Technical Indicators and GRU/LSTM to Predict Stock price

Predictive model using multivariate time-Series and Python Code

Sarit Maitra Nov 20, 2019 · 10 min read ★

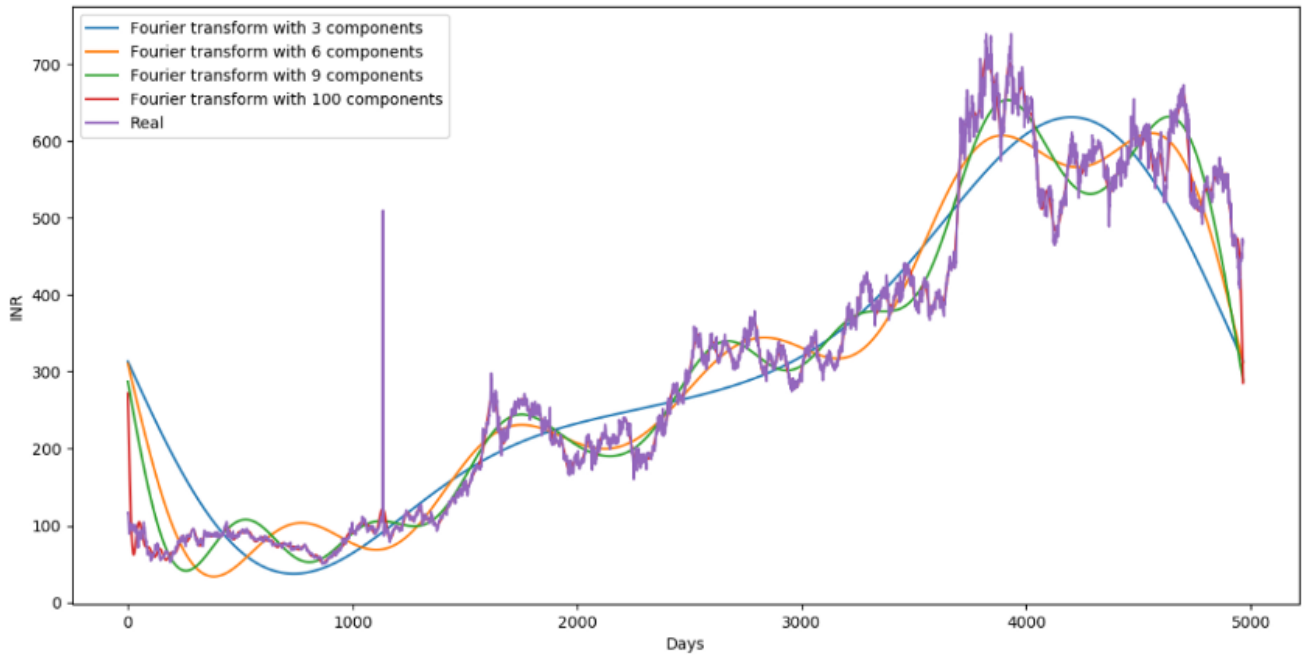


Image by Sarit Maitra

TECHNICAL indicators are available at large numbers and used mostly by stock traders. Most indicators have user-defined variables which allow traders to adapt key inputs such as look-back period which us how much historical data will be used to form the calculations to suit their needs. We will use some of the indicators to create

features in the existing data set. Applying these features, we will see if we can predict the future price of a particular stock.

We will use ***Gated Recurrent Unit (GRU)*** and ***Long Short-Term Memory (LSTM)*** of recurrent units to compare their performance. LSTM is well established on sequence-based tasks with long-term dependencies, and GRU is a new addition in the field of machine learning which is an improvised version of ***Recurrent Neural Network(RNN)***. I found this **article** is quite interesting to know more about these two network architecture.

There are some similarities and differences between GRU and LSTM.

Similarities

- Both the network have additive component from t to t + 1; new content is added on the top of existing content.
- Both addresses vanishing and exploding gradient issue
- The update gate in GRU and forget gate in LSTM takes the linear sum between the existing state and the newly computed state

Differences

- GRU has two gates, reset and update gates compared to LSTM has three gates, input, forget and output. GRU does not have an output gate like LSTM. Update gate in GRU does the work of input and forget gate of LSTM.
- GRU is computationally more efficient considering fewer parameters and need less data to generalize.
- LSTM maintains an internal memory state cell, while GRU does not have a separate memory cell
- GRU does not have any mechanism to control the degree to which its state or memory content is exposed, but exposes the whole state or memory content each time. LSTM can control how much memory content it wants to expose.

You may visit this article to get more theoretical ideas about GRU and LSTM network architecture. I will explain here both feature creation and applying RNN with a

simplified example.

```
# Import Gold data in panda and passed to a variable name "df"
df = pd.read_csv("CIPLA.NS.csv", parse_dates= True)
df.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	1/3/2000	112.800003	116.587997	110.403999	116.587997	69.387695	263250.0
1	1/4/2000	122.959999	122.959999	114.400002	117.220001	69.763824	377688.0
2	1/5/2000	117.919998	117.919998	109.199997	114.804001	68.325935	422488.0
3	1/6/2000	114.720001	114.800003	107.919998	108.468002	64.555061	413538.0
4	1/7/2000	109.599998	111.192001	99.792000	99.804001	59.398636	831700.0

```
df.isnull().sum()
```

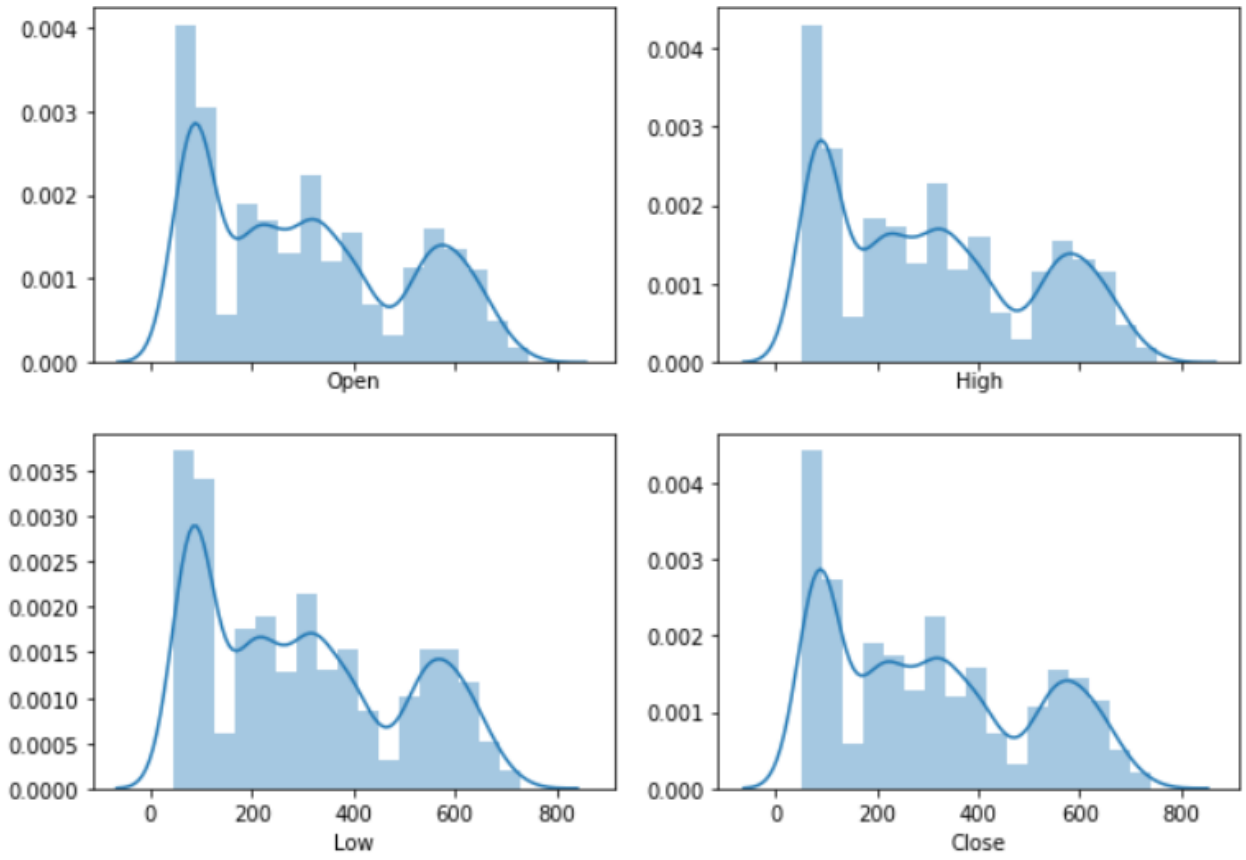
```
Date          0
Open         61
High         61
Low          61
Close        61
Adj Close    61
Volume       61
dtype: int64
```

```
df = df.fillna(method ='pad') # filling the missing values with previous ones
df.isnull().sum()
```

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

```
print('There are {} number of days in the dataset.'.format(df.shape[0]))
```

```
There are 4967 number of days in the dataset.
```




```
dataset.describe()  # overall statistics
```

	Open	High	Low	Close
count	4967.000000	4967.000000	4967.000000	4967.000000
mean	306.183467	310.184928	301.469348	305.608403
std	195.187484	197.185347	192.772437	194.813300
min	48.080002	50.959999	44.891998	50.180000
25%	106.740002	108.320000	104.490002	106.250000
50%	283.500000	287.000000	277.049988	282.200012
75%	468.000000	472.500000	460.800003	467.324997
max	744.950012	752.849976	730.250000	739.599976

I have created 2 copies of data (tek_ind_1 & tek_ind_2) to add columns of different sets of Technical Indicators.

```
import copy
```

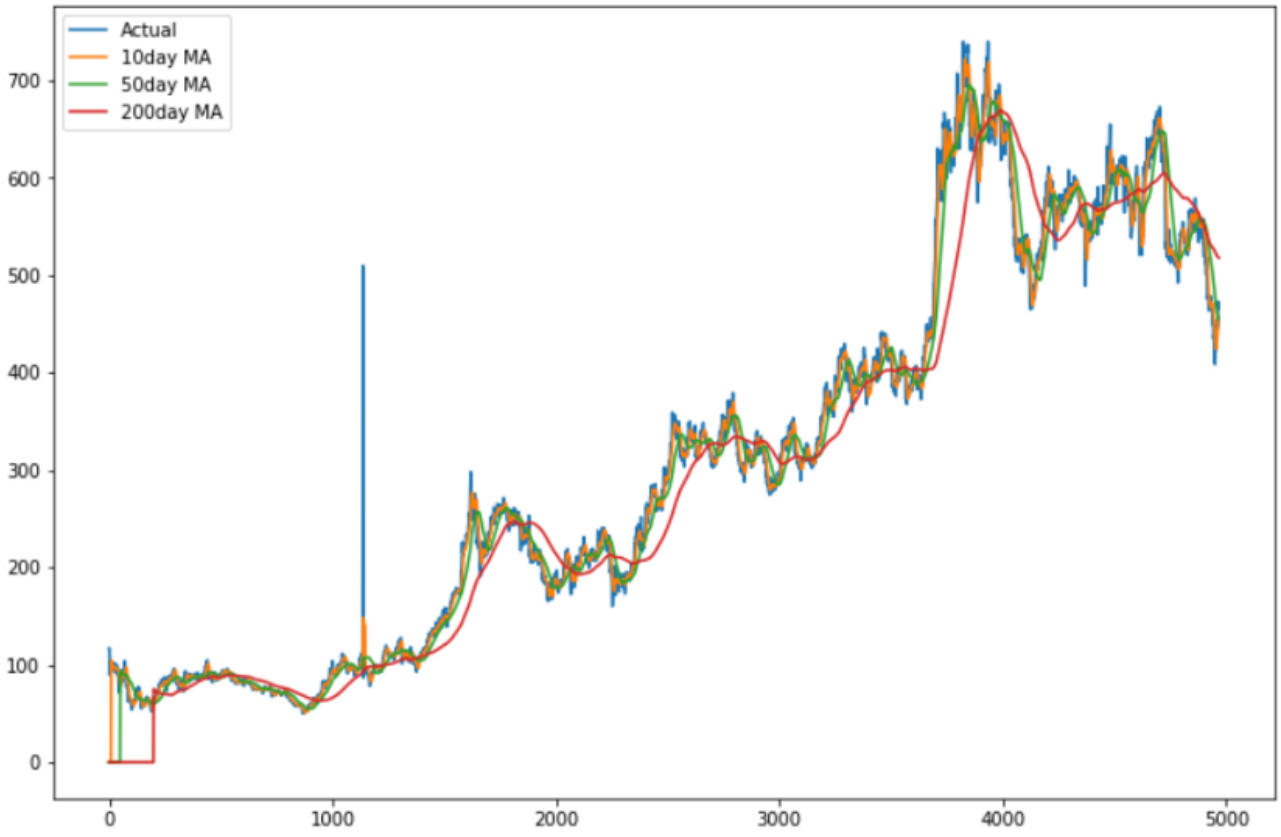
```
data = dataset
tek_ind_1 = copy.deepcopy(data)
tek_ind_2 = copy.deepcopy(data)
```

tek_ind_1 dataframe

```
tek_ind_1['daily_return'] = tek_ind_1.Close.pct_change().fillna(0)
tek_ind_1['cum_daily_return'] = (1 + tek_ind_1['daily_return']).cumprod()

tek_ind_1['H-L'] = tek_ind_1.High - dataset.Low

tek_ind_1 ['C-O'] = tek_ind_1.Close - tek_ind_1.Open

tek_ind_1['10day MA'] = tek_ind_1.Close.shift(1).rolling(window = 10).mean().fillna(0)
tek_ind_1['50day MA'] = tek_ind_1.Close.shift(1).rolling(window = 50).mean().fillna(0)
tek_ind_1['200day MA'] = tek_ind_1.Close.shift(1).rolling(window = 200).mean().fillna(0)

tek_ind_1['rsi'] = talib.RSI(tek_ind_1.Close.values, timeperiod = 14)

tek_ind_1['Williams %R'] = talib.WILLR(tek_ind_1.High.values,
                                       tek_ind_1.Low.values,
                                       tek_ind_1.Close.values, 14)

# Create 7 and 21 days Moving Average
tek_ind_1['ma7'] = tek_ind_1.Close.rolling(window=7).mean().fillna(0)
tek_ind_1['ma21'] = tek_ind_1.Close.rolling(window=21).mean().fillna(0)

# Creating MACD
tek_ind_1['ema_26'] = tek_ind_1.Close.ewm(span=26).mean().fillna(0)
tek_ind_1['ema_12'] = tek_ind_1.Close.ewm(span=12).mean().fillna(0)
tek_ind_1['macd'] = (tek_ind_1['ema_12'] - tek_ind_1['ema_26'])
```

```
# Creating Bollinger Bands
#Set number of days and standard deviations to use for rolling lookback period for Bollinger band calculation
window = 21
no_of_std = 2
#Calculate rolling mean and standard deviation using number of days set above
rolling_mean = tek_ind_1.Close.rolling(window).mean()
rolling_std = tek_ind_1.Close.rolling(window).std()
#create two new DataFrame columns to hold values of upper and lower Bollinger bands
#B['Rolling Mean'] = rolling_mean.fillna(0)
tek_ind_1['bb_high'] = (rolling_mean + (rolling_std * no_of_std)).fillna(0)
tek_ind_1['bb_low'] = (rolling_mean - (rolling_std * no_of_std)).fillna(0)

# Create Exponential moving average
tek_ind_1['ema'] = tek_ind_1.Close.ewm(com=0.5).mean()

# Create Momentum
tek_ind_1['momentum'] = tek_ind_1.Close - 1

tek_ind_1.head(20)
```

```
plt.figure(figsize=(12, 8))
plt.plot(tek_ind_1['Close'], label='Actual')
plt.plot(tek_ind_1['10day MA'], label='10day MA')
plt.plot(tek_ind_1['50day MA'], label='50day MA')
plt.plot(tek_ind_1['200day MA'], label='200day MA')

plt.legend(loc='best')
```




```
tek_ind_1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4967 entries, 0 to 4966
Data columns (total 22 columns):
Open                 4967 non-null float64
High                 4967 non-null float64
Low                  4967 non-null float64
Close                4967 non-null float64
daily_return         4967 non-null float64
cum_daily_return     4967 non-null float64
H-L                  4967 non-null float64
C-O                  4967 non-null float64
10day MA             4967 non-null float64
```

```
50day MA          4967 non-null float64
200day MA         4967 non-null float64
rsi               4953 non-null float64
Williams %R       4954 non-null float64
ma7               4967 non-null float64
ma21              4967 non-null float64
ema_26            4967 non-null float64
ema_12            4967 non-null float64
macd              4967 non-null float64
bb_high           4967 non-null float64
bb_low            4967 non-null float64
ema               4967 non-null float64
momentum          4967 non-null float64
dtypes: float64(22)
memory usage: 853.8 KB
```

tek_ind_2 dataframe

Calculation of Stochastic Oscillator (%K and %D)

```
def stok(df, n):
    tek_ind_2['stok'] = ((tek_ind_2['Close'] - tek_ind_2['Low'].rolling(window=n, center=False).mean()) /
                  (tek_ind_2['High'].rolling(window=n, center=False).max() -
                   tek_ind_2['Low'].rolling(window=n, center=False).min())) * 100
    tek_ind_2['stod'] = tek_ind_2['stok'].rolling(window = 3, center=False).mean()

stok(tek_ind_2, 4)
tek_ind_2 = tek_ind_2.fillna(0)
tek_ind_2.tail()
```

	Open	High	Low	Close	stok	stod
4962	447.100006	461.450012	441.750000	459.850006	77.411786	55.441863
4963	463.899994	478.399994	462.000000	473.049988	66.029996	55.108553
4964	478.950012	478.950012	462.250000	465.000000	33.904567	59.115450
4965	465.000000	472.700012	460.149994	466.850006	27.721785	42.552116
4966	467.299988	474.200012	462.399994	469.200012	39.893659	33.840004

CCI = (typical price − ma) / (0.015 * mean deviation)

- typical price = (high + low + close) / 3
- p = number of periods (20 commonly used)
- ma = moving average
- moving average = typical price / p

- mean deviation = (typical price — MA) / p

Calculations of Ichimoku Cloud

- Turning Line = (Highest High + Lowest Low) / 2, for the past 9 days
- Standard Line = (Highest High + Lowest Low) / 2, for the past 26 days
- Leading Span 1 = (Standard Line + Turning Line) / 2, plotted 26 days ahead of today
- Leading Span 2 = (Highest High + Lowest Low) / 2, for the past 52 days, plotted 26 days ahead of today
- Cloud = Shaded Area between Span 1 and Span 2

```
#Calculation of Price Rate of Change
# ROC = [(Close - Close n periods ago) / (Close n periods ago)] * 100

tek_ind_2['ROC'] = ((tek_ind_2['Close'] - tek_ind_2['Close'].shift(12)) /
                    (tek_ind_2['Close'].shift(12)))*100
tek_ind_2 = tek_ind_2.fillna(0)

#Calculation of Momentum
tek_ind_2['Momentum'] = tek_ind_2['Close'] - tek_ind_2['Close'].shift(4)
tek_ind_2 = tek_ind_2.fillna(0)

#Calculation of Commodity Channel Index
tp = (tek_ind_2['High'] + tek_ind_2['Low'] + tek_ind_2['Close']) / 3
ma = tp / 20
md = (tp - ma) / 20
tek_ind_2['CCI'] = (tp-ma)/(0.015 * md)

# Calculation of Triple Exponential Moving Average
# Triple Exponential MA Formula:
# T-EMA = (3EMA − 3EMA(EMA)) + EMA(EMA(EMA))
# Where:
# EMA = EMA(1) + α * (Close − EMA(1))
# α = 2 / (N + 1)
# N = The smoothing period.

tek_ind_2['ema'] = tek_ind_2['Close'].ewm(span=3,min_periods=0,adjust=True,ignore_na=False).mean()
tek_ind_2 = tek_ind_2.fillna(0)

tek_ind_2['tema'] = (3 * tek_ind_2['ema'] - 3 * tek_ind_2['ema'] * tek_ind_2['ema']) + (tek_ind_2['ema'] *
                                                                                        tek_ind_2['ema'] *
                                                                                        tek_ind_2['ema'])
```

```
# Turning Line
high = tek_ind_2['High'].rolling(window=9,center=False).max()
low = tek_ind_2['Low'].rolling(window=9,center=False).min()
tek_ind_2['turning_line'] = (high + low) / 2
```

```
# Standard Line
p26_high = tek_ind_2['High'].rolling(window=26,center=False).max()
p26_low = tek_ind_2['Low'].rolling(window=26,center=False).min()
tek_ind_2['standard_line'] = (p26_high + p26_low) / 2

# Leading Span 1
tek_ind_2['ichimoku_span1'] = ((tek_ind_2['turning_line'] + tek_ind_2['standard_line']) / 2).shift(26)

# Leading Span 1
tek_ind_2['ichimoku_span1'] = ((tek_ind_2['turning_line'] + tek_ind_2['standard_line']) / 2).shift(26)

# Leading Span 2
p52_high = tek_ind_2['High'].rolling(window=52,center=False).max()
p52_low = tek_ind_2['Low'].rolling(window=52,center=False).min()
tek_ind_2['ichimoku_span2'] = ((p52_high + p52_low) / 2).shift(26)

# The most current closing price plotted 22 time periods behind (optional)
tek_ind_2['chikou_span'] = tek_ind_2['Close'].shift(-22) # 22 according to investopedia
```

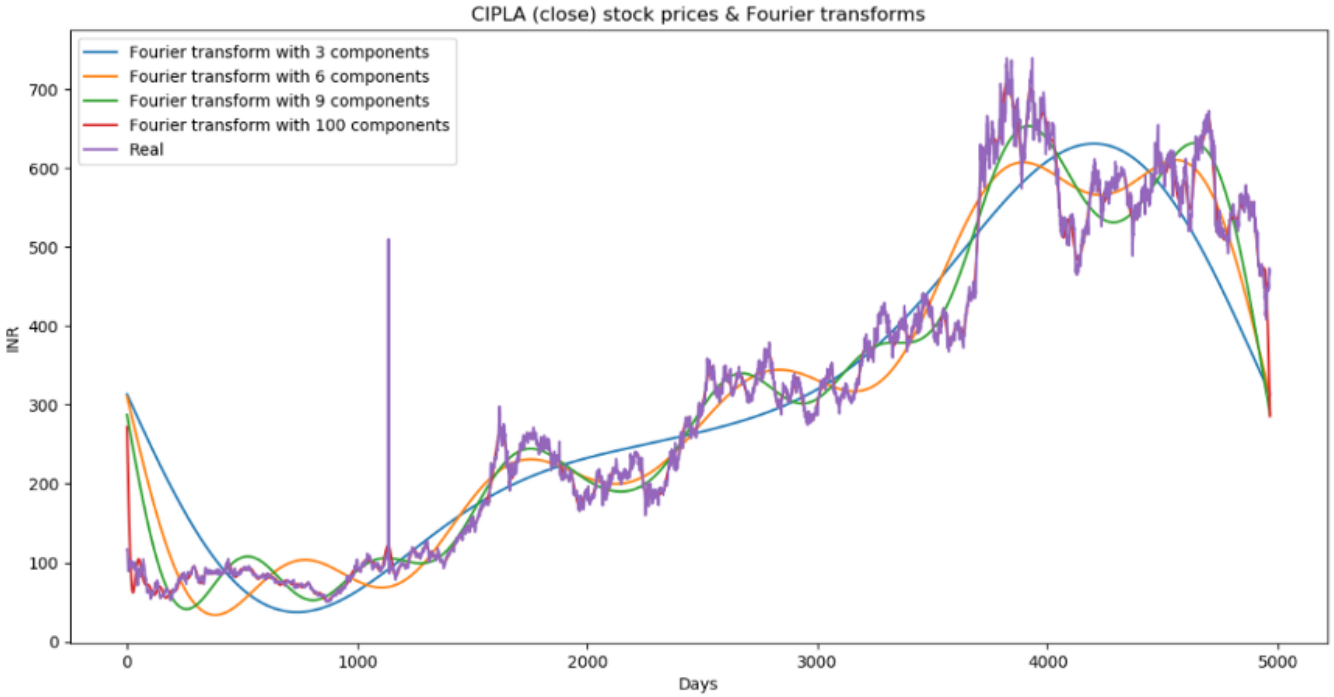
Fourier transformation

For data that is known to have seasonal, or daily patterns we'd like to use Fourier analysis to make predictions. It's like a combination of extrapolation and de-noising. We want to repeat the observed data over multiple periods. and also want to find patterns by finding the dominant frequency components in the observed data. We will compute the Fourier transformation and the spectral density of the signal.

- The first step is to compute the FFT of the signal using the fft() function
- Once the FFT has been obtained, we need to take the square of its absolute value in order to get the power spectral density (PSD)

```
close_fft = np.fft.fft(np.asarray(dataset.Close.tolist()))
fft_df = pd.DataFrame({'fft':close_fft})
fft_df['absolute'] = fft_df['fft'].apply(lambda x: np.abs(x))
fft_df['angle'] = fft_df['fft'].apply(lambda x: np.angle(x))
```

```
plt.figure(figsize=(14, 7), dpi=100)
fft_list = np.asarray(fft_df['fft'].tolist())
for num_ in [3, 6, 9, 100]:
    fft_list_m10= np.copy(fft_list); fft_list_m10[num_:-num_]=0
    plt.plot(np.fft.ifft(fft_list_m10), label='Fourier transform with {} components'.format(num_))
plt.plot(tek_ind_2.Close,  label='Real')
plt.xlabel('Days')
plt.ylabel('INR')
plt.title('CIPLA (close) stock prices & Fourier transforms')
plt.legend()
plt.show()
```



If we compare with 10,50,200 day MA plot, we see here that Fourier transformation smoothing the data better by de-noising.

```
tek_ind_2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4967 entries, 0 to 4966
Data columns (total 18 columns):
Open              4967 non-null float64
High              4967 non-null float64
Low               4967 non-null float64
Close             4967 non-null float64
stok              4967 non-null float64
stod              4967 non-null float64
ROC               4967 non-null float64
Momentum          4967 non-null float64
CCI               4967 non-null float64
ema               4967 non-null float64
tema              4967 non-null float64
turning_line      4959 non-null float64
standard_line     4942 non-null float64
ichimoku_span1    4916 non-null float64
ichimoku_span2    4890 non-null float64
chikou_span       4945 non-null float64
absolute          4967 non-null float64
angle             4967 non-null float64
dtypes: float64(18)
```

```
dtypes: float64(18)
memory usage: 698.6 KB
```

Data preparation for RNN

```
tek_ind_1.columns
```

```
Index(['Open', 'High', 'Low', 'Close', 'daily_return', 'cum_daily_return',
       'H-L', 'C-O', '10day MA', '50day MA', '200day MA', 'rsi', 'Williams %R',
       'ma7', 'ma21', 'ema_26', 'ema_12', 'macd', 'bb_high', 'bb_low', 'ema',
       'momentum'],
      dtype='object')
```

```
a = copy.deepcopy(tek_ind_1)
b = copy.deepcopy(tek_ind_2)
```

```
print('Total dataset has {} samples, and {} features.'.format(a.shape[0], a.shape[1]))
```

```
Total dataset has 4967 samples, and 22 features.
```

```
a.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4967 entries, 0 to 4966
Data columns (total 22 columns):
Open                4967 non-null float64
High                4967 non-null float64
Low                 4967 non-null float64
Close               4967 non-null float64
daily_return        4967 non-null float64
cum_daily_return    4967 non-null float64
H-L                 4967 non-null float64
C-O                 4967 non-null float64
10day MA            4967 non-null float64
50day MA            4967 non-null float64
200day MA           4967 non-null float64
rsi                 4953 non-null float64
Williams %R         4954 non-null float64
ma7                 4967 non-null float64
ma21                4967 non-null float64
ema_26              4967 non-null float64
ema_12              4967 non-null float64
macd                4967 non-null float64
bb_high             4967 non-null float64
bb_low              4967 non-null float64
ema                 4967 non-null float64
momentum            4967 non-null float64
dtypes: float64(22)
memory usage: 853.8 KB
```

```
a = a.fillna(0) # removing NaN from columns
a.info()
```

```
a.shape
```

```
(4967, 22)
```

```
values = a.values
# ensure all data is float
values = values.astype('float32')
values
```

```
array([[112.8    , 116.588  , 110.404  , ...,   0.     , 116.588  ,
        115.588  ],
       [122.96   , 122.96   , 114.4    , ...,   0.     , 117.062  ,
        116.22   ],
       [117.92   , 117.92   , 109.2    , ...,   0.     , 115.49877,
        113.804  ],
       ...,
       [478.95   , 478.95   , 462.25   , ..., 404.06226, 465.81265,
        464.     ],
       [465.     , 472.7    , 460.15   , ..., 403.42627, 466.5042 ,
        465.85   ],
       [467.3    , 474.2    , 462.4    , ..., 402.82706, 468.30142,
        468.2    ]], dtype=float32)
```

```
print("Min:", np.min(values))
print("Max:", np.max(values))
```

```
Min: -100.0
Max: 756.7664
```

```
values =pd.DataFrame(values)
```

We see that data points varies from -100 to 756.77; we need to scale and standardize the data for RNN network. This will be done once we split the train/test data set.

Now, let's define a function to create the time series data set. I have specified a look back interval (60 time steps) and the predicted column. RNN works based on time steps. If we are making 60 time steps which means that, for making future forecast our RNN will

observe previous 60 time steps and every time it will predict the output, it will check previous 60 time steps. So, we need to create the data structure accordingly.

```
def ts (a, look_back = 60, pred_col = 4):
    t = a.copy()
    t["id"] = range(1, len(t)+1)
    t = t.iloc[:-look_back, :]
    t.set_index('id', inplace =True)
    pred_value = a.copy()
    pred_value = pred_value.iloc[look_back:, pred_col]
    pred_value.columns = ["Pred"]
    pred_value = pd.DataFrame(pred_value)

    pred_value["id"] = range(1, len(pred_value)+1)
    pred_value.set_index('id', inplace = True)
    final_df= pd.concat([t, pred_value], axis=1)

    return final_df
```

```
arr_df = ts(values, 60,4)
arr_df.fillna(0, inplace=True)

arr_df.columns = ['v1(t-60)', 'v2(t-60)', 'v3(t-60)', 'v4(t-60)',
                  'v5(t-60)', 'v6(t-60)', 'v7(t-60)', 'v8(t-60)',
                  'v9(t-60)', 'v10(t-60)', 'v11(t-60)', 'v12(t-60)',
                  'v13(t-60)', 'v14(t-60)', 'v15(t-60)', 'v16(t-60)',
                  'v17(t-60)', 'v18(t-60)', 'v19(t-60)', 'v20(t-60)',
                  'v21(t-60)', 'v22(t-60)', 'v1(t)']
print(arr_df.head(4))
```

```
     v1(t-60)    v2(t-60)    v3(t-60)  ...   v21(t-60)   v22(t-60)     v1(t)
id                                     ...
1   112.800003  116.587997  110.403999  ...  116.587997  115.587997 -0.048071
2   122.959999  122.959999  114.400002  ...  117.061996  116.220001 -0.034573
3   117.919998  117.919998  109.199997  ...  115.498772  113.804001  0.020241
4   114.720001  114.800003  107.919998  ...  110.752998  107.468002  0.002289

[4 rows x 23 columns]
```

Above is first 4 rows of the transformed data set. We can see the 22 input variables (input series) and the 1 output variable (Close Price). If we check the new shape of data, we can see the difference of 60 time steps.

```
print(arr_df.describe())
```

```
          v1(t-60)     v2(t-60)  ...    v22(t-60)        v1(t)
count  4907.000000  4907.000000  ...  4907.000000  4907.000000
mean    304.257782   308.236847  ...   302.685883     0.001152
std     195.572800   197.573212  ...   195.198288     0.059529
min      48.080002    50.959999  ...    49.180000    -0.795261
25%     105.320000   107.199997  ...   103.868000    -0.009259
50%     276.000000   282.200012  ...   273.660004     0.000000
75%     445.425003   450.599991  ...   443.574997     0.010135
max     744.950012   752.849976  ...   738.599976     3.861170

[8 rows x 23 columns]
```

```
arr_df.shape
```

```
(4907, 23)
```

Here we split the data-set into train and test sets. Then splits the train and test sets into input and output variables. Finally, the inputs are reshaped into the 3D format expected by LSTMs, namely *[samples, time-steps, features]*. We will fit the model on 90% of the data, then evaluate it on the remaining data.

```
# split into train and test sets
val = arr_df.values
train_sample =int(len(a)*0.8)
train = val[: train_sample, :]
test = val[train_sample:, :]

print(train.shape, test.shape)
```

```
(3973, 23) (934, 23)
```

```
train = pd.DataFrame(train)
test = pd.DataFrame(test)
```

Train/test split

Sequence of values are crucial with time-series data. So, we have split the data in train and test test using a systematic approach as below. Also, normalization is required to fit the data for neural network architecture learning.

```
X, y = train, test
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X= scaler.fit_transform(X)
print(X)
print('\n')
print(X.shape)
```

```
[[0.09287242 0.09350183 0.09558799 ... 0.09640107 0.09632445 0.06072869]
 [0.10745189 0.10258018 0.10141853 ... 0.09709395 0.09724117 0.05039677]
 [0.10021956 0.09539957 0.09383125 ... 0.09480888 0.09373678 0.05579232]
 ...
 [0.80060846 0.83337855 0.8128715  ... 0.8351234  0.8440718  0.8192633 ]
 [0.8364831  0.82639736 0.8230852  ... 0.82444906 0.8134664  0.8120883 ]
 [0.8120166  0.80210584 0.8129445  ... 0.81777245 0.80882484 0.80993587]]


(4470, 23)
```

Shaping data for LSTM

Here I have taken a look back period of 60 days where the model will look through last 60 time steps to predict future price.

Shaping train data

```
# shaping data from neural network
X_train = []
y_train = []
for i in range(60, X.shape[0]):
    X_train.append(X[i-60:i])
    y_train.append(X[i,0])
    if i <= 61:
        print(X_train)
        print('\n')
        print(y_train)
        print()
```

```
[array([[0.09287242, 0.09350183, 0.09558799, ..., 0.09640107, 0.09632445,
        0.06072869],
```

```
        [0.10745189, 0.10258018, 0.10141853, ..., 0.09709395, 0.09724117,
         0.05039677],
        [0.10021956, 0.09539957, 0.09383125, ..., 0.09480888, 0.09373678,
         0.05579232],
        ...,
        [0.06313948, 0.06245994, 0.06523306, ..., 0.06037507, 0.0627368 ,
         0.0242226 ],
        [0.06359867, 0.06189004, 0.06534978, ..., 0.06161238, 0.06241768,
         0.02054903],
        [0.06075165, 0.06063629, 0.06126433, ..., 0.05836066, 0.05696384,
         0.01951583]], dtype=float32)]


[0.06072869]

[array([[0.09287242, 0.09350183, 0.09558799, ..., 0.09640107, 0.09632445,
         0.06072869],
        [0.10745189, 0.10258018, 0.10141853, ..., 0.09709395, 0.09724117,
         0.05039677],
        [0.10021956, 0.09539957, 0.09383125, ..., 0.09480888, 0.09373678,
         0.05579232],
        ...,
```

```
1  import numpy as np
2  X_train, y_train = np.array(X_train), np.array(y_train)
3  print(X_train.shape, y_train.shape)
```

```
(3913, 60, 23) (3913,)
```

LSTM model

We now create the LSTM model with 3 LSTM layers and one Dense layer. I have defined the LSTM with 75 neurons in the first hidden layer, 30 neuron in the 2nd and 3rd layers; 1 neuron in the output layer for predicting price. The input shape will be 60 time step with 22 features.

The model is compiled with Mean Absolute Error (MAE) loss function and the efficient Adam version of stochastic gradient descent.

```
1  #create model
2  model_lstm = tf.keras.Sequential()
3  model_lstm.add(tf.keras.layers.LSTM(units = 75, return_sequences = True,
4                                      input_shape = (X_train.shape[1], X_train.shape[2])))
5  model_lstm.add(tf.keras.layers.LSTM(units = 30,return_sequences = True))
```

```
model_lstm.add(tf.keras.layers.LSTM(units = 30, return_sequences = True))

model_lstm.add(tf.keras.layers.Dense(units = 1))
model_lstm.compile(loss='mae', optimizer='adam')
model_lstm.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 60, 75)            29700
_________________________________________________________________
lstm_1 (LSTM)                (None, 60, 30)            12720
_________________________________________________________________
lstm_2 (LSTM)                (None, 60, 30)            7320
_________________________________________________________________
dense (Dense)                (None, 60, 1)             31
=================================================================
Total params: 49,771
Trainable params: 49,771
Non-trainable params: 0
_________________________________________________________________
```

Prepare test data

Once the model is fit on training data, we want to see the skill of the model on out-of-sample data. Likewise we did in training set, we will prepare the test set and we need to pick up look-back value from training data. The look-back is the number of previous time steps to use as input variables to predict the next time period *(60 in this case)*.

```
look_back = train.tail(60)
data = look_back.append(test)
print(data)
```

```
inputs = scaler.transform(data)
inputs
```

```
# shaping data from neural network
X_test = []
y_test = []
for i in range(60, inputs.shape[0]):
    X_test.append(inputs[i-60:i])
    y_test.append(inputs[i,0])
    if i <= 61:
        print(X_test)
```

```
print('\n')
print(y_test)
print()
```

```
[array([[0.75038385, 0.7414838 , 0.75509155, ..., 0.7532024 , 0.7458734 ,
        0.8106533 ],
       [0.74134344, 0.75010335, 0.7566965 , ..., 0.7624983 , 0.7619013 ,
        0.792429  ],
       [0.75425833, 0.7528816 , 0.75888515, ..., 0.7590191 , 0.7521105 ,
        0.7905635 ],
       ...,
       [0.80060846, 0.83337855, 0.8128715 , ..., 0.8351234 , 0.8440718 ,
        0.8192633 ],
       [0.8364831 , 0.82639736, 0.8230852 , ..., 0.82444906, 0.8134664 ,
        0.8120883 ],
       [0.8120166 , 0.80210584, 0.8129445 , ..., 0.81777245, 0.80882484,
        0.80993587]], dtype=float32)]


[0.8106533]

[array([[0.75038385, 0.7414838 , 0.75509155, ..., 0.7532024 , 0.7458734 ,
        0.8106533 ],
       [0.74134344, 0.75010335, 0.7566965 , ..., 0.7624983 , 0.7619013 ,
        0.792429  ],
       [0.75425833, 0.7528816 , 0.75888515, ..., 0.7590191 , 0.7521105 ,
        0.7905635 ]
```

```
X_test, y_test = np.array(X_test), np.array(y_test)
print(X_test.shape, y_test.shape)
```

```
(934, 60, 23) (934,)
```

The model is fitted with 20 training epochs with a batch size of 32.

```
# fit network
history_lstm = model_lstm.fit(X_train, y_train,
                              epochs = 20,
                              batch_size = 32,
                              validation_data = (X_test, y_test),
                              shuffle=False)
```

```
Epoch 1/20
123/123 [==============================] - 2s 18ms/step - loss: 0.0609 - val_loss: 0.2502
Epoch 2/20
123/123 [==============================] - 1s 11ms/step - loss: 0.1018 - val_loss: 0.2340
```

```
Epoch 3/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0839 - val_loss: 0.2047
Epoch 4/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0704 - val_loss: 0.1512
Epoch 5/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0600 - val_loss: 0.1688
Epoch 6/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0590 - val_loss: 0.0569
Epoch 7/20
123/123 [==============================] - 1s 12ms/step - loss: 0.0463 - val_loss: 0.1532
Epoch 8/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0555 - val_loss: 0.1215
Epoch 9/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0463 - val_loss: 0.1336
Epoch 10/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0574 - val_loss: 0.1243
Epoch 11/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0465 - val_loss: 0.1282
Epoch 12/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0498 - val_loss: 0.1144
Epoch 13/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0413 - val_loss: 0.1246
Epoch 14/20
123/123 [==============================] - 1s 11ms/step - loss: 0.0425 - val_loss: 0.1192
Epoch 15/20
```



```
123/123 [==============================] - 1s 11ms/step - loss: 0.0402 - val_loss: 0.1215
```

GRU

GRU network is created with same parameters and configuration as LSTM.

```
model_gru = tf.keras.Sequential()
model_gru.add(tf.keras.layers.GRU(75, return_sequences = True, input_shape=(X_train.shape[1], X_train.shape[2])))
model_gru.add(tf.keras.layers.GRU(units=30, return_sequences=True))
model_gru.add(tf.keras.layers.GRU(units=30))
model_gru.add(tf.keras.layers.Dense(units=1))

model_gru.compile(loss='mae', optimizer='adam')
model_gru.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 60, 75)            22500
_________________________________________________________________
gru_1 (GRU)                  (None, 60, 30)            9630
_________________________________________________________________
gru_2 (GRU)                  (None, 30)                5580
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 31
=================================================================
Total params: 37,741
Trainable params: 37,741
Non-trainable params: 0
_________________________________________________________________
```

```
# fit network
gru_history = model_gru.fit(X_train, y_train, epochs = 20, batch_size = 64,
                            validation_data = (X_test, y_test), shuffle=False)
```

```
Epoch 1/20
62/62 [==============================] - 2s 25ms/step - loss: 0.0566 - val_loss: 0.1555
Epoch 2/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0891 - val_loss: 0.1611
Epoch 3/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0515 - val_loss: 0.0218
Epoch 4/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0326 - val_loss: 0.1014
Epoch 5/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0231 - val_loss: 0.0145
Epoch 6/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0255 - val_loss: 0.1160
Epoch 7/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0280 - val_loss: 0.0254
Epoch 8/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0116 - val_loss: 0.1269
Epoch 9/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0235 - val_loss: 0.0340
Epoch 10/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0193 - val_loss: 0.0933
Epoch 11/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0218 - val_loss: 0.0752
Epoch 12/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0176 - val_loss: 0.1015
Epoch 13/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0422 - val_loss: 0.0648
Epoch 14/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0286 - val_loss: 0.0812
Epoch 15/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0428 - val_loss: 0.0164
Epoch 16/20
62/62 [==============================] - 1s 14ms/step - loss: 0.0275 - val_loss: 0.0757
Epoch 17/20
```

Here, both LSTM and GRU had the same architecture but the number of parameters in LSTM is 49, 771 whereas GRU in GRU is 37,741.

```
=================================================================
Total params: 49,771
Trainable params: 49,771
Non-trainable params: 0
_________________________________________________________________
```

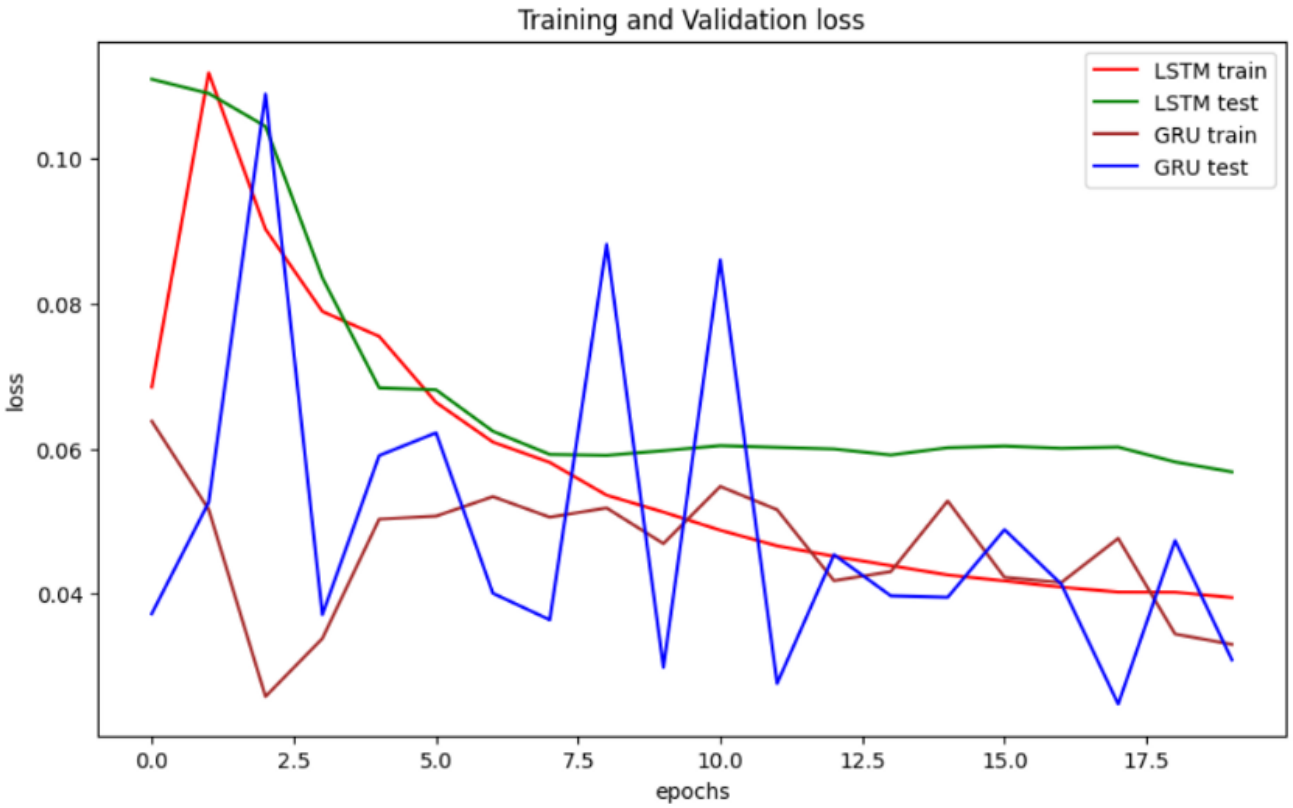
```
Total params: 37,741
Trainable params: 37,741
Non-trainable params: 0
```

As we already discussed the differences, GRU with two gates compared to LSTM that has three gates. GRU has fewer parameters it is computationally more efficient than LSTM.

Diagnostic Plots

The training history of both LSTM & GRU models are used to diagnose the behavior of the models. I have created a single plot for the ease of convenience.

```
plt.figure(figsize=(10, 6), dpi=100)
plt.plot(history_lstm.history['loss'], label='LSTM train', color='red')
plt.plot(history_lstm.history['val_loss'], label='LSTM test', color= 'green')
plt.plot(gru_history.history['loss'], label='GRU train', color='brown')
plt.plot(gru_history.history['val_loss'], label='GRU test', color='blue')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
plt.title('Training and Validation loss')
plt.show()
```



We can see that, in both cases the training losses is considerably lower than validation losses indicating models may have under-fitting the training data. The performance may

be improved by increasing the epochs which can be experimented.

LSTM Prediction

This gives us the scale down values of prediction; we inverse the scale and need to bring these values to normal scale. Let us find out the scaling level of our data.

```
scaler.scale_
```

```
array([0.00143499, 0.00142472, 0.00145909, 0.00145049, 0.21475677,
       0.16911027, 0.01628664, 0.00942507, 0.00138611, 0.00143893,
       0.00149408, 0.01023407, 0.01      , 0.00137255, 0.0014019 ,
       0.00152457, 0.00150257, 0.0141036 , 0.00132141, 0.00133131,
       0.00146176, 0.00145049], dtype=float32)
```

Here, the 1st value is the Open price; to bring this to normal scale, we divide this value by 1 as shown below.

```
normal_scale = 1/0.00143499
normal_scale
```

```
696.8689677279981
```

```
y_pred = y_pred * normal_scale
```

```
y_test = y_test * normal_scale
```

Evaluate Model

Now we can forecast for the entire test data-set combining the forecast with the test data-set and invert the scaling. ith forecasts and actual values in their original scale, we can calculate an error score (Root Mean Squared Error — RMSE) for the model. RMSE gives error in the same units as the variable itself.

The accuracy can be calculated by taking the average of observed values and predicted values and dividing them with each other.

```
mean_y_test = y_test.mean()
mean_y_pred = y_pred.mean()
print(mean_y_test, mean_y_pred)
```

```
524.4402 599.86346
```

```
accuracy = round((mean_y_test / mean_y_pred)*100,2)
accuracy
```

```
87.43
```

We can perform the similar exercise on GRU model.

Moreover, performance of the models can be improved by adding more time variants e.g. adding Weeks (day of weeks, week no.), Months (month no.) and Year Columns, time lags etc. and also adjusting the hyper-parameters and epochs. There are several types of motivation and data analysis available for time series which are appropriate for different purposes and etc.

Conclusion

The key difference between a GRU and an LSTM is that a GRU has two gates (*reset* and *update* gates) whereas an LSTM has three gates (namely *input*, *output* and *forget* gates). GRU network is simpler and thus easier to modify, for example adding new gates in case of additional input to the network. It's just less code in general. However, if sequence is large or accuracy is very critical, I would recommend LSTM

I can be connected here.

Machine Learning | Recurrent Neural Network | Technical Indicator | Predictive Analytics