```
# This is formatted as code
```

Recurrent neural network (RNN) Use the Google stock prices dataset and design a time series analysis and prediction system using RNN.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import datetime
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.metrics import r2_score
```

```
#data=pd.read_csv("/content/Google_Stock_Price_Train.csv",index_col='Date',parse_dates=True)
data=pd.read_csv("/content/Google_Stock_Price_Train.csv")
data
```

	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
1	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
2	1/5/2012	329.83	330.75	326.89	657.21	6,590,300
3	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
4	1/9/2012	322.04	322.29	309.46	620.76	11,688,800
...	...	...	...	...	...	...
1253	12/23/2016	790.90	792.74	787.28	789.91	623,400
1254	12/27/2016	790.68	797.86	787.66	791.55	789,100
1255	12/28/2016	793.70	794.23	783.20	785.05	1,153,800
1256	12/29/2016	783.33	785.93	778.92	782.79	744,300
1257	12/30/2016	782.75	782.78	770.41	771.82	1,770,000

1258 rows × 6 columns

```
data['Close']=data['Close'].str.replace(',','').astype(float)
data['Volume']=data['Volume'].str.replace(',','').astype(float)
```

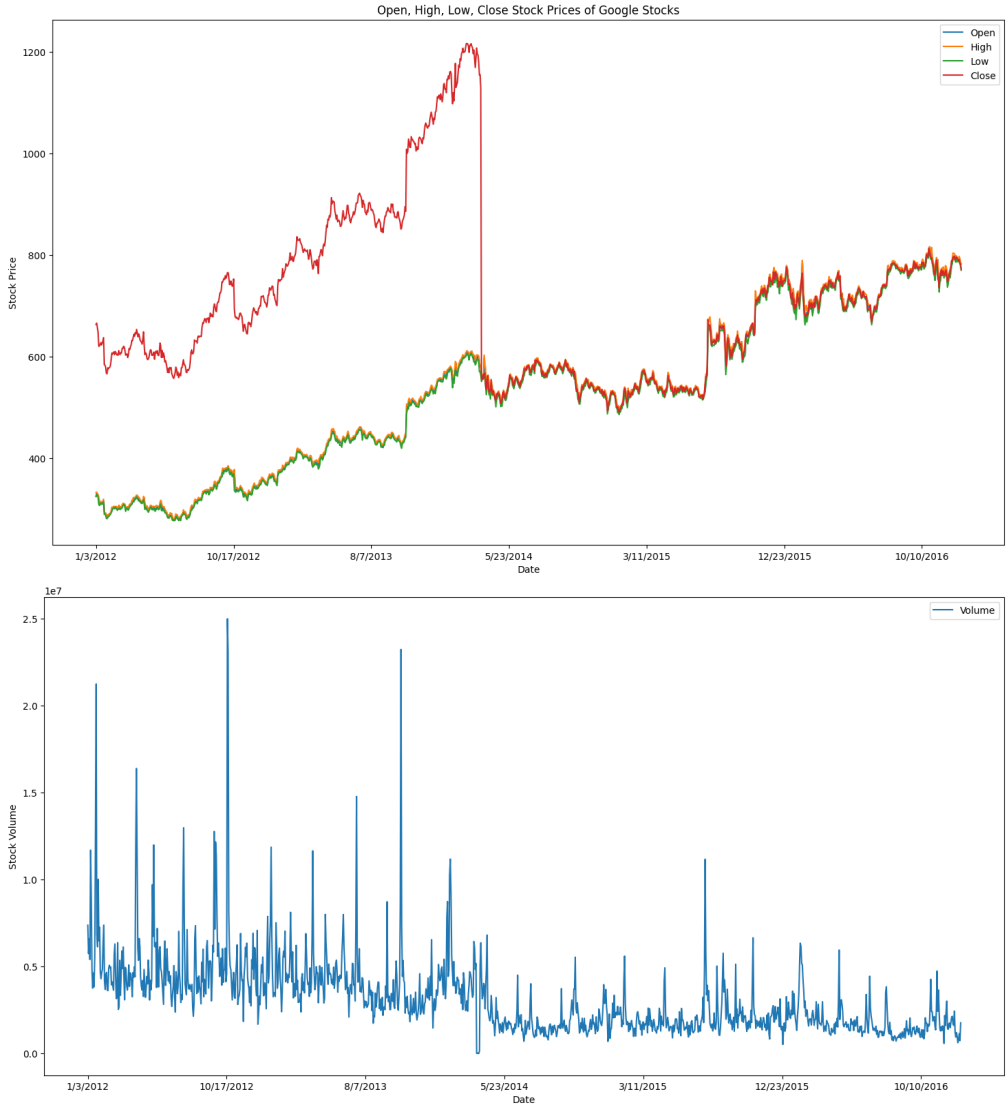
In the dataset, the Volume column had thousand seperated values, which is taken as string when usually loading to Pandas DataFrame. I have specified that ',' is used as the thousand separator when loading, so that the ',' will not be there when loading to our program. Then it'll be loaded as a numerical value.

Plotting the Data

```
ax1 = data.plot(x="Date", y=["Open", "High", "Low", "Close"],  figsize=(18,10),title='Open, High, Low, Close Stock Prices of Google Stocks')
ax1.set_ylabel("Stock Price")

ax2 = data.plot(x="Date", y=["Volume"],  figsize=(18,9))
ax2.set_ylabel("Stock Volume")
```

```
Text(0, 0.5, 'Stock Volume')
```



Double-click (or enter) to edit

When inspecting the first plot, we can see that in the closing value of stock prior to 2014/03/26, there is a significant gap between the closing price and the other prices, where as after this date, the closing price has also been closer to other prices.

When inspecting the second plot, we can see that there are significant fluctuations in the Stock Volume over the time. And it is quite hard to find a trend/pattern in this the data in Stock Volume (the one we need to predict).

Preprocessing Data Preprocessing data includes handling missing values and outliers, applying feature coding techniques if needed, scale & standardize features.

Checking for Missing values

```
# Getting a summary of missing values for each field/attribute
print(data.isnull().sum())
```

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

We can see that there are no missing values in the dataset.

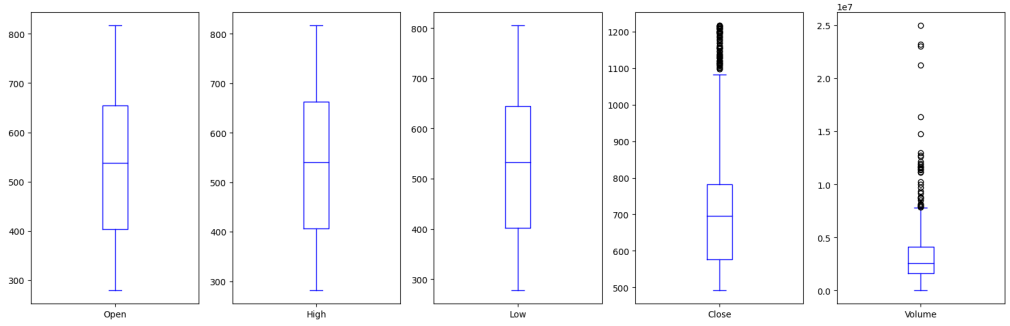
Handling Outliers

According to the above graph plots that indicate the Open, High, Low and Close values of the Stock prices, the Close price is as mentioned earlier, has a significant gap with the other values. The surprising fact is that, the Close value has even become higher than the High value (highest price of stock for a given day), which is not real.

Ref: https://www.mit.edu/~mbarker/formula1/f1help/11-ch-10.htm

Let us also check outliers for attributes by using box plots as follows.

```
data[['Open','High','Low','Close','Volume']].plot(kind= 'box' ,layout=(1,5),subplots=True, sharex=False, sharey=False, figsize=(20,6),color='
plt.show()
```



We can see that there are outliers in the Close and Volume attributes. However, i will not remove any outliers here since there are only a limited number of datapoints (less than 1300) and if we remove outliers, the dataset will become even smaller.

Feature Encoding

When we check the 1st 10 records of the dataset, we could see that all the data in the dataset are numerical data, and there are no categorical data that needs encoding. Therefore, no feature encoding process was carried out on this dataset.

Feature Scaling

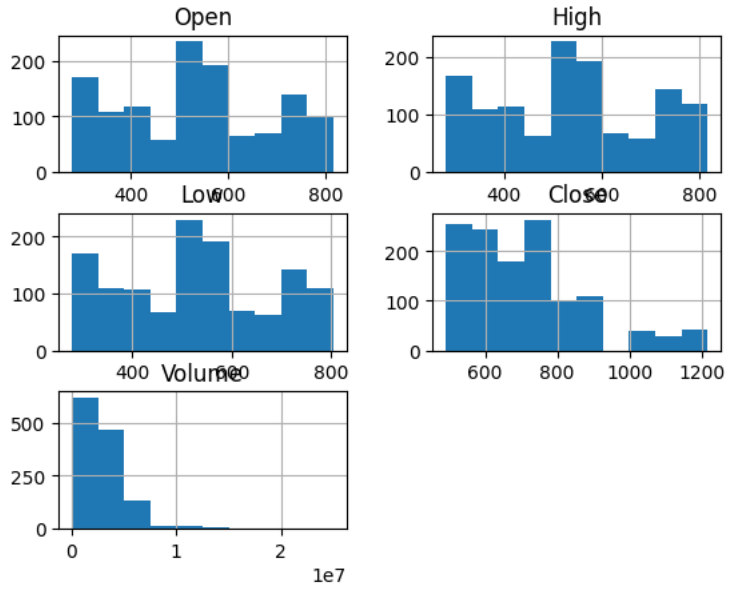
MinMaxScaler, which is said to be better to be used with time-series data, was used on this dataset for the feature scaling purposes.

https://machinelearningmastery.com/normalize-standardize-time-series-data-python/

The following set of graphs show the attribute histogram before scaling

```
data.hist()
```

```
array([[<Axes: title={'center': 'Open'}>,
        <Axes: title={'center': 'High'}>],
       [<Axes: title={'center': 'Low'}>,
        <Axes: title={'center': 'Close'}>],
       [<Axes: title={'center': 'Volume'}>, <Axes: >]], dtype=object)
```




```
data
```

	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7380500.0
1	1/4/2012	331.27	333.87	329.08	666.45	5749400.0
2	1/5/2012	329.83	330.75	326.89	657.21	6590300.0
3	1/6/2012	328.34	328.77	323.68	648.24	5405900.0
4	1/9/2012	322.04	322.29	309.46	620.76	11688800.0
...	...	...	...	...	...	...
1253	12/23/2016	790.90	792.74	787.28	789.91	623400.0
1254	12/27/2016	790.68	797.86	787.66	791.55	789100.0
1255	12/28/2016	793.70	794.23	783.20	785.05	1153800.0
1256	12/29/2016	783.33	785.93	778.92	782.79	744300.0
1257	12/30/2016	782.75	782.78	770.41	771.82	1770000.0

1258 rows × 6 columns

```
scaler = MinMaxScaler()
data_without_date = data[['Open','High','Low','Close','Volume']]
data_scaled = pd.DataFrame(scaler.fit_transform(data_without_date))
```

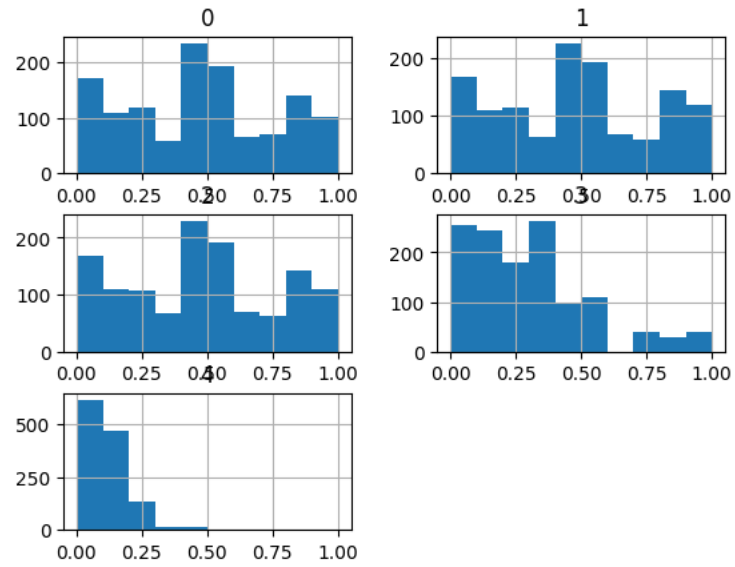
```
data_scaled
```

	0	1	2	3	4
0	0.085814	0.096401	0.090449	0.237573	0.295258
1	0.097012	0.098344	0.098235	0.241514	0.229936
2	0.094334	0.092517	0.094086	0.228781	0.263612
3	0.091562	0.088819	0.088006	0.216419	0.216179
4	0.079842	0.076718	0.061070	0.178548	0.467797
...	...	...	...	...	...
1253	0.952043	0.955292	0.966169	0.411656	0.024650
1254	0.951633	0.964853	0.966889	0.413916	0.031286
1255	0.957251	0.958074	0.958441	0.404958	0.045891
1256	0.937960	0.942574	0.950333	0.401844	0.029491
1257	0.936881	0.936691	0.934214	0.386726	0.070569

1258 rows × 5 columns

```
data_scaled.hist()
```

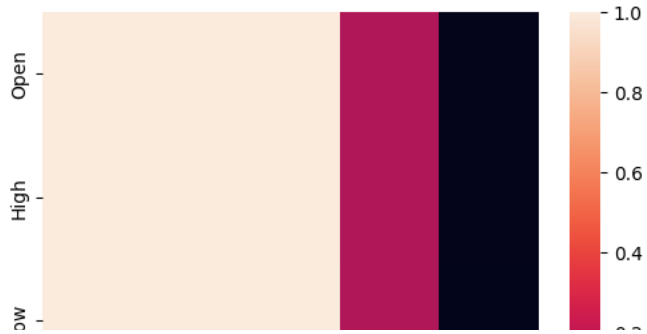
```
array([[<Axes: title={'center': '0'}>, <Axes: title={'center': '1'}>],
       [<Axes: title={'center': '2'}>, <Axes: title={'center': '3'}>],
       [<Axes: title={'center': '4'}>, <Axes: >]], dtype=object)
```



Feature Engineering Drawing the Correlation Matrix Correlation Coefficient checking mechanism checks the relationship between the different features with the predicting attribute.

```
plt.figure(figsize=(6,6))
sns.heatmap(data.corr())
```

```
<ipython-input-12-373595b0d5cd>:2: FutureWarning: The default value of numeric_only in DataFrame.c
  sns.heatmap(data.corr())
<Axes: >
```



We can see from the correlation matrix that the features Open, High, and Low, all are highly correlated with each other. Therefore, it is sufficient to consider only one feature from the above 3 features and remove the rest.

Furthermore, I am removing the Close field since it contains many abnormal data values, which is like a contexual outlier.

```
data_scaled=data_scaled.drop([1,2,4], axis=1)
data_scaled
```

	0	3
0	0.085814	0.237573
1	0.097012	0.241514
2	0.094334	0.228781
3	0.091562	0.216419
4	0.079842	0.178548
...	...	...
1253	0.952043	0.411656
1254	0.951633	0.413916
1255	0.957251	0.404958
1256	0.937960	0.401844
1257	0.936881	0.386726

1258 rows × 2 columns

Developing the RNN Model

When developing an RNN Model, we have to reshape the data that we are feeding into the model. To do that, first we have to find a pattern in the data available and define the number of timesteps according to the pattern.

When considering the plots we have created for this we could not see a clear cut pattern/trend in the data by just visual inspection. Therefore, I have taken the following code snippet to split the data into a sequence by trying to identify any pattern available.

Since there are two features in our dataset after cleaning, I have used a split sequence method for a multivariate dataset.

```
def split_seq_multivariate(sequence, n_past, n_future):

    '''
    n_past ==> no of past observations
    n_future ==> no of future observations
    '''
    x, y = [], []
    for window_start in range(len(sequence)):
        past_end = window_start + n_past
        future_end = past_end + n_future
        if future_end > len(sequence):
            break
        # slicing the past and future parts of the window
        past   = sequence[window_start:past_end, :]
```

```
        future = sequence[past_end:future_end, -1]
        x.append(past)
        y.append(future)

    return np.array(x), np.array(y)
```

In RNN, since we are dealing with time series data, we have to specify how many past data points we will be considering when generating the sequence.

In here, i have taken 60 past data points (time steps) when generating the data sequences.

```
# specify the window size
n_steps = 60

data_scaled = data_scaled.to_numpy()
data_scaled.shape
```

```
(1258, 2)
```

Next, I am using the split_seq_multivariate function to split the dataset into sequences.

```
# split into samples
X, y = split_seq_multivariate(data_scaled, n_steps,1)


X[1].shape
```

```
(60, 2)
```

```
y
```

```
array([[0.21420007],
       [0.20434657],
       [0.21216047],
       ...,
       [0.40495845],
       [0.40184391],
       [0.38672602]])
```

```
# X is in the shape of [samples, timesteps, features]
print(X.shape)
print(y.shape)

# make y to the shape of [samples]
y=y[:,0]
y.shape
```

```
(1198, 60, 2)
(1198, 1)
(1198,)
```

Splitting the Data

In this step, I will be splitting the sequenced data into train and test sections with 0.2 test size.

Furthermore, i have split the training set again to train and validation data. I have not used the test data to do the validation because validation data are used to fine tune the model, and if i used the testing data for validation purposes, then those data will be already seen by the model when trying to predict them later.

Ref: https://machinelearningmastery.com/difference-test-validation-datasets/

```
# split into train/test
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=50)

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

```
(958, 60, 2) (240, 60, 2) (958,) (240,)
```

```
# further dividing the training set into train and validation data
X_train, X_val, y_train, y_val = train_test_split(X_train,y_train,test_size=0.2,random_state=30)

print(X_train.shape, X_val.shape, y_train.shape, y_val.shape)
```

```
(766, 60, 2) (192, 60, 2) (766,) (192,)
```

Next I am defining the model using Tensorflow Keras LSTM and Dense layers.

In the LSTM layer, i have not specified the activation function explicitly because in the Keras Documentation it is specified that the default activation function for LSTM is tanh and default recurrent activation function is sigmoid. Therefore those default acivation functions will be used here.

Ref: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell

Define Model

```
# define RNN model
model = Sequential()
model.add(LSTM(612, input_shape=(n_steps,2)))
model.add(Dense(50, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(1))
```

The following code line gives a summary of the model we have created, mentioning each layers information.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 612)               1505520

 dense (Dense)               (None, 50)                30650

 dense_1 (Dense)             (None, 50)                2550

 dense_2 (Dense)             (None, 30)                1530

 dense_3 (Dense)             (None, 1)                 31

=================================================================
Total params: 1,540,281
Trainable params: 1,540,281
Non-trainable params: 0
_________________________________________________________________
```

Compiling and Training the Model

Next we will be compiling and fitting the model with the training data, and using the validation data to fine tune our model.

```
# compile the model
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

```
# fit the model
history = model.fit(X_train, y_train, epochs=250, batch_size=32, verbose=2, validation_data=(X_val, y_val))  # has used mini batch learning,
```

```
Epoch 1/250
24/24 - 13s - loss: 0.0382 - mae: 0.1368 - val_loss: 0.0158 - val_mae: 0.0940 - 13s/epoch - 551ms/step
Epoch 2/250
24/24 - 0s - loss: 0.0056 - mae: 0.0490 - val_loss: 0.0043 - val_mae: 0.0323 - 319ms/epoch - 13ms/step
Epoch 3/250
24/24 - 0s - loss: 0.0027 - mae: 0.0277 - val_loss: 0.0036 - val_mae: 0.0287 - 353ms/epoch - 15ms/step
Epoch 4/250
24/24 - 0s - loss: 0.0026 - mae: 0.0271 - val_loss: 0.0057 - val_mae: 0.0473 - 319ms/epoch - 13ms/step
Epoch 5/250
24/24 - 0s - loss: 0.0027 - mae: 0.0283 - val_loss: 0.0033 - val_mae: 0.0270 - 316ms/epoch - 13ms/step
Epoch 6/250
24/24 - 0s - loss: 0.0018 - mae: 0.0208 - val_loss: 0.0028 - val_mae: 0.0278 - 328ms/epoch - 14ms/step
```

```
Epoch 7/250
24/24 - 0s - loss: 0.0023 - mae: 0.0258 - val_loss: 0.0032 - val_mae: 0.0227 - 323ms/epoch - 13ms/step
Epoch 8/250
24/24 - 0s - loss: 0.0021 - mae: 0.0223 - val_loss: 0.0031 - val_mae: 0.0226 - 352ms/epoch - 15ms/step
Epoch 9/250
24/24 - 0s - loss: 0.0021 - mae: 0.0240 - val_loss: 0.0028 - val_mae: 0.0243 - 360ms/epoch - 15ms/step
Epoch 10/250
24/24 - 0s - loss: 0.0021 - mae: 0.0264 - val_loss: 0.0031 - val_mae: 0.0346 - 327ms/epoch - 14ms/step
Epoch 11/250
24/24 - 0s - loss: 0.0020 - mae: 0.0238 - val_loss: 0.0025 - val_mae: 0.0245 - 318ms/epoch - 13ms/step
Epoch 12/250
24/24 - 0s - loss: 0.0017 - mae: 0.0217 - val_loss: 0.0026 - val_mae: 0.0245 - 318ms/epoch - 13ms/step
Epoch 13/250
24/24 - 0s - loss: 0.0018 - mae: 0.0239 - val_loss: 0.0030 - val_mae: 0.0303 - 350ms/epoch - 15ms/step
Epoch 14/250
24/24 - 0s - loss: 0.0015 - mae: 0.0188 - val_loss: 0.0023 - val_mae: 0.0271 - 356ms/epoch - 15ms/step
Epoch 15/250
24/24 - 0s - loss: 0.0019 - mae: 0.0254 - val_loss: 0.0019 - val_mae: 0.0229 - 359ms/epoch - 15ms/step
Epoch 16/250
24/24 - 0s - loss: 0.0016 - mae: 0.0212 - val_loss: 0.0019 - val_mae: 0.0211 - 390ms/epoch - 16ms/step
Epoch 17/250
24/24 - 0s - loss: 0.0014 - mae: 0.0175 - val_loss: 0.0014 - val_mae: 0.0204 - 387ms/epoch - 16ms/step
Epoch 18/250
24/24 - 0s - loss: 0.0015 - mae: 0.0212 - val_loss: 0.0016 - val_mae: 0.0221 - 363ms/epoch - 15ms/step
Epoch 19/250
24/24 - 0s - loss: 0.0015 - mae: 0.0186 - val_loss: 0.0016 - val_mae: 0.0185 - 347ms/epoch - 14ms/step
Epoch 20/250
24/24 - 0s - loss: 0.0014 - mae: 0.0181 - val_loss: 0.0014 - val_mae: 0.0183 - 322ms/epoch - 13ms/step
Epoch 21/250
24/24 - 0s - loss: 0.0014 - mae: 0.0175 - val_loss: 0.0015 - val_mae: 0.0217 - 321ms/epoch - 13ms/step
Epoch 22/250
24/24 - 0s - loss: 0.0014 - mae: 0.0177 - val_loss: 0.0013 - val_mae: 0.0184 - 317ms/epoch - 13ms/step
Epoch 23/250
24/24 - 0s - loss: 0.0014 - mae: 0.0178 - val_loss: 0.0017 - val_mae: 0.0253 - 329ms/epoch - 14ms/step
Epoch 24/250
24/24 - 0s - loss: 0.0015 - mae: 0.0213 - val_loss: 0.0014 - val_mae: 0.0185 - 350ms/epoch - 15ms/step
Epoch 25/250
24/24 - 0s - loss: 0.0016 - mae: 0.0218 - val_loss: 0.0013 - val_mae: 0.0180 - 321ms/epoch - 13ms/step
Epoch 26/250
24/24 - 0s - loss: 0.0014 - mae: 0.0177 - val_loss: 0.0018 - val_mae: 0.0281 - 324ms/epoch - 14ms/step
Epoch 27/250
24/24 - 0s - loss: 0.0016 - mae: 0.0215 - val_loss: 0.0014 - val_mae: 0.0192 - 325ms/epoch - 14ms/step
Epoch 28/250
24/24 - 0s - loss: 0.0014 - mae: 0.0206 - val_loss: 0.0016 - val_mae: 0.0214 - 321ms/epoch - 13ms/step
Epoch 29/250
24/24 - 0s - loss: 0.0012 - mae: 0.0146 - val_loss: 0.0013 - val_mae: 0.0199 - 322ms/epoch - 13ms/step
```

```
from matplotlib import pyplot
# plot learning curves
pyplot.title('Learning Curves')
pyplot.xlabel('Epoch')
pyplot.ylabel('Root Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='val')
pyplot.legend()
pyplot.show()
```

Model Evaluation and Predictions

Next, i will be using the model.evaluate() and model.predict() methods.

The model.evaluate() is used to only check the MSE, RMSE, MAE values of our model when the testing data is used. This does not provide us with the results the model would predict. Therefore to see the results that the model would predict, i have used the model.predict() function. Later I have plotted the actual and predicted values of the test data to see how well our model has performed.

Ref: https://stackoverflow.com/questions/44476706/what-is-the-difference-between-keras-model-evaluate-and-model-predict

```
# evaluate the model
mse, mae = model.evaluate(X_test, y_test, verbose=0)
print('MSE: %.3f, RMSE: %.3f, MAE: %.3f' % (mse, np.sqrt(mse), mae))
```

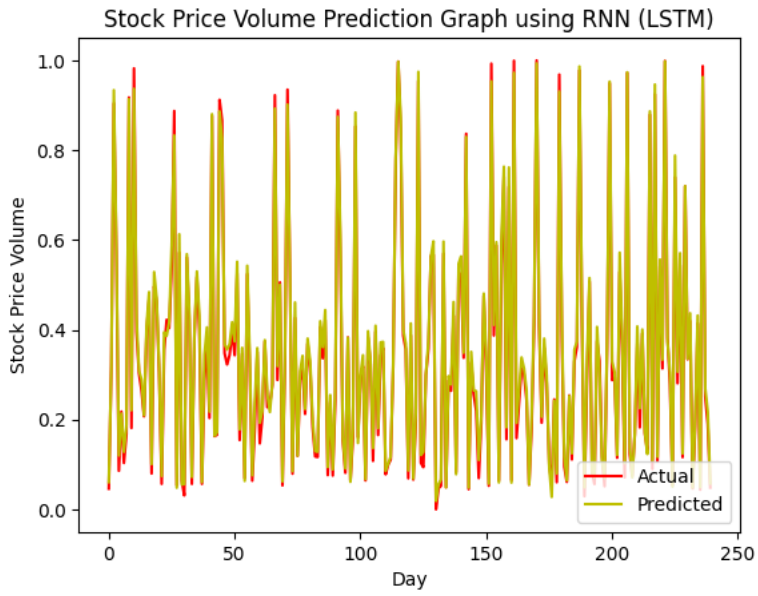
```
MSE: 0.000, RMSE: 0.022, MAE: 0.018
```

```
# predicting y_test values
print(X_test.shape)
predicted_values = model.predict(X_test)
print(predicted_values.shape)
# print(predicted_values)
```

```
(240, 60, 2)
8/8 [==============================] - 0s 5ms/step
(240, 1)
```

```
plt.plot(y_test,c = 'r')
plt.plot(predicted_values,c = 'y')
plt.xlabel('Day')
plt.ylabel('Stock Price Volume')
plt.title('Stock Price Volume Prediction Graph using RNN (LSTM)')
plt.legend(['Actual','Predicted'],loc = 'lower right')
plt.figure(figsize=(10,6))
```

```
<Figure size 1000x600 with 0 Axes>
```




```
<Figure size 1000x600 with 0 Axes>
```

Using R Squared

The R squared is a metric we can use to evaluate how well our regression based (model that handle continuous data) have performed.

R squared can vary between 0 and 1 and we can evaluate our model using this metric, while a negative value would mean that the training has not been done properly.