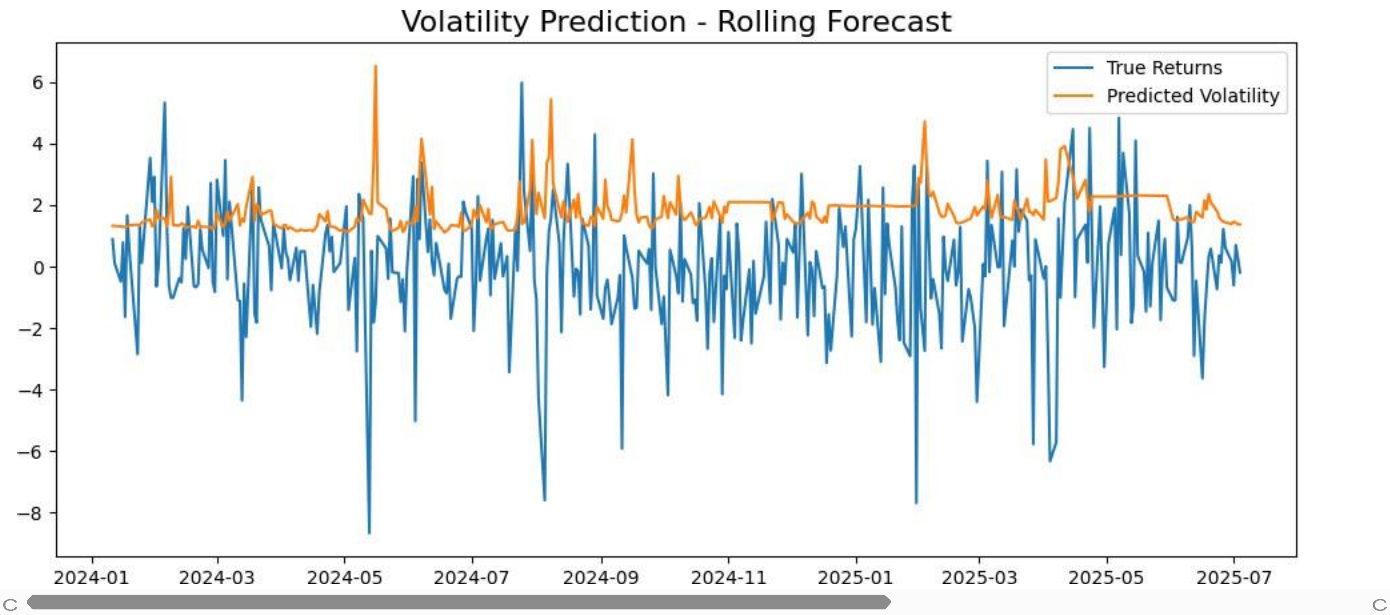
```
Ljung-Box Test Result:
      lb_stat  lb_pvalue
10  13.020091   0.222551
```

```
rolling_predictions = [] test_size = 365

for i in range(test_size):
   train = returns[:-(test_size-i)]
   model = arch_model(train, p=3, q=1) model_fit = model.fit(disp='off')
   pred = model_fit.forecast(horizon=1)
   rolling_predictions.append(np.sqrt(pred.variance.values[-1,:][0]))
```

Predicting the volatility of the stock with the actual returns

```
rolling_predictions = pd.Series(rolling_predictions, index=returns.index[-365:])

plt.figure(figsize=(12,5))
true, = plt.plot(returns[-365:])
preds, = plt.plot(rolling_predictions)
plt.title('Volatility Prediction - Rolling Forecast', fontsize=16)
plt.legend(['True Returns', 'Predicted Volatility'], fontsize=10)
```

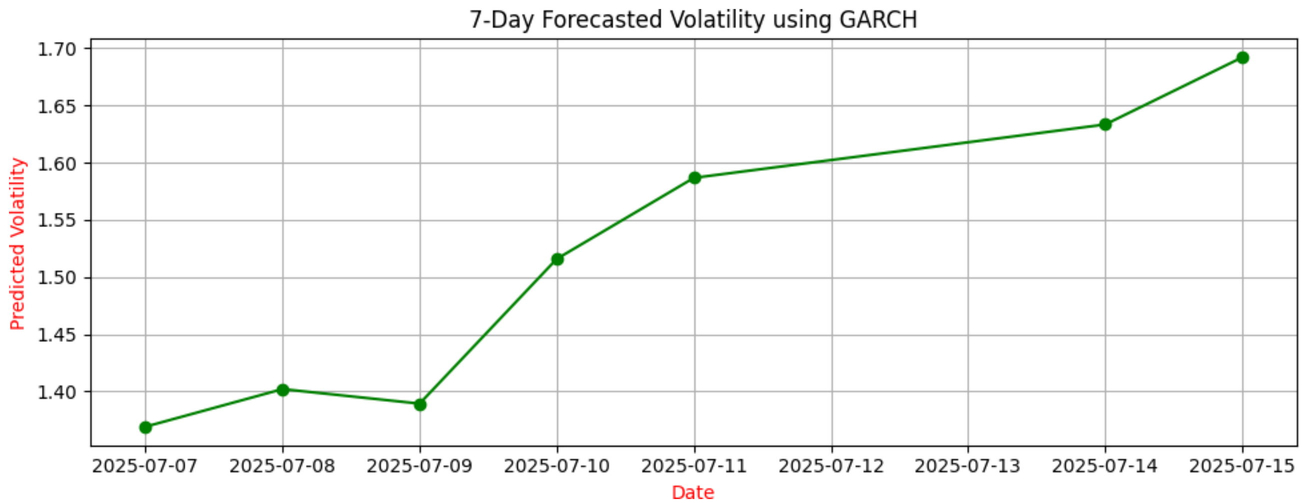
```
<matplotlib.legend.Legend at 0x7bc96f4c4710>
```



Next 7-Day Forecast:

```
# Used simulation-based forecast for horizon > 1 forecast_horizon = 7
forecast = model_fit.forecast(horizon=forecast_horizon, method='simulation')

# Get the forecasted variance (simulation returns mean forecast) predicted_volatility =
np.sqrt(forecast.variance.values[-1])

# Generate future business dates last_date = returns.index[-1]
future_dates = pd.bdate_range(start=last_date, periods=forecast_horizon + 1)[1:]

# Create a Series for predicted volatility
predicted_vol_series = pd.Series(predicted_volatility, index=future_dates)
```

```
# Plot the forecast
plt.figure(figsize=(10, 4))
plt.plot(predicted_vol_series, marker='o', color='green')
plt.title(f'{forecast_horizon}-Day Forecasted Volatility using GARCH')
plt.xlabel('Date', color='red')
plt.ylabel('Predicted Volatility', color='red')
plt.grid(True)
plt.tight_layout()
plt.show()
```



Computing the model Evaluation metrices:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Ensure both series are aligned
actual_vol = returns.rolling(window=7).std().dropna()[-365:]
predicted_vol = rolling_predictions

# Drop any NaNs to avoid metric calculation errors
actual_vol, predicted_vol = actual_vol.align(predicted_vol, join='inner')

# Compute metrics
mse = mean_squared_error(actual_vol, predicted_vol) rmse = np.sqrt(mse)
mae = mean_absolute_error(actual_vol, predicted_vol)

# Display metrics
print("Model Evaluation Metrics:\n")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
```

```
Model Evaluation Metrics:

Mean Squared Error (MSE): 0.4187
Root Mean Squared Error (RMSE): 0.6471
Mean Absolute Error (MAE): 0.4797
```

XGBoost:

```
# Importing the necessary libraries for XG Boost
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
import matplotlib.dates as mdates
```

Data preparation for XGBoost

Prepare the data by creating lag features and splitting it into training and testing sets.

```
# Create lag features for the 'Close' price
n_lags = 7
for i in range(1, n_lags + 1):
   df[f'lag_{i}'] = df['Close'].shift(i)

# Drop rows with NaN values resulting from the lag features
df.dropna(inplace=True)

# Define features (X) and target variable (y)
features = ['High', 'Low', 'Open', 'Volume'] + [f'lag_{i}' for i in range(1, n_lags + 1)]
X = df[['Date'] + features]

# Split data into training and testing sets (80/20 split) train_size = int(len(df) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Display the shapes of the train and test sets print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

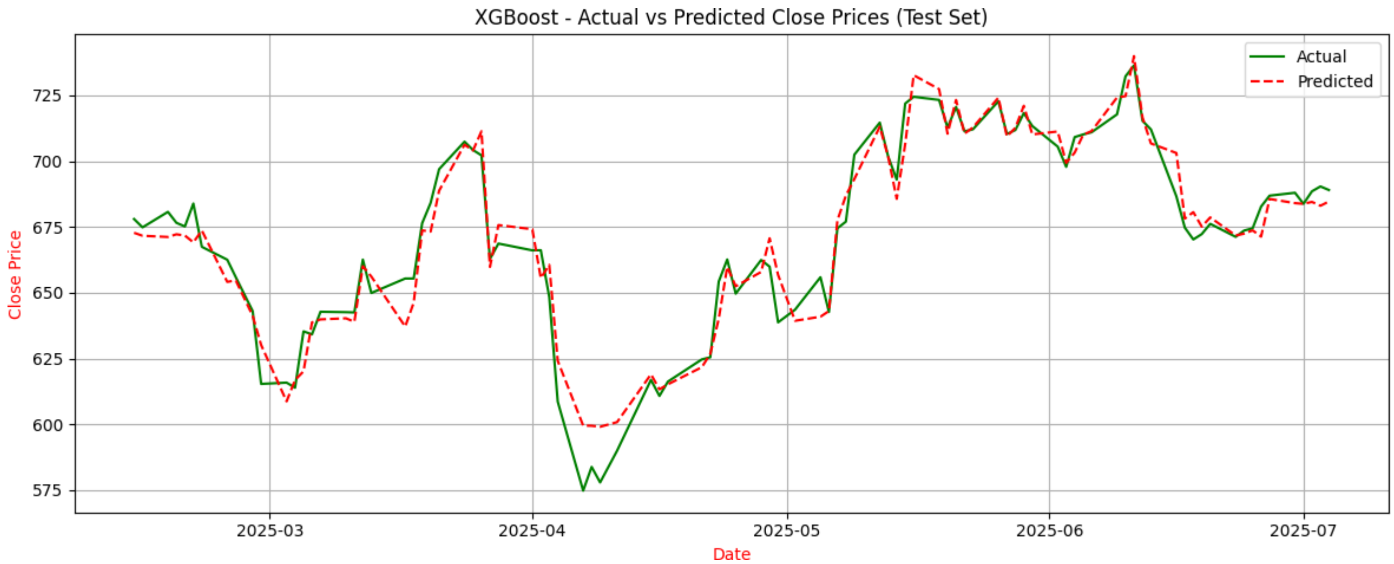
```
Shape of X_train: (377, 12)
Shape of X_test: (95, 12)
Shape of y_train: (377,)
Shape of y_test: (95,)
```

Model training and evaluation

Train the model on the training data, predict on the test data, and evaluate the model's performance with metrics and an actual vs forecast plot.

```
# Instantiate an XGBRegressor model
model = XGBRegressor(objective='reg:squarederror', n_estimators=100)

# Train the model and explicitly drop 'Date' column from X_train before fitting X_train_xgb =
X_train.drop('Date', axis=1)
model.fit(X_train_xgb, y_train)

# Make predictions on the test data and explicitly drop 'Date' column from X_test before predicting
X_test_xgb = X_test.drop('Date', axis=1)
y_pred_test = model.predict(X_test_xgb)
```

```
# Plot actual vs forecast for the test set
plt.figure(figsize=(12, 5))
plt.plot(X_test['Date'], y_test, label='Actual', color='green')
plt.plot(X_test['Date'], y_pred_test, label='Predicted', color='red', linestyle='--')
plt.xlabel('Date', color='red')
plt.ylabel('Close Price', color='red')
plt.title('XGBoost - Actual vs Predicted Close Prices (Test Set)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Next 7-day forecasting

Used the trained model to forecast the closing prices for the next 7 days and visualize the forecast.

```
# Forecast the next 7 days
n_forecast_days = 7
last_date = df['Date'].iloc[-1]

# Create a DataFrame for future dates and initialize with the last known Close price
future_dates = pd.date_range(start=last_date + pd.Timedelta(days=1), periods=n_forecast_days, freq='B')
# Use 'B' for business days

# Recursively forecast and update lag features forecast_xgb = []
last_row = df.iloc[-1].copy()
current_lags = last_row[[f'lag_{i}' for i in range(1, n_lags + 1)]].values.flatten().tolist()
last_close = last_row['Close']
```

```
for i in range(n_forecast_days):
    # Create input for the next prediction including High, Low, Open, and Volume from the last actual day
    # For future predictions, we'll use the last known High, Low, Open, Volume from the actual data for simplicity
    input_data = {
        'High': last_row['High'],
        'Low': last_row['Low'],
        'Open': last_row['Open'],
        'Volume': last_row['Volume']
    }

    for j in range(1, n_lags + 1):
        input_data[f'lag_{j}'] = current_lags[j-1]

    input_df = pd.DataFrame([input_data])

    # Predict the next day's close price
    next_pred = model.predict(input_df)[0]
    forecast_xgb.append(next_pred)

    # Update lags for the next iteration. The new lag_1 is the most recent prediction.
    current_lags.insert(0, next_pred)
    current_lags.pop()
    last_close = next_pred # Update last_close to the new prediction for the next iteration
```

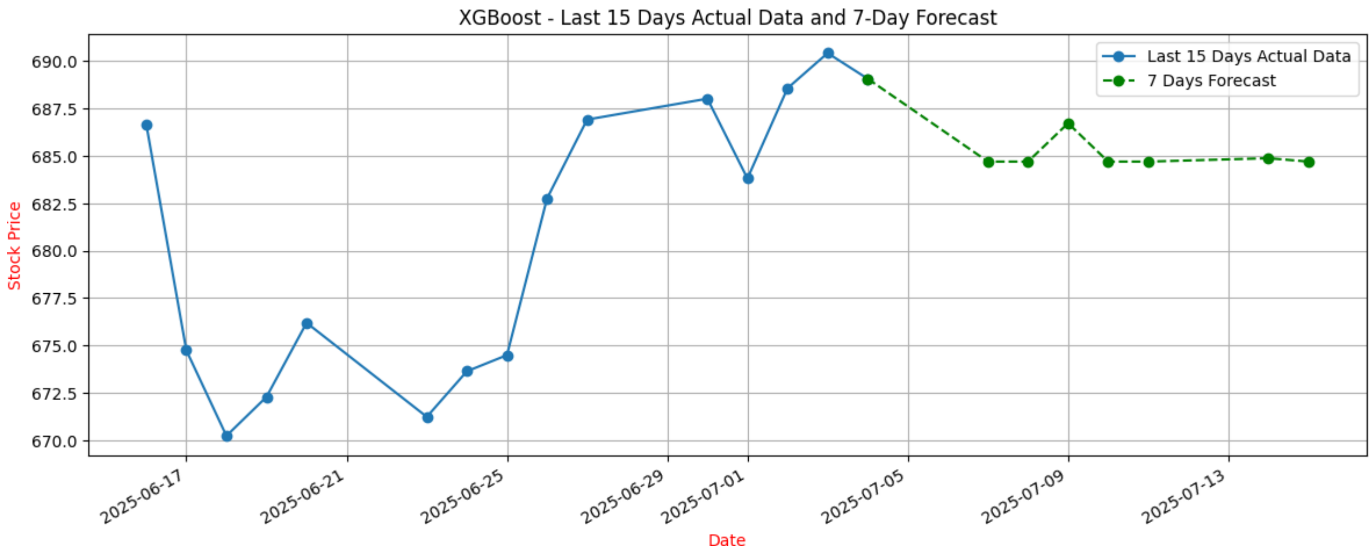
Creating the series for forecast

```
# Create a pandas Series for the forecast
forecast_series_xgb = pd.Series(forecast_xgb, index=future_dates)

# Add the last actual value to the start of the forecast for continuity in plotting
last_actual_date = df['Date'].iloc[-1]
last_actual_price = df['Close'].iloc[-1]

forecast_series_xgb_cont = pd.concat([
    pd.Series([last_actual_price], index=[last_actual_date]), forecast_series_xgb
])
```

Plotting the forecast

```
# Plot the forecast focusing on the last 15 days of actual data and the next 7 day
plt.figure(figsize=(12, 5))

# Plot last 15 days of actual data
plt.plot(df['Date'].iloc[-15:], df['Close'].iloc[-15:], label='Last 15 Days Actual Data', marker='o', linestyle='-')

# Use the forecast_series_xgb_cont which already includes the last actual point for continuity
plt.plot(forecast_series_xgb_cont.index, forecast_series_xgb_cont.values,
label='7 Days Forecast', color='green', marker='o', linestyle='--')
plt.title('XGBoost - Last 15 Days Actual Data and 7-Day Forecast')
plt.xlabel('Date', color='red')
plt.ylabel('Stock Price', color='red')
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
# Improve date formatting on x-axis
plt.gcf().autofmt_xdate()
date_form = mdates.DateFormatter('%Y-%m-%d') # Define date format
plt.gca().xaxis.set_major_formatter(date_form) # Apply date format
plt.show()
```



Calculating the Model Performance metrices

```
# Calculate evaluation metrics
print("XGBoost Model Performance Metrics:\n")

# MSE
mse = mean_squared_error(y_test, y_pred_test)
print(f"MSE: {mse:.2f}")

# RMSE
rmse = np.sqrt(mse)
print(f"RMSE: {rmse:.2f}")

# MAPE
mape = np.mean(np.abs((y_test - y_pred_test) / y_test)) * 100
print(f"MAPE: {mape:.2f}%")

# R² Score
r2 = r2_score(y_test, y_pred_test) print(f"R² Score: {r2:.4f}")
```

```
XGBoost Model Performance Metrics:

MSE: 65.66
RMSE: 8.10
MAPE: 0.92%
R² Score: 0.9512
```

LSTM Modelling:

```
# Installing the required libraries for LSTM modeling
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
```

Fetching the data from yfinance library and preprocessing the data for model training and future predictions

```
# Define dates to fetch stock
data todayLS = dt.date.today()
dLS1 = today.strftime("%Y-%m-%d")

enddateLS = todayLS

dLS2 = todayLS - timedelta(days=365*5)
dLS2 = dLS2.strftime("%Y-%m-%d")
startdateLS = dLS2

print("Start Date:", startdateLS)
print("End Date:", enddateLS)
```

```
Start Date: 2020-07-06
End Date: 2025-07-05
```

```
# Fetching stock data using yfinance
ticker = 'TATAMOTORS.NS'
dfLS = yf.download(ticker, start=startdateLS, end=enddateLS, progress = False)
dfLS.columns = dfLS.columns.droplevel('Ticker')


dfLS.insert(0, 'Date', dfLS.index)
dfLS.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1240 entries, 2020-07-06 to 2025-07-04 Data columns (total 6 columns):
```

#	Column	Non-Null Count	Dtype
0	Date	1240 non-null	datetime64[ns]
1	Close	1240 non-null	float64
2	High	1240 non-null	float64
3	Low	1240 non-null	float64
4	Open	1240 non-null	float64
5	Volume	1240 non-null	int64

```
dtypes: datetime64[ns](1), float64(4), int64(1) memory usage: 67.8 KB
```

```
# Plotting the closing price for the last 2 yrs data for LSTSM modeling
fig = px.line(dfLS, x='Date', y='Close', title=f'{ticker} Closing Price', width=1200, height=500)
fig.show()
```

Data preparation for LSTM

Prepare the `dfLS` data by scaling and creating sequences for the LSTM model.

```
# Select only the 'Close' column
dfLS = dfLS[['Close']]

# Drop any rows with missing values
dfLS.dropna(inplace=True)

# Set datetime as index
dfLS.index = pd.to_datetime(dfLS.index)


# Normalize data to [0, 1] range
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(dfLS)

# Convert to supervised learning format
def create_sequences(data, time_steps=60):
   X, y = [], []
   for i in range(time_steps, len(data)):
      X.append(data[i-time_steps:i, 0])
      y.append(data[i, 0])
   return np.array(X), np.array(y)

# Set time_steps time_steps = 90

# Create sequences
X, y = create_sequences(scaled_data, time_steps)

# Reshape input for LSTM [samples, time_steps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))

display(X.shape) display(y.shape)
```

```
(1150, 90, 1)
```

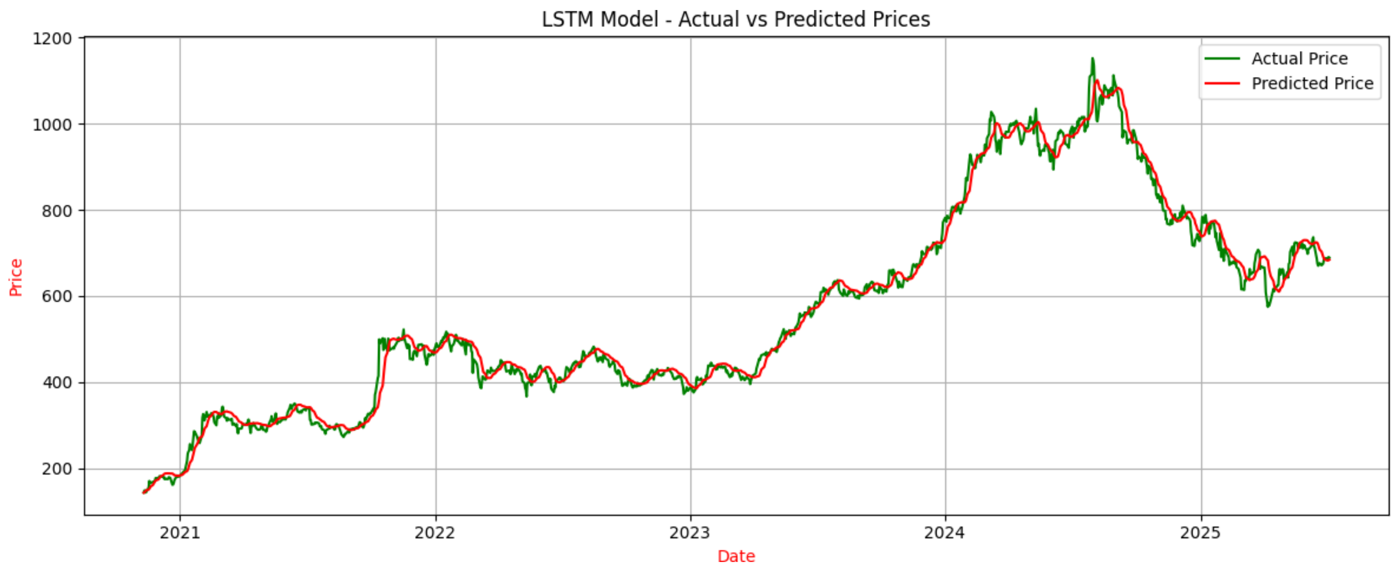
```
# Build LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True,
input_shape=(X.shape[1], 1))) model.add(LSTM(units=50))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X, y, epochs=20, batch_size=32, validation_split=0.1)
```

```
Epoch 1/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 7s 93ms/step - loss: 0.0870 val_loss: 0.0039
Epoch 2/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 86ms/step - loss: 0.0020 - val_loss: 9.5777e-04
Epoch 3/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 6s 112ms/step - loss: 7.0636e-04 - val_loss: 9.6775e-04
Epoch 4/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - loss: 6.1481e-04 - val_loss: 0.0012
Epoch 5/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 82ms/step - loss: 7.3934e-04 - val_loss: 9.5814e-04
Epoch 6/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 6s 109ms/step - loss: 7.2337e-04 - val_loss: 0.0012
Epoch 7/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 80ms/step - loss: 7.6310e-04 - val_loss: 8.9231e-04
Epoch 8/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 83ms/step - loss: 7.3037e-04 - val_loss: 8.9071e-04
Epoch 9/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 5s 90ms/step - loss: 6.4512e-04 - val_loss: 8.8493e-04
Epoch 10/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 94ms/step - loss: 5.9500e-04 - val_loss: 9.3367e-04
Epoch 11/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 5s 79ms/step - loss: 5.5041e-04 - val_loss: 9.0728e-04
Epoch 12/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 86ms/step - loss: 5.7900e-04 - val_loss: 8.3263e-04
Epoch 13/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 5s 91ms/step - loss: 6.1264e-04 - val_loss: 8.3419e-04
Epoch 14/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 5s 82ms/step - loss: 5.1489e-04 - val_loss: 8.1432e-04
Epoch 15/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 82ms/step - loss: 5.8941e-04 - val_loss: 9.1567e-04
Epoch 16/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 4s 109ms/step - loss: 5.6085e-04 - val_loss: 8.1969e-04
Epoch 17/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 4s 76ms/step - loss: 5.0585e-04 - val_loss: 0.0014
Epoch 18/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 5s 79ms/step - loss: 8.0429e-04 - val_loss: 8.1818e-04
Epoch 19/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 4s 130ms/step - loss: 6.1417e-04 - val_loss: 0.0011
Epoch 20/20
33/33 ━━━━━━━━━━━━━━━━━━━━ 3s 79ms/step - loss: 4.6229e-04 - val_loss: 8.8089e-04
<keras.src.callbacks.history.History at 0x7bc977a3b310>
```

```
# Predict on training data
predicted = model.predict(X)
predicted_prices = scaler.inverse_transform(predicted.reshape(-1, 1))
actual_prices = scaler.inverse_transform(y.reshape(-1, 1))

# Create aligned date index (skip first time_steps rows)
aligned_dates = dfLS.index[time_steps:]
```

```
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step
```

```
# Plot with correct dates
plt.figure(figsize=(12, 5))
plt.plot(aligned_dates, actual_prices, label='Actual Price', color='green')
plt.plot(aligned_dates, predicted_prices, label='Predicted Price', color='red')
plt.title('LSTM Model - Actual vs Predicted Prices')
plt.xlabel('Date', color='red')
plt.ylabel('Price', color='red')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Calculating the next 7-Day forecast data

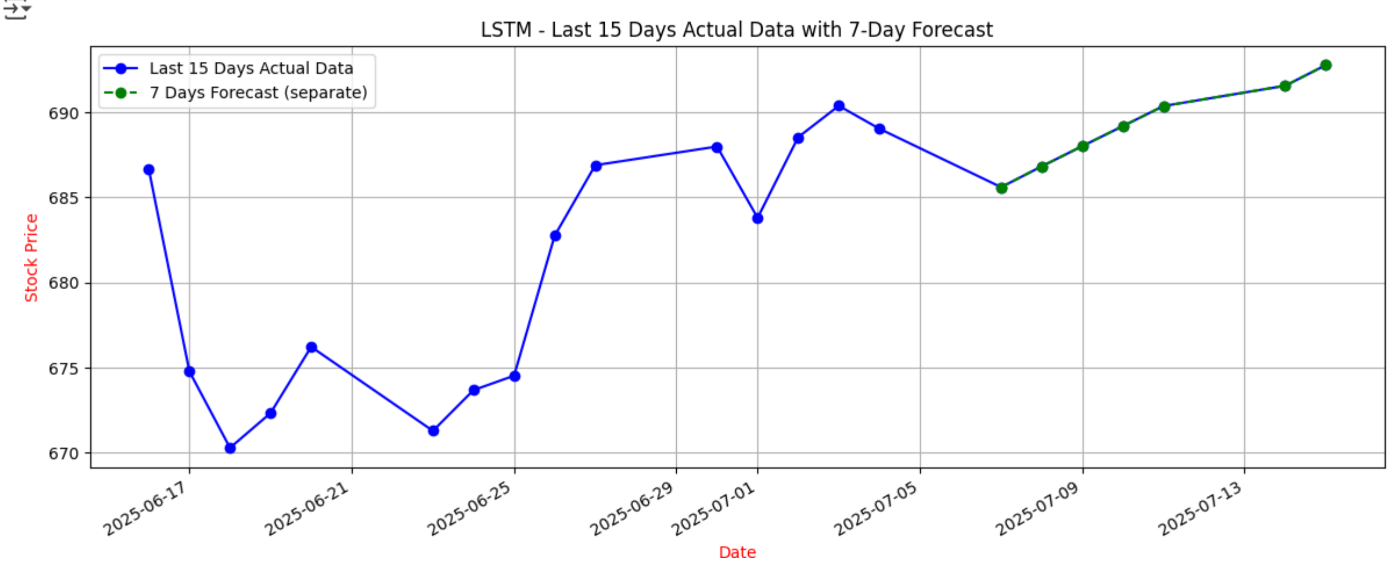
```
# Forecast next 7 days n_forecast_days = 7
last_sequence = scaled_data[-time_steps:].reshape(1, time_steps, 1)
forecasted_scaled_prices = []

for _ in range(n_forecast_days):
   next_pred_scaled = model.predict(last_sequence)[0, 0]
   forecasted_scaled_prices.append(next_pred_scaled)

   # Reshape next_pred_scaled to have 3 dimensions before appending
   next_pred_scaled_reshaped = np.array([[next_pred_scaled]]).reshape(1, 1, 1)
   last_sequence = np.append(last_sequence[:, 1:, :], next_pred_scaled_reshaped, axis=1)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
```

```
# Inverse transform the forecasted prices
forecasted_prices = scaler.inverse_transform(np.array(forecasted_scaled_prices).reshape(-1, 1))

# Create future dates for the forecast last_actual_date = dfLS.index[-1]
future_dates = pd.date_range(start=last_actual_date + pd.Timedelta(days=1), periods=n_forecast_days, freq='B')

# Create a Series for the forecast
forecast_series_lstm = pd.Series(forecasted_prices.flatten(), index=future_dates)
```

Now plotting the next 7-Day forecast data

```
# Plot the forecast focusing on the last 15 days of actual data and the next 7 day
plt.figure(figsize=(12, 5))

# Get the last 15 days of actual data
last_15_days_actual = dfLS.iloc[-15:]

# Created a combined series that includes the last actual point and the forecast
combined_plot_data_index = last_15_days_actual.index.tolist() + forecast_series_lstm.index.tolist()
combined_plot_data_values = last_15_days_actual['Close'].tolist() +
forecast_series_lstm.values.flatten().tolist()

# Plot the combined data
plt.plot(combined_plot_data_index, combined_plot_data_values, label='Last 15 Days Actual Data',
color='blue', marker='o', linestyle='-')

# Added separate markers for the forecast part for clarity
plt.plot(forecast_series_lstm.index, forecast_series_lstm.values, label=f'{n_forecast_days} Days Forecast
(separate)', color='green', marker='o', linestyle='--')

plt.title(f'LSTM - Last 15 Days Actual Data with {n_forecast_days}-Day Forecast')
plt.xlabel('Date', color='red')
plt.ylabel('Stock Price', color='red')
plt.legend()
plt.grid(True)
plt.tight_layout()

# Improve date formatting on x-axis
plt.gcf().autofmt_xdate()
date_form = mdates.DateFormatter('%Y-%m-%d') # Define date format
plt.gca().xaxis.set_major_formatter(date_form) # Apply date format
plt.show()
```



Calculating the Model Performance metrices

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Calculate evaluation metrics
print("LSTM Model Performance Metrics:\n")

# MSE
mse = mean_squared_error(actual_prices, predicted_prices)
print(f"MSE: {mse:.4f}")

# RMSE
rmse = np.sqrt(mse)
print(f"RMSE: {rmse:.4f}")

# MAE
mae = mean_absolute_error(actual_prices, predicted_prices)
print(f"MAE: {mae:.4f}")

# MAPE
# Handle potential division by zero
mape = np.mean(np.abs((actual_prices - predicted_prices) / actual_prices)) * 100
print(f"MAPE: {mape:.4f}%")

# R2 score
r2 = r2_score(actual_prices, predicted_prices)
print(f"R2 Score: {r2:.4f}")
```

```
LSTM Model Performance Metrics:
```

```
MSE: 583.1775
RMSE: 24.2111
MAE: 17.3278
MAPE: 3.2875%
R² Score: 0.9894
```