Predicting Stock Market Index Using LSTM & GRU

Authors: Sofia Mucci, Sebastion Tardieu, Madison Klinefelter

Abstract

This project focuses on predicting stock market behavior using a comprehensive approach that includes data preprocessing, exploratory data analysis, dimensionality reduction, and the application of Long Short-Term Memory (LSTM) neural network models. The dataset, sourced from NEPSE (Nepal Stock Exchange), undergoes preprocessing steps such as date conversion, feature engineering (50 and 200-day moving averages), and correlation analysis. Dimensionality reduction using Principal Component Analysis (PCA) is employed to capture essential features for model training. The LSTM models are constructed and tuned through hyperparameter optimization, considering different neuron configurations, optimizers (Adam, Adagrad, Nadam), learning rates, and batch sizes. The performance is evaluated using metrics such as root mean square error (RMSE) and mean absolute percentage error (MAPE). The project emphasizes the significance of selecting appropriate neural network architectures for time series prediction tasks.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
#data path for Sofia
data_path = "/content/drive/MyDrive/DSC 201/"


#data path for Madison
#data_path = "/content/drive/MyDrive/fall23/dsc201/"


#data path for Sebastian
#data_path = "/content/drive/MyDrive/NEPSE/"
```

```
#importing required libaries
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="whitegrid")
plt.style.use('ggplot')
import tensorflow as tf
%load_ext tensorboard
import warnings
warnings.filterwarnings('ignore')
import os
import datetime as dt
from sklearn.decomposition import PCA


import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import GRU
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow import keras
from tensorflow.keras import optimizers
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
import time
```

Data Visualization

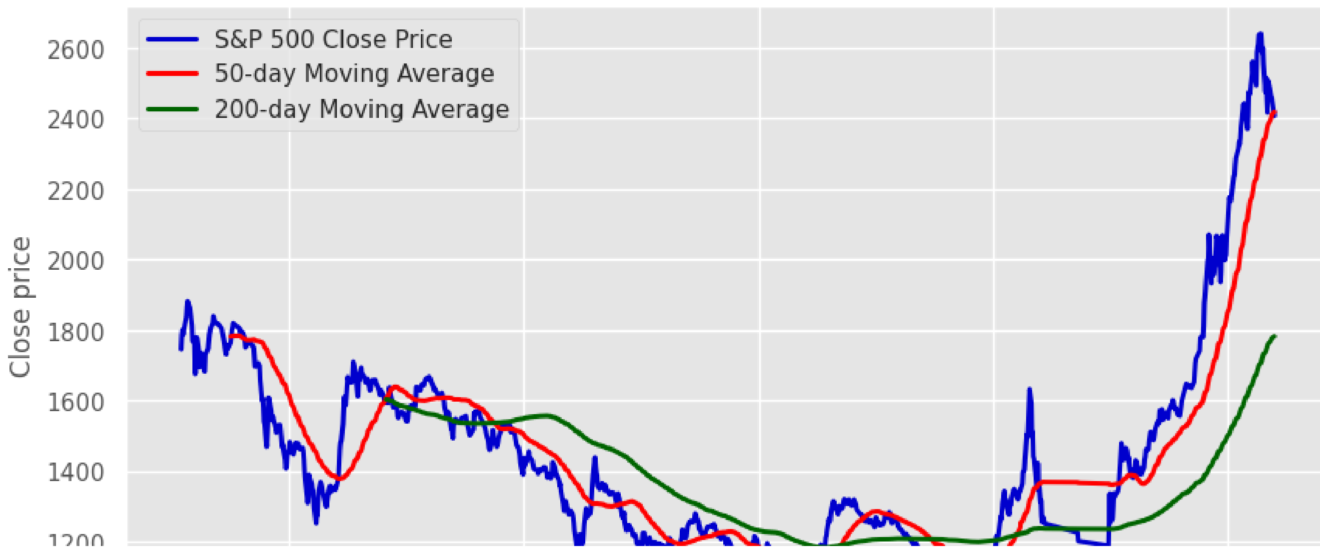
```
nepse_data= pd.read_csv(data_path + 'nepse_data.csv')
#converting Date columns to datetime and only extracting the date
nepse_data['Date']=pd.to_datetime(nepse_data['Date']).dt.date
#setting 'Date' column as the index
nepse_data.set_index("Date",inplace=True)
nepse_data.head()
```

	Unnamed: 0	Open	High	Low	Close	Volume	MACD	RSI	A
Date									
2016-									

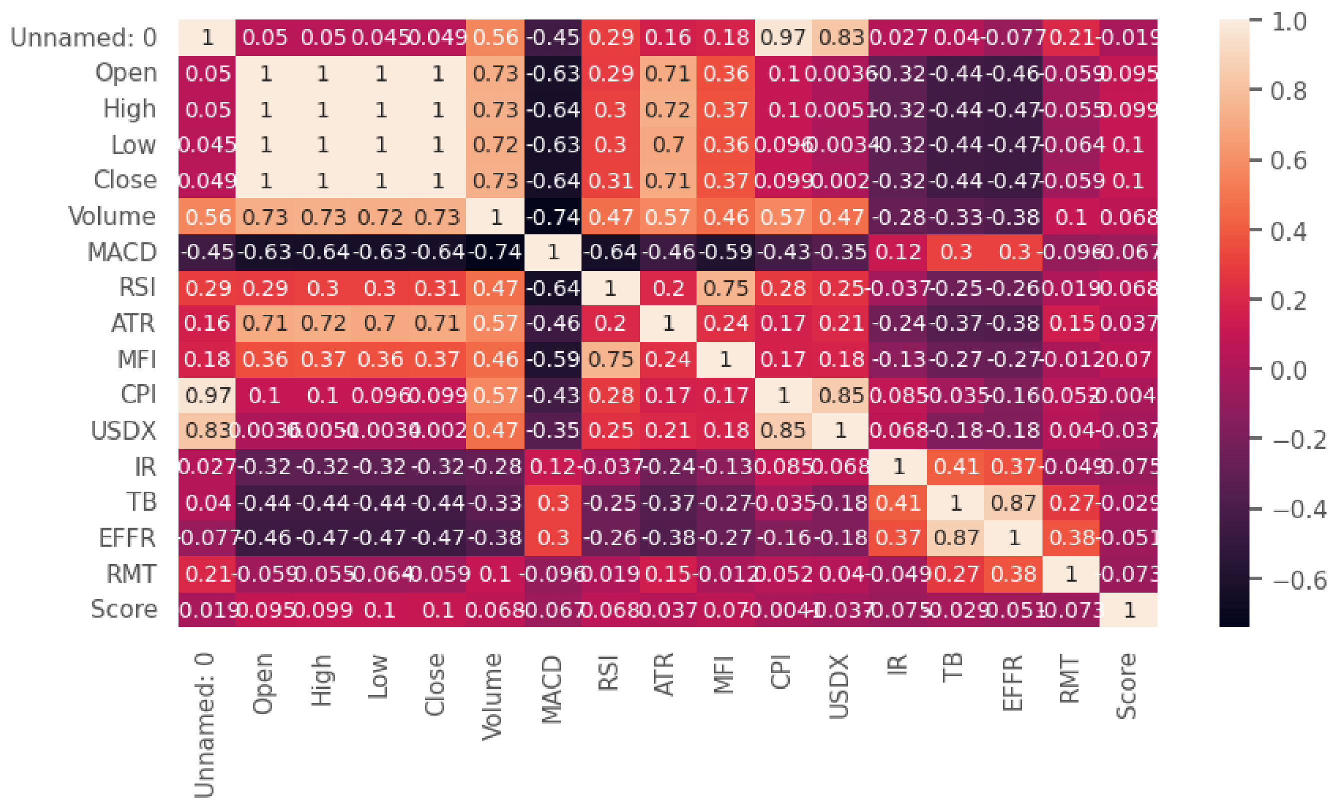
```
nepse_data.tail()
```

	Unnamed: 0	Open	High	Low	Close	Volume	MACD	RSI	
Date									
2021-03-04	1046	2425.20	2506.68	2427.25	2506.68	16622763	-25.944387	48.236208	66.38
2021-03-07	1047	2519.96	2525.30	2473.57	2485.09	11778820	-21.572364	44.946087	65.33
2021-03-09	1048	2494.05	2494.05	2453.53	2461.88	12482428	-16.049640	41.439954	63.56

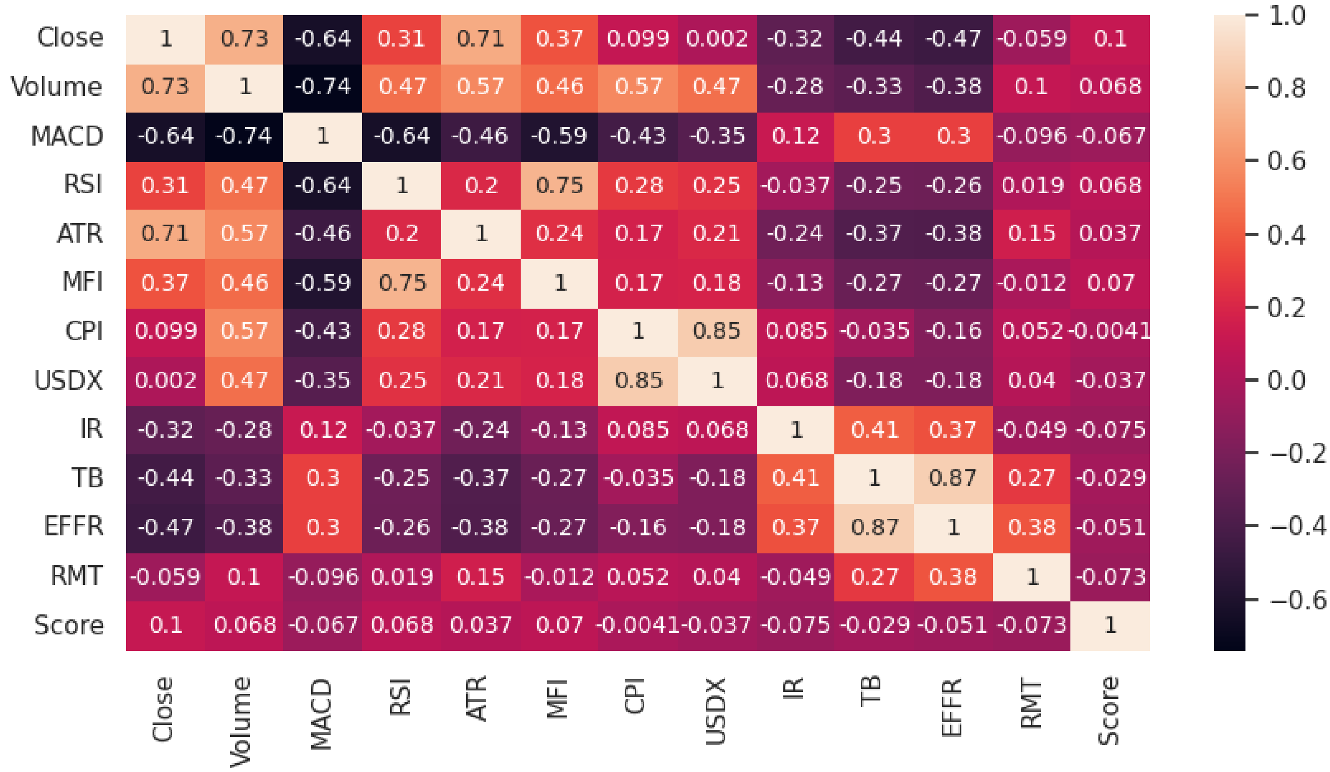
```
df = nepse_data.copy()
#calculating 50 and 200 day moving average and add it as columns to dataset
df['MA_50'] = df['Close'].rolling(50).mean()
df['MA_200'] = df['Close'].rolling(200).mean()

fig = plt.figure(figsize=(10,5))
fig.set(facecolor = "white")
#plotting the close price
plt.plot(df['Close'], 'mediumblue',label=['S&P 500 Close Price'], linewidth = 2.2)
#plotting the 50 day moving average
plt.plot(df['MA_50'],'red', label=['50-day MA'], linewidth = 2.2)
#plotting on the 200 dat moving average
plt.plot(df['MA_200'],'darkgreen', label=['200-day MA'], linewidth = 2.2)
plt.legend(['S&P 500 Close Price', '50-day Moving Average', '200-day Moving Average'
plt.title('')
plt.xlabel('Time (years)')
plt.ylabel('Close price')
#fig.savefig(output_dir_path+ "original_data_plus_moving_averages.png",dpi=600)
plt.show()
```



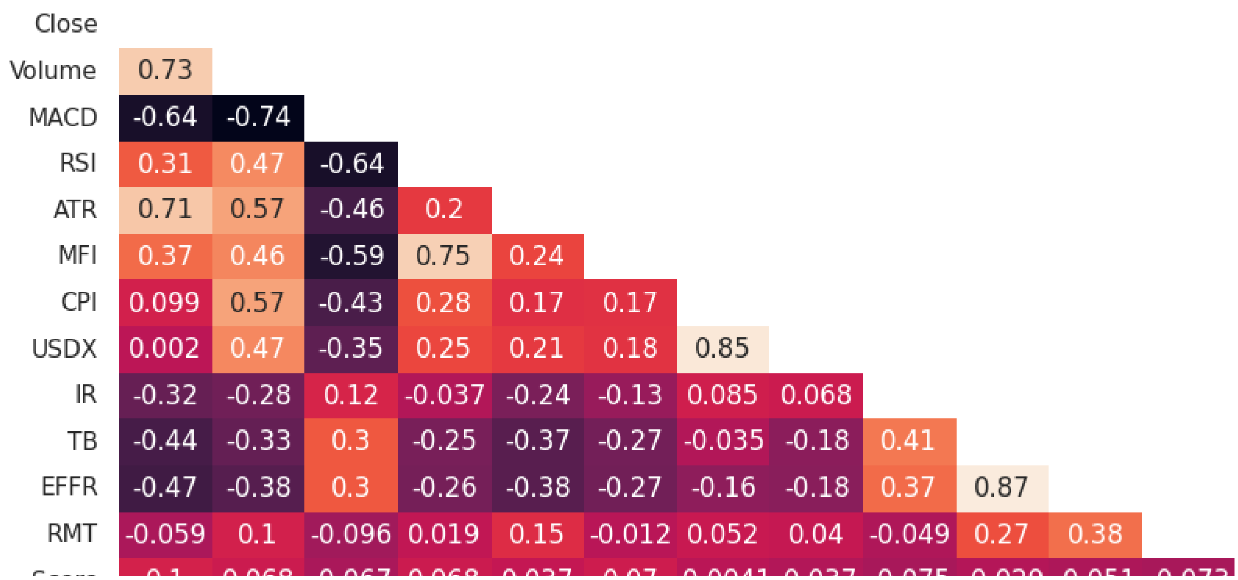

```
fig = plt.figure(figsize= (10,5))
sns.heatmap(nepse_data.corr(), annot=True)
sns.set_style("whitegrid")
#fig.savefig(output_dir_path+"correlation_heatmap.png",dpi=600)
```



```
#Extracting Nepse data for a heatmap starting at column 5
data = nepse_data.iloc[:, 4:]
fig = plt.figure(figsize= (10,5))
sns.heatmap(data.corr(), annot=True)
sns.set_style("whitegrid")
```



```
# Create a lower triangular mask
mask = np.triu(np.ones_like(data.corr(), dtype=bool))


data = nepse_data.iloc[:, 4:]
sns.set_style("white")  # Set the style to plain white background
sns.set(style="white", rc={"axes.grid": False})
fig = plt.figure(figsize= (10,5))
sns.heatmap(data.corr(), annot=True, mask=mask, cbar = False)
plt.show()
```




```
nepse_data.describe()
```

	Unnamed: 0	Open	High	Low	Close	Volume	
count	1051.00000	1051.000000	1051.000000	1051.000000	1051.000000	1.051000e+03	1051.0
mean	525.00000	1448.737869	1459.018934	1437.164500	1447.630276	3.075432e+06	-5.6
std	303.54187	307.807762	310.701300	301.714343	306.109743	4.118234e+06	32.0
min	0.00000	1100.950000	1104.610000	1098.950000	1100.580000	1.052600e+04	-111.0
25%	262.50000	1216.750000	1223.540000	1208.295000	1216.595000	9.654130e+05	-20.5
50%	525.00000	1363.010000	1379.750000	1353.400000	1363.010000	1.428838e+06	1.0
75%	787.50000	1585.080000	1593.445000	1573.570000	1583.585000	2.912336e+06	15.4
max	1050.00000	2667.270000	2673.850000	2619.580000	2640.340000	2.594538e+07	66.3

```
nepse_data.columns
```

```
Index(['Unnamed: 0', 'Open', 'High', 'Low', 'Close', 'Volume', 'MACD', 'RSI',
       'ATR', 'MFI', 'CPI', 'USDX', 'IR', 'TB', 'EFFR', 'RMT', 'Score'],
      dtype='object')
```

```
#dropping unecessary columns
nepse = nepse_data.drop("Unnamed: 0", axis=1)
```

```
nepse.head()
```

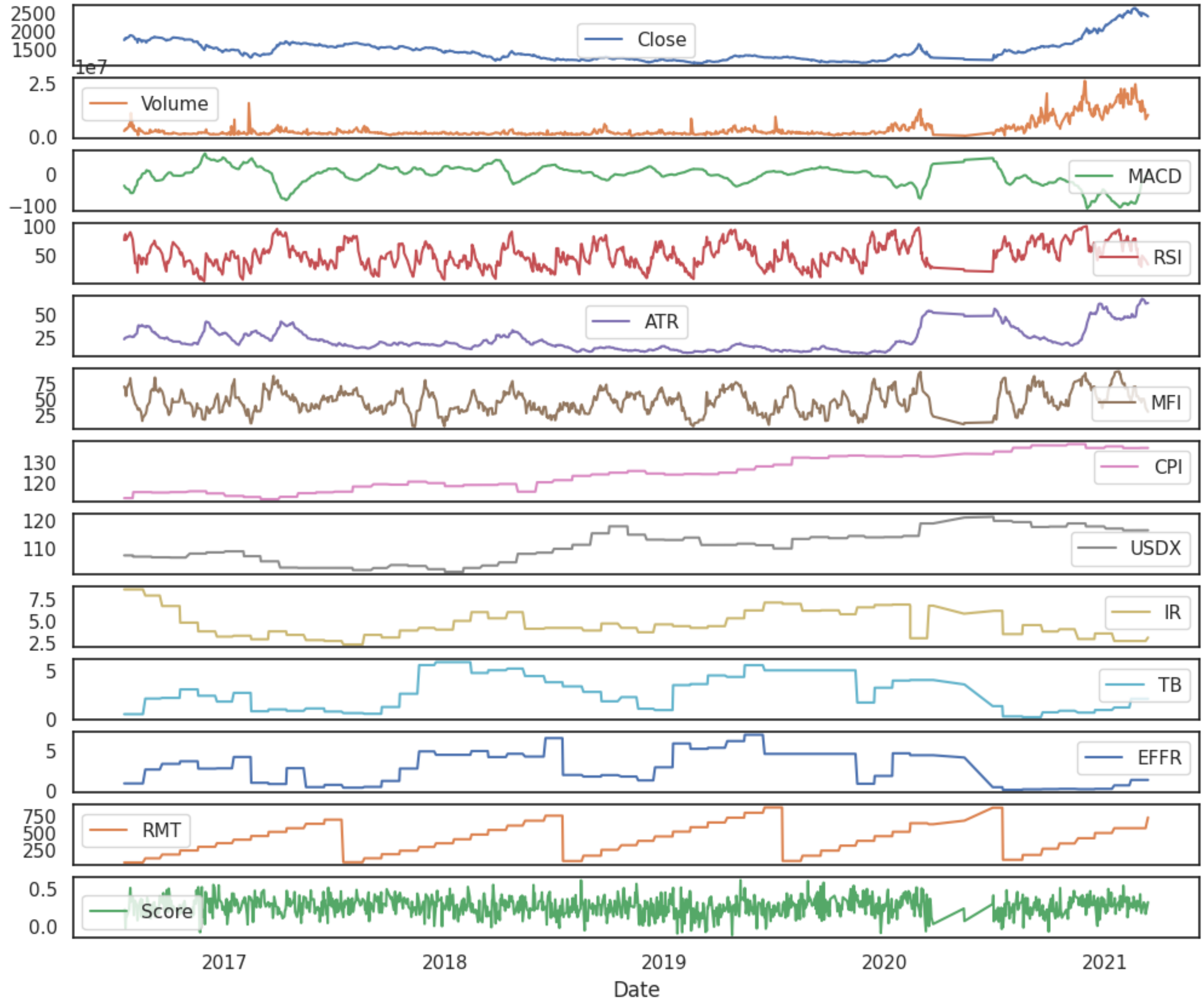
	Open	High	Low	Close	Volume	MACD	RSI	ATR	M
Date									
2016-07-17	1718.15	1749.96	1715.14	1745.74	2272045	-37.509420	74.940143	23.122890	70.1965
2016-07-18	1745.74	1786.68	1745.74	1786.59	2870497	-41.071885	82.962838	24.395540	67.1974
2016-									

```
nepse.describe()
```

	Open	High	Low	Close	Volume	MACD	
count	1051.000000	1051.000000	1051.000000	1051.000000	1.051000e+03	1051.000000	1051.
mean	1448.737869	1459.018934	1437.164500	1447.630276	3.075432e+06	-5.688622	50
std	307.807762	310.701300	301.714343	306.109743	4.118234e+06	32.078118	20.
min	1100.950000	1104.610000	1098.950000	1100.580000	1.052600e+04	-111.082111	4.
25%	1216.750000	1223.540000	1208.295000	1216.595000	9.654130e+05	-20.560546	35.
50%	1363.010000	1379.750000	1353.400000	1363.010000	1.428838e+06	1.054089	47.
75%	1585.080000	1593.445000	1573.570000	1583.585000	2.912336e+06	15.415869	65.
max	2667.270000	2673.850000	2619.580000	2640.340000	2.594538e+07	66.304566	98.

```
fig = plt.figure(figsize = (20, 12))
#fig.axes.get_yaxis().set_visible(False)
data.plot(subplots = True, figsize = (12,10),grid=False)
sns.set_style("whitegrid")
#fig.savefig(output_dir_path+"timeseries.png",dpi=600)
plt.show()
```

<Figure size 2000x1200 with 0 Axes>



Dimensionality Reduction

```
#from operator import index
from sklearn.decomposition import PCA
```

```
nepse.head()
```

	Open	High	Low	Close	Volume	MACD	RSI	ATR	M
Date									
07-17	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
07-18	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

```
from sklearn.preprocessing import StandardScaler

scaler=StandardScaler()

#calculate parameters needed for normalization
nepse_normalized=scaler.fit(nepse)
#apply the normalization transformation to the Nepse data
nepse_normalized=scaler.transform(nepse)
nepse_normalized
```

```
array([[ 0.8756777 ,  0.93684698,  0.92175876, ..., -0.96025412,
        -1.57550256, -0.42943027],
       [ 0.96535424,  1.0550875 ,  1.02322748, ..., -0.96025412,
        -1.57550256, -0.34835518],
       [ 1.09813012,  1.14119184,  1.15450678, ..., -0.96025412,
        -1.57550256, -2.39766991],
       ...,
       [ 3.39760691,  3.33285963,  3.37023875, ..., -0.74125943,
         0.73139818,  0.18288685],
       [ 3.3419613 ,  3.28024389,  3.34636375, ..., -0.74125943,
         0.73139818, -0.83755828],
       [ 3.30728031,  3.24640108,  3.1845112 , ..., -0.74125943,
         1.45295322,  0.45705746]])
```

```
#create a PCA object with 95% variance
pca = PCA(n_components=0.95, svd_solver='full')
# you can try both 0.95 and 0.99


X_pca = pca.fit_transform(nepse_normalized)
#X_pca = pca.fit(nepse.values)


#print the shape of the after applying PCA
X_pca.shape
```

```
(1051, 9)
```

```
X_pca
```

```
array([[ 1.98905314, -0.81932092, -0.75841077, ...,  1.53655513,
        -0.48403268, -0.24408457],
       [ 2.29965747, -0.77191487, -0.66255152, ...,  1.56335573,
        -0.55543441,  0.07303038],
       [ 2.3922143 , -0.7097521 , -0.55378168, ...,  1.19778126,
        -0.5795444 ,  0.22989557],
       ...,
       [ 7.10373573, -1.13228951,  2.0343545 , ..., -0.07235327,
```

```
         1.05035434, -0.41184385],
       [ 6.46312534, -1.40105137,  1.88928154, ..., -0.05042317,
         0.98151247, -0.1360254 ],
       [ 6.2804184 , -1.37349962,  2.27270526, ...,  0.51379841,
         0.60052276, -0.44140933]])
```

```
pca.explained_variance_ratio_
```

```
array([0.43212577, 0.14646813, 0.10448407, 0.0825473 , 0.06388735,
       0.0607044 , 0.03640986, 0.02305484, 0.01498773])
```

```
#trying with n_componenets set to 0.95
pca = PCA(n_components=0.95, svd_solver='full')
```

```
#fit the PCA model to normalized Nepse data
X_pca = pca.fit_transform(nepse_normalized)
```

```
X_pca.shape
```

```
(1051, 9)
```

Data Normalization and Input Preperation

```
# defining a function that gives a dataset and a time step, which then returns the i
def DatasetCreation(dataX,dataY, time_step = 1):
   DataX, DataY = [], []
   for i in range(len(dataX)- time_step -1):
         a = dataX[i:(i+ time_step), ]
         DataX.append(a)
         b= dataY[i + time_step, ]
         DataY.append(b)
   return np.array(DataX), np.array(DataY)
```

```
X = X_pca
y = nepse['Close'].values
#reshape 'y' to have a single column
y=y.reshape(len(y),1)
```

```
#creating a dataset using DatasetCreation
X,y=DatasetCreation(X,y, time_step = 5)
```

```
# shape is (None, time_step, #number of features)
X.shape
```

```
(1045, 5, 9)
```

```
y.shape
```

```
(1045, 1)
```

```
def data_split(data, split = 0.2):
  # creating training and test data
  l1   = int(len(data) * (1- split))
  l2    = len(data) - l1
  train  = data[0:l1,:]
  test   = data[l1:len(data),:]
  return train, test
```

```
X_train, X_test=data_split(X, split = 0.2)
y_train, y_test=data_split(y, split = 0.2)
```

```
print('X_train shape',X_train.shape)
print('y_train shape',y_train.shape)
print('X_test shape',X_test.shape)
print('y_test shape',y_test.shape)
```

```
X_train shape (836, 5, 9)
y_train shape (836, 1)
X_test shape (209, 5, 9)
y_test shape (209, 1)
```

```
y_train[0]
```

```
array([1838.49])
```

```
X_val= data_split(X_train, 0.2)
y_val= data_split(y_train, 0.2)
```

```
def mean_absolute_percentage_error(y_true, y_pred):
    return (np.mean(np.abs((y_true - y_pred)/(y_true))*100)) #some issues with zero

def calculate_scores(y_true, y_pred):
  rmse = math.sqrt(mean_squared_error(y_true, y_pred))
  #R2_score = r2_score(y_true, y_pred)
  R = np.corrcoef(y_true, y_pred)
  #mae = mean_absolute_error(y_true, y_pred)
  mape = mean_absolute_percentage_error(y_true, y_pred)
  #dic = {'rmse':rmse, 'R2_score': R2_score, 'R':R[0,1], 'mae': mae, 'mape': mape}
  dic = {'rmse':rmse, 'R': R[0,1], 'mape': mape}
  return (dic)


def min_max_transform(data, feature_range=(0, 1)):
   scaler = MinMaxScaler(feature_range)
   return scaler.fit_transform(data)

def min_max_inverse_transform(data_scaled, min_original, max_original):
    return min_original +  data_scaled*(max_original - min_original)



def write_dic_to_file(dic_name, file_name):
  file = open(file_name, 'w')
  file.write(str(dic_name))
  file.close()

import ast
def read_dic_from_file(file_name):
  file = open(file_name, "r")
  contents = file.read()
  dictionary = ast.literal_eval(contents)
  file.close()
  return dictionary
```

Build the LSTM Model

```
def build_lstm_model(layers, optimizer = 'Adam', learning_rate = 0.001, verbose = 1)

  model = Sequential()

  for i in range(len(layers)):
    if len(layers)==1:
      model.add(LSTM(np.int(layers[i]), input_shape = (5, 9)))
    else:
      if i < len(layers)-1:
        if i == 0:
          model.add(LSTM(np.int(layers[i]), input_shape=(5, 9), return_sequences= Tr
          #model.add(Dropout(0.10))
        else:
          model.add(LSTM(np.int(layers[i]), return_sequences=True))
          #model.add(Dropout(0.10))
      else:
        model.add(LSTM(np.int(layers[i])))
        #model.add(Dropout(0.10))
  model.add(Dense(1, activation = 'linear'))

  if optimizer == 'Adam':
    opt = optimizers.Adam(learning_rate = learning_rate)
  elif optimizer == 'Adagrad':
    opt = optimizers.Adagrad(learning_rate = learning_rate)
  elif optimizer == 'Nadam':
    opt = optimizers.Nadam(learning_rate = learning_rate)
  elif optimizer == 'Adadelta':
    opt = optimizers.Adadelta(learning_rate= learning_rate)
  elif optimizer == 'RMSprop':
    opt = optimizers.RMSprop(learning_rate= learning_rate)
  else:
    print("No optimizer found in the list(['Adam', 'Adagrad','Nadam', 'Adadelta', 'R

  model.compile(loss='mean_squared_error', optimizer= opt)
  return model
```

Testing

```
#test model on 'Adam', 'Adagrad', and 'Nadam' optimizers
optimizers_names = ['Adam', 'Adagrad', 'Nadam']

build_lstm_model([250], optimizers_names[2], 0.001, 1)
```

```
<keras.src.engine.sequential.Sequential at 0x787bb47cfb80>
```

```
#perform hyperparameter tuning for neural network model
def hyper_parameter_tuning(layers, optimizers_names, learning_rates, batch_sizes, ep


  best_avg_rmse = 99999999999

  collect_rmse = []

  all_avg_rmse = np.zeros((len(optimizers_names), len(learning_rates), len(batch_siz

  best_hyper_parameters = {"model": layers,
                           "optimizer": None,
                           "learning_rate": None,
                           "batch_size": None,
                           "best_avg_rmse": None}

  for opt in range(len(optimizers_names)):

    for lr in range(len(learning_rates)):

      for batch_size in range(len(batch_sizes)):

        for i in range(num_replicates):

          print("Running for " + optimizers_names[opt] + " optimizer " + str(learnin

          model = build_lstm_model(layers,  optimizers_names[opt], learning_rate = l

          callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=

          history = model.fit(X_train, y_train, batch_size = batch_sizes[batch_size]


          y_pred=model.predict(X_test)
          collect_rmse.append(math.sqrt(mean_squared_error(y_test,y_pred)))

        avg_rmse = np.mean(np.array(collect_rmse))
        all_avg_rmse[opt][lr][batch_size] = avg_rmse

        if avg_rmse < best_avg_rmse:
          best_avg_rmse = avg_rmse
          best_hyper_parameters = {"model": layers,
                                   "optimizer": optimizers_names[opt],
                                   "learning_rate": learning_rates[lr],
                                   "batch_size": batch_sizes[batch_size],
                                   "best_avg_rmse": best_avg_rmse}


  output_dictionary = {
      "best_hyper_parameters":  best_hyper_parameters,
      "all_avg_rmse": all_avg_rmse
```

```
    }

  #writing output dictionary in the file

  file_name = data_path+ "sl-lstm-" + str(layers[0])+ "-neurons-validation_results"+
  write_dic_to_file(output_dictionary, file_name)

  print("Best_hyper_parameters: \n", output_dictionary['best_hyper_parameters'])
  print("all_avg_rmse: \n", output_dictionary['all_avg_rmse'])

  return output_dictionary['best_hyper_parameters']
```

Case I: Tuning parameters of 8 neuron single layer LSTM

```
layers = [10]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10


N10_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learni
N10_best_hyper_parameters
```

```
105/105 [==============================] - 1s 7ms/step - loss: 1845762.0000 - va
Epoch 3/50
105/105 [==============================] - 1s 8ms/step - loss: 1812758.2500 - va
Epoch 4/50
105/105 [==============================] - 1s 8ms/step - loss: 1781166.2500 - va
Epoch 5/50
105/105 [==============================] - 1s 8ms/step - loss: 1750382.5000 - va
Epoch 6/50
105/105 [==============================] - 1s 8ms/step - loss: 1720225.1250 - va
Epoch 7/50
105/105 [==============================] - 1s 8ms/step - loss: 1690582.8750 - va
Epoch 8/50
105/105 [==============================] - 1s 8ms/step - loss: 1661431.8750 - va
Epoch 9/50
105/105 [==============================] - 1s 8ms/step - loss: 1632718.8750 - va
Epoch 10/50
105/105 [==============================] - 1s 9ms/step - loss: 1604413.5000 - va
Epoch 11/50
105/105 [==============================] - 1s 9ms/step - loss: 1576495.0000 - va
Epoch 12/50
105/105 [==============================] - 1s 8ms/step - loss: 1548920.7500 - va
Epoch 13/50
105/105 [==============================] - 1s 7ms/step - loss: 1521730.7500 - va
Epoch 14/50
105/105 [==============================] - 1s 8ms/step - loss: 1494867.1250 - va
Epoch 15/50
105/105 [==============================] - 1s 8ms/step - loss: 1468361.5000 - va
Epoch 16/50
105/105 [==============================] - 1s 8ms/step - loss: 1442172.1250 - va
Epoch 17/50
105/105 [==============================] - 1s 8ms/step - loss: 1416301.0000 - va
Epoch 18/50
105/105 [==============================] - 1s 8ms/step - loss: 1390768.5000 - va
Epoch 19/50
```

Case II: Tuning parameters of 30 neuron single layer LSTM

```
layers = [30]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10
```

```
N30_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learni
N30_best_hyper_parameters
```

```
209/209 [==============================] - 2s 7ms/step - loss: 864722.7500 - val
Epoch 9/50
209/209 [==============================] - 2s 8ms/step - loss: 769488.9375 - val
Epoch 10/50
209/209 [==============================] - 1s 7ms/step - loss: 682019.6875 - val
Epoch 11/50
209/209 [==============================] - 2s 8ms/step - loss: 602022.7500 - val
Epoch 12/50
209/209 [==============================] - 1s 7ms/step - loss: 529129.1875 - val
Epoch 13/50
209/209 [==============================] - 2s 8ms/step - loss: 462901.6562 - val
Epoch 14/50
209/209 [==============================] - 2s 7ms/step - loss: 402996.0625 - val
Epoch 15/50
209/209 [==============================] - 2s 7ms/step - loss: 349101.0625 - val
Epoch 16/50
209/209 [==============================] - 2s 8ms/step - loss: 300933.1562 - val
Epoch 17/50
209/209 [==============================] - 2s 9ms/step - loss: 258162.4844 - val
Epoch 18/50
209/209 [==============================] - 2s 8ms/step - loss: 220496.1094 - val
Epoch 19/50
209/209 [==============================] - 2s 8ms/step - loss: 187570.4844 - val
Epoch 20/50
209/209 [==============================] - 2s 8ms/step - loss: 159095.7656 - val
Epoch 21/50
209/209 [==============================] - 2s 8ms/step - loss: 134778.3750 - val
Epoch 22/50
209/209 [==============================] - 2s 7ms/step - loss: 114215.4844 - val
Epoch 23/50
209/209 [==============================] - 2s 8ms/step - loss: 97083.2734 - val_
Epoch 24/50
209/209 [==============================] - 1s 7ms/step - loss: 83101.8984 - val_
Epoch 25/50
209/209 [==============================] - 2s 8ms/step - loss: 71864.2344 - val_
Epoch 26/50
209/209 [==============================] - 2s 8ms/step - loss: 63045.8164 - val_
Epoch 27/50
209/209 [==============================] - 2s 7ms/step - loss: 56293.0820 - val_
Epoch 28/50
209/209 [==============================] - 2s 8ms/step - loss: 51221.1875 - val_
Epoch 29/50
209/209 [==============================] - 2s 8ms/step - loss: 48096.5938 - val_
Epoch 30/50
209/209 [==============================] - 2s 8ms/step - loss: 38198.0156 - val_
Epoch 31/50
209/209 [==============================] - 2s 8ms/step - loss: 31927.5820 - val_
Epoch 32/50
209/209 [==============================] - 2s 8ms/step - loss: 27235.7207 - val_
Epoch 33/50
209/209 [==============================] - 2s 8ms/step - loss: 23228.4336 - val_
Epoch 34/50
209/209 [==============================] - 2s 8ms/step - loss: 19858.7793 - val_
Epoch 35/50
209/209 [==============================] - 2s 8ms/step - loss: 16746.4219 - val_
```

Case III: Tuning parameters of 50 neuron single layer LSTM

```
layers = [50]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10
```

```
N50_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learni
N50_best_hyper_parameters
```

```
105/105 [==============================] - 0s 3ms/step - loss: 1796287.0000 - va
Epoch 11/50
105/105 [==============================] - 0s 4ms/step - loss: 1791876.6250 - va
Epoch 12/50
105/105 [==============================] - 0s 3ms/step - loss: 1787704.3750 - va
Epoch 13/50
105/105 [==============================] - 0s 4ms/step - loss: 1783717.5000 - va
Epoch 14/50
105/105 [==============================] - 0s 3ms/step - loss: 1779895.3750 - va
Epoch 15/50
105/105 [==============================] - 0s 3ms/step - loss: 1776221.0000 - va
Epoch 16/50
105/105 [==============================] - 0s 4ms/step - loss: 1772675.3750 - va
Epoch 17/50
105/105 [==============================] - 0s 3ms/step - loss: 1769252.0000 - va
Epoch 18/50
105/105 [==============================] - 0s 3ms/step - loss: 1765934.3750 - va
Epoch 19/50
105/105 [==============================] - 0s 4ms/step - loss: 1762717.0000 - va
Epoch 20/50
```

Case IV: Tuning parameters of 100 neuron single layer LSTM

```
layers = [100]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10
```

```
N100_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learn
N100_best_hyper_parameters
```

```
7/7 [==============================] - 1s 7ms/step
Running for Adagrad optimizer 0.01 learning_rate 4 batch_size and 9 replicate

Epoch 1/50
209/209 [==============================] - 5s 12ms/step - loss: 1827771.3750 - v
Epoch 2/50
209/209 [==============================] - 2s 7ms/step - loss: 1780996.1250 - va
Epoch 3/50
209/209 [==============================] - 2s 7ms/step - loss: 1753219.5000 - va
Epoch 4/50
209/209 [==============================] - 2s 8ms/step - loss: 1731119.2500 - va
Epoch 5/50
209/209 [==============================] - 1s 7ms/step - loss: 1712265.5000 - va
Epoch 6/50
209/209 [==============================] - 2s 7ms/step - loss: 1695576.0000 - va
Epoch 7/50
209/209 [==============================] - 2s 8ms/step - loss: 1680467.8750 - va
Epoch 8/50
209/209 [==============================] - 2s 7ms/step - loss: 1666580.8750 - va
Epoch 9/50
209/209 [==============================] - 2s 7ms/step - loss: 1653676.1250 - va
Epoch 10/50
209/209 [==============================] - 2s 9ms/step - loss: 1641582.2500 - va
Epoch 11/50
209/209 [==============================] - 2s 8ms/step - loss: 1630170.7500 - va
Epoch 12/50
209/209 [==============================] - 2s 9ms/step - loss: 1619348.3750 - va
Epoch 13/50
209/209 [==============================] - 2s 8ms/step - loss: 1609035.5000 - va
Epoch 14/50
209/209 [==============================] - 2s 8ms/step - loss: 1599173.5000 - va
Epoch 15/50
209/209 [==============================] - 2s 8ms/step - loss: 1589712.8750 - va
Epoch 16/50
209/209 [==============================] - 2s 8ms/step - loss: 1580611.8750 - va
Epoch 17/50
209/209 [==============================] - 2s 9ms/step - loss: 1571837.0000 - va
Epoch 18/50
209/209 [==============================] - 2s 9ms/step - loss: 1563357.0000 - va
```

Case V: Tuning parameters of 150 neuron single layer LSTM

```
layers = [150]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10


N150_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learn
N150_best_hyper_parameters
```

```
Epoch 46/50
209/209 [==============================] - 2s 10ms/step - loss: 1181334.3750 - v
Epoch 47/50
209/209 [==============================] - 2s 11ms/step - loss: 1175046.0000 - v
Epoch 48/50
209/209 [==============================] - 2s 12ms/step - loss: 1168848.5000 - v
Epoch 49/50
209/209 [==============================] - 2s 9ms/step - loss: 1162739.5000 - va
Epoch 50/50
209/209 [==============================] - 2s 10ms/step - loss: 1156716.3750 - v
7/7 [==============================] - 0s 8ms/step
Running for Adagrad optimizer 0.01 learning_rate 4 batch_size and 9 replicate

Epoch 1/50
209/209 [==============================] - 4s 12ms/step - loss: 1794420.2500 - v
```

Case VI: Tuning parameters of 200 neuron single layer LSTM

```
layers = [200]
time_step = 5
optimizers_names = ['Adam', 'Adagrad', 'Nadam']
learning_rates =  [0.1, 0.01, 0.001]
batch_sizes =  [4, 8, 16]
epochs = 50
num_replicates = 10
```

```
N200_best_hyper_parameters = hyper_parameter_tuning(layers,  optimizers_names, learn
N200_best_hyper_parameters
```

```
Epoch 16/50
105/105 [==============================] - 2s 21ms/step - loss: 1459183.0000 - v
Epoch 17/50
105/105 [==============================] - 2s 19ms/step - loss: 1447541.0000 - v
Epoch 18/50
105/105 [==============================] - 2s 18ms/step - loss: 1436319.1250 - v
Epoch 19/50
105/105 [==============================] - 2s 19ms/step - loss: 1425477.2500 - v
Epoch 20/50
105/105 [==============================] - 2s 21ms/step - loss: 1414982.6250 - v
Epoch 21/50
105/105 [==============================] - 2s 20ms/step - loss: 1404796.1250 - v
Epoch 22/50
105/105 [==============================] - 2s 22ms/step - loss: 1394912.1250 - v
Epoch 23/50
105/105 [==============================] - 2s 20ms/step - loss: 1385306.3750 - v
Epoch 24/50
105/105 [==============================] - 2s 21ms/step - loss: 1375965.5000 - v
Epoch 25/50
105/105 [==============================] - 3s 24ms/step - loss: 1366856.6250 - v
Epoch 26/50
105/105 [==============================] - 2s 21ms/step - loss: 1357978.1250 - v
Epoch 27/50
105/105 [==============================] - 2s 20ms/step - loss: 1349314.0000 - v
Epoch 28/50
105/105 [==============================] - 2s 19ms/step - loss: 1340851.8750 - v
Epoch 29/50
105/105 [==============================] - 2s 21ms/step - loss: 1332579.8750 - v
Epoch 30/50
105/105 [==============================] - 2s 18ms/step - loss: 1324487.5000 - v
Epoch 31/50
105/105 [==============================] - 2s 20ms/step - loss: 1316567.1250 - v
Epoch 32/50
105/105 [==============================] - 2s 22ms/step - loss: 1308806.3750 - v
Epoch 33/50
```

Building and running single layer models in full scale

```
#========== Model hyper parameters settting ==========================#
def LSTM_model(neurons, hyper_parameters,  epochs = 20,  num_replicates = 2):

  #====== data transformation==========#
  print("Progress: Performing data preparation steps.......\n")

  #======= creating training and test data===#

 # train_data, test_data = data_split(data, test_split)
  train_data, test_data = X_train, X_test

  num_features = train_data.shape[1]

  # min_train, max_train  = train_data["Close"].min(), train_data["Close"].max()
  # min_test, max_test   =  test_data["Close"].min(), test_data["Close"].max()

  # train_data_scaled  =  min_max_transform(train_data)
  # test_data_scaled   = min_max_transform(test_data)


  # X_train, y_train  =   DatasetCreation(train_data_scaled, time_step)
  # X_test, y_test    =   DatasetCreation(test_data_scaled, time_step)

  # y_train_original  =  min_max_inverse_transform(y_train, min_train, max_train) #i
  # y_test_original  =  min_max_inverse_transform(y_test, min_test, max_test) #in or




  print("Progress: Building and training models.......\n")

  neurons = np.array(neurons)
  #============ arrays for collecting test scores ================#
  rmse_array = np.zeros((len(neurons), num_replicates))
  #mae_array  = np.zeros((len(neurons), num_replicates))
  mape_array = np.zeros((len(neurons), num_replicates))
  #R2_array   = np.zeros((len(neurons), num_replicates))
  R_array    = np.zeros((len(neurons), num_replicates))
  elapsed_time_array = np.zeros((len(neurons), num_replicates))


  #========== array for collecting history and predictions =======#
  models_history = []
  train_predictions = []
  test_predictions  = []

  for i in range(len(neurons)):

    print("Model hyperparameters used: \n ", hyper_parameters[i])
    #========== saving history and predictions per replicate=====#
    model_history_per_replicate = []
```

```
    train_predictions_per_replicate = []
    test_predictions_per_replicate  = []

    hidden_nodes = np.int(neurons[i])

   # print("Program is running for %d neurons ----->\n" %np.int(neurons[i]))


    for k in range(num_replicates):

      print("Program is running for %d neurons and %d replicate ----->\n" %(hidden_n



      layers = [hidden_nodes]


      ### model = build_lstm_model(layers, time_step, num_features, optimizer = hype
      model = build_lstm_model(layers, optimizer = hyper_parameters[i][0], learning_
      callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience= 5)
      # This callback will stop the training when there is no improvement in
      # the loss for three consecutive epochs
      start = time.time()
      history = model.fit(X_train, y_train, batch_size = hyper_parameters[i][2], epo
      end = time.time()
      elapsed_time = end - start

      model_history_per_replicate.append(history)


      #==============Making train and test prediction in original scales ==========
      train_pred   =  model.predict(X_train) #in original scale
      test_pred    =  model.predict(X_test)

      train_predictions_per_replicate.append(train_pred)
      test_predictions_per_replicate.append(test_pred)

      #============== Calculating performance scores=========
      scores =   calculate_scores(y_test, test_pred)
      rmse_array[i][k] = scores['rmse']
      mape_array[i][k] =  scores['mape']
      R_array[i][k] = scores['R']
      elapsed_time_array[i][k] = elapsed_time

    models_history.append(model_history_per_replicate)
    train_predictions.append(train_predictions_per_replicate)
    test_predictions.append(test_predictions_per_replicate)

  print("Progress: Collecting outputs.......\n")

  neurons_df = pd.DataFrame(neurons)
```

```
    rmse_df = pd.DataFrame(rmse_array)
    #mae_df  = pd.DataFrame(mae_array)
    mape_df  = pd.DataFrame(mape_array)
    #R2_df   = pd.DataFrame(R2_array)
    R_df    = pd.DataFrame(R_array)
    elapsed_time_df =  pd.DataFrame(elapsed_time_array)

    train_predictions  = np.array(train_predictions)
    test_predictions   = np.array(test_predictions)

    #==== Idendifying  the best model results based on rmse ===============#
    min_index = pd.DataFrame(rmse_df.min(axis = 1)).idxmin()[0]
    min_col =   pd.DataFrame(rmse_df.min(axis = 0)).idxmin()[0]

    num_neurons_with_best_rmse = neurons_df.loc[min_index,0]

    best_rmse = rmse_df.loc[min_index, min_col]
    #mae_with_best_rmse = mae_df.loc[min_index, min_col]
    mape_with_best_rmse = mape_df.loc[min_index, min_col]
    #R2_with_best_rmse = R2_df.loc[min_index, min_col]
    R_with_best_rmse =  R_df.loc[min_index, min_col]
    elapsed_time_with_best_rmse = elapsed_time_df.loc[min_index, min_col]

    train_predictions_with_best_rmse = train_predictions[min_index][min_col]
    test_predictions_with_best_rmse = test_predictions[min_index][min_col]

    loss_with_best_rmse = models_history[min_index][min_col].history['loss']
    #val_loss_with_best_rmse = models_history[min_index][min_col].history['val_loss']

    #======= Collecting hyperparameters=============#
    hyper_parameters = { 'neurons': neurons,
                         'model_specific_hyper_parameters': hyper_parameters,#addition
                         'epochs': epochs,

                         'num_replicates': num_replicates,
                        #'validataion_split':validation_split
                         }

    #======= Collecting test scores =============#
    scores = {'neurons': neurons_df, 'rmse': rmse_df, 'mape': mape_df, 'R': R_df, 'ela

    #======= Collecting average test scores =============#
    avg_scores = pd.DataFrame({'neurons': neurons,
                               'rmse': rmse_df.mean(axis = 1),
                               'mape': mape_df.mean(axis = 1),
                               'R': R_df.mean(axis = 1),
                               'elapsed_time': elapsed_time_df.mean(axis = 1)})

   #======= Collecting average test scores =============#
    all_stds = pd.DataFrame({'neurons': neurons,
                               'rmse': rmse_df.std(axis = 1),
```

```
                          'mape': mape_df.std(axis = 1),
                          'R': R_df.std(axis = 1),
                          'elapsed_time': elapsed_time_df.std(axis = 1)})


 #======= Collecting average test scores =============#
  all_minimums = pd.DataFrame({'neurons': neurons,
                          'rmse': rmse_df.min(axis = 1),
                          'mape': mape_df.min(axis = 1),
                          'R': R_df.min(axis = 1),
                          'elapsed_time': elapsed_time_df.min(axis = 1)})

  #======= Collecting average test scores =============#
  all_maximums = pd.DataFrame({'neurons': neurons,
                          'rmse': rmse_df.max(axis = 1),
                          'mape': mape_df.max(axis = 1),
                          'R': R_df.max(axis = 1),
                          'elapsed_time': elapsed_time_df.max(axis = 1)})



  #======= Collecting the best model results =============#
  model_with_best_rmse = {  'neurons': num_neurons_with_best_rmse,
                          'replicate': min_col,
                          'rmse': best_rmse,
                          'mape': mape_with_best_rmse,
                          'R':  R_with_best_rmse,
                          'elapsed_time': elapsed_time_with_best_rmse,
                          'train_predictions':train_predictions_with_best_rmse,
                          'test_predictions': test_predictions_with_best_rmse,
                          'loss':loss_with_best_rmse,

                       }


  #======= Collecting all the outputs together =============#
  output_dictionary = { 'hyper_parameters': hyper_parameters,
                      'best_model': model_with_best_rmse,
                      'scores': scores,
                      'avg_scores': avg_scores,
                      'all_stds': all_stds,
                      'all_minimums': all_minimums,
                      'all_maximums': all_maximums,
                      'train_predictions': train_predictions,
                      'test_predictions':  test_predictions,
                      'models_history': models_history
                     }

  print("\nBest model (neurons, replicate, rmse): ", num_neurons_with_best_rmse, min
  print('\nAverage scores:\n', avg_scores)
  print('\nStandard_deviations:\n', all_stds)
```

```
  print('\nMinimums:\n', all_minimums)
  print('\nMaximums:\n', all_maximums)
  print("\nProgress: All works are done successfully, congratulations!!\n")



  #Save all rmses in a file for statistical study
  scores['rmse'].to_csv(data_path+'sl-lstm-all-rmse.csv')

  #writing output dictionary in the file
  file_name = data_path + "sl-lstm-results.txt"
  write_dic_to_file(output_dictionary, file_name)

  return (output_dictionary)
```

Final Step: Models Executions and Results Visualization

```
t=np.zeros(10)
t.shape
```

```
(10,)
```

```
s=[0,0,0,0,0]
np.array(s).shape
```

```
(5,)
```

```
np.empty_like(nepse.values[:,0]).shape
```

```
(1051,)
```

Supporting model for visualization

```
def test_scores_plot(model_output):
  neurons = model_output['avg_scores']['neurons']
  rmse = model_output['avg_scores']['rmse']
  #mae =  model_output['avg_scores']['mae']
  mape =  model_output['avg_scores']['mape']
  #R2 =   model_output['avg_scores']['R2']
  R =    model_output['avg_scores']['R']
  #time =  model_output['avg_scores']['elapsed_time']

  fig = plt.figure(figsize = (18, 4))
  plt.subplot(131)
  plt.plot(neurons, rmse, '--o', linewidth = 2, color = 'indigo')
  plt.title("(a)")
  plt.xlabel("Neurons")
  plt.ylabel("Avg. RMSE")
  sns.set_style("whitegrid")


  plt.subplot(132)
  plt.plot(neurons, mape, '--o', linewidth = 2, color = 'darkgreen')
  plt.title("(b)")
  plt.xlabel("Neurons")
  plt.ylabel("Avg. MAPE")


  plt.subplot(133)
  plt.plot(neurons, R, '--o', linewidth = 2, color = 'darkred')
  plt.title("(c)")
  plt.xlabel("Neurons")
  plt.ylabel("Avg. R ")

  fig.savefig(data_path+"multiple_avg_scores_plots.png",dpi=600)
  plt.show()


def true_pred_plot(model_output):

  train_pred = model_output['best_model']['train_predictions']
  test_pred = model_output['best_model']['test_predictions']

  ##====== Visualizing true vs predicted plots ========#
  fig = plt.figure(figsize= (14,5))
  plt.subplot(121)
  #sns.relplot(x = y_train_original, y = train_pred_original)
  plt.scatter(y_train, train_pred, marker= "+", color = 'mediumblue')
  identity_line = np.linspace(max(min(y_train), min(train_pred)), min(max(y_train),
  plt.plot(identity_line, identity_line, color="red", linestyle="dashed", linewidth

  plt.xlabel("True")
  plt.ylabel("Predicted")
  plt.title("(a)")
```

```
  plt.subplot(122)
  #sns.relplot(x = y_test_original, y = test_pred_original)
  plt.scatter(y_test, test_pred, marker = "+", color = 'mediumblue')
  identity_line = np.linspace(max(min(y_test), min(test_pred)), min(max(y_test), ma
  plt.plot(identity_line, identity_line, color="red", linestyle="dashed", linewidth
  plt.xlabel("True")
  plt.ylabel("Predicted")
  plt.title("(b)")
  fig.savefig(data_path+"True_vs_predicted_plot.png", dpi=600)
  plt.show()


def prediction_plot(model_output):
  time_step =  5
  best_replicate = model_output['best_model']['replicate']

  train_pred = model_output['train_predictions'][0][best_replicate]
  test_pred = model_output['test_predictions'][0][best_replicate]

  train_predict_plot_data = np.empty_like(nepse.values[:,0])# extracting closing pr
  train_predict_plot_data[:] = np.nan



  test_predict_plot_data = np.empty_like(nepse.values[:,0])
  test_predict_plot_data[:] = np.nan



  fig1 = plt.figure(figsize = (18,12))

  plt.subplot(231)



  train_predict_plot_data[time_step:len(train_pred)+ time_step] =  train_pred.flatt
  test_predict_plot_data[len(train_pred)+(time_step)+1:len(nepse.values)] = test_pr



  plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
  plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
  plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
  plt.xlabel('')
  plt.ylabel('Close price')
  plt.title("(a)")
  plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes

  plt.subplot(232)
```

```
train_pred = model_output['train_predictions'][1][best_replicate]
test_pred = model_output['test_predictions'][1][best_replicate]

train_predict_plot_data[time_step:len(train_pred)+ time_step] =  train_pred
test_predict_plot_data[len(train_pred)+(time_step*2)+1:len(data.values)-1] =  tes

plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(b)")
plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes

plt.subplot(233)

train_pred = model_output['train_predictions'][2][best_replicate]
test_pred = model_output['test_predictions'][2][best_replicate]

train_predict_plot_data[time_step:len(train_pred)+ time_step] = train_pred
test_predict_plot_data[len(train_pred)+(time_step*2)+1:len(nepse.values)-1] =  te

plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(c)")
plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes


plt.subplot(234)

train_pred = model_output['train_predictions'][3][best_replicate]
test_pred = model_output['test_predictions'][3][best_replicate]

train_predict_plot_data[time_step:len(train_pred)+ time_step] = train_pred
test_predict_plot_data[len(train_pred)+(time_step*2)+1:len(nepse.values)-1] = tes

plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(d)")
plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes


plt.subplot(235)

train_pred = model_output['train_predictions'][4][best_replicate]
```

```
test_pred = model_output['test_predictions'][4][best_replicate]

train_predict_plot_data[time_step:len(train_pred)+ time_step] = train_pred
test_predict_plot_data[len(train_pred)+(time_step*2)+1:len(nepse.values)-1] = tes

plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(e)")
plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes

plt.subplot(236)

train_pred = model_output['train_predictions'][5][best_replicate]
test_pred = model_output['test_predictions'][5][best_replicate]

train_predict_plot_data[time_step:len(train_pred)+ time_step] = train_pred
test_predict_plot_data[len(train_pred)+(time_step*2)+1:len(nepse.values)-1] = tes

plt.plot(nepse.values[:,0],'k',linewidth = 1.5)
plt.plot(train_predict_plot_data,'mediumblue',linewidth = 1.5)
plt.plot(test_predict_plot_data,'darkgreen',linewidth = 1.5)
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(f)")
plt.legend(['True value', 'Predicted value in train set', 'Predicted value in tes


fig1.savefig(data_path+"predictions_plots_fullset.png",dpi=600)
plt.show()

fig2 = plt.figure(figsize = (18,12))

plt.subplot(231)
plt.plot(nepse.values[len(train_pred)+(time_step*2)+1:-1, 0],'mediumblue',linewic
plt.plot(model_output['test_predictions'][0][best_replicate], 'darkgreen',linewic
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(a)")
plt.legend(['True value', 'Predicted value'], loc='upper left')


plt.subplot(232)

plt.plot(nepse.values[len(train_pred)+(time_step*2)+1:-1, 0],'mediumblue',linewic
plt.plot(model_output['test_predictions'][1][best_replicate], 'darkgreen',linewic
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(b)")
```

```
plt.legend(['True value', 'Predicted value'], loc='upper left')


plt.subplot(233)

plt.plot(nepse.values[len(train_pred)+(time_step*2)+1:-1, 0],'mediumblue',linewic
plt.plot(model_output['test_predictions'][2][best_replicate], 'darkgreen',linewic
plt.xlabel('')
plt.ylabel('Close price')
plt.title("(c)")
plt.legend(['True value', 'Predicted value'], loc='upper left')
```