untitled13

July 19, 2023

[2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.graph_objects as go
```

[3]:
```
# Load the data from the CSV file
df = pd.read_csv('/content/amazon_stock_price.csv')
```

[4]:
```
# Convert the 'Date' column to datetime type
df['Date'] = pd.to_datetime(df['Date'])
```

[5]:
```
# Sort the DataFrame by date in ascending order
df = df.sort_values('Date')
```

[6]:
```
# Extract the 'Close' column (our target variable)
dataset = df[['Close']].values.astype(float)
```

[7]:
```
# Normalize the dataset using Min-Max scaling to bring values between 0 and 1
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

[8]:
```
# Function to create input sequences and corresponding target values
def create_sequences(dataset, look_back=1):
    data_X, data_y = [], []
    for i in range(len(dataset) - look_back):
        data_X.append(dataset[i:(i + look_back), 0])
        data_y.append(dataset[i + look_back, 0])
    return np.array(data_X), np.array(data_y)
```

[9]:
```
# Set the look-back period (number of previous time steps to use for prediction)
look_back = 30
```

[10]:
```
# Create input sequences and target values
X, y = create_sequences(dataset, look_back)
```

[11]:
```
# Split the data into training and testing sets
train_size = int(len(X) * 0.7)
test_size = len(X) - train_size
X_train, X_test = X[0:train_size], X[train_size:len(X)]
y_train, y_test = y[0:train_size], y[train_size:len(y)]
```

[12]:
```
# Reshape the input data to fit the LSTM input shape (samples, time steps,␣
 ↪features)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

[13]:
```
# Create the LSTM model
model = Sequential()
model.add(LSTM(50, input_shape=(look_back, 1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

[14]:
```
# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=1, verbose=2)
```

```
Epoch 1/100
4287/4287 - 21s - loss: 1.0084e-05 - 21s/epoch - 5ms/step
Epoch 2/100
4287/4287 - 14s - loss: 3.4736e-06 - 14s/epoch - 3ms/step
Epoch 3/100
4287/4287 - 15s - loss: 2.6222e-06 - 15s/epoch - 3ms/step
Epoch 4/100
4287/4287 - 15s - loss: 2.2404e-06 - 15s/epoch - 3ms/step
Epoch 5/100
4287/4287 - 14s - loss: 2.0190e-06 - 14s/epoch - 3ms/step
Epoch 6/100
4287/4287 - 14s - loss: 2.0103e-06 - 14s/epoch - 3ms/step
Epoch 7/100
4287/4287 - 14s - loss: 1.9650e-06 - 14s/epoch - 3ms/step
Epoch 8/100
4287/4287 - 15s - loss: 1.8073e-06 - 15s/epoch - 3ms/step
Epoch 9/100
4287/4287 - 15s - loss: 1.8052e-06 - 15s/epoch - 3ms/step
Epoch 10/100
4287/4287 - 14s - loss: 1.6952e-06 - 14s/epoch - 3ms/step
Epoch 11/100
4287/4287 - 14s - loss: 1.5117e-06 - 14s/epoch - 3ms/step
Epoch 12/100
4287/4287 - 14s - loss: 1.6446e-06 - 14s/epoch - 3ms/step
```

```
Epoch 13/100
4287/4287 - 14s - loss: 1.6566e-06 - 14s/epoch - 3ms/step
Epoch 14/100
4287/4287 - 15s - loss: 1.5561e-06 - 15s/epoch - 3ms/step
Epoch 15/100
4287/4287 - 14s - loss: 1.6875e-06 - 14s/epoch - 3ms/step
Epoch 16/100
4287/4287 - 15s - loss: 1.5007e-06 - 15s/epoch - 3ms/step
Epoch 17/100
4287/4287 - 14s - loss: 1.4980e-06 - 14s/epoch - 3ms/step
Epoch 18/100
4287/4287 - 14s - loss: 1.4851e-06 - 14s/epoch - 3ms/step
Epoch 19/100
4287/4287 - 14s - loss: 1.5725e-06 - 14s/epoch - 3ms/step
Epoch 20/100
4287/4287 - 15s - loss: 1.4640e-06 - 15s/epoch - 3ms/step
Epoch 21/100
4287/4287 - 14s - loss: 1.6016e-06 - 14s/epoch - 3ms/step
Epoch 22/100
4287/4287 - 14s - loss: 1.5054e-06 - 14s/epoch - 3ms/step
Epoch 23/100
4287/4287 - 14s - loss: 1.5702e-06 - 14s/epoch - 3ms/step
Epoch 24/100
4287/4287 - 14s - loss: 1.5336e-06 - 14s/epoch - 3ms/step
Epoch 25/100
4287/4287 - 14s - loss: 1.5560e-06 - 14s/epoch - 3ms/step
Epoch 26/100
4287/4287 - 15s - loss: 1.4375e-06 - 15s/epoch - 3ms/step
Epoch 27/100
4287/4287 - 14s - loss: 1.4461e-06 - 14s/epoch - 3ms/step
Epoch 28/100
4287/4287 - 14s - loss: 1.5250e-06 - 14s/epoch - 3ms/step
Epoch 29/100
4287/4287 - 15s - loss: 1.4602e-06 - 15s/epoch - 3ms/step
Epoch 30/100
4287/4287 - 14s - loss: 1.4994e-06 - 14s/epoch - 3ms/step
Epoch 31/100
4287/4287 - 15s - loss: 1.4189e-06 - 15s/epoch - 4ms/step
Epoch 32/100
4287/4287 - 15s - loss: 1.3942e-06 - 15s/epoch - 3ms/step
Epoch 33/100
4287/4287 - 15s - loss: 1.4227e-06 - 15s/epoch - 3ms/step
Epoch 34/100
4287/4287 - 15s - loss: 1.3610e-06 - 15s/epoch - 3ms/step
Epoch 35/100
4287/4287 - 14s - loss: 1.3813e-06 - 14s/epoch - 3ms/step
Epoch 36/100
4287/4287 - 15s - loss: 1.3564e-06 - 15s/epoch - 3ms/step
```

```
Epoch 37/100
4287/4287 - 14s - loss: 1.3986e-06 - 14s/epoch - 3ms/step
Epoch 38/100
4287/4287 - 14s - loss: 1.3590e-06 - 14s/epoch - 3ms/step
Epoch 39/100
4287/4287 - 14s - loss: 1.4233e-06 - 14s/epoch - 3ms/step
Epoch 40/100
4287/4287 - 14s - loss: 1.4069e-06 - 14s/epoch - 3ms/step
Epoch 41/100
4287/4287 - 15s - loss: 1.3593e-06 - 15s/epoch - 3ms/step
Epoch 42/100
4287/4287 - 15s - loss: 1.3206e-06 - 15s/epoch - 4ms/step
Epoch 43/100
4287/4287 - 14s - loss: 1.3790e-06 - 14s/epoch - 3ms/step
Epoch 44/100
4287/4287 - 15s - loss: 1.4119e-06 - 15s/epoch - 3ms/step
Epoch 45/100
4287/4287 - 15s - loss: 1.3975e-06 - 15s/epoch - 4ms/step
Epoch 46/100
4287/4287 - 15s - loss: 1.2880e-06 - 15s/epoch - 4ms/step
Epoch 47/100
4287/4287 - 15s - loss: 1.3948e-06 - 15s/epoch - 3ms/step
Epoch 48/100
4287/4287 - 15s - loss: 1.3907e-06 - 15s/epoch - 3ms/step
Epoch 49/100
4287/4287 - 14s - loss: 1.3706e-06 - 14s/epoch - 3ms/step
Epoch 50/100
4287/4287 - 14s - loss: 1.2828e-06 - 14s/epoch - 3ms/step
Epoch 51/100
4287/4287 - 15s - loss: 1.3711e-06 - 15s/epoch - 4ms/step
Epoch 52/100
4287/4287 - 15s - loss: 1.3700e-06 - 15s/epoch - 3ms/step
Epoch 53/100
4287/4287 - 14s - loss: 1.3971e-06 - 14s/epoch - 3ms/step
Epoch 54/100
4287/4287 - 14s - loss: 1.3496e-06 - 14s/epoch - 3ms/step
Epoch 55/100
4287/4287 - 15s - loss: 1.3897e-06 - 15s/epoch - 4ms/step
Epoch 56/100
4287/4287 - 15s - loss: 1.3530e-06 - 15s/epoch - 3ms/step
Epoch 57/100
4287/4287 - 14s - loss: 1.2930e-06 - 14s/epoch - 3ms/step
Epoch 58/100
4287/4287 - 14s - loss: 1.2812e-06 - 14s/epoch - 3ms/step
Epoch 59/100
4287/4287 - 14s - loss: 1.3567e-06 - 14s/epoch - 3ms/step
Epoch 60/100
4287/4287 - 14s - loss: 1.3169e-06 - 14s/epoch - 3ms/step
```

```
Epoch 61/100
4287/4287 - 15s - loss: 1.3484e-06 - 15s/epoch - 3ms/step
Epoch 62/100
4287/4287 - 14s - loss: 1.3307e-06 - 14s/epoch - 3ms/step
Epoch 63/100
4287/4287 - 14s - loss: 1.3178e-06 - 14s/epoch - 3ms/step
Epoch 64/100
4287/4287 - 14s - loss: 1.3508e-06 - 14s/epoch - 3ms/step
Epoch 65/100
4287/4287 - 14s - loss: 1.4073e-06 - 14s/epoch - 3ms/step
Epoch 66/100
4287/4287 - 14s - loss: 1.3101e-06 - 14s/epoch - 3ms/step
Epoch 67/100
4287/4287 - 15s - loss: 1.3073e-06 - 15s/epoch - 3ms/step
Epoch 68/100
4287/4287 - 15s - loss: 1.2521e-06 - 15s/epoch - 3ms/step
Epoch 69/100
4287/4287 - 14s - loss: 1.3597e-06 - 14s/epoch - 3ms/step
Epoch 70/100
4287/4287 - 14s - loss: 1.3267e-06 - 14s/epoch - 3ms/step
Epoch 71/100
4287/4287 - 14s - loss: 1.3267e-06 - 14s/epoch - 3ms/step
Epoch 72/100
4287/4287 - 15s - loss: 1.3259e-06 - 15s/epoch - 3ms/step
Epoch 73/100
4287/4287 - 15s - loss: 1.2714e-06 - 15s/epoch - 3ms/step
Epoch 74/100
4287/4287 - 14s - loss: 1.3243e-06 - 14s/epoch - 3ms/step
Epoch 75/100
4287/4287 - 14s - loss: 1.2862e-06 - 14s/epoch - 3ms/step
Epoch 76/100
4287/4287 - 14s - loss: 1.2596e-06 - 14s/epoch - 3ms/step
Epoch 77/100
4287/4287 - 14s - loss: 1.3445e-06 - 14s/epoch - 3ms/step
Epoch 78/100
4287/4287 - 15s - loss: 1.2877e-06 - 15s/epoch - 3ms/step
Epoch 79/100
4287/4287 - 14s - loss: 1.2548e-06 - 14s/epoch - 3ms/step
Epoch 80/100
4287/4287 - 14s - loss: 1.3572e-06 - 14s/epoch - 3ms/step
Epoch 81/100
4287/4287 - 15s - loss: 1.3013e-06 - 15s/epoch - 3ms/step
Epoch 82/100
4287/4287 - 14s - loss: 1.2956e-06 - 14s/epoch - 3ms/step
Epoch 83/100
4287/4287 - 15s - loss: 1.2820e-06 - 15s/epoch - 3ms/step
Epoch 84/100
4287/4287 - 14s - loss: 1.2368e-06 - 14s/epoch - 3ms/step
```

```
Epoch 85/100
4287/4287 - 14s - loss: 1.3056e-06 - 14s/epoch - 3ms/step
Epoch 86/100
4287/4287 - 14s - loss: 1.2639e-06 - 14s/epoch - 3ms/step
Epoch 87/100
4287/4287 - 14s - loss: 1.2613e-06 - 14s/epoch - 3ms/step
Epoch 88/100
4287/4287 - 14s - loss: 1.2721e-06 - 14s/epoch - 3ms/step
Epoch 89/100
4287/4287 - 14s - loss: 1.2580e-06 - 14s/epoch - 3ms/step
Epoch 90/100
4287/4287 - 14s - loss: 1.2925e-06 - 14s/epoch - 3ms/step
Epoch 91/100
4287/4287 - 14s - loss: 1.2499e-06 - 14s/epoch - 3ms/step
Epoch 92/100
4287/4287 - 14s - loss: 1.2990e-06 - 14s/epoch - 3ms/step
Epoch 93/100
4287/4287 - 14s - loss: 1.3008e-06 - 14s/epoch - 3ms/step
Epoch 94/100
4287/4287 - 15s - loss: 1.2107e-06 - 15s/epoch - 4ms/step
Epoch 95/100
4287/4287 - 15s - loss: 1.2112e-06 - 15s/epoch - 3ms/step
Epoch 96/100
4287/4287 - 14s - loss: 1.2257e-06 - 14s/epoch - 3ms/step
Epoch 97/100
4287/4287 - 14s - loss: 1.2921e-06 - 14s/epoch - 3ms/step
Epoch 98/100
4287/4287 - 14s - loss: 1.3463e-06 - 14s/epoch - 3ms/step
Epoch 99/100
4287/4287 - 14s - loss: 1.2003e-06 - 14s/epoch - 3ms/step
Epoch 100/100
4287/4287 - 15s - loss: 1.2969e-06 - 15s/epoch - 3ms/step
```

[14]: <keras.callbacks.History at 0x7e57a3a7bc40>

[15]:
```
# Generate predictions on the training and test data
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)
```

```
134/134 [==============================] - 1s 2ms/step
58/58 [==============================] - 0s 3ms/step
```

[16]:
```
# Inverse transform the predictions to the original scale
train_predict = scaler.inverse_transform(train_predict)
y_train = scaler.inverse_transform([y_train])
test_predict = scaler.inverse_transform(test_predict)
y_test = scaler.inverse_transform([y_test])
```

[17]:
```python
# Calculate the root mean squared error (RMSE) to evaluate the model's␣
↪performance
train_score = np.sqrt(np.mean((y_train[0] - train_predict[:, 0])**2))
print(f"Train RMSE: {train_score:.2f}")
```

```
Train RMSE: 5.14
```

[18]:
```python
test_score = np.sqrt(np.mean((y_test[0] - test_predict[:, 0])**2))
print(f"Test RMSE: {test_score:.2f}")
```

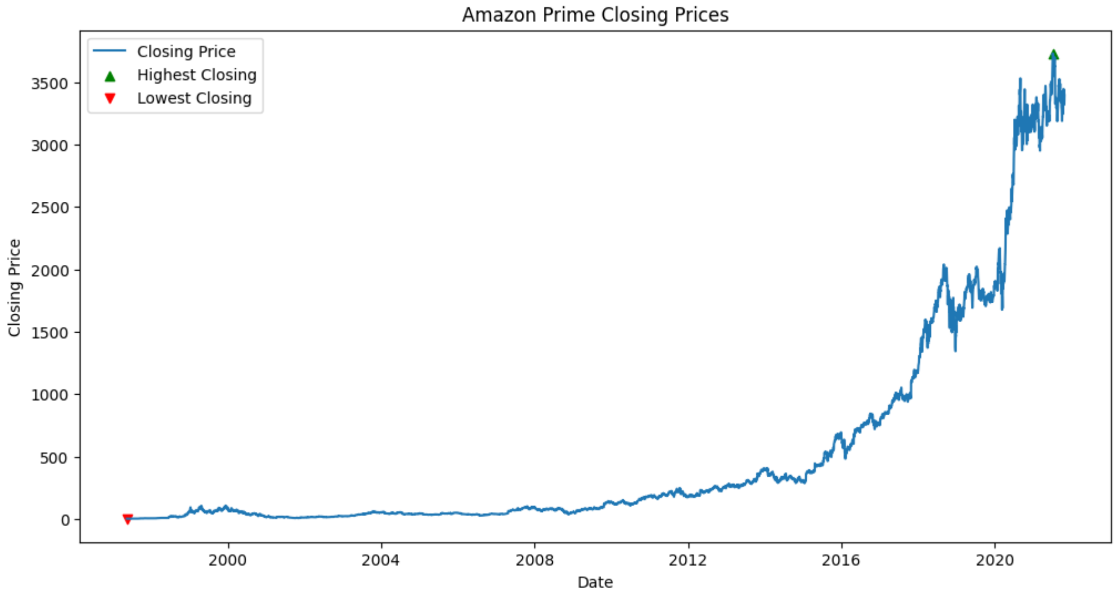
```
Test RMSE: 344.77
```

[19]:
```python
# Find the day with the highest and lowest closing value
max_close_day = df.loc[df['Close'].idxmax()]['Date']
min_close_day = df.loc[df['Close'].idxmin()]['Date']

print(f"Day with highest closing value: {max_close_day}")
print(f"Day with lowest closing value: {min_close_day}")
```

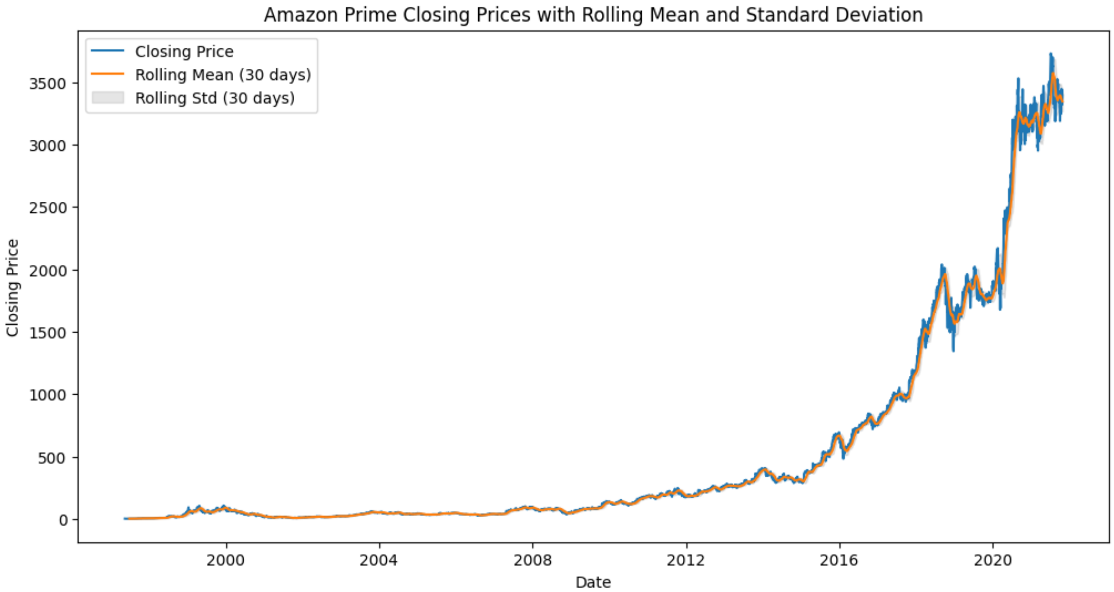
```
Day with highest closing value: 2021-07-08 00:00:00
Day with lowest closing value: 1997-05-22 00:00:00
```

[20]:
```python
# Plot the historical closing prices over time
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['Close'], label='Closing Price')
plt.scatter(max_close_day, df.loc[df['Close'].idxmax()]['Close'],␣
↪color='green', label='Highest Closing', marker='^')
plt.scatter(min_close_day, df.loc[df['Close'].idxmin()]['Close'], color='red',␣
↪label='Lowest Closing', marker='v')
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('Amazon Prime Closing Prices')
plt.legend()
plt.show()
```

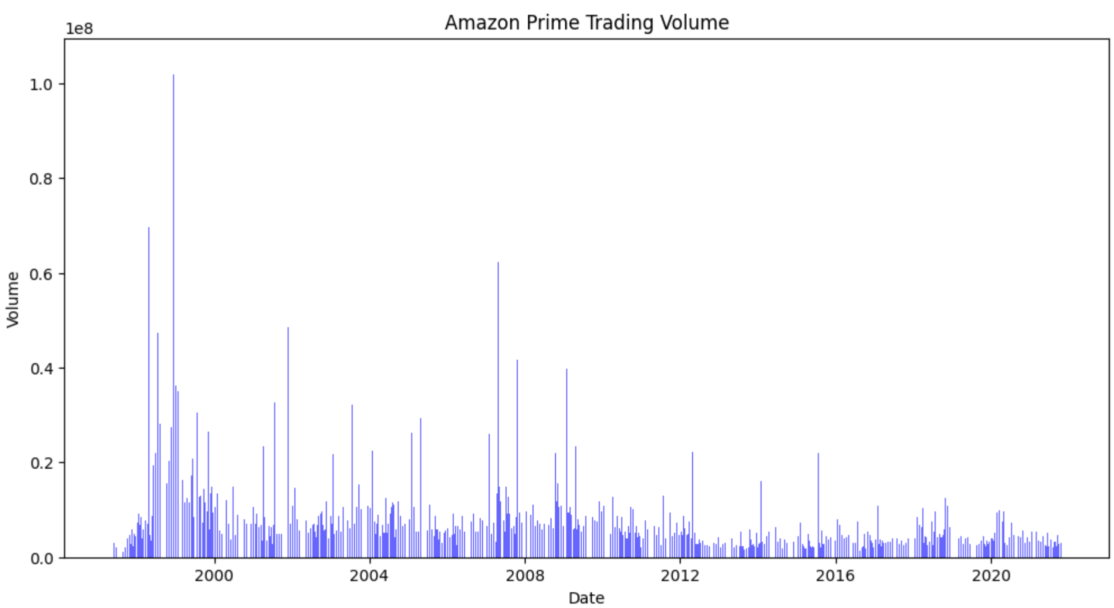


[21]:
```
# Calculate the rolling mean and standard deviation of the closing prices
rolling_mean = df['Close'].rolling(window=30).mean()
rolling_std = df['Close'].rolling(window=30).std()
```

[22]:
```
# Plot the rolling mean and standard deviation
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['Close'], label='Closing Price')
plt.plot(df['Date'], rolling_mean, label='Rolling Mean (30 days)')
plt.fill_between(df['Date'], rolling_mean - rolling_std, rolling_mean +␣
 ↪rolling_std, alpha=0.2, color='gray', label='Rolling Std (30 days)')
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('Amazon Prime Closing Prices with Rolling Mean and Standard␣
 ↪Deviation')
plt.legend()
plt.show()
```



[23]:
```python
# Plot the trading volume over time
plt.figure(figsize=(12, 6))
plt.bar(df['Date'], df['Volume'], color='blue', alpha=0.6)
plt.xlabel('Date')
plt.ylabel('Volume')
plt.title('Amazon Prime Trading Volume')
plt.show()
```



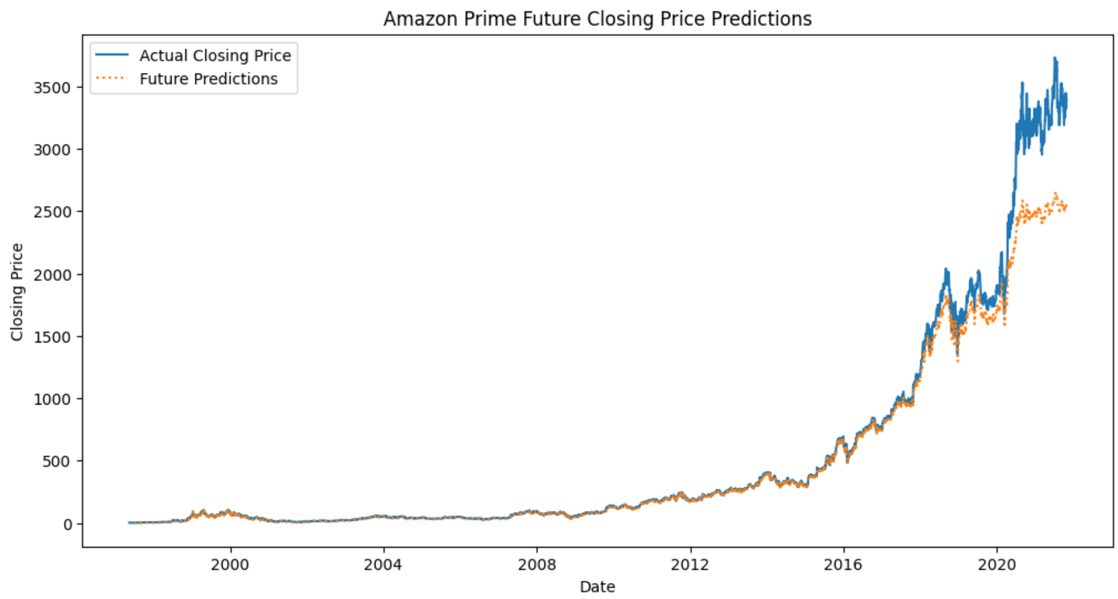
[24]:
```
# Extend the dataset to simulate future predictions (e.g., 30 days beyond the␣
↪available data)
extended_dates = pd.date_range(start=df['Date'].max(), periods=30, freq='D')
extended_dates = pd.DataFrame({'Date': extended_dates})
extended_df = pd.concat([df, extended_dates], ignore_index=True)
```

[25]:
```
# Preprocess the extended dataset for prediction
extended_dataset = scaler.transform(extended_df[['Close']].values.astype(float))
X_extended, y_extended = create_sequences(extended_dataset, look_back)
X_extended = np.reshape(X_extended, (X_extended.shape[0], X_extended.shape[1],␣
↪1))
```

[26]:
```
# Generate predictions for the extended dataset
extended_predict = model.predict(X_extended)
extended_predict = scaler.inverse_transform(extended_predict)
```

```
193/193 [==============================] - 0s 2ms/step
```

[27]:
```
# Plot the actual data and future predictions
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['Close'], label='Actual Closing Price')
plt.plot(extended_df.iloc[look_back:]['Date'], extended_predict, label='Future␣
↪Predictions', linestyle='dotted')
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('Amazon Prime Future Closing Price Predictions')
plt.legend()
plt.show()
```



[37]:
```python
# Filter the data for the year 2020
df_2020 = df[df['Date'].dt.year == 2020]

# Create a DataFrame for actual and predicted closing prices in 2020
dates_2020 = df_2020['Date'][look_back + 1:].reset_index(drop=True)  # Adjusted␣
↪indexing
actual_prices_2020 = df_2020['Close'][look_back:-1].reset_index(drop=True)  #␣
↪Adjusted indexing
predicted_prices_2020 = extended_predict[-len(dates_2020):].flatten()  #␣
↪Flattening the predicted prices array

closing_prices_2020_df = pd.DataFrame({'Date': dates_2020, 'Actual':␣
↪actual_prices_2020, 'Predicted': predicted_prices_2020})

# Display the actual and predicted closing prices for the year 2020 in a␣
↪tabular form
print(closing_prices_2020_df)
```

```
           Date       Actual    Predicted
0    2020-02-18  2134.870117  2511.926270
1    2020-02-19  2155.669922  2519.838623
2    2020-02-20  2170.219971  2504.669678
3    2020-02-21  2153.100098  2498.729492
4    2020-02-24  2095.969971  2489.710449
..          …            …            …
217  2020-12-24  3185.270020          NaN
218  2020-12-28  3172.689941          NaN
219  2020-12-29  3283.959961          NaN
220  2020-12-30  3322.000000          NaN
221  2020-12-31  3285.850098          NaN

[222 rows x 3 columns]
```

[28]:
```python
# Calculate the 50-day and 200-day moving averages
df['MA_50'] = df['Close'].rolling(window=50).mean()
df['MA_200'] = df['Close'].rolling(window=200).mean()
```

[29]:
```python
# Plot the moving averages
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['Close'], label='Closing Price')
plt.plot(df['Date'], df['MA_50'], label='50-day Moving Average',␣
↪linestyle='dashed')
plt.plot(df['Date'], df['MA_200'], label='200-day Moving Average',␣
↪linestyle='dotted')
plt.xlabel('Date')
```

```
plt.ylabel('Closing Price')
plt.title('Amazon Prime Closing Prices with Moving Averages')
plt.legend()
plt.show()
```