**#importing libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import mplfinance as mpf

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

from sklearn.metrics import mean\_absolute\_error

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

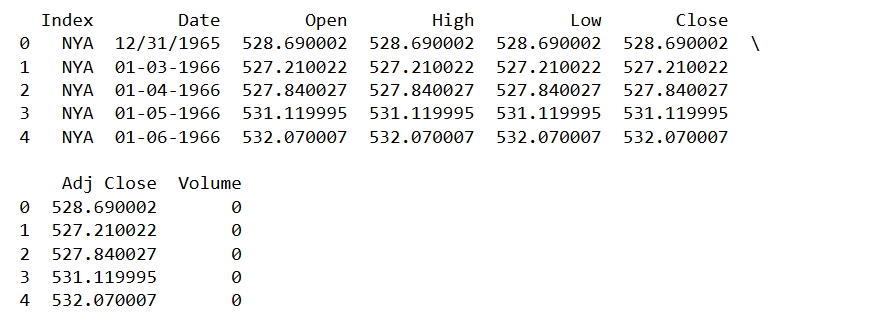
from tensorflow.keras.callbacks import EarlyStopping, LearningRateScheduler

**# Read the CSV file with a specified header row (if the first row doesn't contain column names)**

data = pd.read\_csv(r'file location')

**# Inspect the first few rows of the DataFrame to identify any issues**

print(data.head())

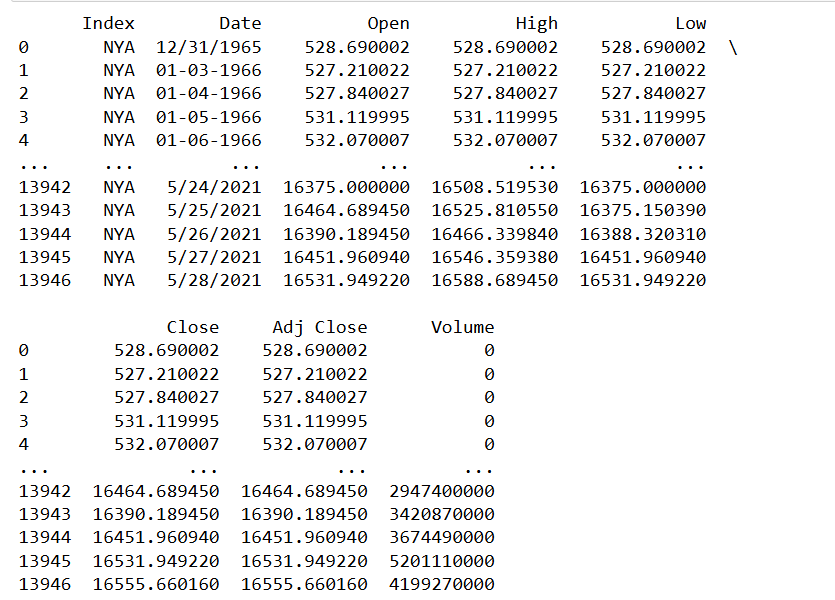


**# Create a DataFrame from the data**

df = pd.DataFrame(data)

**# Display the DataFrame**

print(df)



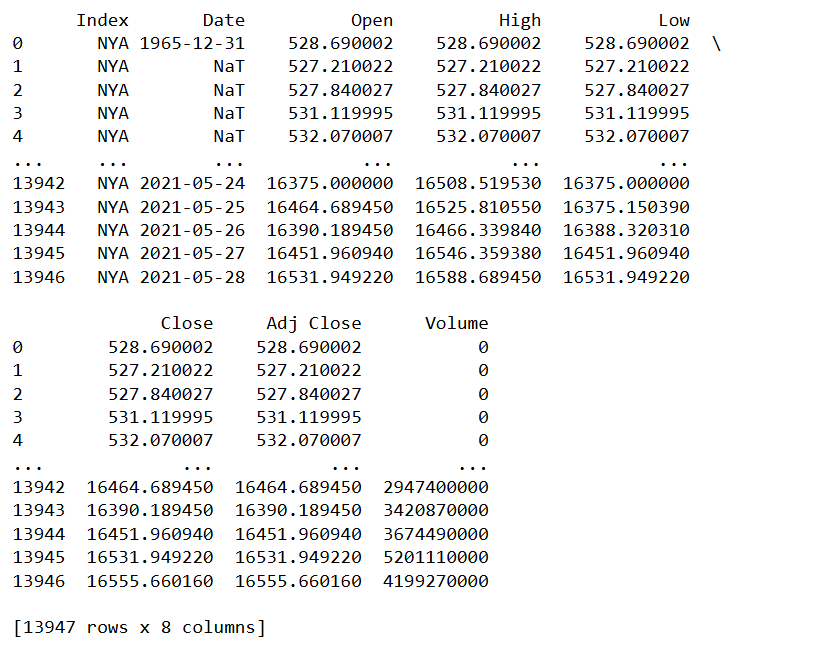
**# Convert the 'Date' column to datetime format**

df['Date'] = pd.to\_datetime(df['Date'], format='%m/%d/%Y', errors='coerce')

df['Date'] = df['Date'].fillna(pd.to\_datetime(df['Date'], format='%m-%d-%Y', errors='coerce'))

**# Display the updated DataFrame**

print(df)



**# Sort the DataFrame by the 'Date' column**

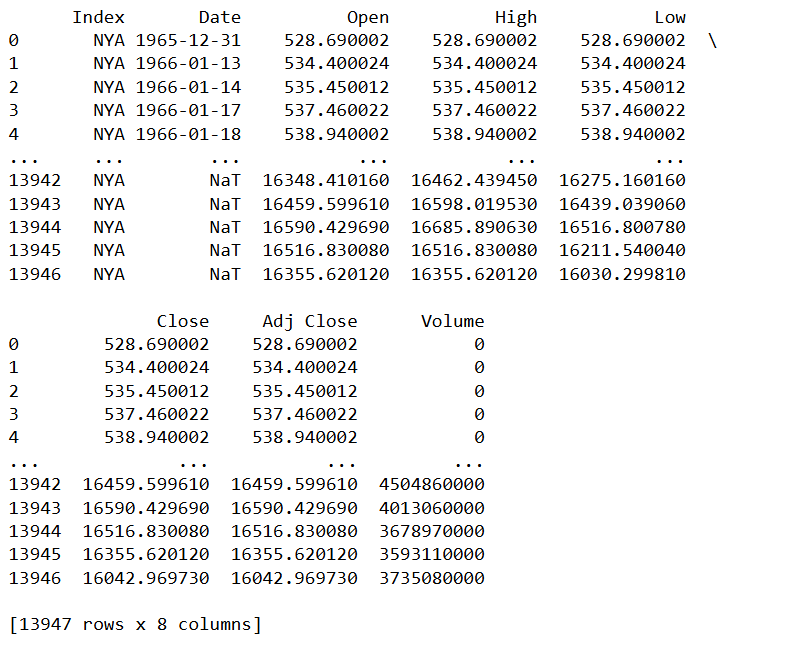
df = df.sort\_values(by='Date')

**# Reset the index**

df = df.reset\_index(drop=True)

**# Display the updated DataFrame**

print(df)

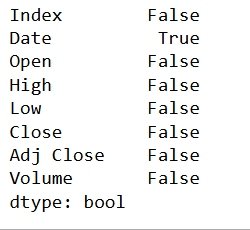


**# Check for NaN values in the DataFrame**

nan\_values = df.isna().any()

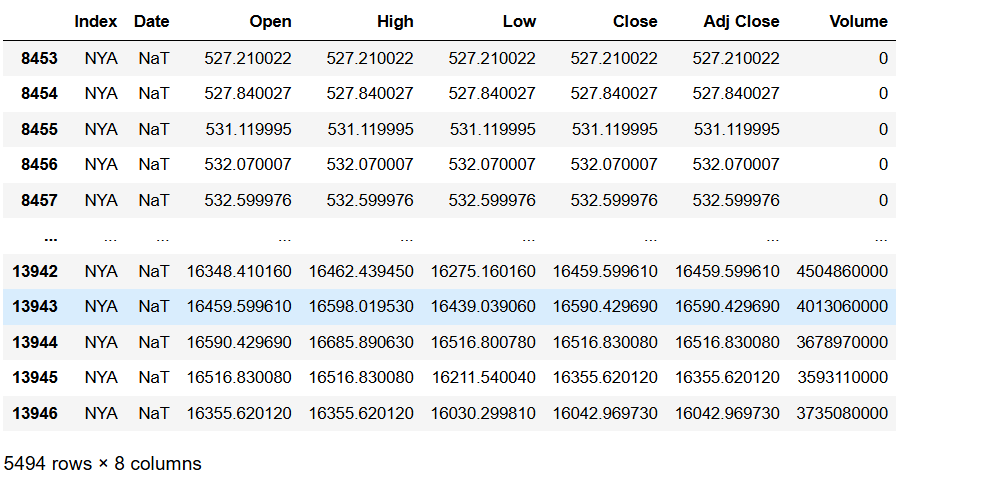
**# Print the columns with NaN values**

print(nan\_values)



**#getting the exact place of the NaN Values in the data for making better decisions for the fate of NaN values**

df.loc[pd.isna(df.values) , :]



**# Drop rows where 'Date' is NaN**

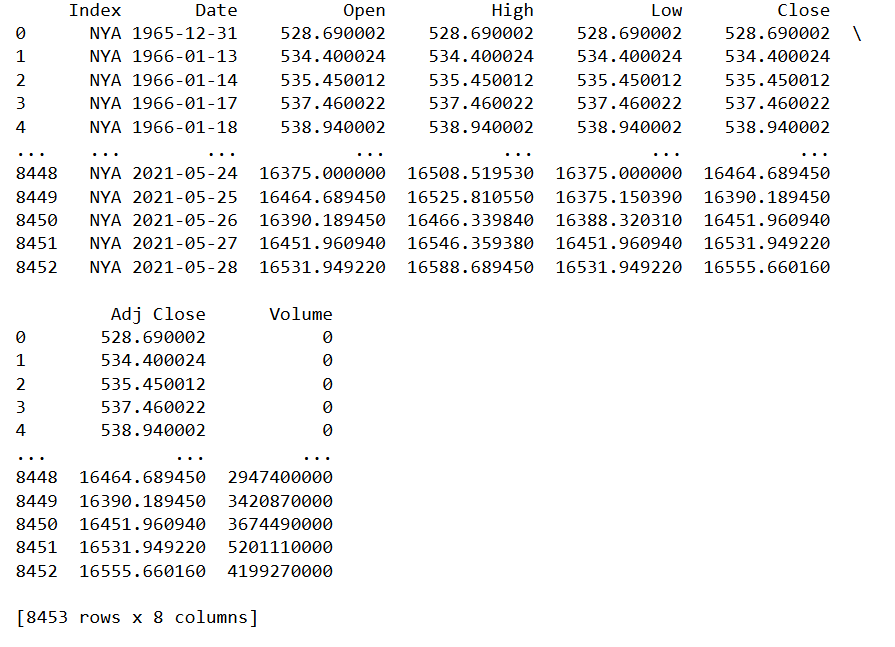
df = df[pd.notna(df['Date'])]

**# Reset the index**

df = df.reset\_index(drop=True)

**# Display the updated DataFrame**

print(df)



**#getting the NaN Values again in the data for making better decisions for the fate of NaN values**

df.loc[pd.isna(df.values) , :]



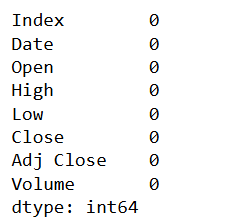
**#filling the NaN values with 'pad' method**

#because in each row that has a NaN value all the other values are repeated and are the same

df = df.fillna(method = 'pad' , axis = 1)

**#seeing if there are any NaN values left**

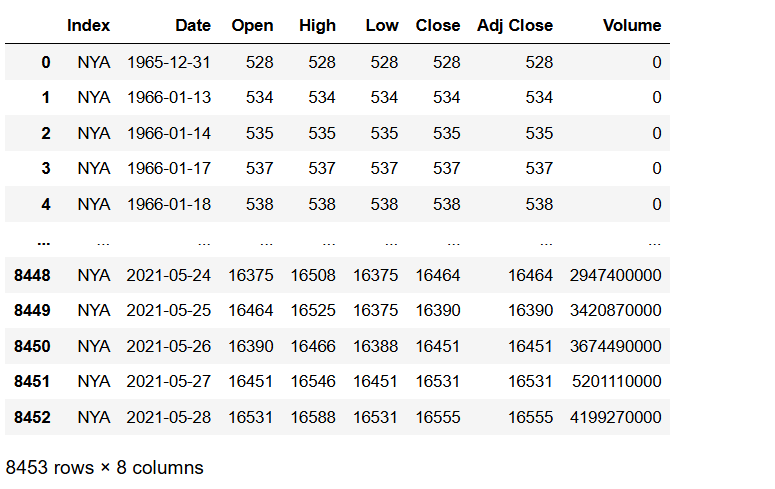
df.isna().sum()



**#rounding the numbers and making them int instead of float**

df.iloc[:,2:7] = df.iloc[:,2:7] .round(0).astype(int)

df



import mplfinance as mpf

**# Drop unnecessary columns**

df2 = df.drop(columns=['Adj Close', 'Volume', 'Index'])

**# Convert the 'Date' column to a datetime index**

df2.index = pd.to\_datetime(df2['Date'])

**# Fill NaN values with a forward fill method**

df2 = df2.ffill()

**# Create the Line chart**

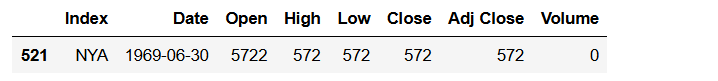
mpf.plot(df2, type='line', title="Line Chart", style='yahoo', figsize=(12, 8))



**#dealing with the noises in " 'Open' , 'High' , 'Adj Close' "**

**#locating the noise in 'Open'**

df.loc[(df['Date']< '1971-12-31') & (df['Open'] > 5500)]

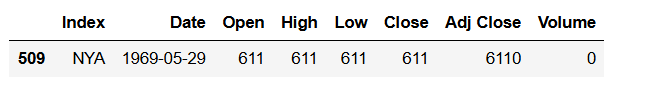


**#correcting the noise in 'Open' on 521 index**

df.at[521,'Open'] = 572

**#locating the noise in 'Adj Close'**

df.loc[(df['Date'] < '1971-12-31') & (df['Adj Close'] > 5500)]

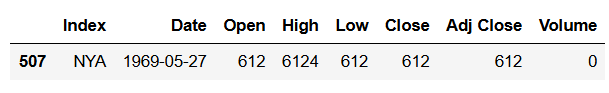


**#correcting the noise in 'Adj Close' on 509 index**

df.at[509,'Adj Close'] = 611

**#locating the noise in 'High'**

df.loc[(df['Date'] < '1971-12-31') & (df['High'] > 5500)]



**#correcting the noises in 'High' on 507 indexes**

df.at[507,'High'] = 612

import mplfinance as mpf

**# Drop unnecessary columns**

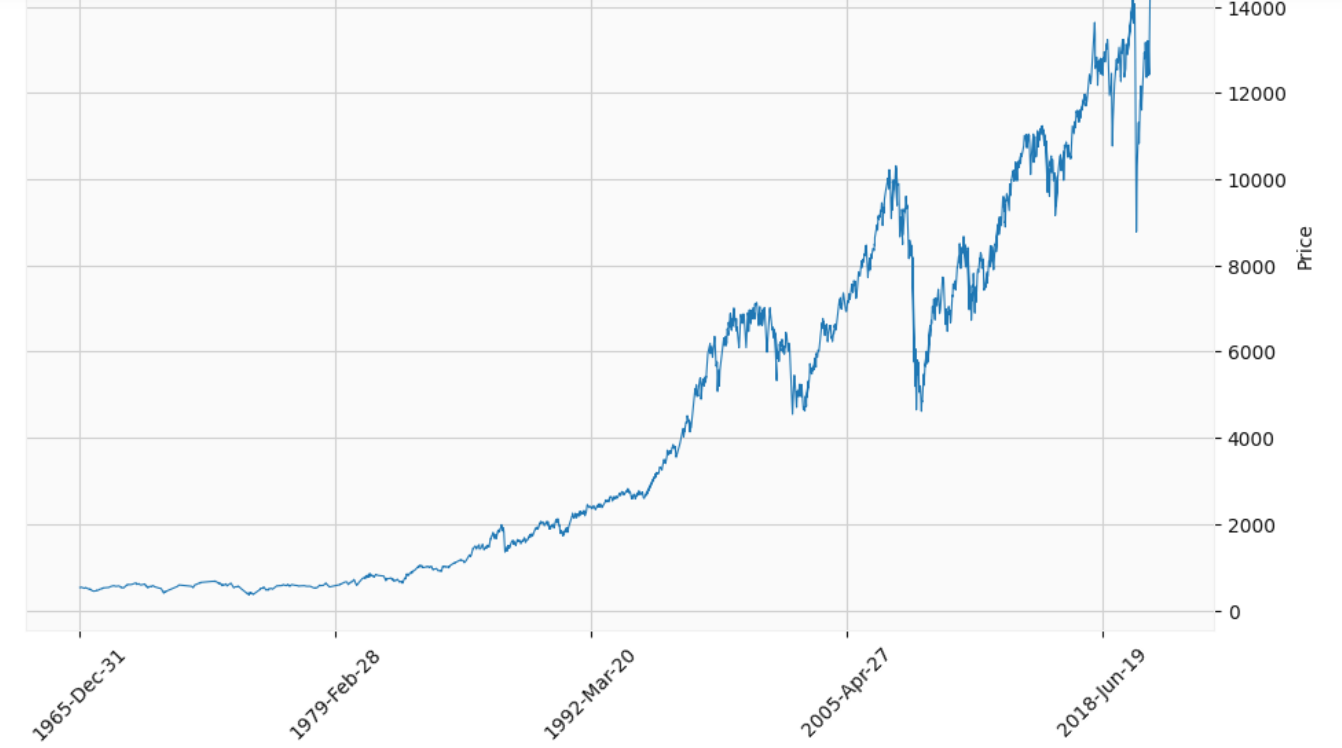
df2 = df.drop(columns=['Adj Close', 'Volume', 'Index'])

**# Convert the 'Date' column to a datetime index**

df2.index = pd.to\_datetime(df2['Date'])

**# Create the line chart**

mpf.plot(df2, type='line', title="Line Chart", style='yahoo', figsize=(12, 8))

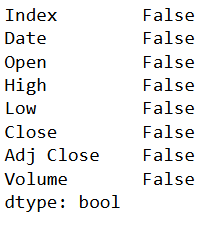


**# Check for NaN values in the DataFrame**

nan\_values = df.isna().any()

**# Print the columns with NaN values**

print(nan\_values)



**# Normalize the data**

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

scaled\_data = scaler.fit\_transform(data[['Close']])

print(data.columns)



**# Split the data into training and testing sets**

train\_size = int(len(scaled\_data) \* 0.8)

train\_data = scaled\_data[:train\_size]

test\_data = scaled\_data[train\_size:]

**# Create sequences for training and testing**

def create\_sequences(data, seq\_length):

sequences = []

for i in range(len(data) - seq\_length):

X = data[i:i+seq\_length]

y = data[i+seq\_length]

sequences.append((X, y))

return np.array(sequences, dtype=object)

sequence\_length = 10

train\_sequences = create\_sequences(train\_data, sequence\_length)

test\_sequences = create\_sequences(test\_data, sequence\_length)

X\_train = np.array([x for x, \_ in train\_sequences])

y\_train = np.array([y for \_, y in train\_sequences])

X\_test = np.array([x for x, \_ in test\_sequences])

y\_test = np.array([y for \_, y in test\_sequences])

**# Create the LSTM model**

model = Sequential()

model.add(LSTM(units=100, return\_sequences=True, input\_shape=(X\_train.shape[1], X\_train.shape[2])))

model.add(Dropout(0.2))

model.add(LSTM(units=100))

model.add(Dropout(0.2))

model.add(Dense(units=1))

**# Compile the model**

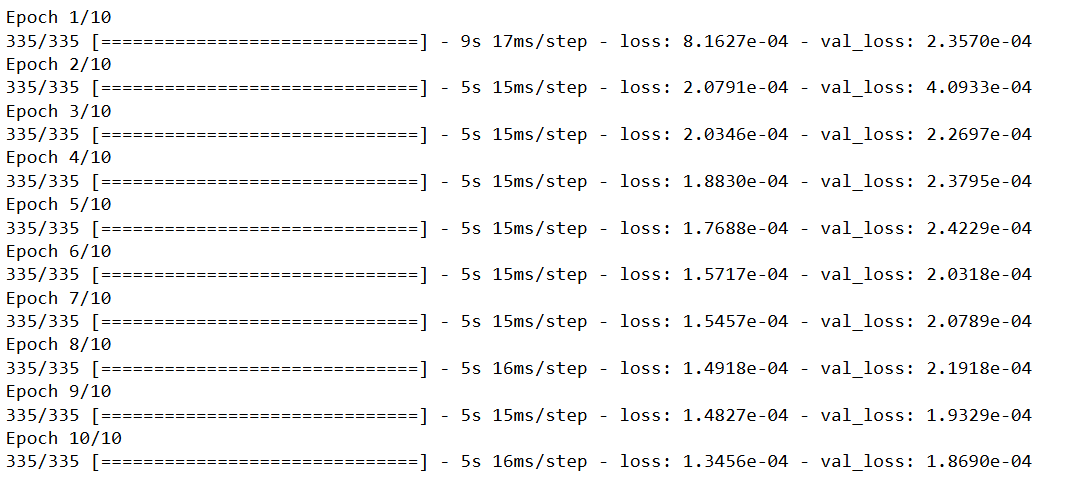
model.compile(optimizer='adam', loss='mean\_squared\_error')

**# Implement early stopping**

early\_stopping = EarlyStopping(patience=10, restore\_best\_weights=True)

**# Train the model**

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.04, callbacks=[early\_stopping])



**# Evaluate the model**

predictions = model.predict(X\_test)

**# Inverse transform**

predictions = scaler.inverse\_transform(predictions)

y\_test = scaler.inverse\_transform(y\_test)



**# Check the shapes of predictions and y\_test**

print("Shape of predictions:", predictions.shape)

print("Shape of y\_test:", y\_test.shape)

**# Ensure y\_test has the same length as predictions**

y\_test = y\_test[:len(predictions)]

**# Check the shapes again after ensuring the same length**

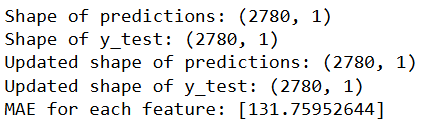
print("Updated shape of predictions:", predictions.shape)

print("Updated shape of y\_test:", y\_test.shape)

**# Calculate the Mean Absolute Error for each feature**

MAE = np.mean(np.abs(predictions - y\_test), axis=0)

print(f'MAE for each feature: {MAE}')



**# Plot training and validation loss**

plt.figure(figsize=(12, 6))

plt.plot(history.history['loss'], label='Training Loss')

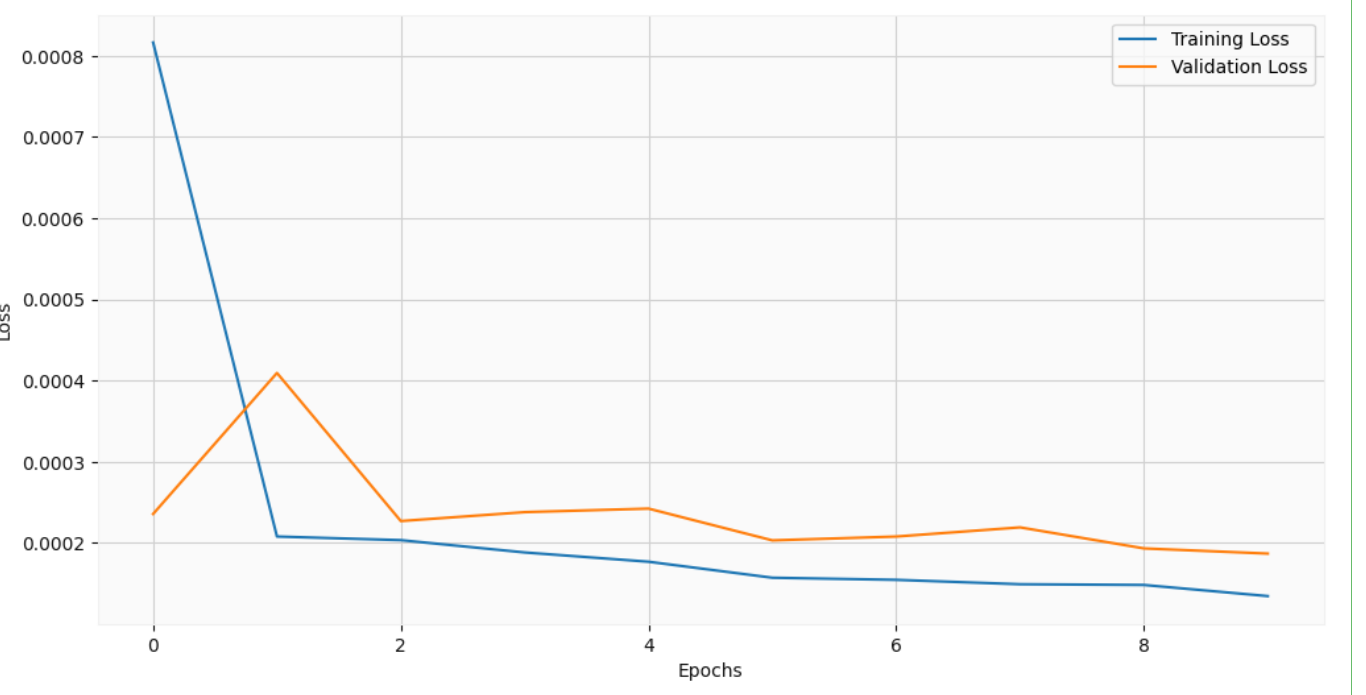
plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()



**# Plot the actual vs. predicted prices**

plt.figure(figsize=(12, 6))

plt.plot(y\_test, label='Actual')

plt.plot(predictions, label='Predicted')

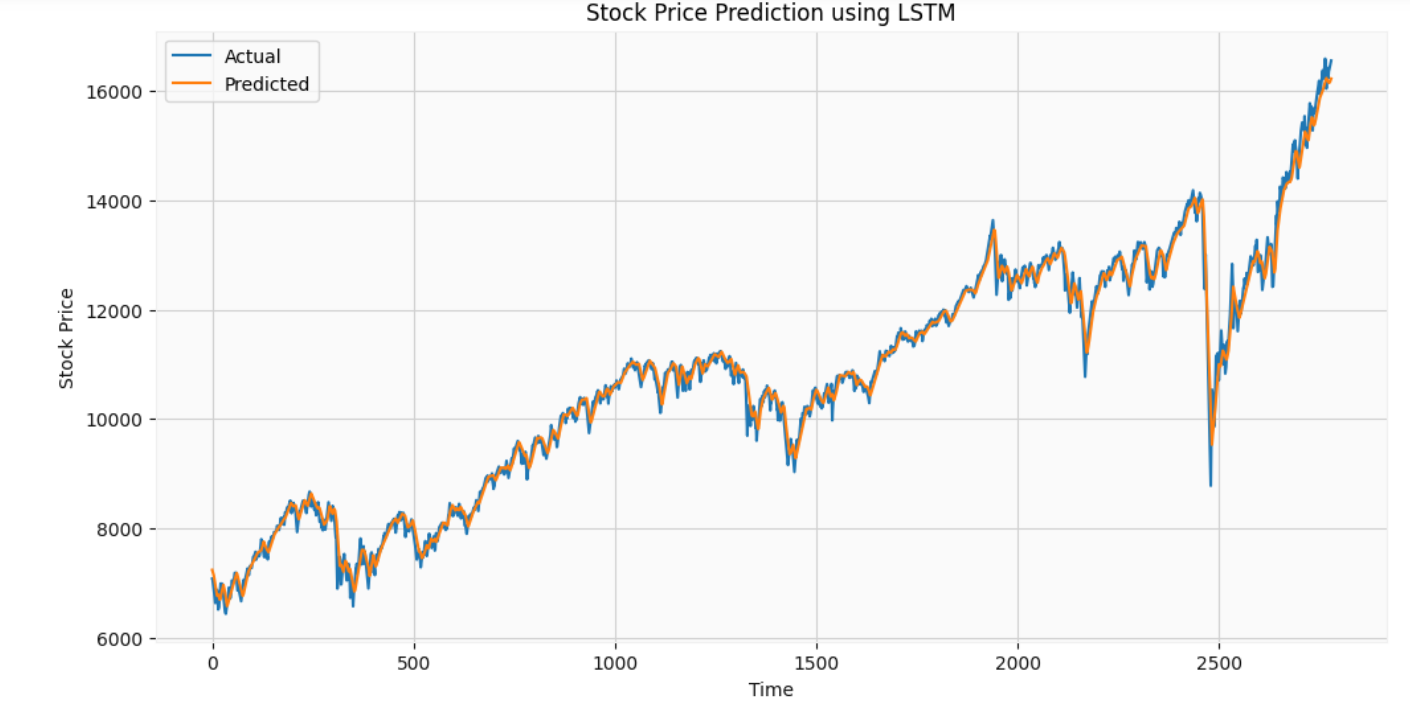
plt.legend()

plt.xlabel('Time')

plt.ylabel('Stock Price')

plt.title('Stock Price Prediction using LSTM')

plt.show()



**# Calculate the Mean Squared Error (MSE)**

MSE = mean\_squared\_error(y\_test, predictions)

**# Calculate the Root Mean Squared Error (RMSE)**

RMSE = np.sqrt(MSE)

print(f'MSE: {MSE}')

print(f'RMSE: {RMSE}')

