Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 019
Code Title:	Emerging Technologies in CpE 2 - 2nd Semester
ACTIVITY NO.	**Hands-on Activity 11.1**
Name	Dela Cruz, Irish
Section	CPE32S3
Date Performed:	05/01/2024
Date Submitted:	05/11/2024
Instructor:	Engr. Roman M. Richard

INSTRUCTION

Given an IBM stocks dataset between 2006 to 2018. You are task to do the following:

- Load the dataset and examine it.
- Check for missing values.
- Scale the training set from 0 to 1. Use MinMaxScaler and fit_transform function to do this.
- LSTM stores long-term memory states. To do this, create a data structure with 60 timesteps and 1 output. Thus, for each element of the training set, we shall have 60 previous training set elements.
- Reshape the X_train for efficient modeling

Importing libraries and packages

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Explonatory Data Analysis

Loading Dataset

```
data = pd.read_csv('/content/IBM_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=['Date'])
```

Dataset types and entries

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3020 entries, 2006-01-03 to 2017-12-29
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Open    3019 non-null   float64
 1   High    3020 non-null   float64
 2   Low     3019 non-null   float64
 3   Close   3020 non-null   float64
 4   Volume  3020 non-null   int64
 5   Name    3020 non-null   object
dtypes: float64(4), int64(1), object(1)
memory usage: 165.2+ KB
```

Remarks for dataset information:

The dataset contains 3020 entries with 7 columns, which include date, open, high, low, close, volume, and name. The data types involved are objects, floats, and objects.

```
data.head(20)
```

Date	Open	High	Low	Close	Volume	Name
2006-01-03	82.45	82.55	80.81	82.06	11715200	IBM
2006-01-04	82.20	82.50	81.33	81.95	9840600	IBM
2006-01-05	81.40	82.90	81.00	82.50	7213500	IBM
2006-01-06	83.95	85.03	83.41	84.95	8197400	IBM
2006-01-09	84.10	84.25	83.38	83.73	6858200	IBM
2006-01-10	83.15	84.12	83.12	84.07	5701000	IBM
2006-01-11	84.37	84.81	83.40	84.17	5776500	IBM
2006-01-12	83.82	83.96	83.40	83.57	4926500	IBM
2006-01-13	83.00	83.45	82.50	83.17	6921700	IBM
2006-01-17	82.80	83.16	82.54	83.00	8761700	IBM
2006-01-18	84.00	84.70	83.52	84.46	11032800	IBM
2006-01-19	84.14	84.39	83.02	83.09	6484000	IBM
2006-01-20	83.04	83.05	81.25	81.36	8614500	IBM
2006-01-23	81.33	81.92	80.92	81.41	6114100	IBM
2006-01-24	81.39	82.15	80.80	80.85	6069000	IBM
2006-01-25	81.05	81.62	80.61	80.91	6374300	IBM
2006-01-26	81.50	81.65	80.59	80.72	7810200	IBM
2006-01-27	80.75	81.77	80.75	81.02	6103400	IBM
2006-01-30	80.21	81.81	80.21	81.63	5325100	IBM
2006-01-31	81.50	82.00	81.17	81.30	6771600	IBM

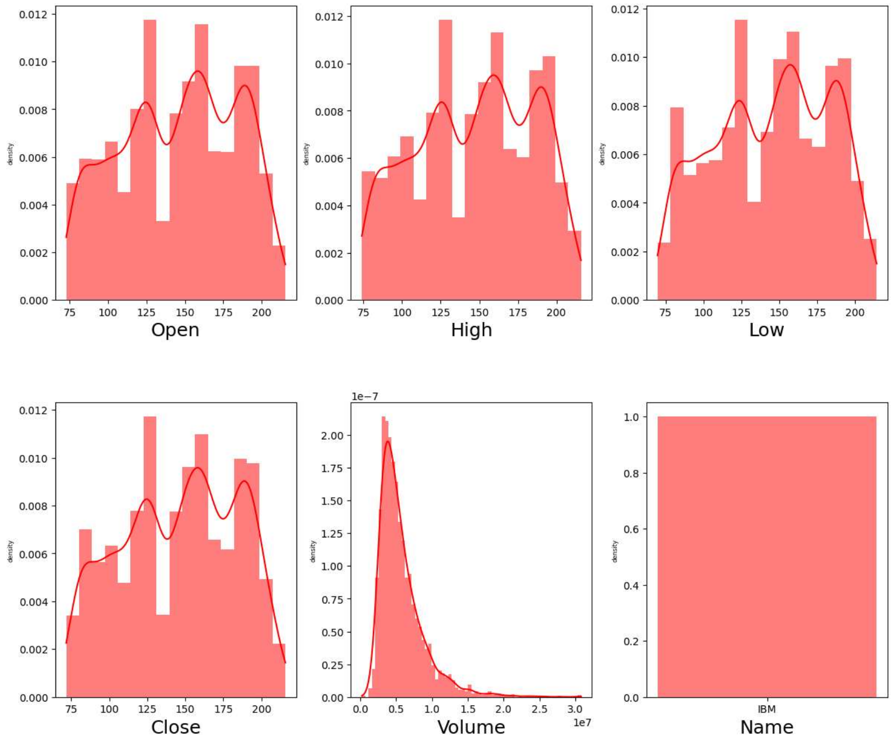
Next steps: View recommended plots

```
data.tail(20)
```

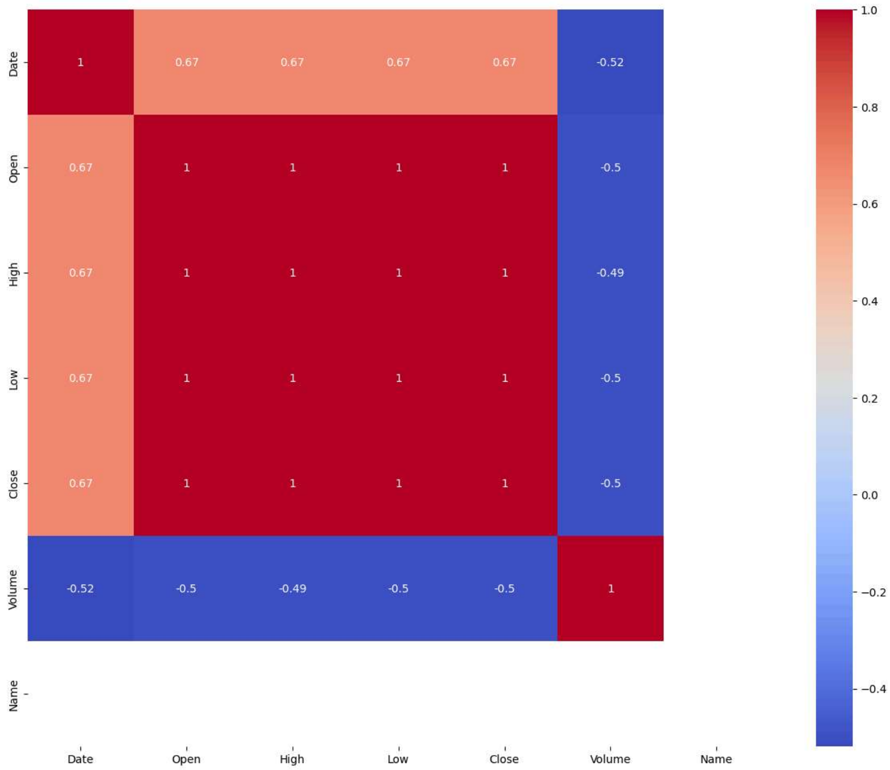
	Open	High	Low	Close	Volume	Name
Date						
2017-12-01	154.40	155.02	152.91	154.76	5567852	IBM
2017-12-04	155.96	156.80	155.07	156.46	4664316	IBM
2017-12-05	156.45	156.74	154.68	155.35	5068043	IBM
2017-12-06	154.10	156.22	154.09	154.10	3410728	IBM
2017-12-07	153.59	154.45	153.26	153.57	3771429	IBM
2017-12-08	154.81	155.03	153.55	154.81	3520281	IBM
2017-12-11	155.46	155.89	154.57	155.41	4102719	IBM
2017-12-12	156.74	157.85	155.16	156.74	6321801	IBM
2017-12-13	156.60	156.73	153.89	153.91	5661618	IBM
2017-12-14	154.60	155.11	153.70	154.00	4637440	IBM
2017-12-15	153.61	153.80	152.03	152.50	11279854	IBM
2017-12-18	153.59	154.18	153.21	153.33	5092838	IBM
2017-12-19	154.05	154.17	153.09	153.23	4116449	IBM
2017-12-20	153.65	153.89	152.78	152.95	3785667	IBM
2017-12-21	153.17	153.46	151.49	151.50	4153935	IBM
2017-12-22	151.82	153.00	151.50	152.50	2990583	IBM
2017-12-26	152.51	153.86	152.50	152.83	2479017	IBM
2017-12-27	152.95	153.18	152.61	153.13	2149257	IBM
2017-12-28	153.20	154.12	153.20	154.04	2687624	IBM
2017-12-29	154.17	154.72	153.42	153.42	3327087	IBM

Features demonstrating by graph

```
num_cols = min(len(data.columns), 3)
num_rows = (len(data.columns) + num_cols - 1) // num_cols

fig, ax = plt.subplots(ncols=num_cols, nrows= num_rows, figsize=(12, 10))
index = 0
ax = ax.flatten()

for col, value in data.items():
    col_dist = sns.histplot(value, ax=ax[index], color='red', kde=True, stat="density", linewidth=0)
    col_dist.set_xlabel(col, fontsize=18)
    col_dist.set_ylabel('density', fontsize=6)
    index += 1

plt.tight_layout(pad=0.5, w_pad=0.7, h_pad=5.0)
```




```
plt.figure(figsize=(15, 12))
sns.heatmap(data.corr(), annot=True, cmap='coolwarm')
plt.show()
```



Data Pre-Processing

Handling Missing Values

```
data.isnull().sum()
```

```
Open      1
High      0
Low       1
Close     0
Volume    0
Name      0
dtype: int64
```

```
missing_values = data.isnull().sum().sum()
rows_with_missing_values = data.isnull().any(axis=1).sum()

print(f'The number of missing values: {missing_values}')
print(f'The number of rows with missing values: {rows_with_missing_values}')
```

```
The number of missing values: 2
The number of rows with missing values: 1
```

Remarks for mising values

As you can see above, there's a 3 missing values in dataset. By improving the learning of the model we must fill all the missing values to predict accurately and effectively learn the machine well.

```
data["Open"].fillna(data["Open"].mean(), inplace=True)
data["Low"].fillna(data["Low"].mean(), inplace=True)
```

```
data.isnull().sum()
```

```
Open      0
High      0
Low       0
Close     0
Volume    0
Name      0
dtype: int64
```

Remarks for verifying missing values

As you can see here, by using the `fillna` function (filling all the missing values), there are no more missing values in the 7 columns.

```
data.shape
```

```
(3020, 6)
```

```
numerical_features = data.select_dtypes(include=['float64', 'int64'])
categorical_features = data.select_dtypes(include=['object'])

print("Numerical Features:")
print(numerical_features.columns)

print("\nCategorical Features:")
print(categorical_features.columns)
```

```
Numerical Features:
Index(['Open', 'High', 'Low', 'Close', 'Volume'], dtype='object')

Categorical Features:
Index(['Name'], dtype='object')
```

Remarks for seperating numerical and categorical variables

Since there's an object types within the dataset. We must need to convert those string into int or float type. Some of the time-series approach can't applied string dtype and must be numerical to actually learn by the model.

Handling Categorical Values

```
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()
data['Name'] = label_encoder.fit_transform(data['Name'])
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3020 entries, 2006-01-03 to 2017-12-29
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Open    3020 non-null   float64
 1   High    3020 non-null   float64
 2   Low     3020 non-null   float64
 3   Close   3020 non-null   float64
 4   Volume  3020 non-null   int64
```

```
 5   Name    3020 non-null   int64
dtypes: float64(4), int64(2)
memory usage: 165.2 KB
```

Remarks for categorical values

After the label being done converted the categorical transform into numerical datatype. All of the variables there are now ready to handle any pattern or model since there all in numerical form

Describe Table

```
data.describe()
```

	Open	High	Low	Close	Volume	Name
count	3020.000000	3020.000000	3020.000000	3020.000000	3.020000e+03	3020.0
mean	145.515545	146.681738	144.471597	145.617278	5.773301e+06	0.0
std	37.548726	37.613446	37.471433	37.529387	3.192831e+06	0.0
min	72.740000	73.940000	69.500000	71.740000	2.542560e+05	0.0
25%	116.407500	117.765000	115.500000	116.525000	3.622681e+06	0.0
50%	149.605000	150.330000	148.425000	149.315000	4.928852e+06	0.0
75%	178.437500	179.762500	177.320000	178.685000	6.965014e+06	0.0
max	215.380000	215.900000	214.300000	215.800000	3.077428e+07	0.0

Remarks for describe

The result above, regarding with the 25% and 75% is a big gap compared to each of them respectively. This proves that there's an outliers goes within the variables, to visualized and to verify those things a box plot will be performed

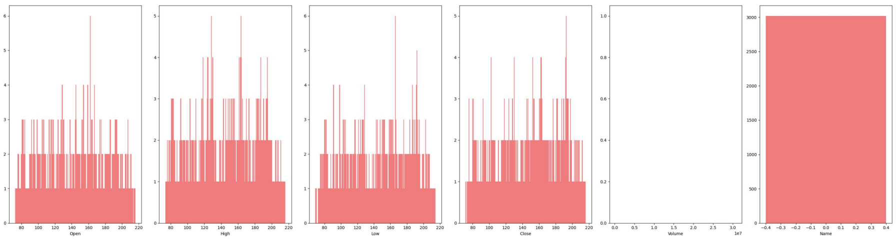
Handling Outliers

```
data.columns
```

```
Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Name'], dtype='object')
```

```
columns = ['Open', 'High', 'Low', 'Close', 'Volume', 'Name']
```

```
fig, axs = plt.subplots(1, len(columns), figsize=(30, 8))

for i, col in enumerate(columns):
    value_counts = data[col].value_counts()
    axs[i].bar(value_counts.index, value_counts.values, color=('lightcoral'))
    axs[i].set_xlabel(col)

plt.tight_layout()
plt.show()
```



```
data.shape
```

```
(3020, 6)
```

Remarks for Outliers

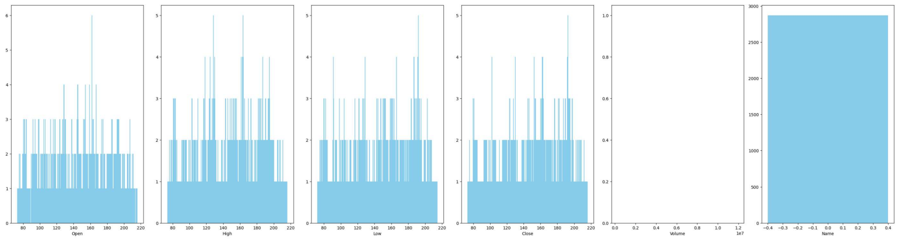
As you can see there's 3020 instances that have been detected.

Removing outliers using IQR

```
def IQR_outliers(data, column_name, thresh=1.5):
  Q1 = data[column_name].quantile(0.25)
  Q3 = data[column_name].quantile(0.75)
  IQR = Q3 - Q1
  lower_bound = Q1 - (thresh * IQR)
  upper_bound = Q3 + (thresh * IQR)

  return data[(data[column_name] >= lower_bound) & (data[column_name] <= upper_bound)]
```

```
indices_to_keep = np.array([])
for col in columns:
    outliers_removed_X = IQR_outliers(data, col)
    indices_to_keep = np.intersect1d(indices_to_keep, outliers_removed_X.index) if indices_to_keep.size else outliers_removed_X.index

data = data.loc[indices_to_keep]
```

```
fig, axs = plt.subplots(1,len(columns), figsize=(30, 8))

for i, col in enumerate(columns):
    value_counts = data[col].value_counts()
    axs[i].bar(value_counts.index, value_counts.values, color='skyblue')
    axs[i].set_xlabel(col)

plt.tight_layout()
plt.show()
```




```
data.shape
```

```
(2871, 6)
```

Remarks for removing outliers

Using IQR Rule the instances are now 2,871 from 3027. 156 instances have been removed

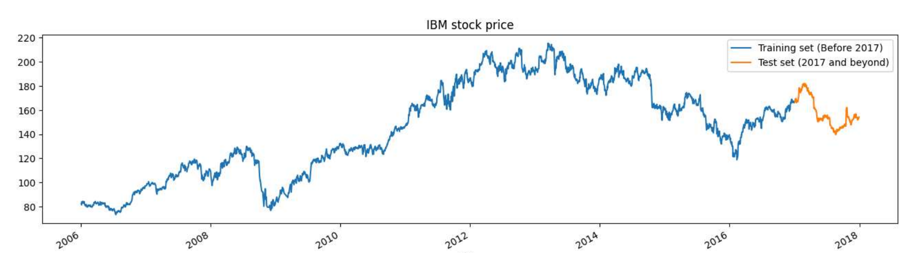
```
data.columns
```

```
Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Name'], dtype='object')
```

Task

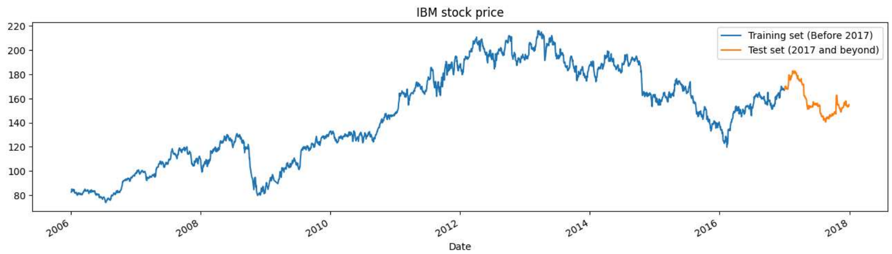
Open Attributes for prices

```
data["Open"][:'2016'].plot(figsize=(16,4),legend=True)
data["Open"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



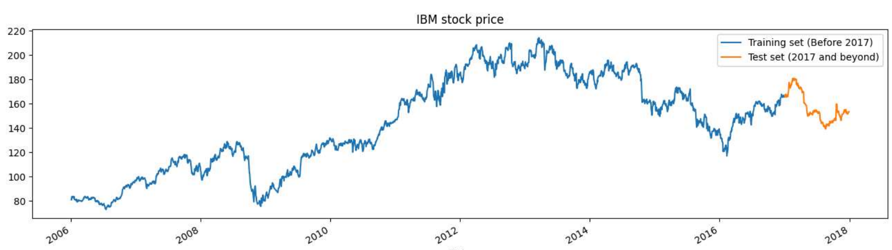
High Attributes for prices.

```
data["High"][:'2016'].plot(figsize=(16,4),legend=True)
data["High"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



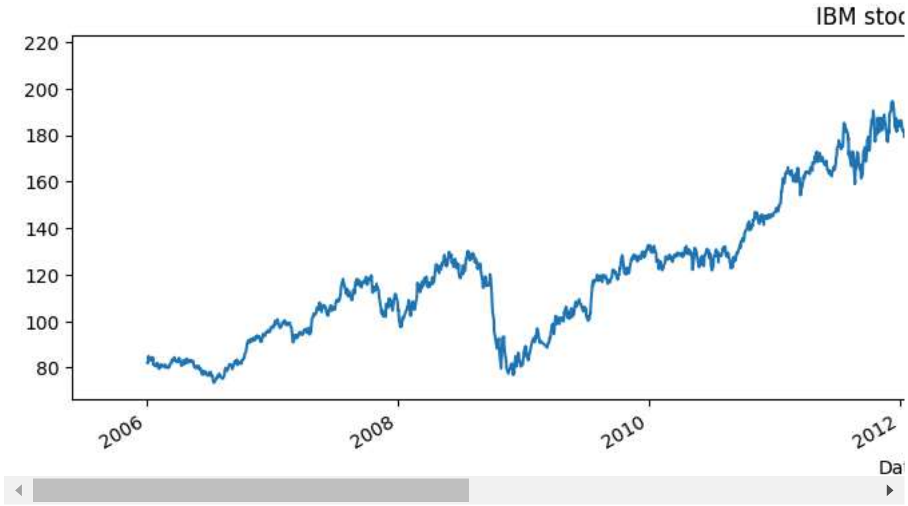
Low Attributes for prices

```
data["Low"][:'2016'].plot(figsize=(16,4),legend=True)
data["Low"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



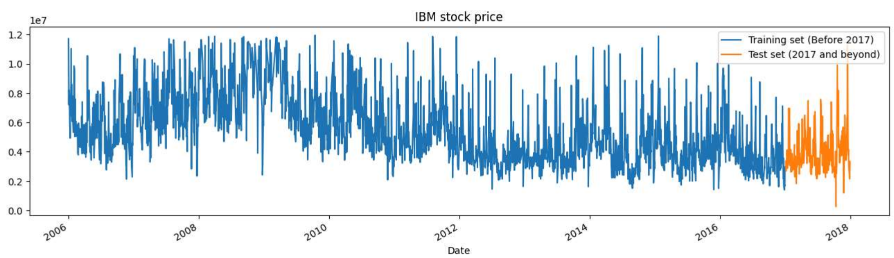
Close Attributes for prices

```
data["Close"][:'2016'].plot(figsize=(16,4),legend=True)
data["Close"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



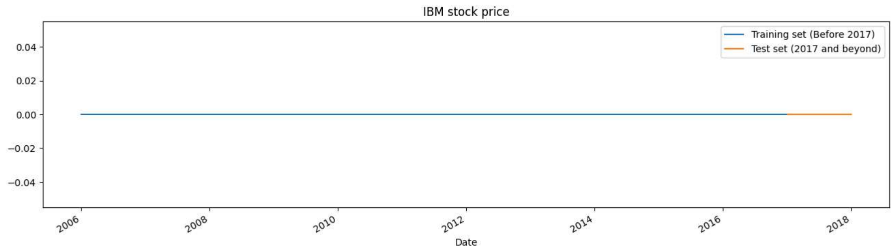
Volume Attributes for prices

```
data["Volume"][:'2016'].plot(figsize=(16,4),legend=True)
data["Volume"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



Name Attributes for prices

```
data["Name"][:'2016'].plot(figsize=(16,4),legend=True)
data["Name"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



Remarks for all the attributes

As observed above, there's minimal variation among variables like open, high, low, and close as they possess similar values. However, the volume component exhibits considerably more noise, primarily due to the presence of significant values. The interquartile range (IQR) method struggles to effectively remove this noise due to the presence of large values.

Scale the training set from 0 to 1. Use MinMaxScaler and fit_transform function to do this.

```
training_set = data[:'2016'].iloc[:,1:2].values
test_set = data['2017':].iloc[:,1:2].values
```

```
from sklearn.preprocessing import MinMaxScaler

sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

Remarks

Scaling the training set from 0 to 1 using MinMaxScaler ensures that features are on a similar scale, accelerating convergence, enhancing stability, and improving optimization effectiveness. This preprocessing step facilitates better model performance and generalization.

LSTM stores long-term memory states.

To do this, create a data structure with 60 timesteps and 1 output. Thus, for each element of the training set, we shall have 60 previous training set elements.

```
X_train = []
y_train = []

for i in range(60, len(training_set_scaled)):
    X_train.append(training_set_scaled[i-60:i, 0])
    y_train.append(training_set_scaled[i, 0])

X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
```

```
X_train shape: (2565, 60, 1)
y_train shape: (2565,)
```

Remarks for LSTM

`X_train` has a shape of (2565, 60, 1), indicating 2565 samples, each comprising a sequence of 60 consecutive time steps, with a single feature representing the high price of IBM stock. `y_train` is of shape (2565,), representing 2565 target values corresponding to the subsequent high price after each sequence. These shapes define the input-output pairs for training the LSTM model to predict future high prices of IBM stock based on historical data.

LSTM Architecture

```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout

regressor = Sequential()
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
regressor.add(Dense(units=1))
regressor.compile(optimizer='rmsprop', loss='mean_squared_error')
```

```
regressor.fit(X_train, y_train, epochs=50, batch_size=32)
```

```
Epoch 21/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0028
Epoch 22/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0029
Epoch 23/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0023
Epoch 24/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 0.0025
Epoch 25/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0022
Epoch 26/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0024
Epoch 27/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0025
Epoch 28/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 0.0023
Epoch 29/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026
Epoch 30/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - loss: 0.0020
Epoch 31/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0020
Epoch 32/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0019
Epoch 33/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0022
Epoch 34/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0021
Epoch 35/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 0.0020
Epoch 36/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0020
Epoch 37/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 0.0018
Epoch 38/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0019
Epoch 39/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - loss: 0.0020
Epoch 40/50
81/81 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - loss: 0.0018
Epoch 41/50
```