**Lab Assignment 10**

**Neural Network & Deep Learning**

**Stock Market Prediction using LSTM**

Name: Nisha Kini

Roll No: 21-02-24

|  |  |
| --- | --- |
| Roll No. C050 | Name: Nisha Kini |
| Program :BTI | Division: B |
| Batch: B2 | Date of Experiment: 21-02-24 |
| Date of Submission: : 21-02-24 | Grade : |

Aim: **Stock Market Prediction using LSTM**

Original file is located at

    https://colab.research.google.com/drive/1WaHhukk52k\_Lf0iLg7sT8Q5S-3oZtNBT

"""

# Make sure that you have all these libaries available to run the code successfully

from pandas\_datareader import data

import matplotlib.pyplot as plt

import pandas as pd

import datetime as dt

import urllib.request, json

import os

import numpy as np

import tensorflow as tf # This code has been tested with TensorFlow 1.6

from sklearn.preprocessing import MinMaxScaler

df = pd.read\_csv('/content/NESTLEIND.csv')

# Sort DataFrame by date

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

# Double check the result

df.head()

# First calculate the mid prices from the highest and lowest

high\_prices = df['High'].values

low\_prices = df['Low'].values

mid\_prices = (high\_prices + low\_prices) / 2.0

import numpy as np

# First calculate the mid prices from the highest and lowest

high\_prices = df['High'].to\_numpy()

low\_prices = df['Low'].to\_numpy()

mid\_prices = (high\_prices + low\_prices) / 2.0

plt.figure(figsize = (18,9))

plt.plot(range(df.shape[0]),(df['Low']+df['High'])/2.0)

plt.xticks(range(0,df.shape[0],500),df['Date'].loc[::500],rotation=45)

plt.xlabel('Date',fontsize=18)

plt.ylabel('Mid Price',fontsize=18)

plt.show()

train\_data = mid\_prices[:11000]

test\_data = mid\_prices[11000:]

# Scale the data to be between 0 and 1

# When scaling remember! You normalize both test and train data with respect to training data

# Because you are not supposed to have access to test data

scaler = MinMaxScaler()

train\_data = train\_data.reshape(-1,1)

test\_data = test\_data.reshape(-1,1)

# Scale the data to be between 0 and 1

# When scaling remember! You normalize both test and train data with respect to training data

# Because you are not supposed to have access to test data

scaler = MinMaxScaler()

train\_data = train\_data.reshape(-1,1)

test\_data = test\_data.reshape(-1,1)

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

# Assuming 'Close' is the feature you want to normalize

feature\_column = 'Close'

# Extract the feature column

train\_data = df[[feature\_column]].values

# Initialize the scaler

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

# Define the smoothing window size

smoothing\_window\_size = 2500

# Train the scaler with training data and smooth the data

for di in range(0, len(train\_data), smoothing\_window\_size):

    window\_data = train\_data[di:di+smoothing\_window\_size, :]

    # Check if there is sufficient data to fit the scaler

    if len(window\_data) > 0:

        scaler.fit(window\_data)

        train\_data[di:di+smoothing\_window\_size, :] = scaler.transform(window\_data)

# Normalize the last bit of remaining data

if di+smoothing\_window\_size < len(train\_data):

    remaining\_data = train\_data[di+smoothing\_window\_size:, :]

    if len(remaining\_data) > 0:

        scaler.fit(remaining\_data)

        train\_data[di+smoothing\_window\_size:, :] = scaler.transform(remaining\_data)

# Now perform exponential moving average smoothing

# So the data will have a smoother curve than the original ragged data

EMA = 0.0

gamma = 0.1

for ti in range(len(train\_data)):

    EMA = gamma \* train\_data[ti] + (1 - gamma) \* EMA

    train\_data[ti] = EMA

# Reshape test\_data if it's 2D

if len(test\_data.shape) > 1:

    test\_data = test\_data[:, 0]

# Used for visualization and test purposes

all\_mid\_data = np.concatenate([train\_data, test\_data], axis=0)

import datetime as dt

import numpy as np

window\_size = 100

N = len(df)  # Assuming df is your DataFrame with time series data

train\_data = df['Close'].values  # Assuming 'Close' column contains the target variable

std\_avg\_predictions = []

std\_avg\_x = []

mse\_errors = []

for pred\_idx in range(window\_size, N):

    if pred\_idx >= N:

        # If prediction index is beyond the available data, add one day to the last date

        date = df['Date'].iloc[-1] + dt.timedelta(days=1)

    else:

        date = df.loc[pred\_idx, 'Date']

    # Check for missing or invalid values in the training data

    if np.any(np.isnan(train\_data[pred\_idx - window\_size:pred\_idx])):

        continue

    # Calculate standard average prediction

    std\_avg\_predictions.append(np.mean(train\_data[pred\_idx - window\_size:pred\_idx]))

    # Check for missing or invalid values in the target variable

    if np.isnan(train\_data[pred\_idx]):

        continue

    # Calculate Mean Squared Error (MSE)

    mse\_errors.append((std\_avg\_predictions[-1] - train\_data[pred\_idx]) \*\* 2)

    std\_avg\_x.append(date)

# Check if there are any NaN values in the MSE\_errors

if np.any(np.isnan(mse\_errors)):

    print("Warning: NaN values present in MSE\_errors.")

# Calculate MSE only if there are valid values

if len(mse\_errors) > 0:

    mse = 0.5 \* np.nanmean(mse\_errors)

    mse = mse/100000000

    print('MSE error for standard averaging: %.5f' % mse)

else:

    print("No valid MSE calculation due to missing or invalid values.")

plt.figure(figsize=(18, 9))

plt.plot(range(len(all\_mid\_data)), all\_mid\_data, color='b', label='True')

plt.plot(range(window\_size, window\_size + len(std\_avg\_predictions)), std\_avg\_predictions, color='orange', label='Prediction')

plt.xlabel('Index')

plt.ylabel('Mid Price')

plt.legend(fontsize=18)

plt.show()

import datetime as dt

import numpy as np

window\_size = 100

N = len(df)  # Assuming df is your DataFrame with time series data

train\_data = df['Close'].values  # Assuming 'Close' column contains the target variable

run\_avg\_predictions = []

run\_avg\_x = []

mse\_errors = []

running\_mean = train\_data[0]

run\_avg\_predictions.append(running\_mean)

decay = 0.5

for pred\_idx in range(1, N):

    if np.isnan(train\_data[pred\_idx]):

        continue  # Skip iterations with missing or invalid values

    running\_mean = running\_mean \* decay + (1.0 - decay) \* train\_data[pred\_idx - 1]

    run\_avg\_predictions.append(running\_mean)

    mse\_errors.append((run\_avg\_predictions[-1] - train\_data[pred\_idx]) \*\* 2)

    run\_avg\_x.append(pred\_idx)  # Use index as a substitute for 'Date'

# Drop NaN values from the MSE\_errors

mse\_errors = [mse for mse in mse\_errors if not np.isnan(mse)]

# Check if there are any NaN values remaining

if np.any(np.isnan(mse\_errors)):

    print("Warning: NaN values present in MSE\_errors after dropping NaN values.")

# Calculate MSE only if there are valid values

if len(mse\_errors) > 0:

    mse = 0.5 \* np.mean(mse\_errors)

    mse = mse / 100000000

    print('MSE error for EMA averaging (after dropping NaN values): %.5f' % mse)

else:

    print("No valid MSE calculation due to missing or invalid values.")

import matplotlib.pyplot as plt

# Ensure both arrays have the same length

all\_mid\_data = all\_mid\_data[:len(train\_data)]

plt.figure(figsize=(18, 9))

plt.plot(range(len(train\_data)), all\_mid\_data, color='b', label='True')

plt.plot(range(window\_size, window\_size + len(truncated\_predictions)), truncated\_predictions, color='orange', label='Prediction')

plt.xlabel('Index')

plt.ylabel('Mid Price')

plt.legend(fontsize=18)

plt.show()

class DataGeneratorSeq(object):

    def \_\_init\_\_(self,prices,batch\_size,num\_unroll):

        self.\_prices = prices

        self.\_prices\_length = len(self.\_prices) - num\_unroll

        self.\_batch\_size = batch\_size

        self.\_num\_unroll = num\_unroll

        self.\_segments = self.\_prices\_length //self.\_batch\_size

        self.\_cursor = [offset \* self.\_segments for offset in range(self.\_batch\_size)]

    def next\_batch(self):

        batch\_data = np.zeros((self.\_batch\_size),dtype=np.float32)

        batch\_labels = np.zeros((self.\_batch\_size),dtype=np.float32)

        for b in range(self.\_batch\_size):

            if self.\_cursor[b]+1>=self.\_prices\_length:

                #self.\_cursor[b] = b \* self.\_segments

                self.\_cursor[b] = np.random.randint(0,(b+1)\*self.\_segments)

            batch\_data[b] = self.\_prices[self.\_cursor[b]]

            batch\_labels[b]= self.\_prices[self.\_cursor[b]+np.random.randint(0,5)]

            self.\_cursor[b] = (self.\_cursor[b]+1)%self.\_prices\_length

        return batch\_data,batch\_labels

    def unroll\_batches(self):

        unroll\_data,unroll\_labels = [],[]

        init\_data, init\_label = None,None

        for ui in range(self.\_num\_unroll):

            data, labels = self.next\_batch()

            unroll\_data.append(data)

            unroll\_labels.append(labels)

        return unroll\_data, unroll\_labels

    def reset\_indices(self):

        for b in range(self.\_batch\_size):

            self.\_cursor[b] = np.random.randint(0,min((b+1)\*self.\_segments,self.\_prices\_length-1))

dg = DataGeneratorSeq(train\_data,5,5)

u\_data, u\_labels = dg.unroll\_batches()

for ui,(dat,lbl) in enumerate(zip(u\_data,u\_labels)):

    print('\n\nUnrolled index %d'%ui)

    dat\_ind = dat

    lbl\_ind = lbl

    print('\tInputs: ',dat )

    print('\n\tOutput:',lbl)

from tensorflow.keras.callbacks import EarlyStopping

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, Dropout

from sklearn.metrics import mean\_squared\_error

from tensorflow.keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

# Step 1: Load the dataset

df = pd.read\_csv('/content/NESTLEIND.csv')

df.head()

# Step 2: Select the appropriate feature (e.g., 'Close' price)

feature\_column = 'Close'

df = df[[feature\_column]]

# Step 3: Normalize the features and convert it into timestamps of 60

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

df\_scaled = scaler.fit\_transform(df)

# Convert data to sequences with a timestep of 60

timestep = 60

X, y = [], []

for i in range(len(df\_scaled) - timestep):

    X.append(df\_scaled[i:i + timestep, 0])

    y.append(df\_scaled[i + timestep, 0])

X, y = np.array(X), np.array(y)

# Split data into training and testing sets

split = int(0.8 \* len(X))

X\_train, X\_test, y\_train, y\_test = X[:split], X[split:], y[:split], y[split:]

# Step 4: Reshape the data for applying to the LSTM model

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))

# Step 5: Create a sequential LSTM model using Keras

model = Sequential()

model.add(LSTM(units=50, return\_sequences=True, input\_shape=(X\_train.shape[1], 1)))

model.add(Dropout(0.2))  # Add dropout layer

model.add(LSTM(units=50, return\_sequences=False))

model.add(Dropout(0.2))  # Add dropout layer

model.add(Dense(units=1))

# Step 6: Compile the model and train it using the training data

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

# Add early stopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train the model with early stopping

model.fit(X\_train, y\_train, epochs=15, batch\_size=32, validation\_split=0.1, callbacks=[early\_stopping])

# Step 7: Predict using the test data

predicted\_price = model.predict(X\_test)

# Inverse transform the predicted values

predicted\_price = scaler.inverse\_transform(np.reshape(predicted\_price, (predicted\_price.shape[0], 1)))

# Calculate and print accuracy

mse = mean\_squared\_error(y\_test, predicted\_price)

print(f'Mean Squared Error: {mse}')

# # Visualize the results

# plt.plot(df.index[split + timestep:], df['Close'].values[split + timestep:], label='Actual Price')

# plt.plot(df.index[split + timestep:], predicted\_price, color='red', label='Predicted Price')

# plt.title('NIFTY50 Stock Price Prediction using LSTM')

# plt.xlabel('Date')

# plt.ylabel('Stock Price')

# plt.legend()

# plt.show()

# Visualize the results

plt.plot(df.index[split + timestep:], df['Close'].values[split + timestep:], label='Actual Price')

plt.plot(df.index[split + timestep:], predicted\_price, color='red', label='Predicted Price')

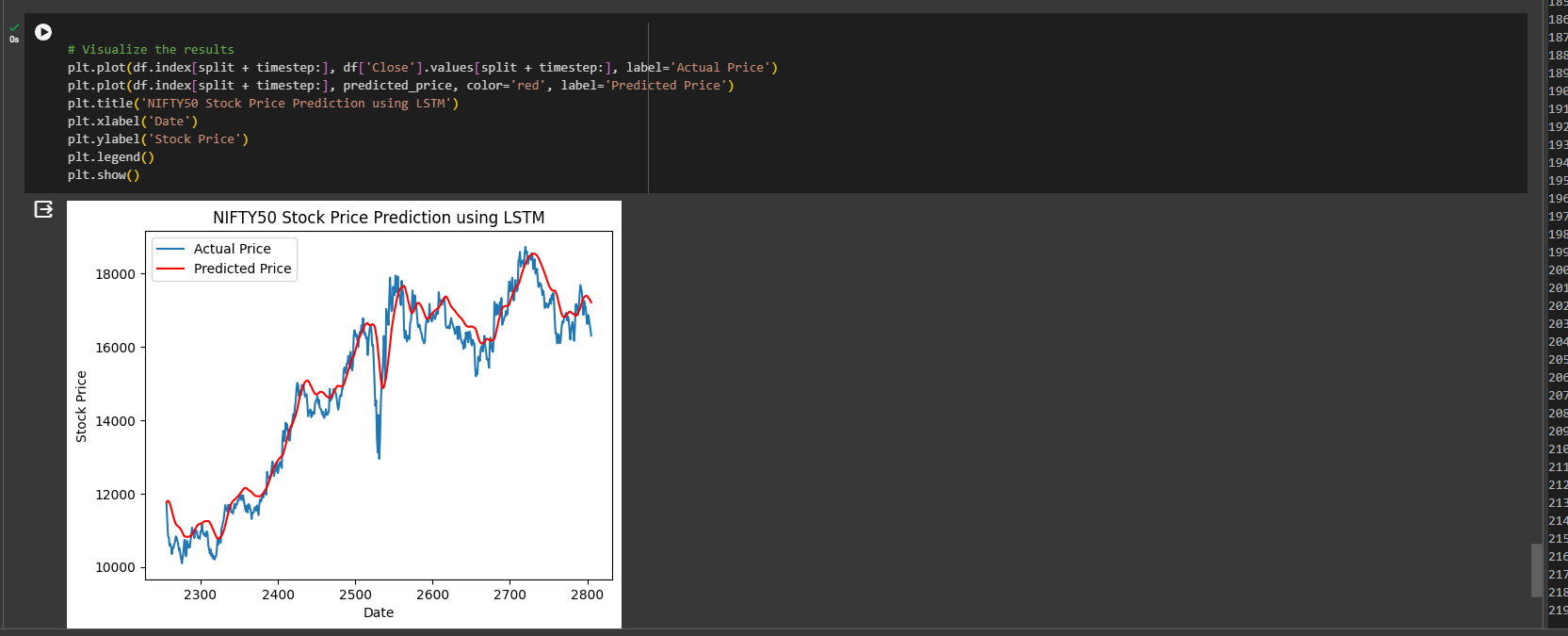
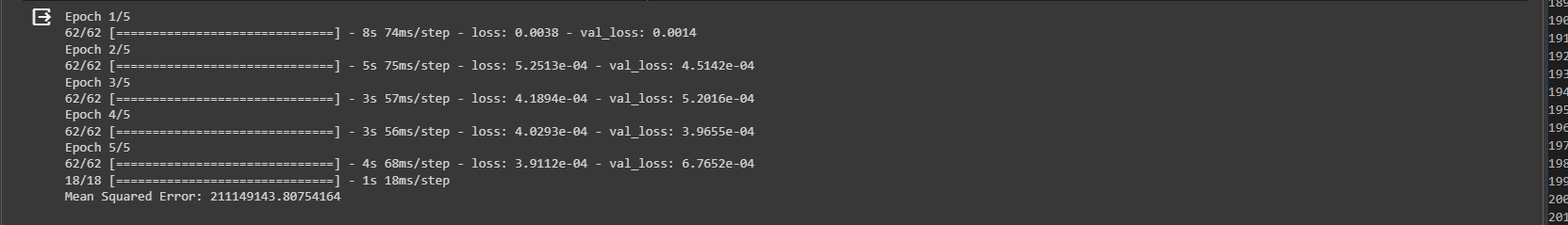
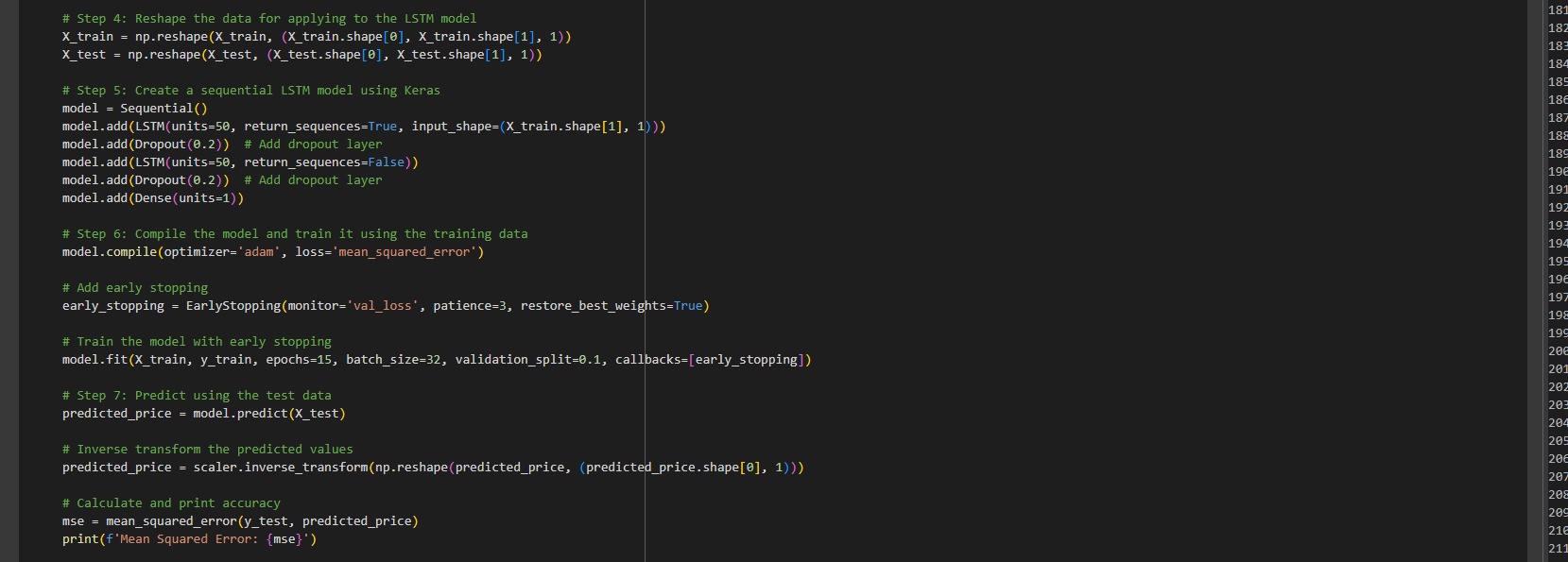
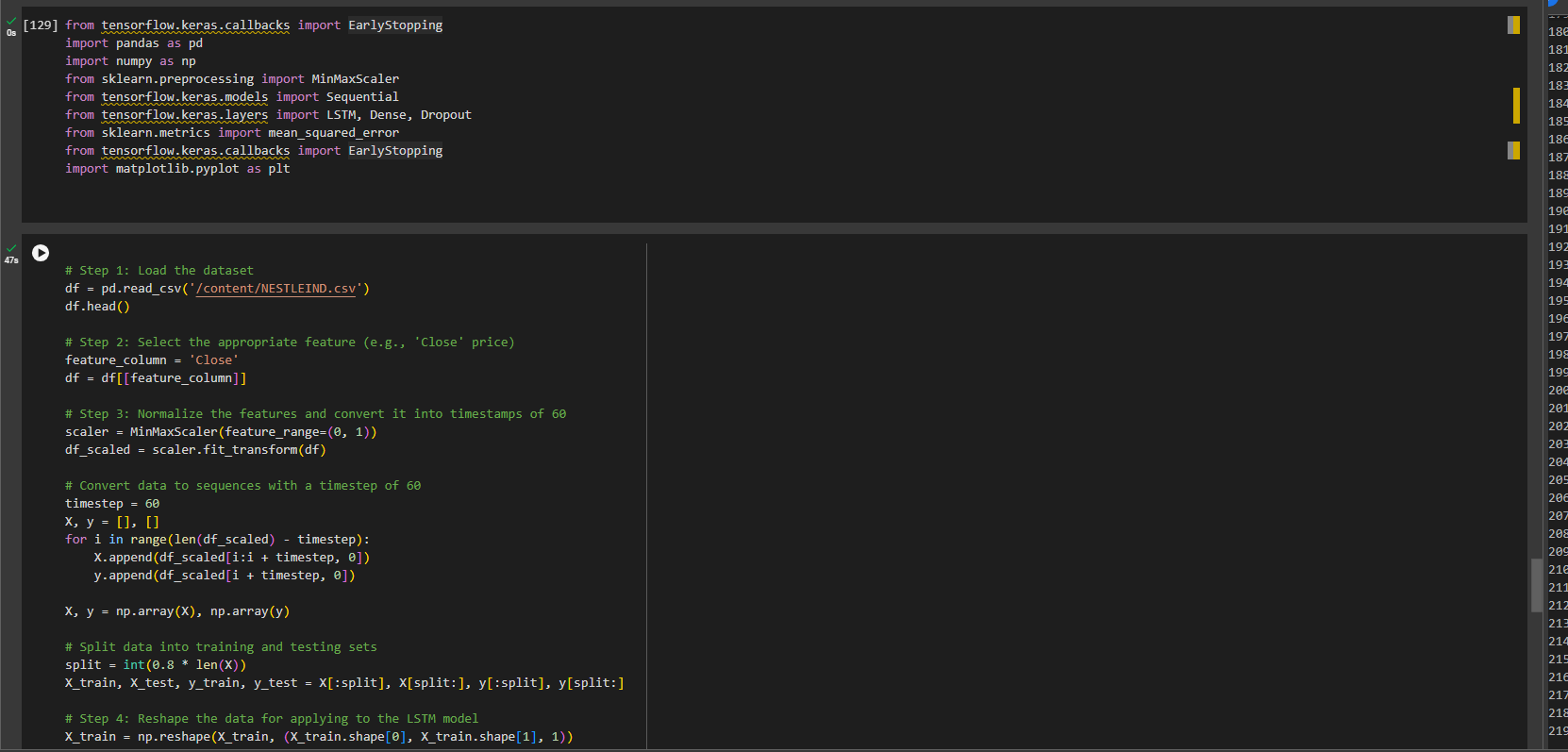
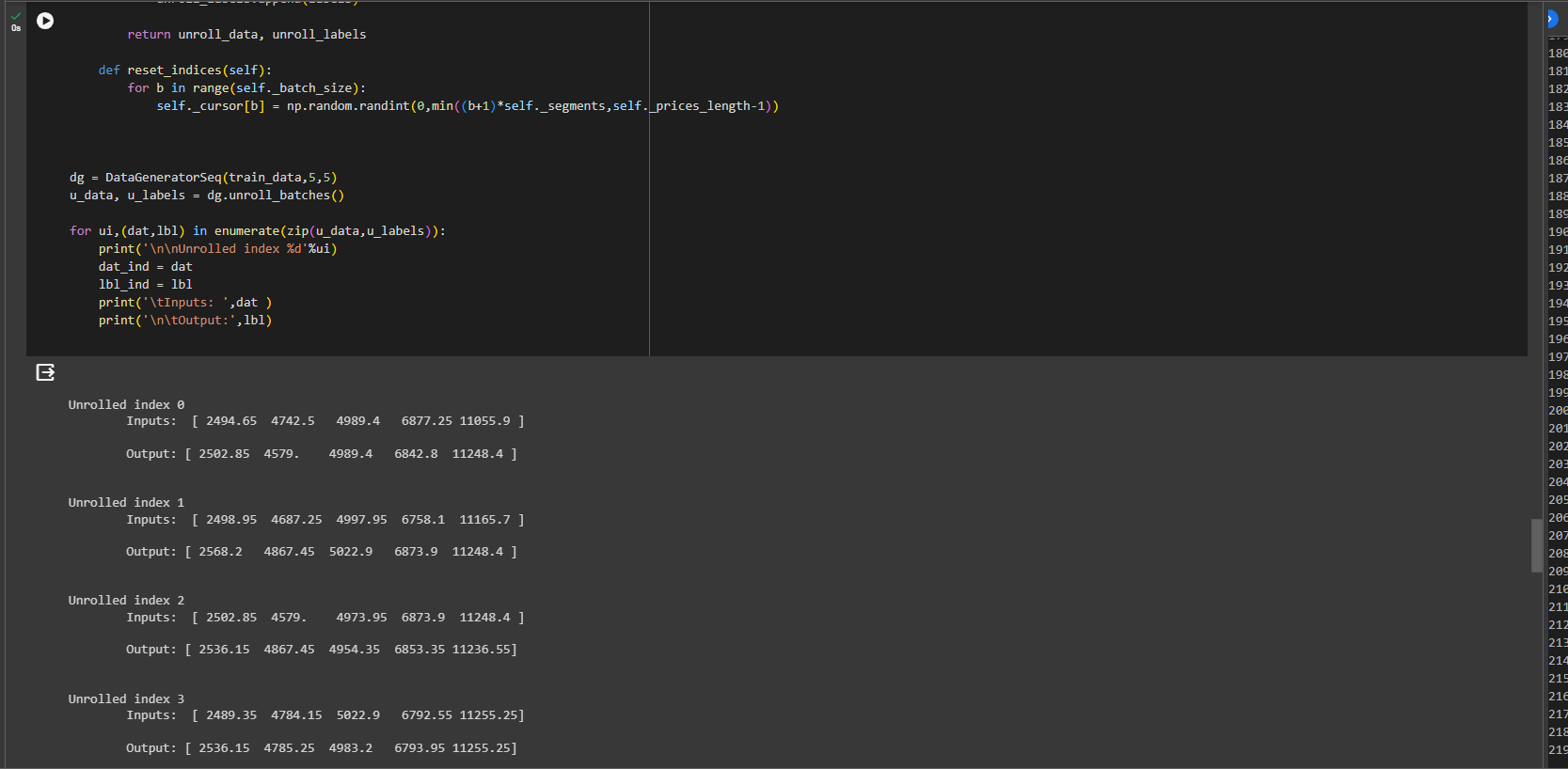
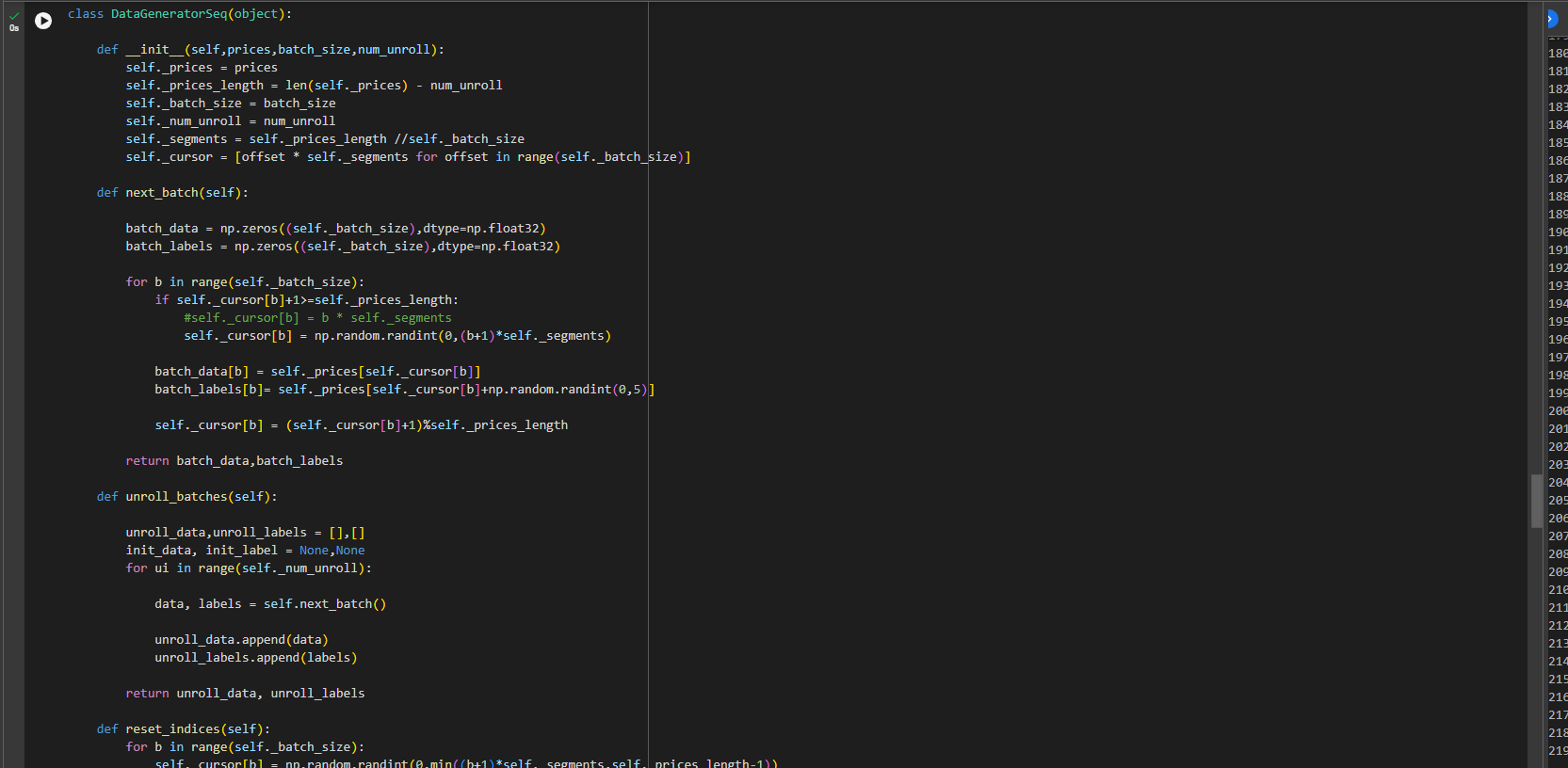
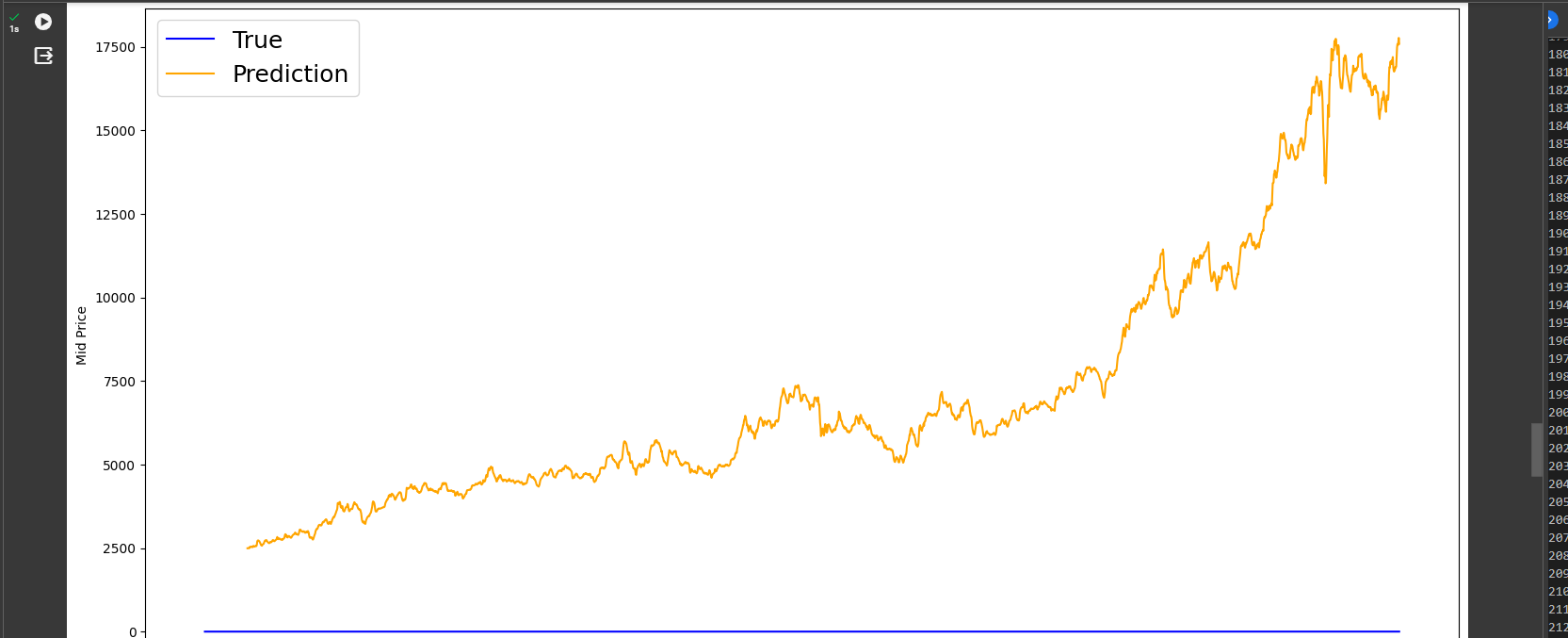
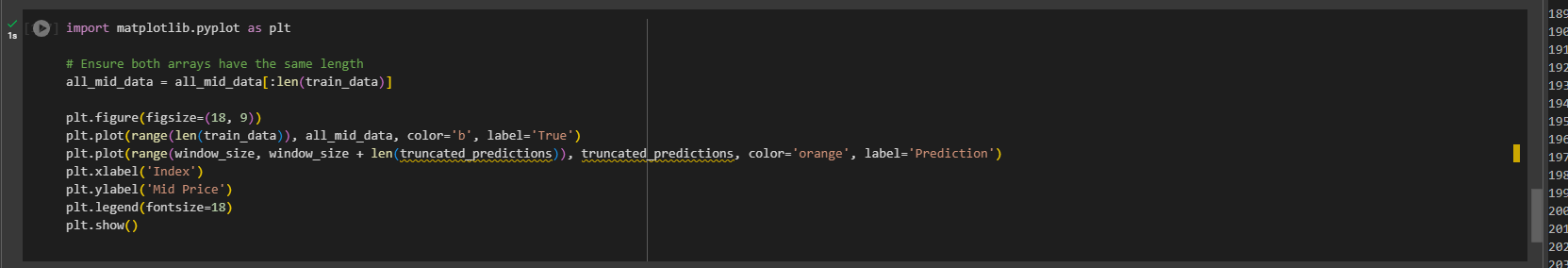
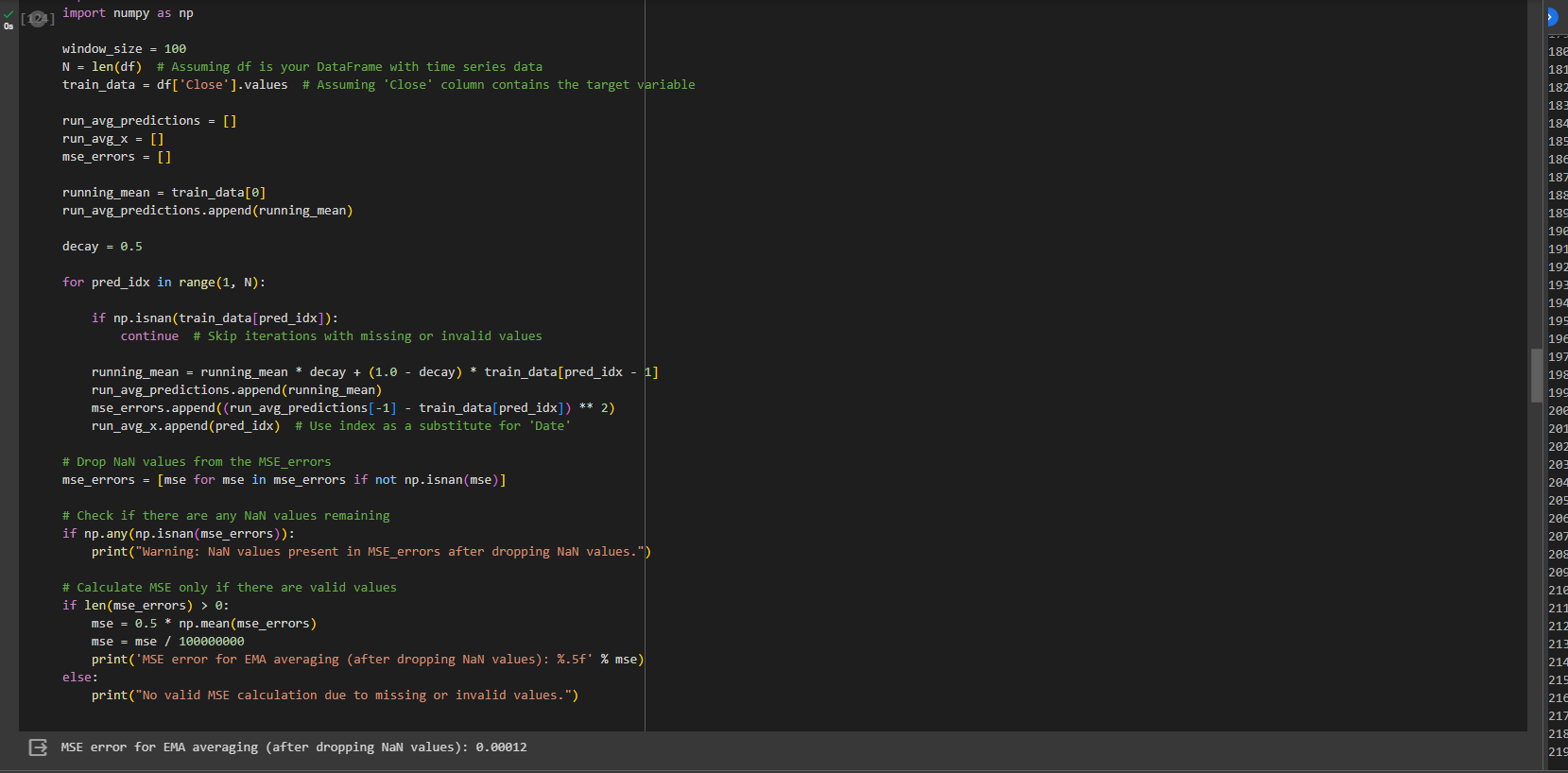
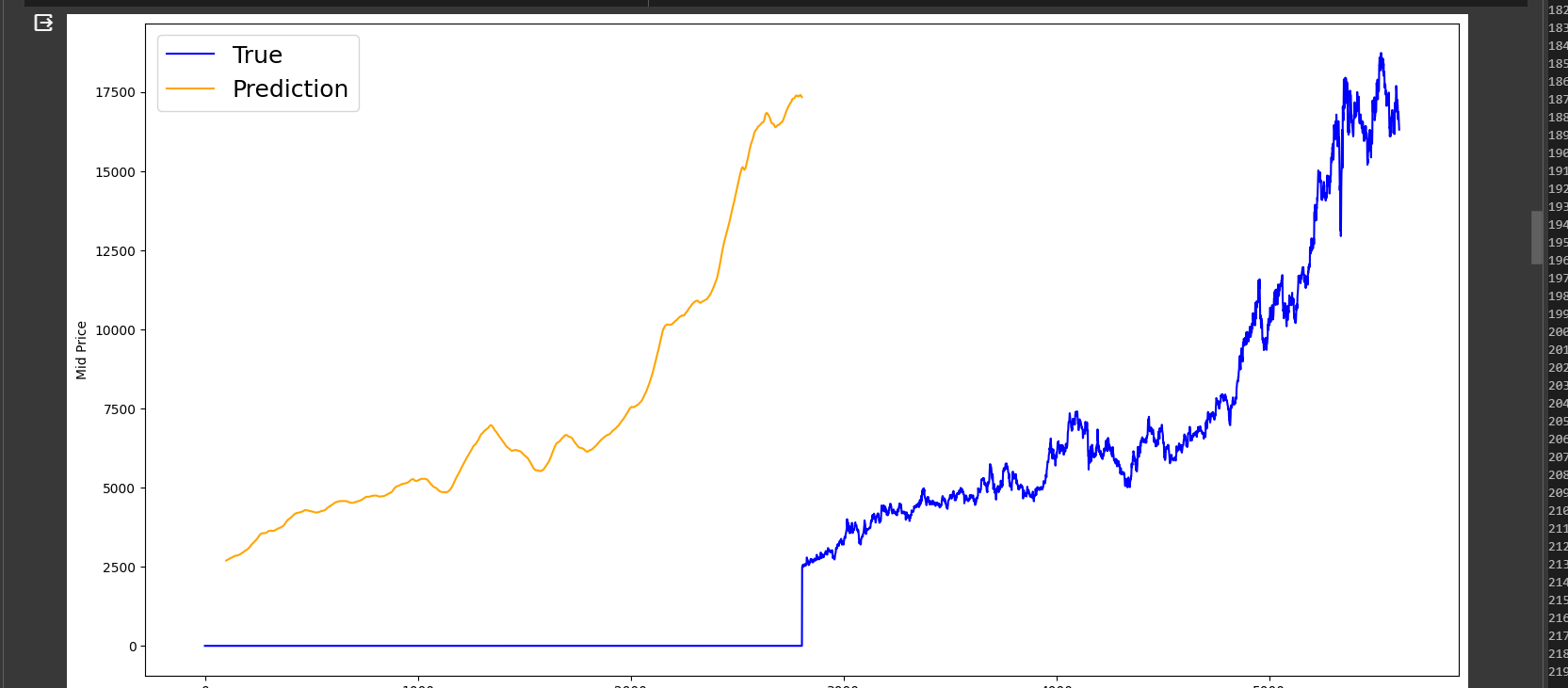
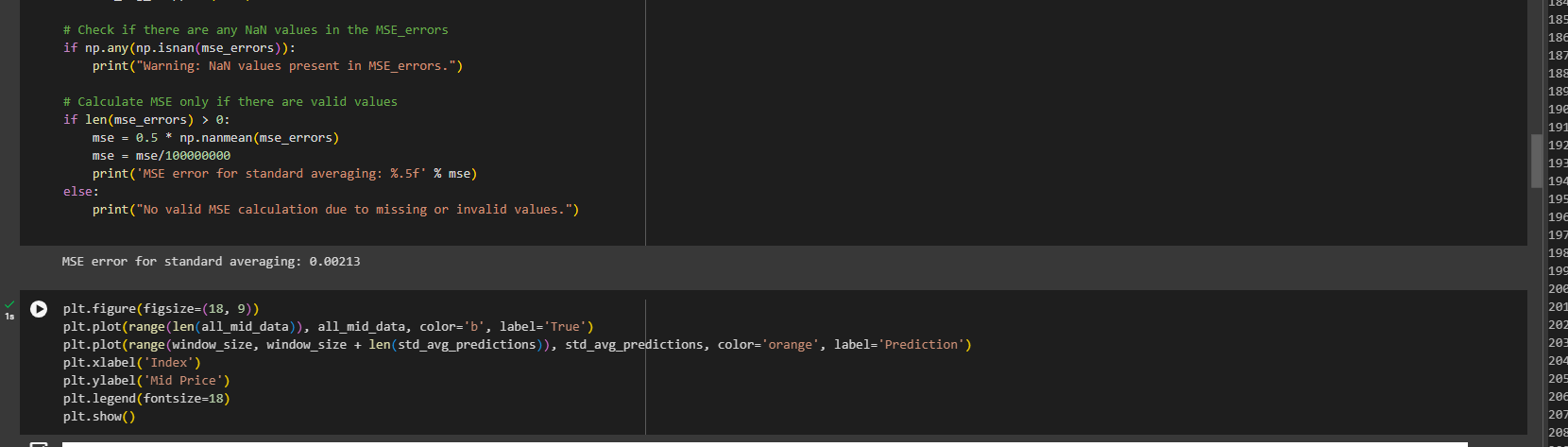
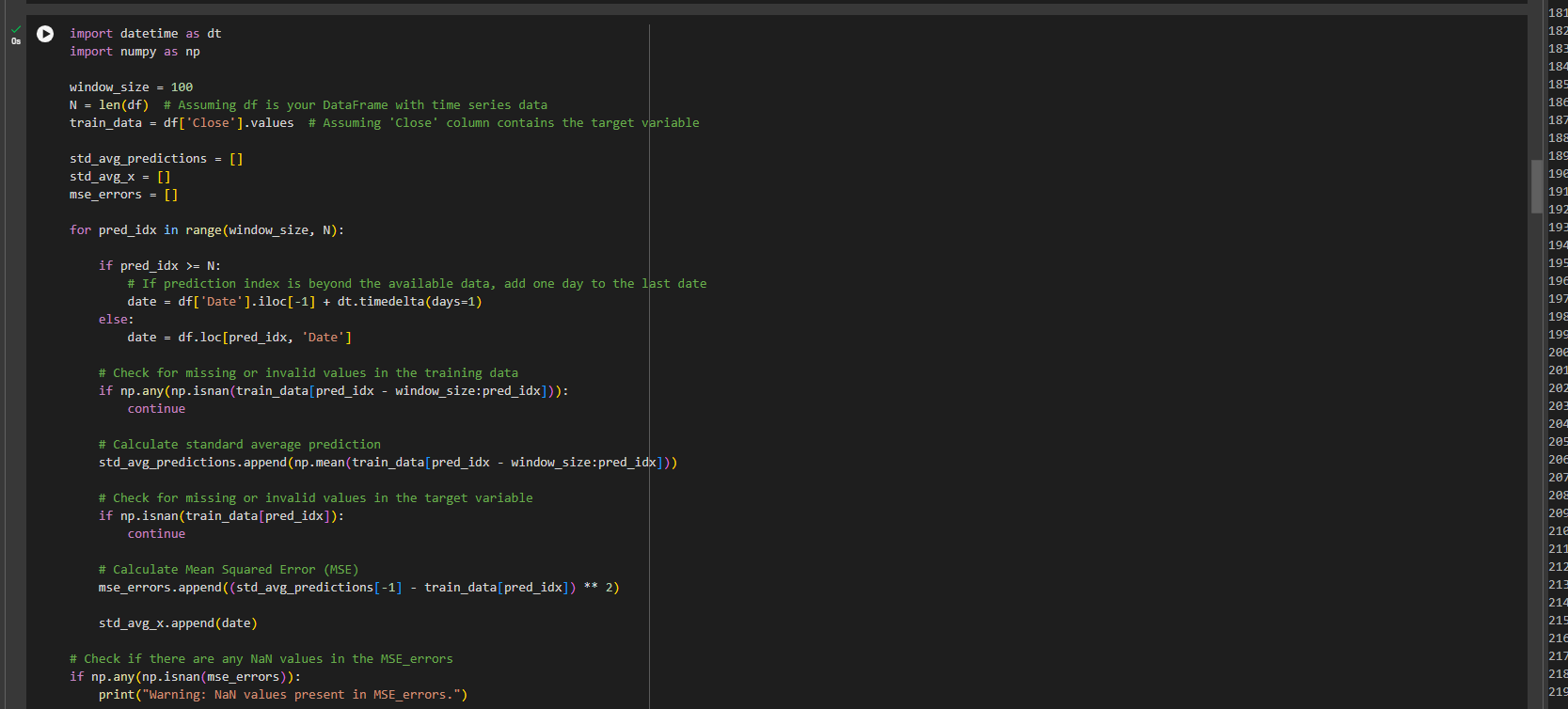
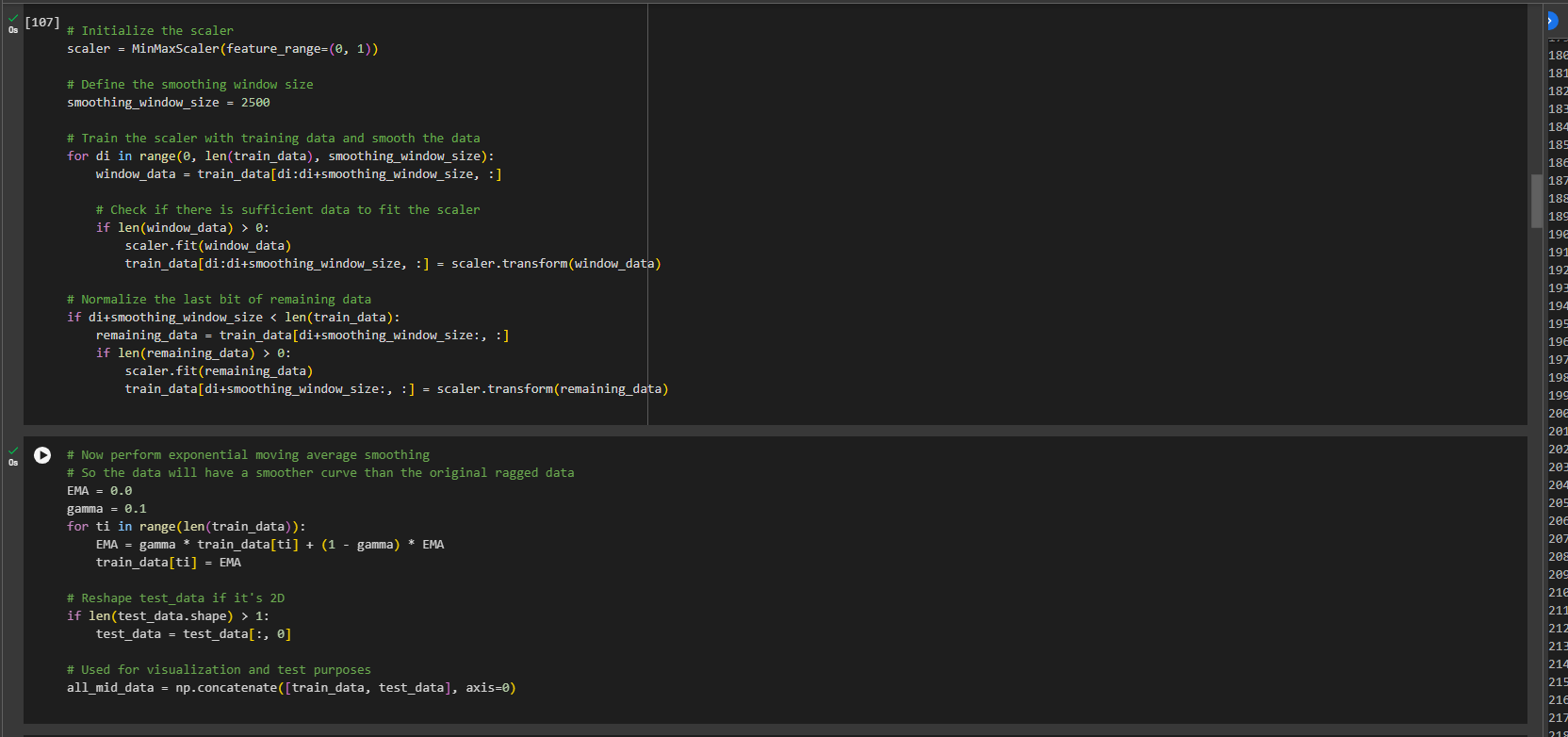
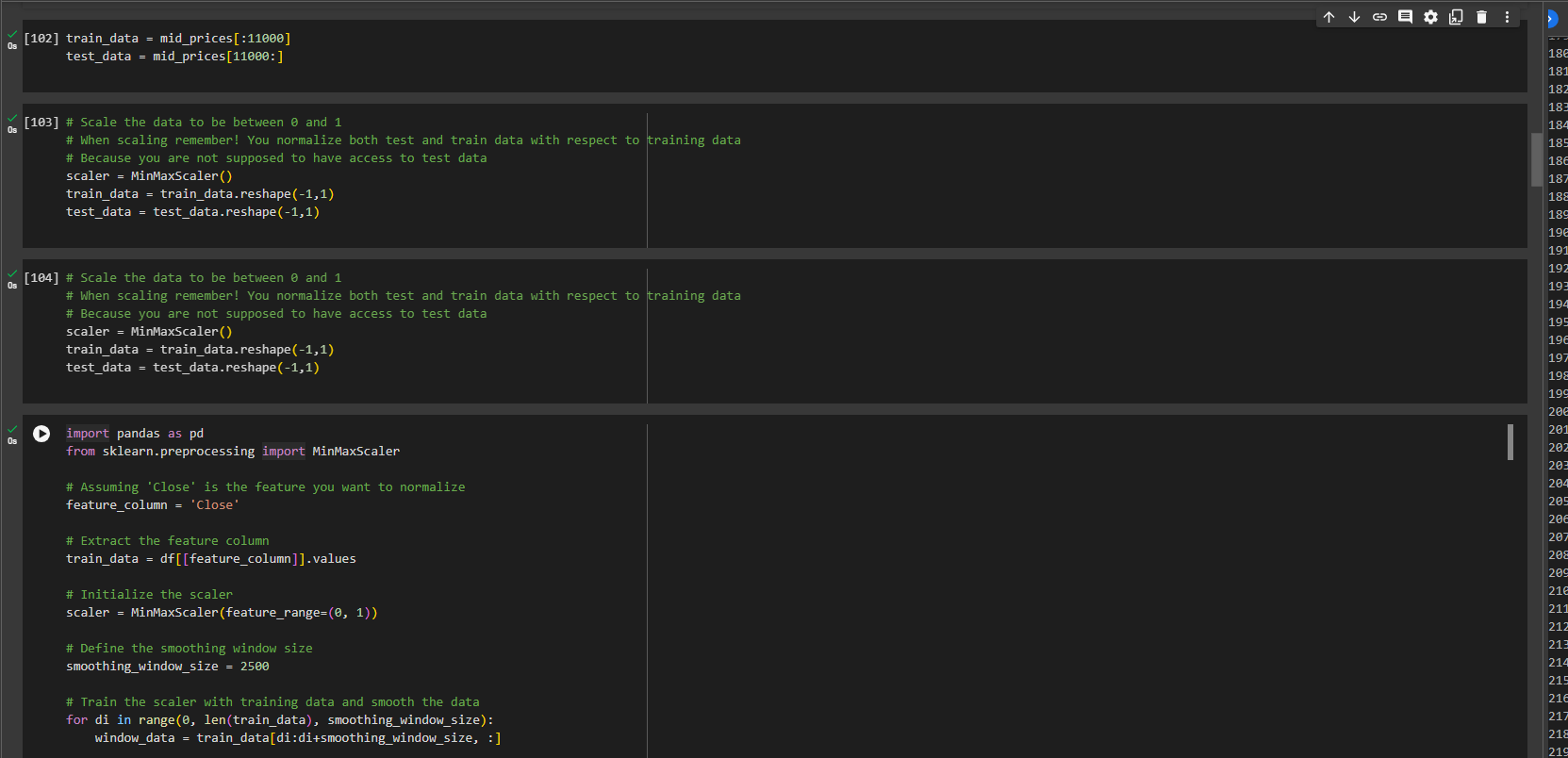
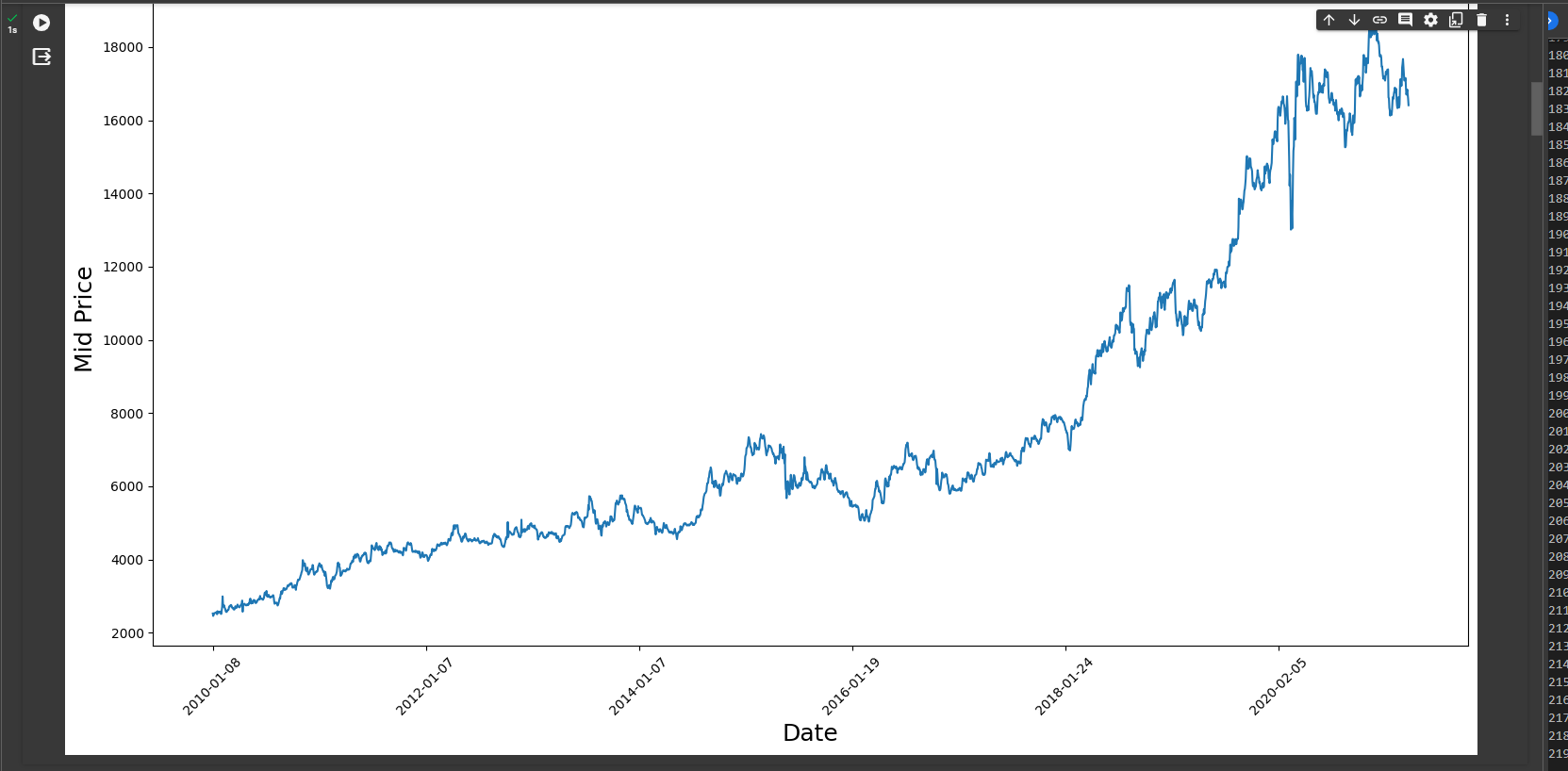
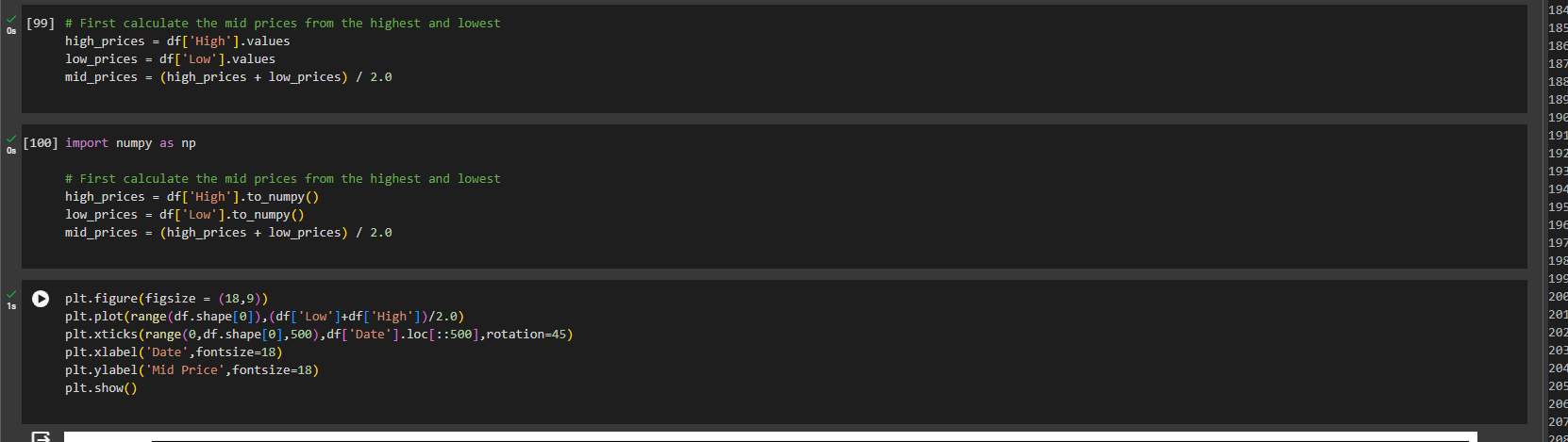
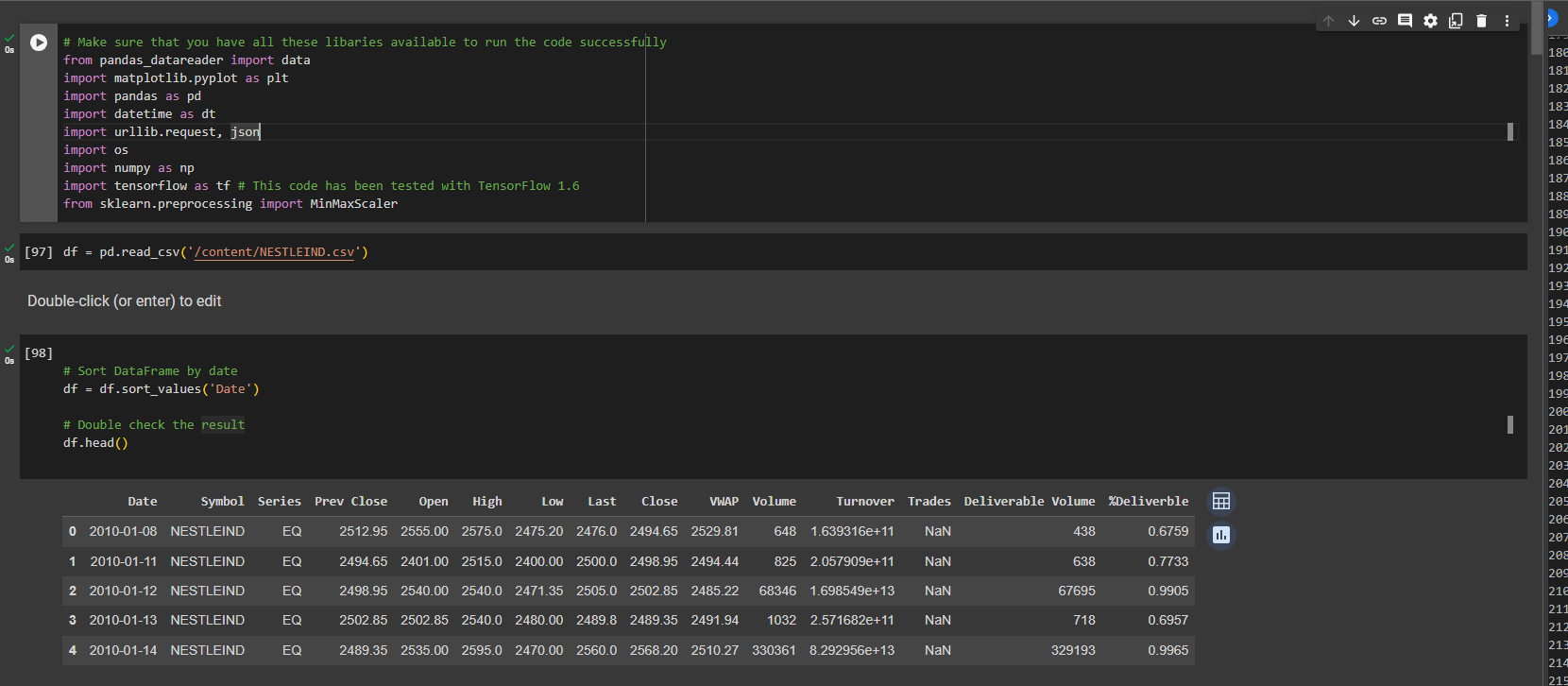
plt.title('NIFTY50 Stock Price Prediction using LSTM')

plt.xlabel('Date')

plt.ylabel('Stock Price')

plt.legend()

plt.show()



Observation and learning:

Step 1: Loading the dataset in the notebook is a crucial initial step to ensure that the necessary data is available for further analysis and model development.

Step 2: Selecting the appropriate features from the training data is essential for creating a model that accurately captures the underlying patterns in the data. This step requires a thorough understanding of the dataset and the problem at hand.

Step 3: Normalizing the features and converting them into timestamps of 60 allows for consistent and standardized input to the model. This preprocessing step is crucial for improving the convergence and performance of the LSTM model.

Conclusion:

The successful completion of these steps contributes to the development of an LSTM model for the given dataset. Loading, selecting, normalizing, and reshaping the data are essential preprocessing steps that set the foundation for effective model training. Creating and training the sequential LSTM model using Keras allows for capturing temporal dependencies in the data. Finally, predicting on test data provides valuable insights into the model's ability to generalize and make accurate predictions on new, unseen data. Overall, this process represents a comprehensive approach to building and evaluating an LSTM model for time-series data.