Neural Network Predictions before and After stationarizing time series data.

April 28, 2018

0.1 Working with a Non Stationary data:

0.2 Data set: Light weight truck sales data.

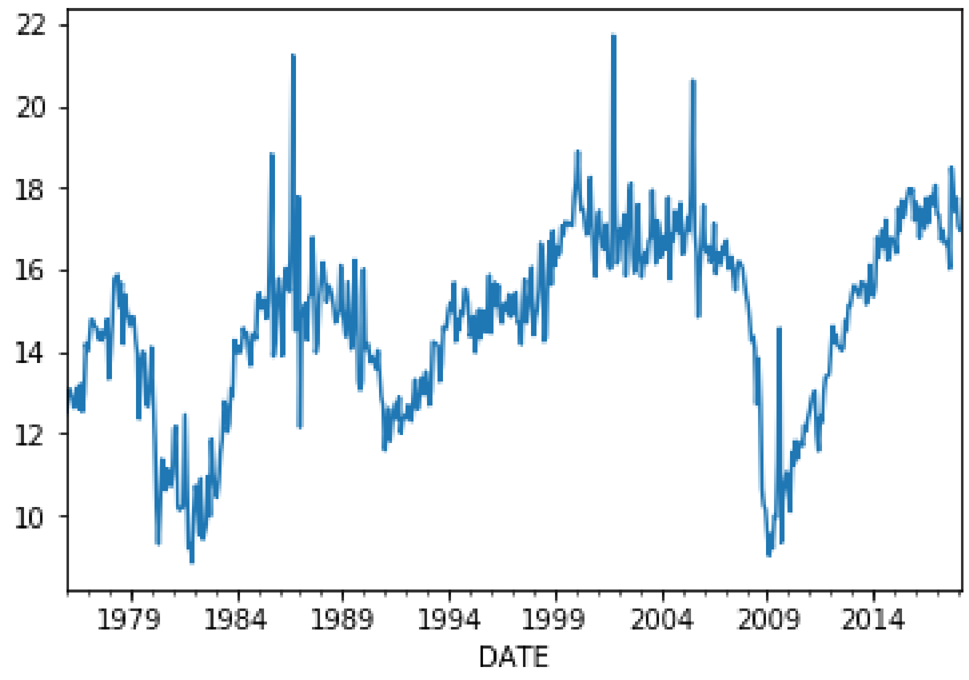
```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        from pandas import Series
```

```
In [19]: import os
         os.chdir('/Users/revanthkota/downloads')
```

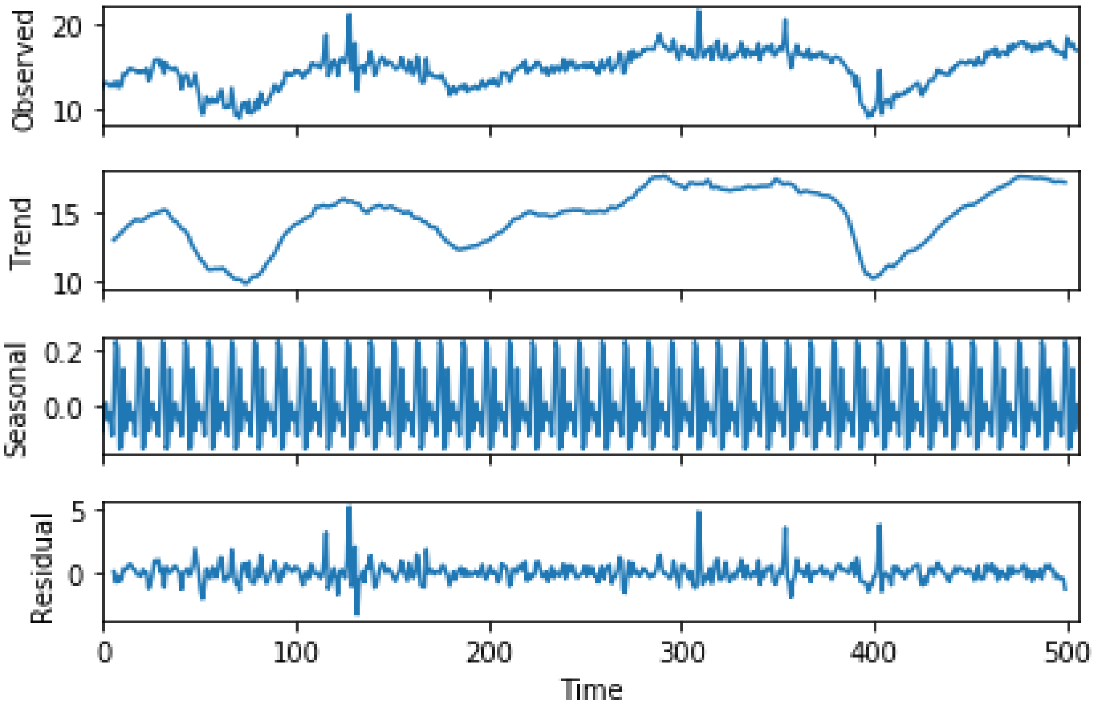
```
In [20]: series1 = Series.from_csv('ALTSALES.csv', header=0)
```

0.2.1 ALTSALES.csv : Light weight trucks sales data.

```
In [21]: series1.plot()
         plt.show()
```



```
In [105]: import statsmodels.api as sm
          from statsmodels.tsa.stattools import adfuller

In [106]: decomposed = sm.tsa.seasonal_decompose(series1.values, model='additive' ,freq=12)

In [107]: decomposed.plot()
          plt.show()
```

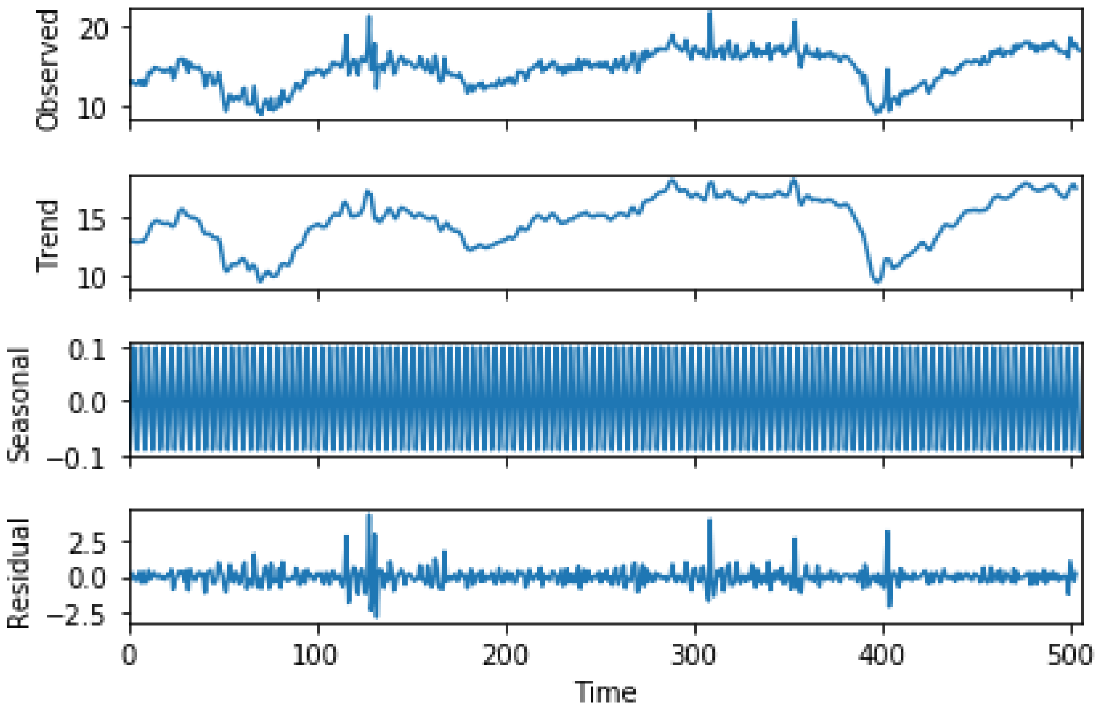


0.2.2 By decomposing time series, we can see that there is no standard trend in data

0.2.3 As sales increase and decrease over the time

0.2.4 We can see that there is seasonality

```
In [110]: decomposed = sm.tsa.seasonal_decompose(series1.values, model='additive' ,freq=4)

In [111]: decomposed.plot()
          plt.show()
```



```
In [112]: # Dickey fuller test:

          x = series1.values
          result = adfuller(x)
          print('ADF Statistic: %f' % result[0])
          print('p-value: %f' % result[1])
          print('Critical Values:')
          for key, value in result[4].items():
                  print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -2.034039
p-value: 0.271850
Critical Values:
        1%: -3.443
        5%: -2.867
        10%: -2.570
```

0.2.5 The above time series is Non stationary as P values is greater than 0.05

0.2.6 Performed seasonal differencing of 1st Order in an attempt to stationarize the data

```
In [113]: seasonal_difference = series1 - series1.shift(12)
In [114]: # Dickey fuller test:

          y = seasonal_difference.dropna()
          result = adfuller(y)
          print('ADF Statistic: %f' % result[0])
          print('p-value: %f' % result[1])
          print('Critical Values:')
          for key, value in result[4].items():
                  print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -3.912829
p-value: 0.001942
Critical Values:
        1%: -3.444
        5%: -2.868
        10%: -2.570
```

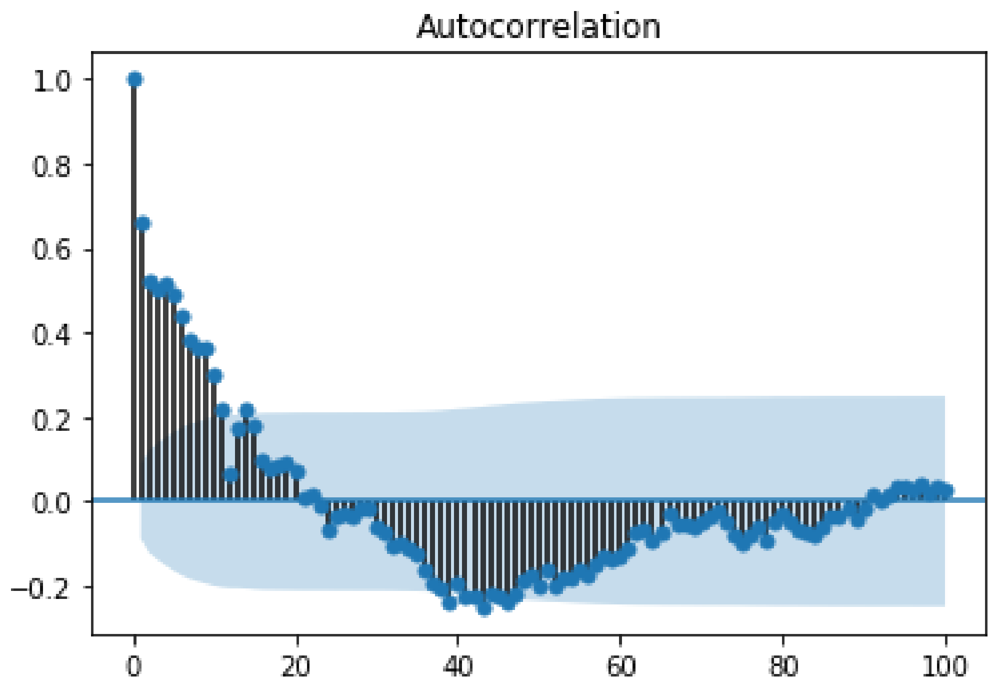
0.2.7 Here we can see that after first differencing the sales data of light weight trucks is stationarized as it's P value is less than 0.05

0.2.8 Plotted ACF AND PACF Plots to determine the order of differenced sales data

```
In [77]: import statsmodels.graphics.tsaplots as tsaplots
```

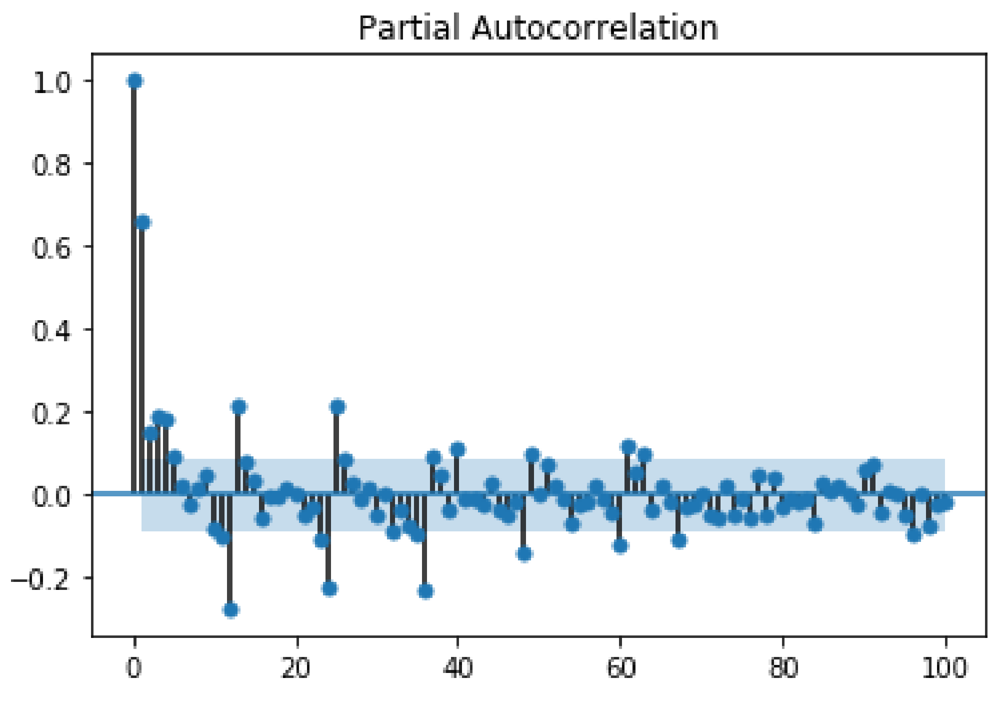
```
In [79]: tsaplots.plot_acf(seasonal_difference.dropna(), lags = 100)
```

```
Out[79]:
```



```
In [80]: tsaplots.plot_pacf(seasonal_difference.dropna(), lags = 100)
```

Out[80]:



0.2.9 From the above plot, we can see that differenced sales data is an ARMA process.

0.2.10 So, instead of using a Linear model to forecast sales

0.2.11 I used Non linear model, LSTM Neural Nets to forecast sales before and after stabilizing data by seasonal differencing

0.2.12 Used LSTM Model with 4 neurons and 100 epochs.

0.2.13 Below are two forecast results

0.2.14 before and after stationarizing sales data by seasonal differencing

0.3 Case 1: Before stationarizing Data

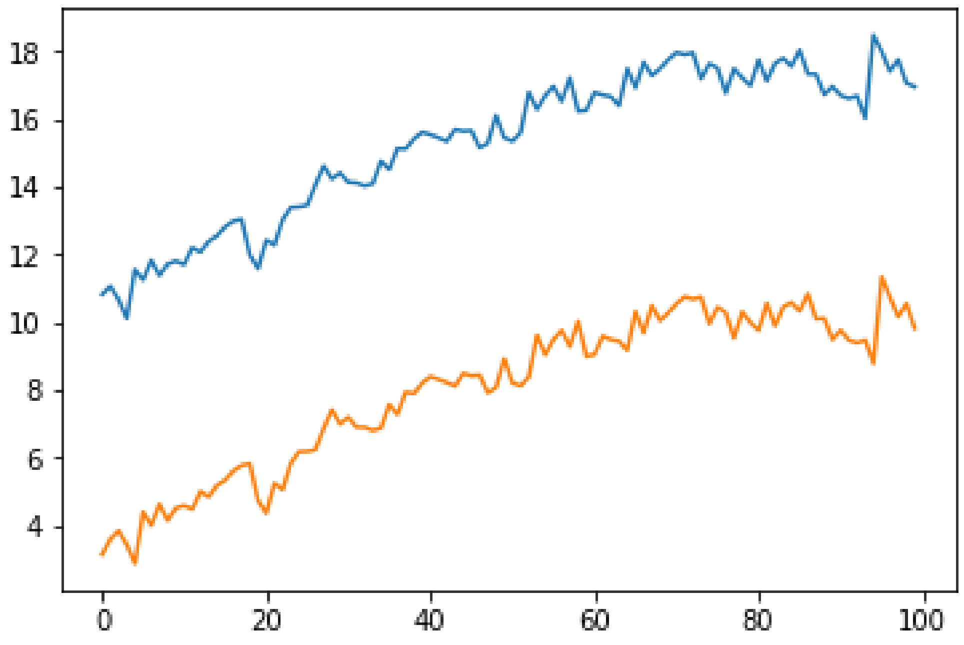
```
In [101]: from pandas import DataFrame
          from pandas import Series
          from pandas import concat
          from pandas import read_csv
          from pandas import datetime
          from sklearn.metrics import mean_squared_error
          from sklearn.preprocessing import MinMaxScaler
          from keras.models import Sequential
          from keras.layers import Dense
          from keras.layers import LSTM
          from math import sqrt
          from matplotlib import pyplot
          import numpy

          # date-time parsing function for loading the dataset
          def parser(x):
                  return datetime.strptime(x, '%Y-%m')

          # frame a sequence as a supervised learning problem
          def timeseries_to_supervised(data, lag=1):
                  df = DataFrame(data)
                  columns = [df.shift(i) for i in range(1, lag+1)]
                  columns.append(df)
                  df = concat(columns, axis=1)
                  df.fillna(0, inplace=True)
                  return df

          # create a differenced series
          def difference(dataset, interval=12):
                  diff = list()
                  for i in range(interval, len(dataset)):
                          value = dataset[i] - dataset[i - interval]
                          diff.append(value)
                  return Series(diff)

          # invert differenced value
          def inverse_difference(history, yhat, interval=1):
                  return yhat + history[-interval]

          # scale train and test data to [-1, 1]
          def scale(train, test):
                  # fit scaler
                  scaler = MinMaxScaler(feature_range=(-1, 1))
                  scaler = scaler.fit(train)
                  # transform train
                  train = train.reshape(train.shape[0], train.shape[1])
                  train_scaled = scaler.transform(train)
                  # transform test
                  test = test.reshape(test.shape[0], test.shape[1])
                  test_scaled = scaler.transform(test)
                  return scaler, train_scaled, test_scaled

          # inverse scaling for a forecasted value
          def invert_scale(scaler, X, value):
                  new_row = [x for x in X] + [value]
                  array = numpy.array(new_row)
                  array = array.reshape(1, len(array))
                  inverted = scaler.inverse_transform(array)
                  return inverted[0, -1]

          # fit an LSTM network to training data
          def fit_lstm(train, batch_size, nb_epoch, neurons):
                  X, y = train[:, 0:-1], train[:, -1]
                  X = X.reshape(X.shape[0], 1, X.shape[1])
                  model = Sequential()
                  model.add(LSTM(neurons, batch_input_shape=(batch_size, X.shape[1], X.shape[2
                  model.add(Dense(1))
                  model.compile(loss='mean_squared_error', optimizer='adam')
                  for i in range(nb_epoch):
                          model.fit(X, y, epochs=1, batch_size=batch_size, verbose=0, shuffle=F
                          model.reset_states()
                  return model

          # make a one-step forecast
          def forecast_lstm(model, batch_size, X):
                  X = X.reshape(1, 1, len(X))
                  yhat = model.predict(X, batch_size=batch_size)
                  return yhat[0,0]

          # load dataset
          series = read_csv('ALTSALES.csv', header=0, parse_dates=[0], index_col=0, squeeze=Tru

          # transform data to be stationary
          raw_values = series.values
          diff_values = difference(raw_values, 1)

          # transform data to be supervised learning
          supervised = timeseries_to_supervised(diff_values, 1)
          supervised_values = supervised.values

          # split data into train and test-sets
          train, test = supervised_values[0:-100], supervised_values[-100:]

          # transform the scale of the data
          scaler, train_scaled, test_scaled = scale(train, test)

          # fit the model
          lstm_model = fit_lstm(train_scaled, 1, 100, 4)
          # forecast the entire training dataset to build up state for forecasting
          train_reshaped = train_scaled[:, 0].reshape(len(train_scaled), 1, 1)
          lstm_model.predict(train_reshaped, batch_size=1)
```

```
# walk-forward validation on the test data
predictions = list()
for i in range(len(test_scaled)):
        # make one-step forecast
        X, y = test_scaled[i, 0:-1], test_scaled[i, -1]
        yhat = forecast_lstm(lstm_model, 1, X)
        # invert scaling
        yhat = invert_scale(scaler, X, yhat)
        # invert differencing
        yhat = inverse_difference(raw_values, yhat, len(test_scaled)+1-i)
        # store forecast
        predictions.append(yhat)
        expected = raw_values[len(train) + i + 1]
        print('Month=%d, Predicted=%f, Expected=%f' % (i+1, yhat, expected))

# report performance
rmse = sqrt(mean_squared_error(raw_values[-100:], predictions))
print('Test RMSE: %.3f' % rmse)
# line plot of observed vs predicted
pyplot.plot(raw_values[-100:])
pyplot.plot(predictions)
pyplot.show()
```

```
Month=1, Predicted=3.128731, Expected=10.817000
Month=2, Predicted=3.580904, Expected=11.060000
Month=3, Predicted=3.825219, Expected=10.668000
Month=4, Predicted=3.416617, Expected=10.108000
Month=5, Predicted=2.862783, Expected=11.553000
Month=6, Predicted=4.381949, Expected=11.249000
Month=7, Predicted=3.997144, Expected=11.822000
Month=8, Predicted=4.613437, Expected=11.385000
Month=9, Predicted=4.137127, Expected=11.715000
Month=10, Predicted=4.501349, Expected=11.802000
Month=11, Predicted=4.575414, Expected=11.703000
Month=12, Predicted=4.469539, Expected=12.199000
Month=13, Predicted=4.991833, Expected=12.070000
Month=14, Predicted=4.832768, Expected=12.383000
Month=15, Predicted=5.167731, Expected=12.547000
Month=16, Predicted=5.323410, Expected=12.815000
Month=17, Predicted=5.596010, Expected=12.984000
Month=18, Predicted=5.760161, Expected=13.037000
Month=19, Predicted=5.808889, Expected=11.980000
Month=20, Predicted=4.731242, Expected=11.581000
Month=21, Predicted=4.344960, Expected=12.419000
Month=22, Predicted=5.231709, Expected=12.277000
Month=23, Predicted=5.039033, Expected=13.026000
Month=24, Predicted=5.827531, Expected=13.391000
Month=25, Predicted=6.171001, Expected=13.406000
```

```
Month=26, Predicted=6.172569, Expected=13.455000
Month=27, Predicted=6.226449, Expected=14.058000
Month=28, Predicted=6.852857, Expected=14.617000
Month=29, Predicted=7.405494, Expected=14.237000
Month=30, Predicted=6.988934, Expected=14.408000
Month=31, Predicted=7.187075, Expected=14.135000
Month=32, Predicted=6.894165, Expected=14.118000
Month=33, Predicted=6.891503, Expected=14.027000
Month=34, Predicted=6.796856, Expected=14.092000
Month=35, Predicted=6.868954, Expected=14.759000
Month=36, Predicted=7.558696, Expected=14.514000
Month=37, Predicted=7.270214, Expected=15.130000
Month=38, Predicted=7.926651, Expected=15.121000
Month=39, Predicted=7.887199, Expected=15.415000
Month=40, Predicted=8.197115, Expected=15.614000
Month=41, Predicted=8.391101, Expected=15.541000
Month=42, Predicted=8.307157, Expected=15.447000
Month=43, Predicted=8.214796, Expected=15.339000
Month=44, Predicted=8.107582, Expected=15.695000
Month=45, Predicted=8.483667, Expected=15.646000
Month=46, Predicted=8.414250, Expected=15.661000
Month=47, Predicted=8.433857, Expected=15.153000
Month=48, Predicted=7.911165, Expected=15.283000
Month=49, Predicted=8.065442, Expected=16.108000
Month=50, Predicted=8.914203, Expected=15.453000
Month=51, Predicted=8.203400, Expected=15.345000
Month=52, Predicted=8.115163, Expected=15.596000
Month=53, Predicted=8.381017, Expected=16.788000
Month=54, Predicted=9.604420, Expected=16.289000
Month=55, Predicted=9.035831, Expected=16.693000
Month=56, Predicted=9.480209, Expected=16.969000
Month=57, Predicted=9.748297, Expected=16.529000
Month=58, Predicted=9.283649, Expected=17.218000
Month=59, Predicted=10.019589, Expected=16.237000
Month=60, Predicted=8.985476, Expected=16.261000
Month=61, Predicted=9.039818, Expected=16.788000
Month=62, Predicted=9.584264, Expected=16.714000
Month=63, Predicted=9.480005, Expected=16.663000
Month=64, Predicted=9.432717, Expected=16.399000
Month=65, Predicted=9.160463, Expected=17.495000
Month=66, Predicted=10.311933, Expected=16.928000
Month=67, Predicted=9.676964, Expected=17.686000
Month=68, Predicted=10.488518, Expected=17.287000
Month=69, Predicted=10.039914, Expected=17.495000
Month=70, Predicted=10.276223, Expected=17.758000
Month=71, Predicted=10.539688, Expected=17.972000
Month=72, Predicted=10.750524, Expected=17.911000
Month=73, Predicted=10.677937, Expected=17.966000
```

```
Month=74, Predicted=10.740160, Expected=17.195000
Month=75, Predicted=9.950060, Expected=17.640000
Month=76, Predicted=10.436872, Expected=17.515000
Month=77, Predicted=10.281211, Expected=16.770000
Month=78, Predicted=9.526670, Expected=17.494000
Month=79, Predicted=10.302037, Expected=17.221000
Month=80, Predicted=9.979204, Expected=16.992000
Month=81, Predicted=9.755591, Expected=17.752000
Month=82, Predicted=10.557137, Expected=17.130000
Month=83, Predicted=9.882014, Expected=17.650000
Month=84, Predicted=10.446080, Expected=17.803000
Month=85, Predicted=10.578336, Expected=17.561000
Month=86, Predicted=10.320413, Expected=18.048000
Month=87, Predicted=10.841178, Expected=17.337000
Month=88, Predicted=10.089806, Expected=17.331000
Month=89, Predicted=10.107492, Expected=16.721000
Month=90, Predicted=9.480570, Expected=16.968000
Month=91, Predicted=9.757233, Expected=16.701000
Month=92, Predicted=9.463378, Expected=16.608000
Month=93, Predicted=9.379721, Expected=16.688000
Month=94, Predicted=9.466529, Expected=16.020000
Month=95, Predicted=8.777744, Expected=18.485000
Month=96, Predicted=11.340847, Expected=18.006000
Month=97, Predicted=10.737523, Expected=17.418000
Month=98, Predicted=10.163523, Expected=17.754000
Month=99, Predicted=10.540044, Expected=17.067000
Month=100, Predicted=9.820640, Expected=16.962000
Test RMSE: 7.311
```



0.3.1 Based on the results predicted on non stationarized data

0.3.2 we can see that LSTM Models are just replicating some pattern in data

0.3.3 LSTM models are failing in forecasting sales data as RMS error in forecasting Is 7.311

0.4 Case 2: After stationarizing Data

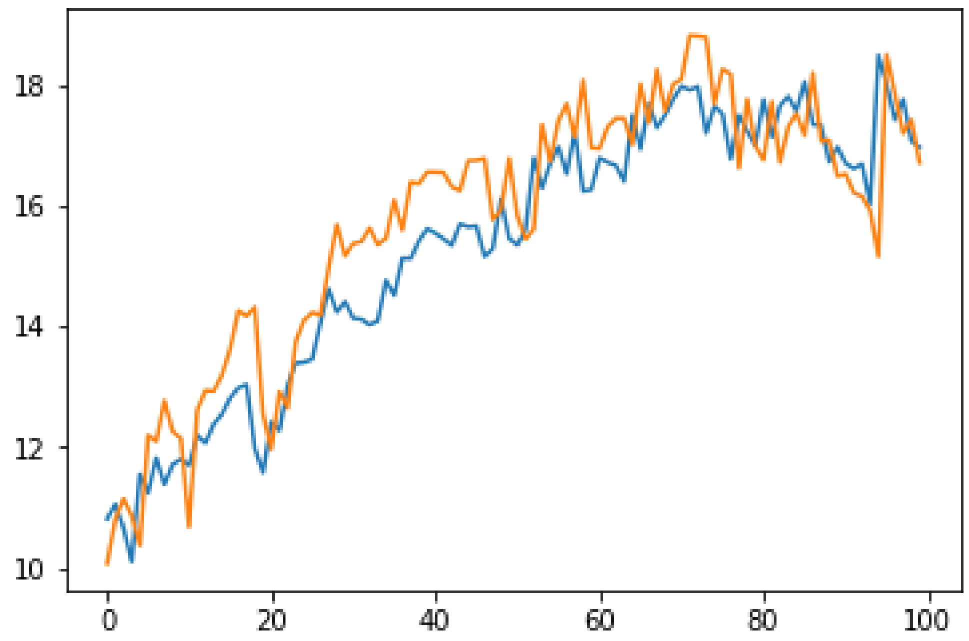
```
In [115]: from pandas import DataFrame
          from pandas import Series
          from pandas import concat
          from pandas import read_csv
          from pandas import datetime
          from sklearn.metrics import mean_squared_error
          from sklearn.preprocessing import MinMaxScaler
          from keras.models import Sequential
          from keras.layers import Dense
          from keras.layers import LSTM
          from math import sqrt
          from matplotlib import pyplot
          import numpy

          # date-time parsing function for loading the dataset
          def parser(x):
                  return datetime.strptime(x, '%Y-%m')

          # frame a sequence as a supervised learning problem
```

```
def timeseries_to_supervised(data, lag=1):
        df = DataFrame(data)
        columns = [df.shift(i) for i in range(1, lag+1)]
        columns.append(df)
        df = concat(columns, axis=1)
        df.fillna(0, inplace=True)
        return df

# create a differenced series
def difference(dataset, interval=12):
        diff = list()
        for i in range(interval, len(dataset)):
                value = dataset[i] - dataset[i - interval]
                diff.append(value)
        return Series(diff)

# invert differenced value
def inverse_difference(history, yhat, interval=1):
        return yhat + history[-interval]

# scale train and test data to [-1, 1]
def scale(train, test):
        # fit scaler
        scaler = MinMaxScaler(feature_range=(-1, 1))
        scaler = scaler.fit(train)
        # transform train
        train = train.reshape(train.shape[0], train.shape[1])
        train_scaled = scaler.transform(train)
        # transform test
        test = test.reshape(test.shape[0], test.shape[1])
        test_scaled = scaler.transform(test)
        return scaler, train_scaled, test_scaled

# inverse scaling for a forecasted value
def invert_scale(scaler, X, value):
        new_row = [x for x in X] + [value]
        array = numpy.array(new_row)
        array = array.reshape(1, len(array))
        inverted = scaler.inverse_transform(array)
        return inverted[0, -1]

# fit an LSTM network to training data
def fit_lstm(train, batch_size, nb_epoch, neurons):
        X, y = train[:, 0:-1], train[:, -1]
        X = X.reshape(X.shape[0], 1, X.shape[1])
        model = Sequential()
        model.add(LSTM(neurons, batch_input_shape=(batch_size, X.shape[1], X.shape[2
        model.add(Dense(1))
        model.compile(loss='mean_squared_error', optimizer='adam')
        for i in range(nb_epoch):
                model.fit(X, y, epochs=1, batch_size=batch_size, verbose=0, shuffle=
                model.reset_states()
        return model

# make a one-step forecast
def forecast_lstm(model, batch_size, X):
        X = X.reshape(1, 1, len(X))
        yhat = model.predict(X, batch_size=batch_size)
        return yhat[0,0]

# load dataset
series = read_csv('ALTSALES.csv', header=0, parse_dates=[0], index_col=0, squeeze=Tr

# transform data to be stationary
raw_values = series.values
diff_values = difference(raw_values, 12)
# Performed a seasonal yearly differencing (interval = 12 months) to stationarize th

# transform data to be supervised learning
supervised = timeseries_to_supervised(diff_values, 1)
supervised_values = supervised.values

# split data into train and test-sets
train, test = supervised_values[0:-100], supervised_values[-100:]

# transform the scale of the data
scaler, train_scaled, test_scaled = scale(train, test)

# fit the model
lstm_model = fit_lstm(train_scaled, 1, 100, 4)
# forecast the entire training dataset to build up state for forecasting
train_reshaped = train_scaled[:, 0].reshape(len(train_scaled), 1, 1)
lstm_model.predict(train_reshaped, batch_size=1)

# walk-forward validation on the test data
predictions = list()
for i in range(len(test_scaled)):
        # make one-step forecast
        X, y = test_scaled[i, 0:-1], test_scaled[i, -1]
        yhat = forecast_lstm(lstm_model, 1, X)
        # invert scaling
        yhat = invert_scale(scaler, X, yhat)
        # invert differencing
        yhat = inverse_difference(raw_values, yhat, len(test_scaled)+1-i)
        # store forecast
        predictions.append(yhat)
        expected = raw_values[len(train) + i + 1]
        print('Month=%d, Predicted=%f, Expected=%f' % (i+1, yhat, expected))

# report performance
rmse = sqrt(mean_squared_error(raw_values[-100:], predictions))
print('Test RMSE: %.3f' % rmse)
# line plot of observed vs predicted
pyplot.plot(raw_values[-100:])
pyplot.plot(predictions)
pyplot.show()
```

```
Month=1, Predicted=10.077825, Expected=10.141000
Month=2, Predicted=10.804842, Expected=9.574000
Month=3, Predicted=11.146881, Expected=9.023000
Month=4, Predicted=10.872128, Expected=9.552000
Month=5, Predicted=10.376185, Expected=9.197000
Month=6, Predicted=12.198038, Expected=9.996000
Month=7, Predicted=12.101322, Expected=9.956000
Month=8, Predicted=12.769087, Expected=11.370000
Month=9, Predicted=12.262516, Expected=14.567000
Month=10, Predicted=12.144432, Expected=9.346000
Month=11, Predicted=10.684989, Expected=10.370000
Month=12, Predicted=12.618979, Expected=10.817000
Month=13, Predicted=12.940367, Expected=11.060000
Month=14, Predicted=12.929556, Expected=10.668000
Month=15, Predicted=13.192876, Expected=10.108000
Month=16, Predicted=13.621494, Expected=11.553000
Month=17, Predicted=14.257914, Expected=11.249000
Month=18, Predicted=14.172553, Expected=11.822000
Month=19, Predicted=14.317385, Expected=11.385000
Month=20, Predicted=12.549443, Expected=11.715000
Month=21, Predicted=11.965282, Expected=11.802000
Month=22, Predicted=12.919897, Expected=11.703000
Month=23, Predicted=12.650022, Expected=12.199000
Month=24, Predicted=13.754080, Expected=12.070000
Month=25, Predicted=14.102367, Expected=12.383000
Month=26, Predicted=14.227843, Expected=12.547000
Month=27, Predicted=14.183150, Expected=12.815000
Month=28, Predicted=14.969557, Expected=12.984000
Month=29, Predicted=15.682302, Expected=13.037000
Month=30, Predicted=15.167169, Expected=11.980000
Month=31, Predicted=15.368930, Expected=11.581000
Month=32, Predicted=15.402339, Expected=12.419000
Month=33, Predicted=15.625723, Expected=12.277000
Month=34, Predicted=15.350335, Expected=13.026000
Month=35, Predicted=15.458713, Expected=13.391000
Month=36, Predicted=16.087971, Expected=13.406000
Month=37, Predicted=15.594749, Expected=13.455000
```

```
Month=38, Predicted=16.387597, Expected=14.058000
Month=39, Predicted=16.361448, Expected=14.617000
Month=40, Predicted=16.553410, Expected=14.237000
Month=41, Predicted=16.551900, Expected=14.408000
Month=42, Predicted=16.544021, Expected=14.135000
Month=43, Predicted=16.311581, Expected=14.118000
Month=44, Predicted=16.244986, Expected=14.027000
Month=45, Predicted=16.740017, Expected=14.092000
Month=46, Predicted=16.747972, Expected=14.759000
Month=47, Predicted=16.778871, Expected=14.514000
Month=48, Predicted=15.765435, Expected=15.130000
Month=49, Predicted=15.926090, Expected=15.121000
Month=50, Predicted=16.779728, Expected=15.415000
Month=51, Predicted=15.828281, Expected=15.614000
Month=52, Predicted=15.442656, Expected=15.541000
Month=53, Predicted=15.611367, Expected=15.447000
Month=54, Predicted=17.333817, Expected=15.339000
Month=55, Predicted=16.727320, Expected=15.695000
Month=56, Predicted=17.397985, Expected=15.646000
Month=57, Predicted=17.681125, Expected=15.661000
Month=58, Predicted=17.130073, Expected=15.153000
Month=59, Predicted=18.074002, Expected=15.283000
Month=60, Predicted=16.955883, Expected=16.108000
Month=61, Predicted=16.940338, Expected=15.453000
Month=62, Predicted=17.300716, Expected=15.345000
Month=63, Predicted=17.442161, Expected=15.596000
Month=64, Predicted=17.440515, Expected=16.788000
Month=65, Predicted=17.004860, Expected=16.289000
Month=66, Predicted=18.011366, Expected=16.693000
Month=67, Predicted=17.366604, Expected=16.969000
Month=68, Predicted=18.252227, Expected=16.529000
Month=69, Predicted=17.563159, Expected=17.218000
Month=70, Predicted=18.009675, Expected=16.237000
Month=71, Predicted=18.088541, Expected=16.261000
Month=72, Predicted=18.810229, Expected=16.788000
Month=73, Predicted=18.803059, Expected=16.714000
Month=74, Predicted=18.782775, Expected=16.663000
Month=75, Predicted=17.677436, Expected=16.399000
Month=76, Predicted=18.253630, Expected=17.495000
Month=77, Predicted=18.174011, Expected=16.928000
Month=78, Predicted=16.620274, Expected=17.686000
Month=79, Predicted=17.759747, Expected=17.287000
Month=80, Predicted=16.979771, Expected=17.495000
Month=81, Predicted=16.756858, Expected=17.758000
Month=82, Predicted=17.717218, Expected=17.972000
Month=83, Predicted=16.712353, Expected=17.911000
Month=84, Predicted=17.302666, Expected=17.966000
Month=85, Predicted=17.525207, Expected=17.195000
```

```
Month=86, Predicted=17.160622, Expected=17.640000
Month=87, Predicted=18.195011, Expected=17.515000
Month=88, Predicted=17.064917, Expected=16.770000
Month=89, Predicted=17.077499, Expected=17.494000
Month=90, Predicted=16.483681, Expected=17.221000
Month=91, Predicted=16.521955, Expected=16.992000
Month=92, Predicted=16.210244, Expected=17.752000
Month=93, Predicted=16.150091, Expected=17.130000
Month=94, Predicted=15.916218, Expected=17.650000
Month=95, Predicted=15.152416, Expected=17.803000
Month=96, Predicted=18.497286, Expected=17.561000
Month=97, Predicted=17.857805, Expected=18.048000
Month=98, Predicted=17.193012, Expected=17.337000
Month=99, Predicted=17.414470, Expected=17.331000
Month=100, Predicted=16.711072, Expected=16.721000
Test RMSE: 0.923
```



0.4.1 After stationarizing data by removing seasonality,

0.4.2 we can see that the error in forecast has decreased drastically by 7 units.

0.5 Conclusions:

0.5.1 Based on above observations we can say that Neural Networks are best at capturing underlying patterns in data and replicating them for future values.

0.5.2 Hence, neural nets are to be used in specific applications like voice or pattern recognition, image processing where output data has a pattern and we are trying to understand and replicate the same pattern.

0.5.3 In time series application, we should use neural networks for predictions once the data is stationarized as by stationarizing data we are trying to say that future values depend on past values and the pattern continues.

0.5.4 By applying neural networks on non stationary data, though neural net might be able to capture the pattern in data it does a poor job in forecasting as it fails to understand underlying data generation process and does not replicate data generation process there by performing poorly on new test data.