task-1-stock-prediction

September 14, 2023

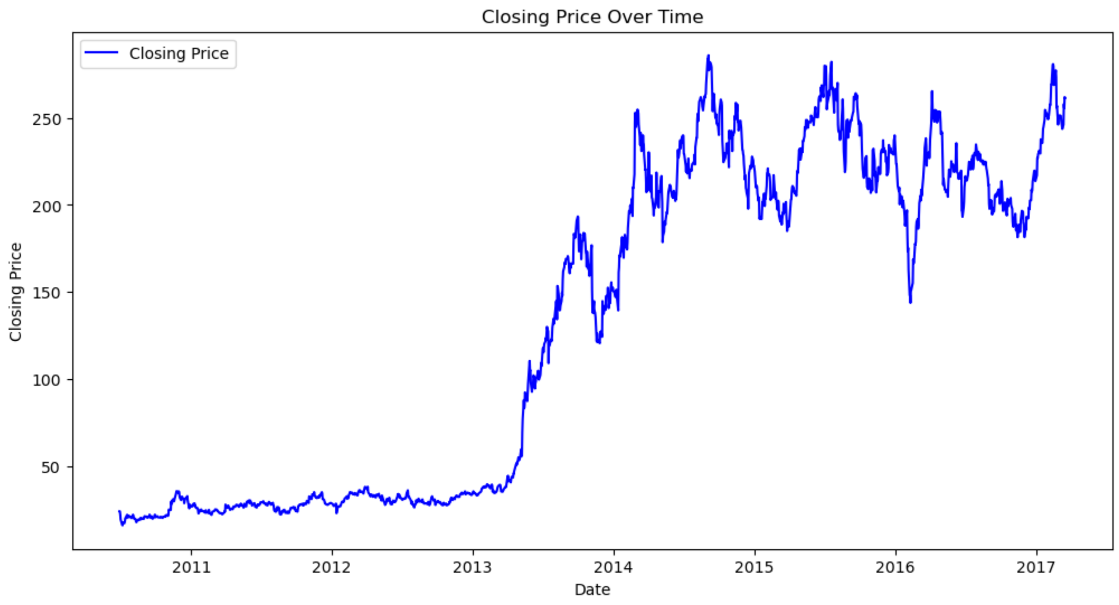
1 TASK - 1

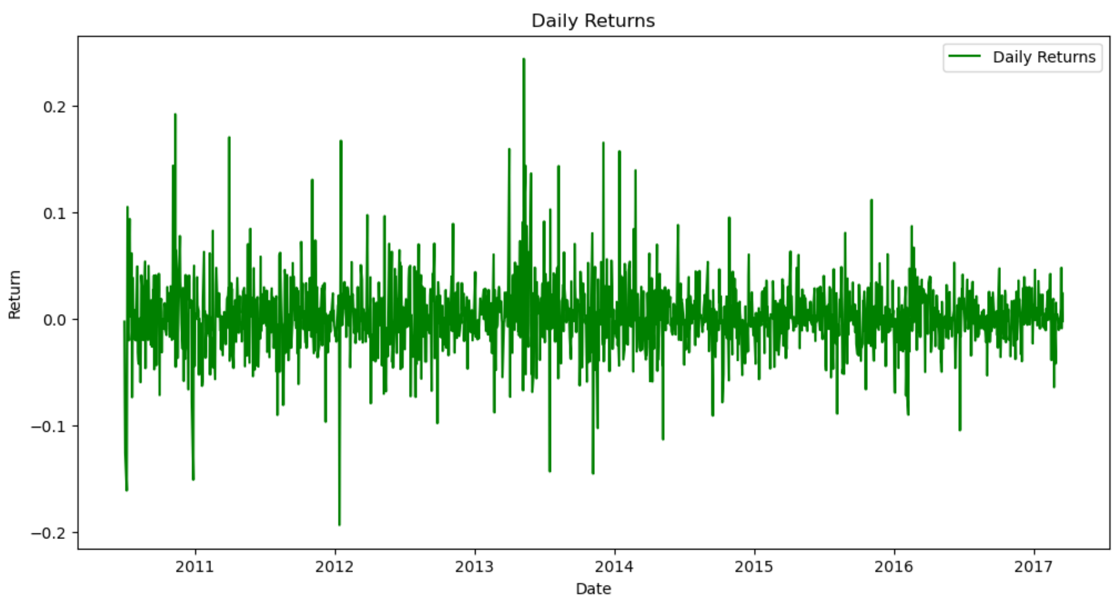
2 Stock Prediction

DATASET USED :- https://www.kaggle.com/datasets/rpaguirre/tesla-stock-price

[5]:
```
# Step 1: Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Step 2: Load the dataset
data = pd.read_csv("C:\\Users\\Narthana\\Downloads\\Tesla.csv - Tesla.csv.csv")
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)

# Step 3: Perform EDA

# Plot the closing price over time
plt.figure(figsize=(12, 6))
plt.title('Closing Price Over Time')
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.plot(data['Close'], label='Closing Price', color='blue')
plt.legend()
plt.show()

# Calculate and plot daily returns
daily_returns = data['Close'].pct_change().dropna()

plt.figure(figsize=(12, 6))
plt.title('Daily Returns')
plt.xlabel('Date')
plt.ylabel('Return')
plt.plot(daily_returns, label='Daily Returns', color='green')
plt.legend()
plt.show()
```

```python
# Calculate basic statistics
mean_return = daily_returns.mean()
std_return = daily_returns.std()
min_return = daily_returns.min()
max_return = daily_returns.max()

# Print the statistics
print(f"Mean Daily Return: {mean_return:.4f}")
print(f"Standard Deviation of Daily Return: {std_return:.4f}")
print(f"Minimum Daily Return: {min_return:.4f}")
print(f"Maximum Daily Return: {max_return:.4f}")
```





```
Mean Daily Return: 0.0020
Standard Deviation of Daily Return: 0.0329
Minimum Daily Return: -0.1933
Maximum Daily Return: 0.2440
```

Inferences from EDA

Inference 1: The closing price of the stock has shown both upward and downward trends over time. Inference 2: Daily returns fluctuate around a mean daily return value. Inference 3: The mean daily return gives an estimate of the average daily return of the stock. Inference 4: The standard deviation of daily returns measures the volatility or risk of the stock.

[7]:
```
# Get the number of rows and columns
num_rows, num_columns = data.shape

# Print the number of rows and columns
print(f"Number of rows: {num_rows}")
print(f"Number of columns: {num_columns}")
```

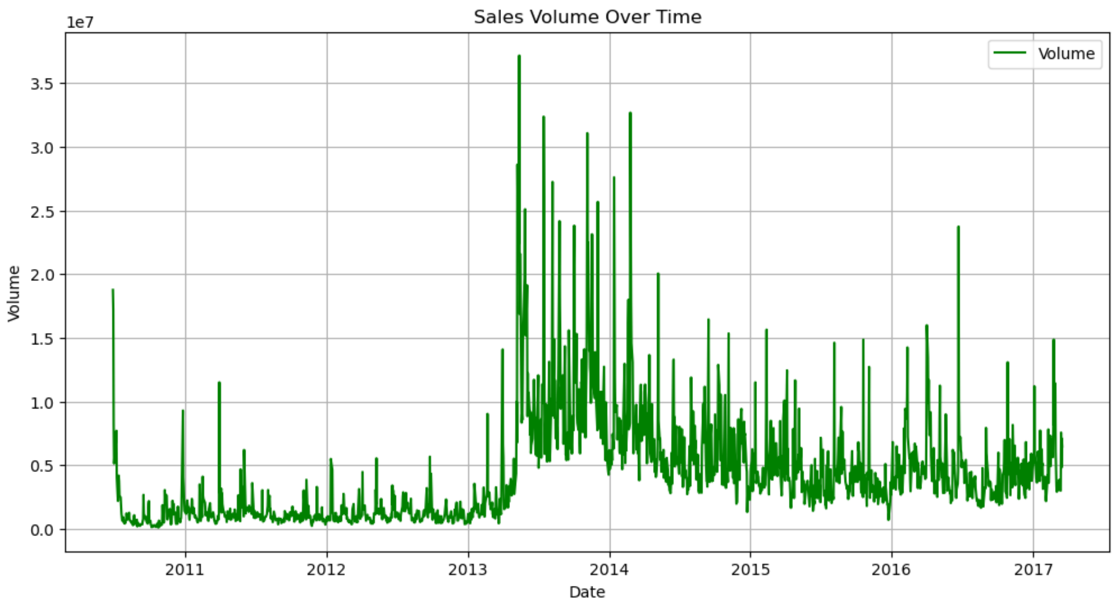
```
Number of rows: 1692
Number of columns: 6
```

[8]:
```
# Print the first few rows of the dataset
print(data.head())
```

```
                 Open   High        Low      Close    Volume  Adj Close
Date
2010-06-29  19.000000  25.00  17.540001  23.889999  18766300  23.889999
2010-06-30  25.790001  30.42  23.299999  23.830000  17187100  23.830000
```

```
2010-07-01  25.000000  25.92  20.270000  21.959999   8218800  21.959999
2010-07-02  23.000000  23.10  18.709999  19.200001   5139800  19.200001
2010-07-06  20.000000  20.00  15.830000  16.110001   6866900  16.110001
```

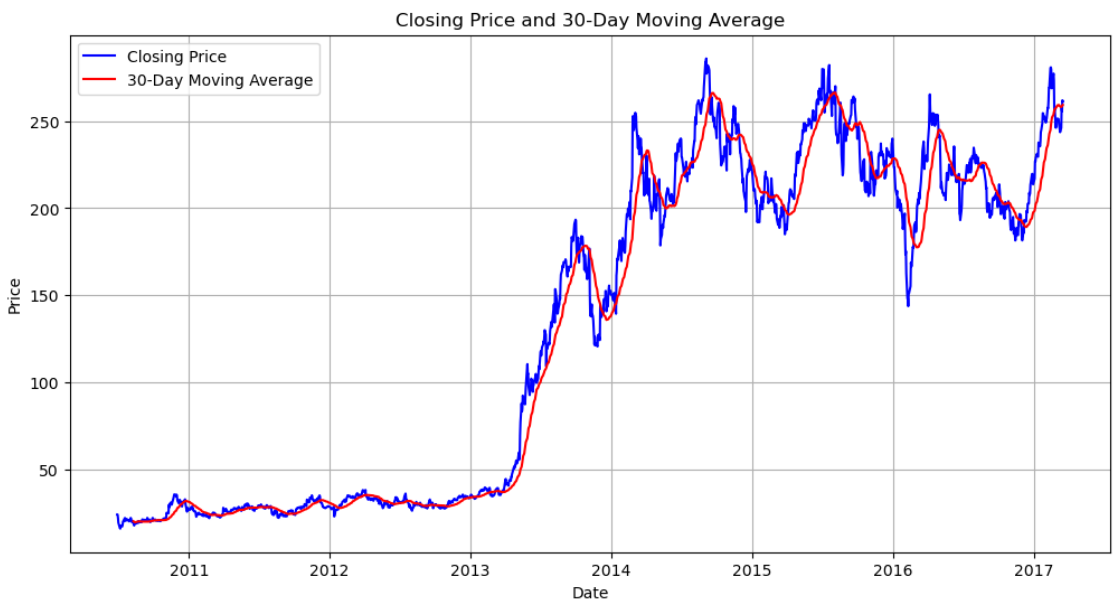
[11]:
```python
# Extract 'Date' and 'Volume' columns
sales_data = data[['Volume']]

# Plot the sales volume over time
plt.figure(figsize=(12, 6))
plt.title('Sales Volume Over Time')
plt.xlabel('Date')
plt.ylabel('Volume')
plt.plot(sales_data, label='Volume', color='green')
plt.legend()
plt.grid(True)
plt.show()
```



Inferences from the Sales Volume Graph

Inference 1: The sales volume fluctuates over time, indicating varying trading activity. Inference 2: High spikes in volume may correspond to significant events or news related to the stock. Inference 3: Low-volume periods may indicate reduced trading interest or market stability.

[12]:
```python
# Calculate the moving average of the closing price
window = 30  # Adjust the window size as needed
data['MovingAverage'] = data['Close'].rolling(window=window).mean()

# Plot the closing price and the moving average
plt.figure(figsize=(12, 6))
plt.title(f'Closing Price and {window}-Day Moving Average')
plt.xlabel('Date')
plt.ylabel('Price')
plt.plot(data['Close'], label='Closing Price', color='blue')
plt.plot(data['MovingAverage'], label=f'{window}-Day Moving Average',␣
 ↪color='red')
plt.legend()
plt.grid(True)
```



Inferences from the Moving Average Graph

The closing price (blue line) shows the daily fluctuations in the stock price. The moving average (red line) smooths out short-term price fluctuations and shows the overall trend. Crossovers between the closing price and moving average may signal potential buying or selling opportunities.

[18]:
```
# Calculate daily returns
data['Daily_Return'] = data['Close'].pct_change()

# Calculate the standard deviation of daily returns as a measure of risk
daily_std = data['Daily_Return'].std()

# Calculate the annualized standard deviation by multiplying by the square root␣
 ↪of the number of trading days in a year (usually 252)
annualized_std = daily_std * (252**0.5)

# Calculate the value at risk (VaR) for a given confidence level (e.g., 95%)
confidence_level = 0.95
z_score = -1.645  # Corresponding to a 5% loss for a one-tailed test
initial_investment = 10000  # Replace with your investment amount
```

```python
# Calculate the VaR
var = initial_investment * annualized_std * z_score

# Print the results
print(f"Daily Standard Deviation (Risk): {daily_std:.4f}")
print(f"Annualized Standard Deviation: {annualized_std:.4f}")
print(f"Value at Risk (VaR) at {confidence_level*100}% confidence: ${var:.2f}")
```

```
Daily Standard Deviation (Risk): 0.0329
Annualized Standard Deviation: 0.5225
Value at Risk (VaR) at 95.0% confidence: $-8594.31
```

Interpretation VaR at 95

[3]:
```python
# Step 1: Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.metrics import mean_squared_error

# Step 2: Load and preprocess the dataset
data = pd.read_csv("C:\\Users\\Narthana\\Downloads\\Tesla.csv - Tesla.csv.csv")
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)
data = data['Close'].values.reshape(-1, 1)  # We will use 'Close' prices for␣
↪prediction
scaler = MinMaxScaler()
data = scaler.fit_transform(data)

# Step 3: Split the dataset into training and testing sets
train_size = int(len(data) * 0.8)
test_size = len(data) - train_size
train_data, test_data = data[0:train_size], data[train_size:len(data)]

# Step 4: Create an LSTM model
def create_lstm_model():
    model = Sequential()
    model.add(LSTM(units=50, activation='relu', input_shape=(1, 1)))
    model.add(Dense(units=1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Step 5: Train the model
model = create_lstm_model()
```

```
# Prepare the target data
target_data = np.roll(train_data, -1)  # Shift the data by one step
target_data[-1] = train_data[-1]  # Set the last element to match the original␣
  ↪shape
target_data = target_data.reshape(-1, 1, 1)

model.fit(train_data, target_data, epochs=100, batch_size=1, verbose=2)

# Step 6: Make predictions
predictions = model.predict(test_data)

# Step 7: Evaluate the model
test_data = scaler.inverse_transform(test_data.reshape(-1, 1))
mse = mean_squared_error(test_data, predictions)
print(f"Mean Squared Error: {mse}")

# Plot the results
plt.figure(figsize=(12,6))
plt.title('Stock Price Prediction')
plt.xlabel('Date')
plt.ylabel('Price')
plt.plot(test_data, label='Actual Price', color='blue')
plt.plot(predictions, label='Predicted Price', color='red')
plt.legend()
plt.show()
```

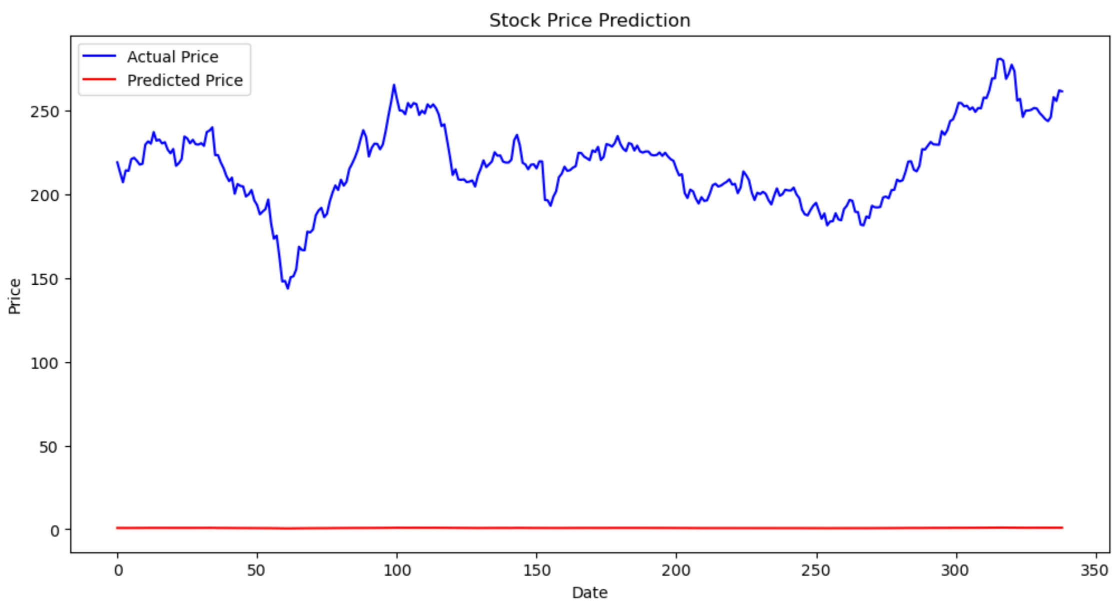
```
Epoch 1/100
1353/1353 - 5s - loss: 0.0252 - 5s/epoch - 3ms/step
Epoch 2/100
1353/1353 - 4s - loss: 5.3789e-04 - 4s/epoch - 3ms/step
Epoch 3/100
1353/1353 - 3s - loss: 5.0317e-04 - 3s/epoch - 2ms/step
Epoch 4/100
1353/1353 - 2s - loss: 4.1910e-04 - 2s/epoch - 2ms/step
Epoch 5/100
1353/1353 - 2s - loss: 3.6130e-04 - 2s/epoch - 2ms/step
Epoch 6/100
1353/1353 - 2s - loss: 3.1815e-04 - 2s/epoch - 2ms/step
Epoch 7/100
1353/1353 - 2s - loss: 3.2285e-04 - 2s/epoch - 2ms/step
Epoch 8/100
1353/1353 - 2s - loss: 3.0103e-04 - 2s/epoch - 2ms/step
Epoch 9/100
1353/1353 - 2s - loss: 3.0946e-04 - 2s/epoch - 2ms/step
Epoch 10/100
1353/1353 - 2s - loss: 3.0313e-04 - 2s/epoch - 2ms/step
Epoch 11/100
```

```
1353/1353 - 2s - loss: 2.9124e-04 - 2s/epoch - 2ms/step
Epoch 12/100
1353/1353 - 2s - loss: 3.1183e-04 - 2s/epoch - 2ms/step
Epoch 13/100
1353/1353 - 2s - loss: 3.0882e-04 - 2s/epoch - 2ms/step
Epoch 14/100
1353/1353 - 2s - loss: 3.1120e-04 - 2s/epoch - 2ms/step
Epoch 15/100
1353/1353 - 2s - loss: 2.9110e-04 - 2s/epoch - 2ms/step
Epoch 16/100
1353/1353 - 2s - loss: 3.1186e-04 - 2s/epoch - 2ms/step
Epoch 17/100
1353/1353 - 2s - loss: 2.9517e-04 - 2s/epoch - 2ms/step
Epoch 18/100
1353/1353 - 2s - loss: 2.9921e-04 - 2s/epoch - 2ms/step
Epoch 19/100
1353/1353 - 2s - loss: 3.1845e-04 - 2s/epoch - 2ms/step
Epoch 20/100
1353/1353 - 3s - loss: 3.1698e-04 - 3s/epoch - 2ms/step
Epoch 21/100
1353/1353 - 3s - loss: 3.0494e-04 - 3s/epoch - 2ms/step
Epoch 22/100
1353/1353 - 2s - loss: 3.0815e-04 - 2s/epoch - 2ms/step
Epoch 23/100
1353/1353 - 2s - loss: 3.0105e-04 - 2s/epoch - 1ms/step
Epoch 24/100
1353/1353 - 2s - loss: 3.1507e-04 - 2s/epoch - 1ms/step
Epoch 25/100
1353/1353 - 2s - loss: 2.9368e-04 - 2s/epoch - 1ms/step
Epoch 26/100
1353/1353 - 2s - loss: 3.0341e-04 - 2s/epoch - 2ms/step
Epoch 27/100
1353/1353 - 2s - loss: 3.2237e-04 - 2s/epoch - 2ms/step
Epoch 28/100
1353/1353 - 2s - loss: 2.9372e-04 - 2s/epoch - 1ms/step
Epoch 29/100
1353/1353 - 2s - loss: 3.0352e-04 - 2s/epoch - 2ms/step
Epoch 30/100
1353/1353 - 2s - loss: 2.9418e-04 - 2s/epoch - 2ms/step
Epoch 31/100
1353/1353 - 2s - loss: 2.9297e-04 - 2s/epoch - 2ms/step
Epoch 32/100
1353/1353 - 2s - loss: 3.0126e-04 - 2s/epoch - 2ms/step
Epoch 33/100
1353/1353 - 2s - loss: 3.0314e-04 - 2s/epoch - 2ms/step
Epoch 34/100
1353/1353 - 2s - loss: 2.9104e-04 - 2s/epoch - 2ms/step
Epoch 35/100
```

```
1353/1353 - 2s - loss: 2.8758e-04 - 2s/epoch - 2ms/step
Epoch 36/100
1353/1353 - 3s - loss: 2.9503e-04 - 3s/epoch - 2ms/step
Epoch 37/100
1353/1353 - 2s - loss: 3.0524e-04 - 2s/epoch - 2ms/step
Epoch 38/100
1353/1353 - 2s - loss: 3.0399e-04 - 2s/epoch - 2ms/step
Epoch 39/100
1353/1353 - 2s - loss: 3.1074e-04 - 2s/epoch - 2ms/step
Epoch 40/100
1353/1353 - 2s - loss: 3.0451e-04 - 2s/epoch - 2ms/step
Epoch 41/100
1353/1353 - 2s - loss: 2.9306e-04 - 2s/epoch - 2ms/step
Epoch 42/100
1353/1353 - 2s - loss: 2.9603e-04 - 2s/epoch - 1ms/step
Epoch 43/100
1353/1353 - 2s - loss: 3.0799e-04 - 2s/epoch - 1ms/step
Epoch 44/100
1353/1353 - 2s - loss: 3.0258e-04 - 2s/epoch - 1ms/step
Epoch 45/100
1353/1353 - 2s - loss: 2.9896e-04 - 2s/epoch - 1ms/step
Epoch 46/100
1353/1353 - 2s - loss: 2.9851e-04 - 2s/epoch - 1ms/step
Epoch 47/100
1353/1353 - 2s - loss: 3.0167e-04 - 2s/epoch - 2ms/step
Epoch 48/100
1353/1353 - 2s - loss: 2.9744e-04 - 2s/epoch - 1ms/step
Epoch 49/100
1353/1353 - 2s - loss: 3.0904e-04 - 2s/epoch - 2ms/step
Epoch 50/100
1353/1353 - 2s - loss: 2.9920e-04 - 2s/epoch - 1ms/step
Epoch 51/100
1353/1353 - 2s - loss: 2.9985e-04 - 2s/epoch - 2ms/step
Epoch 52/100
1353/1353 - 3s - loss: 2.8833e-04 - 3s/epoch - 2ms/step
Epoch 53/100
1353/1353 - 3s - loss: 2.9705e-04 - 3s/epoch - 2ms/step
Epoch 54/100
1353/1353 - 2s - loss: 2.8190e-04 - 2s/epoch - 2ms/step
Epoch 55/100
1353/1353 - 2s - loss: 3.0249e-04 - 2s/epoch - 1ms/step
Epoch 56/100
1353/1353 - 2s - loss: 3.0362e-04 - 2s/epoch - 1ms/step
Epoch 57/100
1353/1353 - 2s - loss: 2.9536e-04 - 2s/epoch - 2ms/step
Epoch 58/100
1353/1353 - 2s - loss: 2.9363e-04 - 2s/epoch - 1ms/step
Epoch 59/100
```

```
1353/1353 - 2s - loss: 2.9783e-04 - 2s/epoch - 1ms/step
Epoch 60/100
1353/1353 - 2s - loss: 2.9938e-04 - 2s/epoch - 1ms/step
Epoch 61/100
1353/1353 - 2s - loss: 3.0275e-04 - 2s/epoch - 1ms/step
Epoch 62/100
1353/1353 - 2s - loss: 3.0506e-04 - 2s/epoch - 1ms/step
Epoch 63/100
1353/1353 - 2s - loss: 3.0622e-04 - 2s/epoch - 1ms/step
Epoch 64/100
1353/1353 - 2s - loss: 2.9780e-04 - 2s/epoch - 1ms/step
Epoch 65/100
1353/1353 - 2s - loss: 3.0172e-04 - 2s/epoch - 2ms/step
Epoch 66/100
1353/1353 - 2s - loss: 3.1303e-04 - 2s/epoch - 1ms/step
Epoch 67/100
1353/1353 - 2s - loss: 3.0401e-04 - 2s/epoch - 1ms/step
Epoch 68/100
1353/1353 - 2s - loss: 2.8896e-04 - 2s/epoch - 2ms/step
Epoch 69/100
1353/1353 - 2s - loss: 3.0474e-04 - 2s/epoch - 1ms/step
Epoch 70/100
1353/1353 - 2s - loss: 2.9176e-04 - 2s/epoch - 2ms/step
Epoch 71/100
1353/1353 - 2s - loss: 2.9130e-04 - 2s/epoch - 2ms/step
Epoch 72/100
1353/1353 - 2s - loss: 3.0026e-04 - 2s/epoch - 2ms/step
Epoch 73/100
1353/1353 - 2s - loss: 2.9119e-04 - 2s/epoch - 2ms/step
Epoch 74/100
1353/1353 - 2s - loss: 2.8123e-04 - 2s/epoch - 2ms/step
Epoch 75/100
1353/1353 - 2s - loss: 2.8438e-04 - 2s/epoch - 2ms/step
Epoch 76/100
1353/1353 - 2s - loss: 2.8714e-04 - 2s/epoch - 2ms/step
Epoch 77/100
1353/1353 - 2s - loss: 2.9779e-04 - 2s/epoch - 2ms/step
Epoch 78/100
1353/1353 - 2s - loss: 2.9694e-04 - 2s/epoch - 2ms/step
Epoch 79/100
1353/1353 - 2s - loss: 3.0064e-04 - 2s/epoch - 2ms/step
Epoch 80/100
1353/1353 - 2s - loss: 2.9217e-04 - 2s/epoch - 2ms/step
Epoch 81/100
1353/1353 - 2s - loss: 3.0155e-04 - 2s/epoch - 2ms/step
Epoch 82/100
1353/1353 - 2s - loss: 2.8288e-04 - 2s/epoch - 2ms/step
Epoch 83/100
```

```
1353/1353 - 2s - loss: 2.9888e-04 - 2s/epoch - 2ms/step
Epoch 84/100
1353/1353 - 2s - loss: 2.9387e-04 - 2s/epoch - 1ms/step
Epoch 85/100
1353/1353 - 2s - loss: 3.0026e-04 - 2s/epoch - 1ms/step
Epoch 86/100
1353/1353 - 2s - loss: 2.8723e-04 - 2s/epoch - 1ms/step
Epoch 87/100
1353/1353 - 2s - loss: 2.8223e-04 - 2s/epoch - 1ms/step
Epoch 88/100
1353/1353 - 2s - loss: 2.9805e-04 - 2s/epoch - 1ms/step
Epoch 89/100
1353/1353 - 2s - loss: 3.1058e-04 - 2s/epoch - 1ms/step
Epoch 90/100
1353/1353 - 2s - loss: 2.8392e-04 - 2s/epoch - 1ms/step
Epoch 91/100
1353/1353 - 2s - loss: 2.9318e-04 - 2s/epoch - 1ms/step
Epoch 92/100
1353/1353 - 2s - loss: 2.9600e-04 - 2s/epoch - 1ms/step
Epoch 93/100
1353/1353 - 2s - loss: 2.9898e-04 - 2s/epoch - 2ms/step
Epoch 94/100
1353/1353 - 2s - loss: 2.9983e-04 - 2s/epoch - 2ms/step
Epoch 95/100
1353/1353 - 2s - loss: 2.9733e-04 - 2s/epoch - 2ms/step
Epoch 96/100
1353/1353 - 2s - loss: 2.9535e-04 - 2s/epoch - 2ms/step
Epoch 97/100
1353/1353 - 2s - loss: 2.9574e-04 - 2s/epoch - 2ms/step
Epoch 98/100
1353/1353 - 2s - loss: 3.0511e-04 - 2s/epoch - 2ms/step
Epoch 99/100
1353/1353 - 3s - loss: 2.9741e-04 - 3s/epoch - 2ms/step
Epoch 100/100
1353/1353 - 3s - loss: 2.9306e-04 - 3s/epoch - 2ms/step
11/11 [==============================] - 0s 3ms/step
Mean Squared Error: 47736.7271493383
```



[19]:
```
# Step 7: Evaluate the model
test_data = scaler.inverse_transform(test_data.reshape(-1, 1))
mse = mean_squared_error(test_data, predictions)
rmse = np.sqrt(mse)
print(f"Mean Squared Error: {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")

# Calculate Accuracy
accuracy = 100 - (rmse / np.mean(test_data) * 100)
print(f"Model Accuracy: {accuracy:.2f}%")
```

```
Mean Squared Error: 3511841910.489239
Root Mean Squared Error (RMSE): 59260.79572946383
Model Accuracy: -0.63%
```