1. Introduction

Stock price prediction is a challenging task due to the highly volatile and nonlinear nature of financial markets. Traditional statistical models often fail to capture long-term dependencies in time-series data. In this project, we use **Long Short-Term Memory (LSTM)**, a type of recurrent neural network (RNN), to predict stock prices based on historical data.

Why LSTM?

LSTM is well-suited for stock price prediction because:

- It can learn and retain long-term dependencies in time-series data.
- It mitigates the vanishing gradient problem, which affects traditional RNNs.
- It captures sequential patterns and trends more effectively than standard machine learning models.

Project Workflow

1. **Data Collection**: Fetch historical stock price data using `yfinance`.
2. **Exploratory Data Analysis (EDA)**: Visualize trends, check correlations, and analyze patterns.
3. **Feature Engineering**: Compute technical indicators like Moving Averages, RSI, and MACD.
4. **Data Preprocessing**: Normalize data and prepare sequences for LSTM input.
5. **Model Development**: Build and train an LSTM-based deep learning model.
6. **Evaluation & Prediction**: Assess performance using metrics like MAPE, RMSE, and R^2 Score.
7. **Visualization & Insights**: Compare actual vs. predicted stock prices to analyze model effectiveness.

This project aims to provide a **data-driven approach to stock price forecasting**, helping investors and traders make more informed decisions.

📌 Importing Necessary Libraries

To build an **LSTM-based stock price prediction model**, we need libraries for **data handling, model training, and evaluation**. Here's why each category is important:

Data Handling & Collection

- `pandas & numpy` : Load, manipulate, and process stock data efficiently.
- `yfinance` : Fetch historical stock prices from Yahoo Finance.

Data Preprocessing

- `MinMaxScaler`, `StandardScaler` : Normalize data for better model performance.
- `train_test_split`, `TimeSeriesSplit` : Maintain time-series structure while splitting data.

LSTM Model Development

- `Sequential` : Builds a stack of deep learning layers.
- `LSTM` : Captures sequential patterns in stock prices.
- `Dense`, `Dropout` : Defines output layers and prevents overfitting.
- `EarlyStopping` : Stops training automatically when performance plateaus.

Evaluation Metrics

- `MSE`, `MAPE`, `MAE`, `R²` : Measure prediction accuracy and model performance.

Visualization

- `matplotlib` : Plot stock prices and predictions.
- `datetime` : Handle time-based financial data.

This structured approach ensures **efficient data processing, accurate predictions, and clear insights** for stock market forecasting.

```
# ==============================
# Importing Necessary Libraries
# ==============================

# Data Handling and Numerical Operations
import pandas as pd  # Data manipulation and processing
import numpy as np  # Numerical computations

# Data Collection
import yfinance as yf  # Fetching stock market data

# Data Preprocessing
from sklearn.preprocessing import MinMaxScaler, StandardScaler  # Feature scali
from sklearn.model_selection import train_test_split, TimeSeriesSplit  # Splitt

# Model Building using TensorFlow/Keras
from tensorflow.keras.models import Sequential  # Defines the deep learning moc
from tensorflow.keras.layers import Dense, LSTM, Dropout  # LSTM layer for time
from tensorflow.keras.callbacks import EarlyStopping  # Prevents overfitting by

# Model Evaluation Metrics
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error,

# Data Visualization
import matplotlib.pyplot as plt  # Plotting and visualizing stock trends and pr
import matplotlib.dates as mdates  # Handling date formatting for plots
import datetime as dt  # Working with timestamps in financial data
```

Data Gathering

We use `yfinance` to fetch historical stock data, including Open, High, Low, Close prices, and Volume.

```
import yfinance as yf
stock_data = yf.download("AAPL", start="2015-03-04", end="2022-03-31")
```

```
[*********************100%***********************]  1 of 1 completed
```

```
stock_data
```

Price	Close	High	Low	Open	Volume
Ticker	AAPL	AAPL	AAPL	AAPL	AAPL
Date					
2015-03-04	28.738100	28.966146	28.688917	28.863304	126665200
2015-03-05	28.261892	28.785053	28.116569	28.747046	226068400
2015-03-06	28.304371	28.923668	28.228357	28.706802	291368400
2015-03-09	28.425098	28.968383	27.960066	28.608428	354114000
2015-03-10	27.837107	28.442990	27.678370	28.261896	275426400
...	...	...	...	...	...
2022-03-24	171.482758	171.551709	167.680129	168.517486	90131400
2022-03-25	172.123123	172.674797	170.182402	171.295612	80546200
2022-03-28	172.990051	173.118109	169.443552	169.611024	90371900
2022-03-29	176.300110	176.349355	173.719041	174.063845	100589400
2022-03-30	175.127762	176.940410	174.073658	175.896167	92633200

1783 rows × 5 columns

Next steps: Generate code with `stock_data` View recommended plots New interactive sheet

This data will be used for analysis and model training.

EDA

Exploratory Data Analysis (EDA)

EDA helps us **understand trends, detect anomalies, and check data quality** before feeding it into the LSTM model. Here, we analyze **missing values, data distribution, and price trends** over time.

1️⃣ Checking Data Structure & Cleaning Column Names

Before proceeding, we inspect and clean column names for easier processing.

```
stock_data.columns
```

```
MultiIndex([( 'Close', 'AAPL'),
            (  'High', 'AAPL'),
            (   'Low', 'AAPL'),
            (  'Open', 'AAPL'),
            ('Volume', 'AAPL')],
           names=['Price', 'Ticker'])
```

If the column names have a multi-level index (which happens sometimes with `yfinance`), we **drop the extra level** to simplify them.

```
# Drop extra column level if present
stock_data.columns = stock_data.columns.droplevel(1)

# Remove column index name for clarity
stock_data.columns.name = None

# Ensure 'Date' is the index
stock_data.index.name = "Date"
print(stock_data.index)  # Verify if it's a DateTimeIndex
```

```
DatetimeIndex(['2015-03-04', '2015-03-05', '2015-03-06', '2015-03-09',
               '2015-03-10', '2015-03-11', '2015-03-12', '2015-03-13',
               '2015-03-16', '2015-03-17',
               ...
               '2022-03-17', '2022-03-18', '2022-03-21', '2022-03-22',
               '2022-03-23', '2022-03-24', '2022-03-25', '2022-03-28',
               '2022-03-29', '2022-03-30'],
              dtype='datetime64[ns]', name='Date', length=1783, freq=None)
```

2️⃣ Summary Statistics & Missing Values

We check basic statistics and identify any missing values.

```
# Summary statistics of numerical columns
stock_data.describe()

# Check for missing values
stock_data.isnull().sum()
```

	0
Close	0
High	0
Low	0
Open	0
Volume	0

dtype: int64

🔷 Why?

- `.describe()` helps us understand the range, mean, and distribution of stock prices.
- `.isnull().sum()` detects missing values that might affect the model.

3️⃣ Save Cleaned Data as CSV (For Reusability)

Saving the cleaned dataset helps **avoid re-fetching and processing** every time.

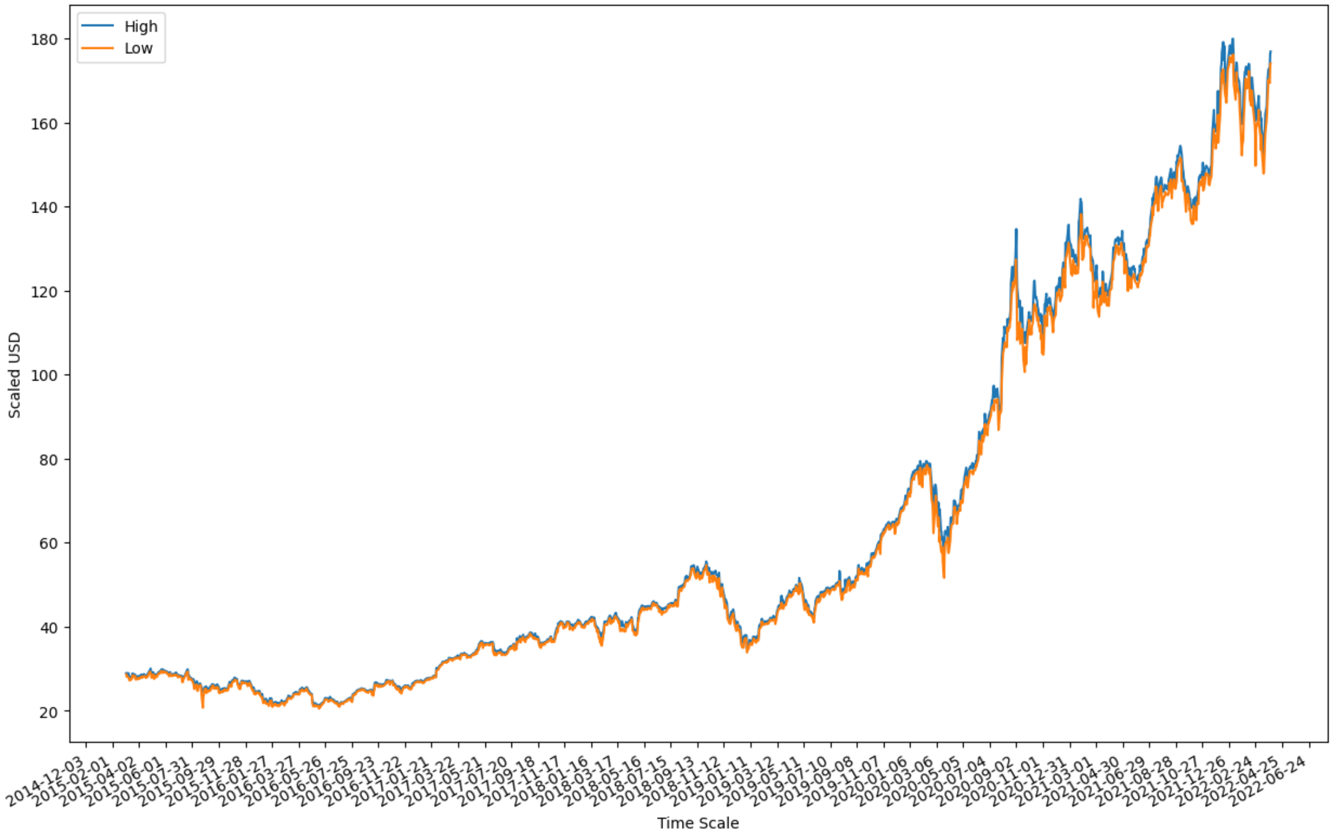
```
stock_data.to_csv("AAPL_cleaned.csv")
#saving the data as csv
```

4️⃣ Data Visualization: Stock Price Trends

📊 High & Low Price Trends Over Time

```
# Ensure index is in datetime format
stock_data.index = pd.to_datetime(stock_data.index)
```

```
plt.figure(figsize=(15, 10))

# Set date formatting and locator for X-axis
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.gca().xaxis.set_major_locator(mdates.DayLocator(interval=60))

# Use stock_data.index directly (it's already datetime)
x_dates = stock_data.index

# Plot High and Low prices
plt.plot(x_dates, stock_data['High'], label='High')
plt.plot(x_dates, stock_data['Low'], label='Low')

# Labels and legend
plt.xlabel('Time Scale')
plt.ylabel('Scaled USD')
plt.legend()

# Automatically format x-axis labels to prevent overlap
plt.gcf().autofmt_xdate()

# Show plot
plt.show()
```

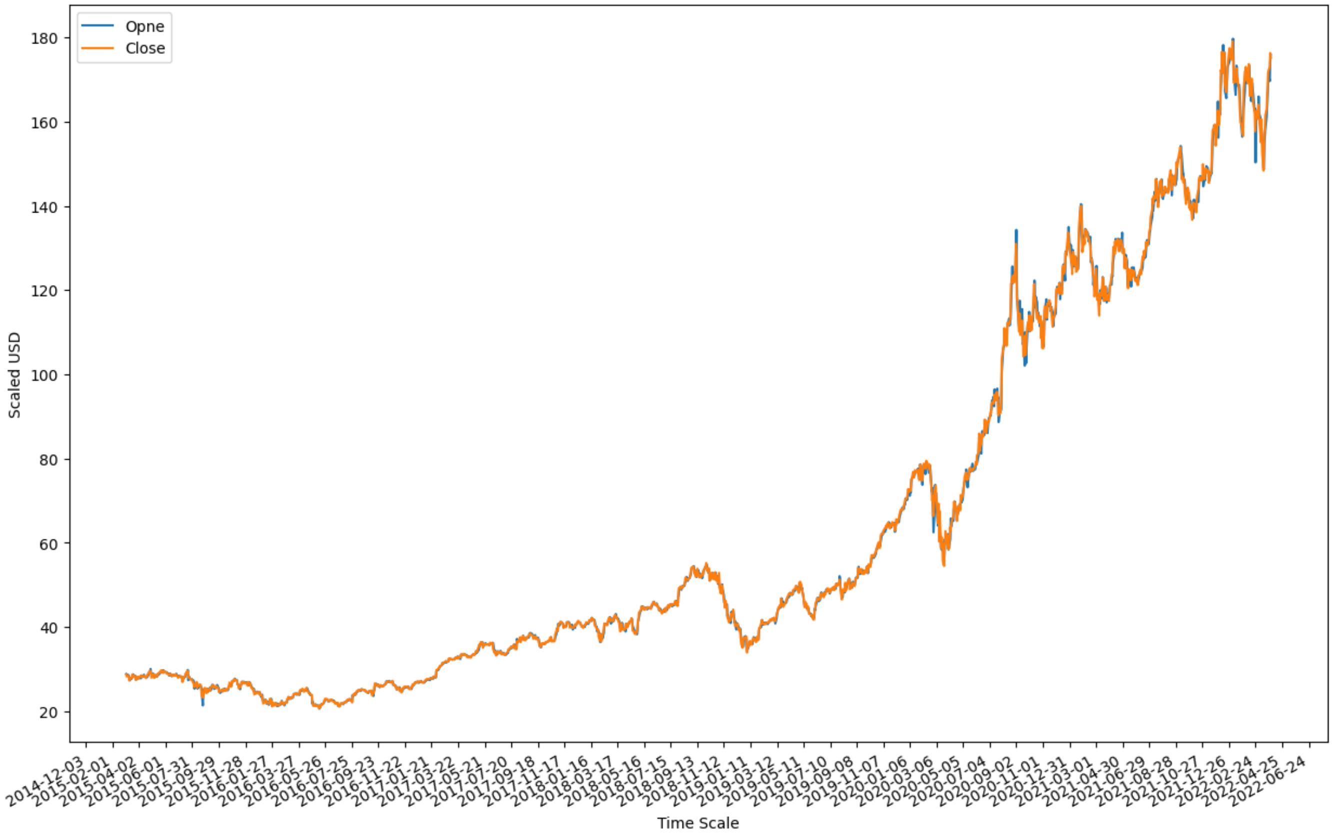


🔷 Why?

This plot **visualizes price fluctuations** and gives insights into volatility.

📊 Open & Close Price Trends Over Time

```
plt.figure(figsize=(15, 10))

# Set date formatting and locator for X-axis
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.gca().xaxis.set_major_locator(mdates.DayLocator(interval=60))

# Use stock_data.index directly (it's already datetime)
x_dates = stock_data.index

# Plot High and Low prices
plt.plot(x_dates, stock_data['Open'], label='Opne')
plt.plot(x_dates, stock_data['Close'], label='Close')

# Labels and legend
plt.xlabel('Time Scale')
plt.ylabel('Scaled USD')
plt.legend()

# Automatically format x-axis labels to prevent overlap
plt.gcf().autofmt_xdate()

# Show plot
plt.show()
```



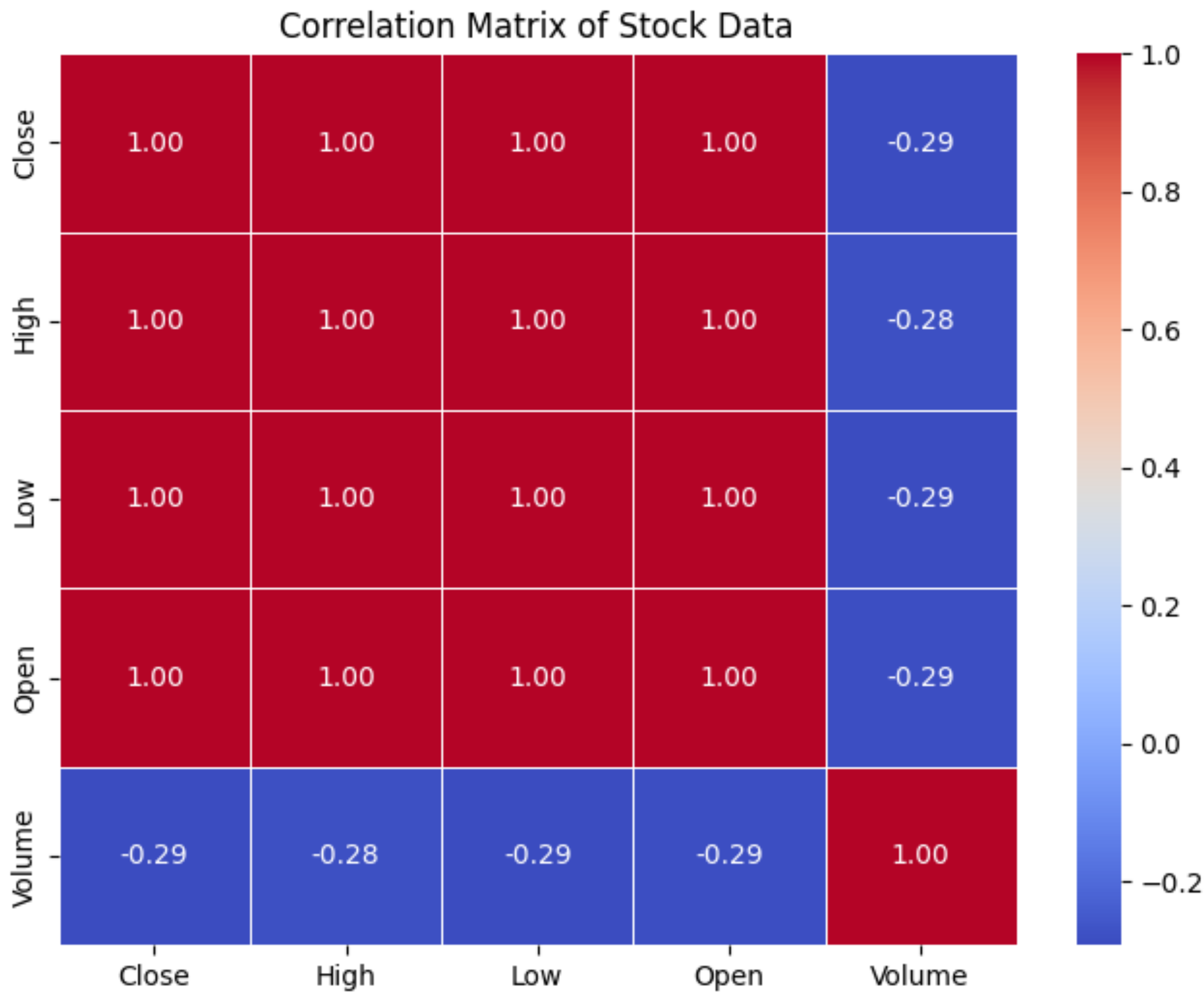
🔷 **Why?**

This helps **identify price gaps** between opening and closing prices, which may indicate market sentiment.

5️⃣ Correlation Analysis (Understanding Feature Relationships)

To check how different stock attributes are related, we use a **correlation matrix**.

```
import seaborn as sns

plt.figure(figsize=(8,6))
sns.heatmap(stock_data.corr(), annot=True, cmap='coolwarm', fmt=".2f", linewidth
plt.title("Correlation Matrix of Stock Data")
plt.show()
```

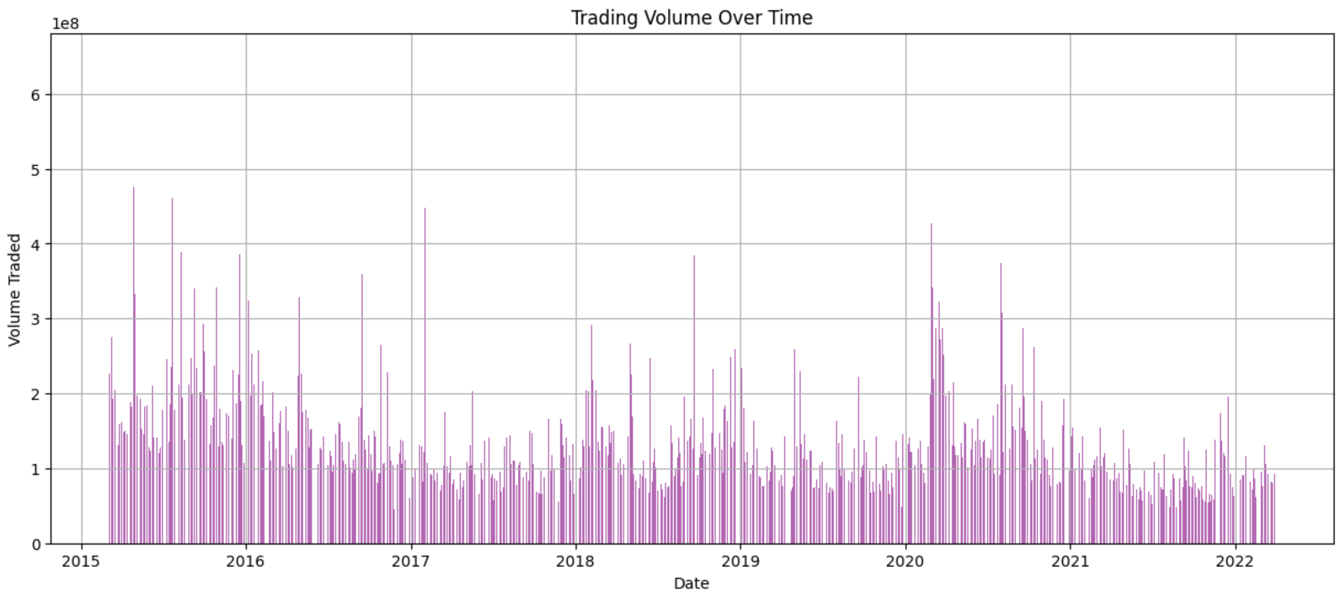


🔷 Why?

- Helps us **understand which features are most relevant** for predicting stock prices.
- Highly correlated features may contain **redundant information**, which could affect model performance.

6️⃣ Volume Analysis (Trading Activity Over Time)

```
plt.figure(figsize=(15, 6))

plt.bar(stock_data.index, stock_data['Volume'], color='purple', alpha=0.6)

plt.xlabel("Date")
plt.ylabel("Volume Traded")
plt.title("Trading Volume Over Time")
plt.grid()

plt.show()
```



🔷 **Why?**

- A **spike in volume** may indicate market events (earnings reports, news, etc.).
- Understanding volume trends helps in **identifying market sentiment and volatility**.

🔷 Summary

✔️ **Checked & cleaned** dataset for missing values and structure.
✔️ **Plotted stock price trends** (High, Low, Open, Close) for better understanding.
✔️ **Analyzed correlations** between different stock attributes.
✔️ **Visualized trading volume** to identify major market movements.

This EDA ensures that we move into **feature engineering and model training with clean and well-understood data**.

```
stock_data = pd.DataFrame(stock_data)
stock_data[['Open', 'Close']] = stock_data[['Close', 'Open']]
# stock_data.columns = ['Open','High','Low','Close','Volume']
# stock_data
```

```
stock_data.columns = ['Open','High','Low','Close','Volume']
stock_data
```

Date	Open	High	Low	Close	Volume
2015-03-04	28.863304	28.966146	28.688917	28.738100	126665200
2015-03-05	28.747046	28.785053	28.116569	28.261892	226068400
2015-03-06	28.706802	28.923668	28.228357	28.304371	291368400
2015-03-09	28.608428	28.968383	27.960066	28.425098	354114000
2015-03-10	28.261896	28.442990	27.678370	27.837107	275426400
...	...	...	...	...	...
2022-03-24	168.517486	171.551709	167.680129	171.482758	90131400
2022-03-25	171.295612	172.674797	170.182402	172.123123	80546200
2022-03-28	169.611024	173.118109	169.443552	172.990051	90371900
2022-03-29	174.063845	176.349355	173.719041	176.300110	100589400
2022-03-30	175.896167	176.940410	174.073658	175.127762	92633200

1783 rows × 5 columns

Next steps: Generate code with `stock_data` | View recommended plots | New interactive sheet

Feature Engineering & Data Preprocessing

Feature engineering enhances predictive power by adding **technical indicators**. We also preprocess data for **LSTM**, ensuring proper scaling and sequence creation.

5️⃣ Feature Engineering (Adding Technical Indicators)

We compute widely used **technical indicators** that help identify stock trends and momentum.

📌 Adding Technical Indicators

```
# Technical indicators functions remain the same
def add_moving_averages(df, window=10):
    result = df.copy()
    result[f'SMA_{window}'] = result['Close'].rolling(window=window).mean()
    result[f'EMA_{window}'] = result['Close'].ewm(span=window, adjust=False).me
    return result

def compute_rsi(df, window=14):
    result = df.copy()
    delta = result['Close'].diff(1)
    gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()
    rs = gain / loss
    result['RSI'] = 100 - (100 / (1 + rs))
    return result

def compute_macd(df):
    result = df.copy()
    short_ema = result['Close'].ewm(span=12, adjust=False).mean()
    long_ema = result['Close'].ewm(span=26, adjust=False).mean()
    result['MACD'] = short_ema - long_ema
    result['MACD_Signal'] = result['MACD'].ewm(span=9, adjust=False).mean()
    return result

def compute_bollinger_bands(df, window=20):
    result = df.copy()
    result['BB_Mid'] = result['Close'].rolling(window=window).mean()
    result['BB_Upper'] = result['BB_Mid'] + (result['Close'].rolling(window=wir
    result['BB_Lower'] = result['BB_Mid'] - (result['Close'].rolling(window=wir
    return result

def compute_volatility(df, window=10):
    result = df.copy()
    result['Volatility'] = result['Close'].rolling(window=window).std()
    return result

# Add all technical indicators
stock_data_with_indicators = stock_data.copy()
stock_data_with_indicators = add_moving_averages(stock_data_with_indicators)
stock_data_with_indicators = compute_rsi(stock_data_with_indicators)
stock_data_with_indicators = compute_macd(stock_data_with_indicators)
stock_data_with_indicators = compute_bollinger_bands(stock_data_with_indicators
stock_data_with_indicators = compute_volatility(stock_data_with_indicators)
```

```
# Handle missing values
stock_data_with_indicators.fillna(method='ffill', inplace=True)
stock_data_with_indicators.fillna(method='bfill', inplace=True)
```

```
<ipython-input-15-6b9665e10a84>:2: FutureWarning: DataFrame.fillna with 'me
  stock_data_with_indicators.fillna(method='ffill', inplace=True)
<ipython-input-15-6b9665e10a84>:3: FutureWarning: DataFrame.fillna with 'me
  stock_data_with_indicators.fillna(method='bfill', inplace=True)
```

🔷 **Why?**

- **Moving Averages (SMA & EMA)** smooth out price fluctuations.
- **RSI** identifies overbought/oversold conditions.
- **MACD** signals trend reversals.
- **Bollinger Bands** show volatility and price range.
- **Volatility** indicates market stability.

6️⃣ Data Preprocessing for LSTM

LSTM requires **scaled numerical input** and **sequential data formatting** for proper learning.

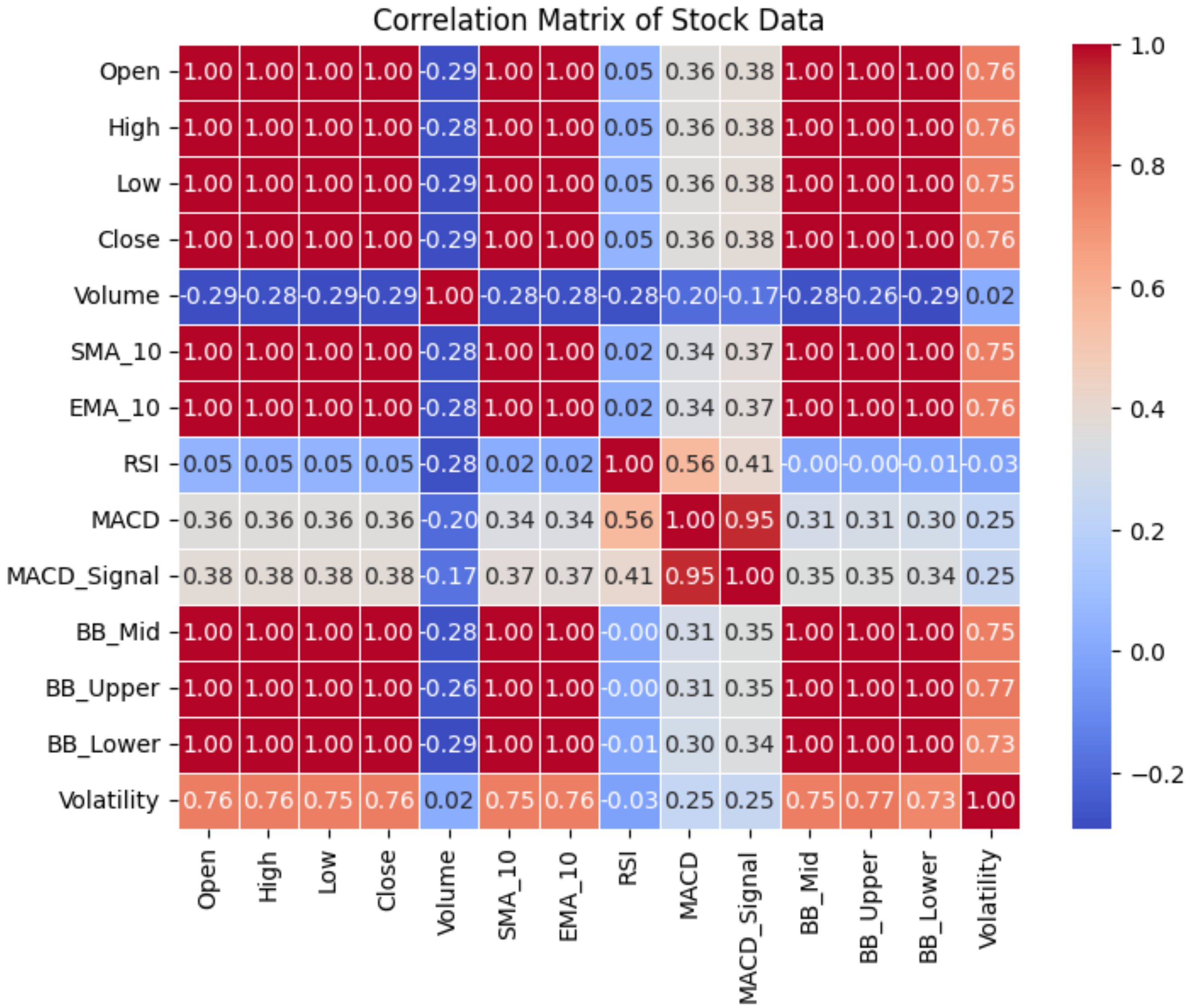
📊 Correlation Analysis with Heatmap

Now, we visualize feature correlations to validate our selection.

📌 **What this does:**

- Helps **visualize relationships between features**.
- Confirms that we have **reduced redundant features**.
- Ensures selected features **are not strongly correlated**, improving model robustness.

```
import seaborn as sns

plt.figure(figsize=(8,6))
sns.heatmap(stock_data_with_indicators.corr(), annot=True, cmap='coolwarm', fmt
plt.title("Correlation Matrix of Stock Data")
plt.show()
```



📌 Feature Selection & Correlation Analysis

To improve our LSTM model, we analyze feature correlations to **reduce redundancy and multicollinearity**, ensuring better generalization.

🔍 Feature Selection Analysis

1️⃣ Price-Related Features (Open, High, Low, Close)

✔ **Problem: Perfect correlation (1.0)** among all price features → causes redundancy.
✔ **Solution: Remove Open, High, and Low** since we are predicting Close.

2️⃣ Volume

✔ **Problem:** None significant, low correlation (~ -0.29 with Close).
✔ **Solution: Keep Volume** as it provides independent trading activity information.

3️⃣ Moving Averages (SMA_10, EMA_10)

✔ **Problem: Perfect correlation (1.0) between SMA and EMA, high correlation with BB_Mid**.
✔ **Solution: Remove both** since **BB_Mid provides similar trend information**.

4️⃣ RSI (Relative Strength Index)

✔ **Problem:** No major correlation issues.
✔ **Solution: Keep RSI**, as it provides momentum information.

5️⃣ MACD and MACD_Signal

✔ **Problem: High correlation (0.95) between them**.
✔ **Solution: Keep MACD, remove MACD_Signal** (MACD is slightly more independent).

6️⃣ Bollinger Bands (BB_Mid, BB_Upper, BB_Lower)

✔ **Problem: Perfect correlation between bands**.
✔ **Solution: Keep only BB_Mid**, as it represents the price trend.

7️⃣ Volatility

✔ **Problem:** Moderate correlation (~0.73-0.77) with BB components.
✔ **Solution: Keep Volatility**, as it provides unique market volatility insights.

✅ Final Feature Set

```
features = ['Volume', 'RSI', 'MACD', 'BB_Mid', 'Volatility']
```

💡 **Why this feature set?**
✔️ **Balances information on price, volume, momentum, and volatility**.
✔️ **Minimizes multicollinearity**, preventing overfitting.

✔ **Improves generalization**, ensuring stable training/testing performance.
✔ **More stable predictions**, reducing gaps between training and testing errors.

🔷 Summary

✔ Selected **5 key features** (Volume, RSI, MACD, BB_Mid, Volatility).
✔ Removed **highly correlated and redundant features** (Open, High, Low, SMA, EMA, MACD_Signal, BB_Upper, BB_Lower).
✔ Used a **correlation heatmap** to validate feature independence.

This refined feature set improves **model efficiency, reduces overfitting, and enhances predictive accuracy**. 🚀

📌 Feature & Target Separation

```
# Separate features and target
# features = ['Open', 'High', 'Low', 'Volume', 'SMA_10', 'EMA_10', 'RSI',
#             'MACD', 'MACD_Signal', 'BB_Mid', 'BB_Upper', 'BB_Lower', 'Volatili
features = ['Volume', 'RSI', 'MACD', 'BB_Mid', 'Volatility']
target = 'Close'
X = stock_data_with_indicators[features]
y = stock_data_with_indicators[target]
```

🔷 Why?

- We **remove Close price** from features since it's our target.
- Selected indicators help improve **model accuracy**.

📌 Scaling Features & Target

LSTMs perform better with **normalized inputs**, preventing large-value dominance.

```
# Scale features and target separately
feature_scaler = MinMaxScaler()
target_scaler = MinMaxScaler()

X_scaled = feature_scaler.fit_transform(X)
y_scaled = target_scaler.fit_transform(y.values.reshape(-1, 1))
```

🔷 **Why?**

- MinMax scaling ensures **all values are between 0 and 1**, preventing bias.
- **Separate scaling for target** avoids information leakage.

📌 Creating Sequential Data for LSTM

LSTM requires input as **sequences** rather than individual rows.

```
# Prepare sequences for LSTM
def create_sequences(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        Xs.append(X[i:(i + time_steps)])
        ys.append(y[i + time_steps])
    return np.array(Xs), np.array(ys)

# Create sequences with 5 time steps
time_steps = 5
X_seq, y_seq = create_sequences(X_scaled, y_scaled, time_steps)
```

🔷 **Why?**

- Converts raw time-series into **sequential inputs** for LSTM.
- Helps **capture short-term price trends** for better predictions.

📌 Train-Test Split

We **split data (80-20)** to train the model on past trends while testing generalization.

```
# Train-test split (80-20)
train_size = int(len(X_seq) * 0.8)
X_train, X_test = X_seq[:train_size], X_seq[train_size:]
y_train, y_test = y_seq[:train_size], y_seq[train_size:]
```

🔷 Why?

- Ensures **model learns from past trends** but **generalizes to unseen data**.
- Maintains **time order** since LSTM needs sequential data.

🔷 Summary

✔️ **Added technical indicators** to enhance feature richness.
✔️ **Normalized data** to improve model performance.
✔️ **Converted data into sequences** suitable for LSTM.
✔️ **Split dataset** to ensure generalization.

Now, we can proceed to **building and training the LSTM model!**

📌 Building, Training & Evaluating the LSTM Model

Now that our data is preprocessed, we **build, train, and evaluate** an LSTM model to predict stock prices.

7️⃣ Building the LSTM Model

We define an **LSTM model** with:

- **Two stacked LSTM layers** to capture sequential patterns.
- **Dropout layers** to prevent overfitting.
- **Dense layer** to output a single predicted value.

```
# Build LSTM model
model = Sequential([
    LSTM(64, input_shape=(time_steps, len(features)), activation='tanh', returr
    Dropout(0.2),
    LSTM(32, activation='tanh', return_sequences=False),
    Dropout(0.2),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: Us
  super().__init__(**kwargs)
```

🔷 **Why?**

- `LSTM(64, return_sequences=True)` : Captures long-term dependencies.
- `Dropout(0.3)` : Prevents overfitting.
- `Dense(1)` : Outputs a single stock price prediction.
- `Adam Optimizer` : Adaptive learning rate for efficient training.

8️⃣ Training the Model

We train the model using **early stopping** to prevent overfitting.

```
# Train model with early stopping
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='loss', patience=10, restore_best_weight

history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    verbose=1,
    callbacks=[early_stopping],
    shuffle=False
)
```

```
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0017
Epoch 27/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 5.5362e-04
```

```
Epoch 28/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 5.5130e-04
Epoch 29/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 2.1360e-04
Epoch 30/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 2.3954e-04
Epoch 31/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 2.4512e-04
Epoch 32/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 5.7247e-04
Epoch 33/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 0.0011
Epoch 34/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 2.9361e-04
Epoch 35/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 1.6877e-04
Epoch 36/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 5.9674e-04
Epoch 37/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 2.8169e-04
Epoch 38/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 4.5450e-04
Epoch 39/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 3.2584e-04
Epoch 40/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 3.5302e-04
Epoch 41/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0013
Epoch 42/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 3.0384e-04
Epoch 43/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 2.7414e-04
Epoch 44/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 1.6251e-04
Epoch 45/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.5542e-04
Epoch 46/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.8240e-04
Epoch 47/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 2.6598e-04
Epoch 48/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.7585e-04
Epoch 49/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 2.0920e-04
Epoch 50/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.3349e-04
Epoch 51/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 2.2345e-04
Epoch 52/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 9.5906e-04
Epoch 53/100
45/45 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 5.0298e-04
Epoch 54/100
```

```
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 3.0809e-04
Epoch 55/100
```

🔷 Why?

- `EarlyStopping` stops training if no improvement is seen for **10 epochs**.
- `batch_size=32` : Balances performance and computational efficiency.

9️⃣ Model Evaluation

We **make predictions** on both training and test sets, then calculate error metrics.

📌 Making Predictions & Inversing Scaling

```
# Make predictions
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Inverse transform predictions
train_predictions = target_scaler.inverse_transform(train_predictions)
test_predictions = target_scaler.inverse_transform(test_predictions)

# Get actual values
train_actual = y[:train_size]
test_actual = y[train_size:len(test_predictions) + train_size]

# Calculate metrics
train_mse = mean_squared_error(train_actual, train_predictions)
test_mse = mean_squared_error(test_actual, test_predictions)
train_mape = mean_absolute_percentage_error(train_actual, train_predictions)
test_mape = mean_absolute_percentage_error(test_actual, test_predictions)
```

```
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
```

📌 Calculate Evaluation Metrics

```
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error,
import numpy as np

# Calculate error metrics
train_mse = mean_squared_error(train_actual, train_predictions)
test_mse = mean_squared_error(test_actual, test_predictions)
train_mape = mean_absolute_percentage_error(train_actual, train_predictions)
test_mape = mean_absolute_percentage_error(test_actual, test_predictions)

train_mae = mean_absolute_error(train_actual, train_predictions)
test_mae = mean_absolute_error(test_actual, test_predictions)
train_rmse = np.sqrt(train_mse)
test_rmse = np.sqrt(test_mse)
train_r2 = r2_score(train_actual, train_predictions)
test_r2 = r2_score(test_actual, test_predictions)

# Print results
print(f"\nTraining MSE: {train_mse:.2f}")
print(f"Testing MSE: {test_mse:.2f}")
print(f"Training MAPE: {train_mape:.2%}")
print(f"Testing MAPE: {test_mape:.2%}")

print(f"\nTraining MAE: {train_mae:.2f}")
print(f"Testing MAE: {test_mae:.2f}")
print(f"Training RMSE: {train_rmse:.2f}")
print(f"Testing RMSE: {test_rmse:.2f}")
print(f"Training R²: {train_r2:.2f}")
print(f"Testing R²: {test_r2:.2f}")
```

```
Training MSE: 15.15
Testing MSE: 23.52
Training MAPE: 8.20%
Testing MAPE: 2.82%

Training MAE: 3.40
Testing MAE: 3.98
Training RMSE: 3.89
Testing RMSE: 4.85
Training R²: 0.97
Testing R²: 0.93
```

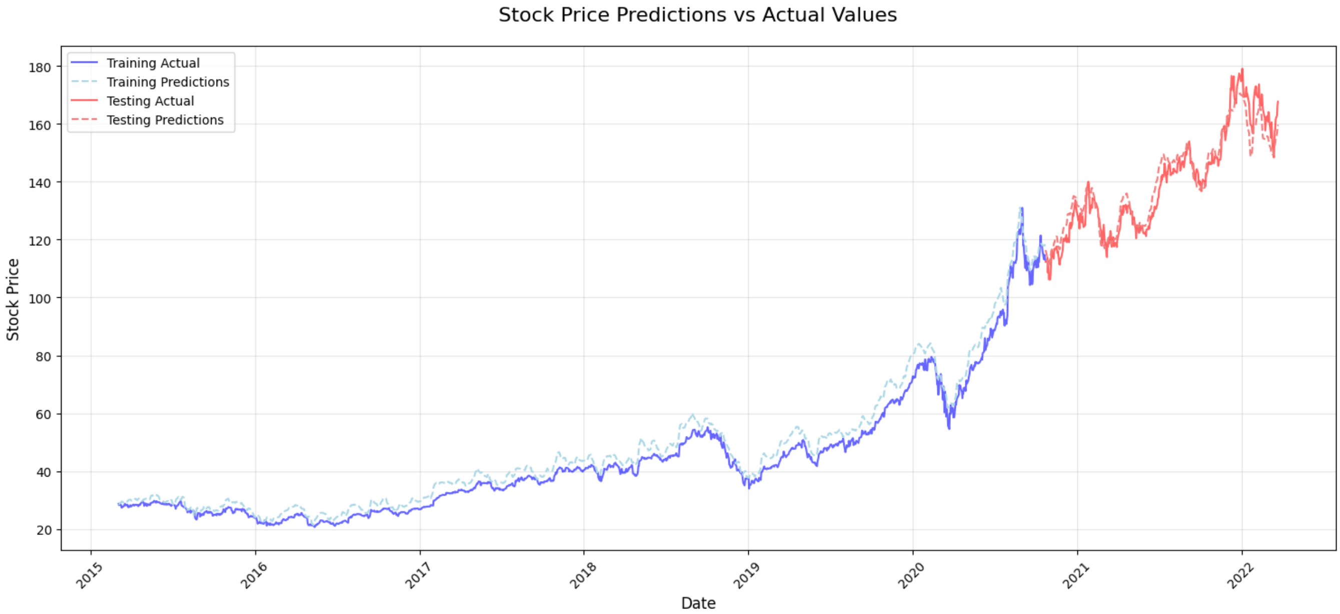
🔷 **Why?**

- **MSE & RMSE**: Penalizes large errors.
- **MAPE**: Measures percentage error, useful for financial data.
- **R^2 Score**: Checks how well the model explains variance (**1.0 is perfect**).

🔟 Visualizing Predictions

We plot **actual vs predicted stock prices** to assess model performance.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 7))

# Plot training predictions
plt.plot(train_actual.index, train_actual.values, label='Training Actual', colo
plt.plot(train_actual.index, train_predictions, label='Training Predictions', c

# Plot testing predictions
plt.plot(test_actual.index, test_actual.values, label='Testing Actual', color='
plt.plot(test_actual.index, test_predictions, label='Testing Predictions', colo

# Titles and labels
plt.title('Stock Price Predictions vs Actual Values', fontsize=16, pad=20)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Stock Price', fontsize=12)
plt.legend(loc='best', fontsize=10)
plt.grid(True, alpha=0.3)
plt.xticks(rotation=45)
plt.tight_layout()

plt.show()
```



🔷 Why?

- Helps **visually compare model predictions** with actual stock prices.
- Detects **patterns & inconsistencies** in model performance.

🔷 Summary

✔️ **Built an LSTM model** with optimized architecture.
✔️ **Trained using early stopping** to prevent overfitting.
✔️ **Evaluated using MSE, MAPE, RMSE, and R^2 Score**.
✔️ **Visualized actual vs predicted stock prices** for validation.

This completes our **LSTM-based stock price prediction model!**

```
# model.save("lstm_stock_model_88.h5")
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```