greatlearning

TIME SERIES FORECASTING MODULE

Business Report

Submitted By- Anmol Dwivedi

DATASET 1 (Sparkling Wine Sales):

For this particular assignment, the data of different types of wine sales in the 20th century is to be analysed. Both of these data are from the same company but of different wines. As an analyst in the ABC Estate Wines, you are tasked to analyse and forecast Wine Sales in the 20th century.

Data set for the Problem: **Sparkling.csv** and Rose.csv

Please do perform the following questions on each of these two data sets separately.

1. Read the data as an appropriate Time Series data and plot the data.

The first step of any analysis is to import the necessary libraries into the python notebook. For our analysis we are going to need pandas, numpy, seaborn and sklearn libraries. These were imported into the jupyter notebook by using the following code. For additional libraries we will import them as we proceed further down into the analysis.

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

Whenever we are performing a Time Series Forecasting Problem, it is a good practice to check the version of libraries that we are using. This is important because varying versions of libraries give slightly different solutions, so we will ensure that we are using the latest version of libraries as shown below:

```
np.__version__
```

```
'1.18.4'
```

```
pd.__version__
```

```
'1.0.3'
```

```
sklearn.__version__
```

```
'0.23.2'
```

```
statsmodels.__version__
```

```
'0.11.1'
```

After this we import the Sparklin.csv dataset into the jupyter environment using the following function:

```
df1 = pd.read_csv('Sparkling.csv')
df1.head()
```

Now we proceed into the initial investigation of our dataset. First, we check the head of the data using the head() function. We can see the layout of our data and the various variables that we will be working with in this assignment. This command yields the following results:

```
df1.head()
```

	YearMonth	Sparkling
0	1980-01	1686
1	1980-02	1591
2	1980-03	2304
3	1980-04	1712
4	1980-05	1471

We can see that we have a univariate time series at our hand. Along with the head of the dataset we are also interested in the shape of the Data frame. To check the shape of the dataset we use shape function. This gives us the output as (187, 3). So, we can see that the given dataset has 187 observations and 3 features.
The current time series has time stamp in the form of year-month. We are going to format the time-stamp to change it from year-moth to YYYY-MM-DD. To do this we first use the date_range function from the pandas library to generate a data range from 1980-01-31 to 1995-07-31 at a monthly frequency.

```
date = pd.date_range(start='1/1/1980', end='8/1/1995', freq='M')
date
```

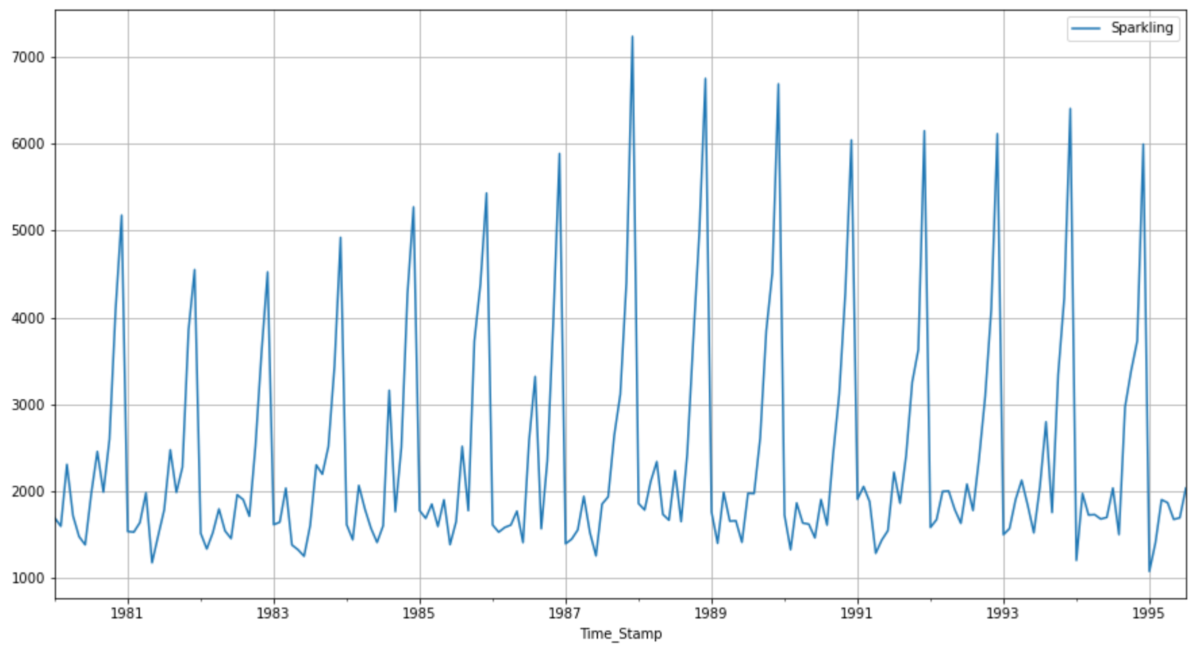
```
DatetimeIndex(['1980-01-31', '1980-02-29', '1980-03-31', '1980-04-30',
               '1980-05-31', '1980-06-30', '1980-07-31', '1980-08-31',
               '1980-09-30', '1980-10-31',
               ...
               '1994-10-31', '1994-11-30', '1994-12-31', '1995-01-31',
               '1995-02-28', '1995-03-31', '1995-04-30', '1995-05-31',
               '1995-06-30', '1995-07-31'],
              dtype='datetime64[ns]', length=187, freq='M')
```

Next, we are going to add the above generated dates as a time stamp and add them to our original time series. After this, we are will use the to_datetime function from pandas library to convert them into time-stamps and set it as index after dropping the original year month column. These steps will give us a final time series dataset as shown below:

```
df = df1.set_index('Time_Stamp')
df.drop(['YearMonth'], axis=1, inplace=True)
df.head()
```

	Sparkling
Time_Stamp	
1980-01-31	1686
1980-02-29	1591
1980-03-31	2304
1980-04-30	1712
1980-05-31	1471

Now, will plot our time series data using the pylab from rcParams. The plot of the time series is as shown below:



2. Perform appropriate Exploratory Data Analysis to understand the data and also perform decomposition.

After plotting the Time Series data, we are ready to do perform exploratory data analysis. Exploratory Data Analysis will help us to get important insights from the data.
The first step, we will check the info about the variables using the info() function. The results of this function are shown below:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 187 entries, 1980-01-31 to 1995-07-31
Data columns (total 1 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Sparkling  187 non-null    int64
dtypes: int64(1)
memory usage: 2.9 KB
```

We can see that our dataset has only one variable and it is int type data. The time column is neglected by this function.
Now, we proceed to descriptive statistics using the describe() function. Descriptive statistics gives a bird's eye view of the summary of the whole dataset. It shows us the Five-Point summary (mean, standard deviation, IQR values) of the dataset's continuous variables as shown below:

```
df.describe()
```

	Sparkling
count	187.000000
mean	2402.417112
std	1295.111540
min	1070.000000
25%	1605.000000
50%	1874.000000
75%	2549.000000
max	7242.000000

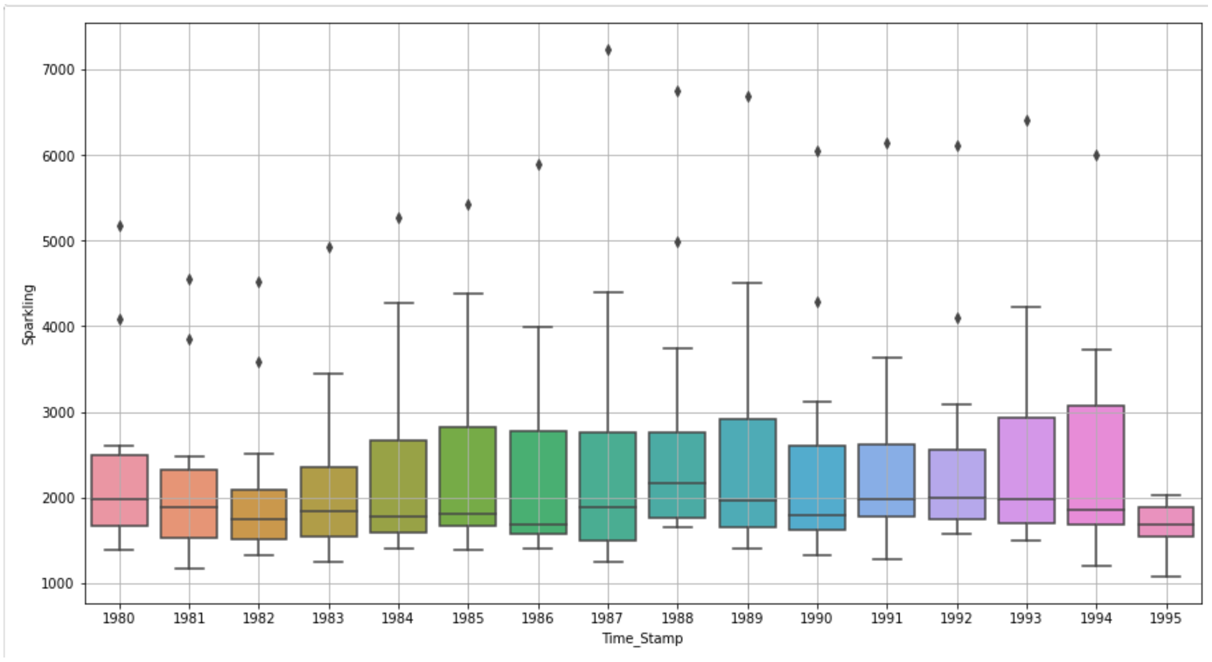
After completing the 5-point summary we proceed to do null value check. We check for null values by df.isnull().sum() function (since isnull is a Boolean operator we use sum function along with it to get the sum of all the values). This command returns the following results:

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

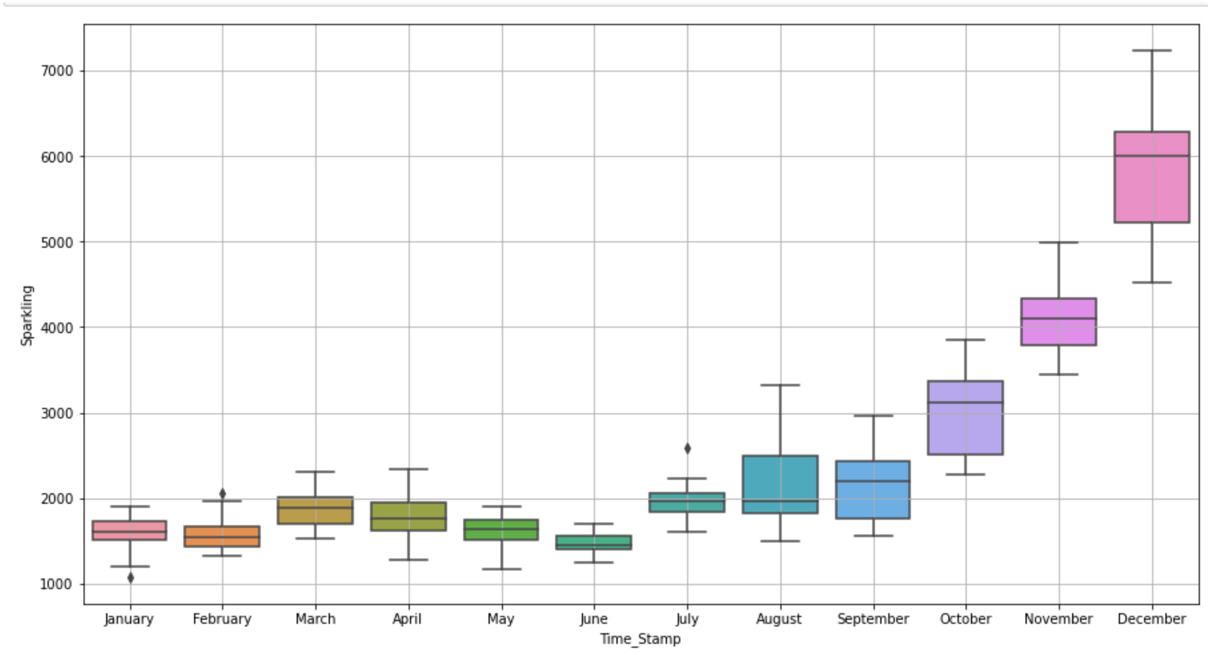
```
Sparkling    0
dtype: int64
```

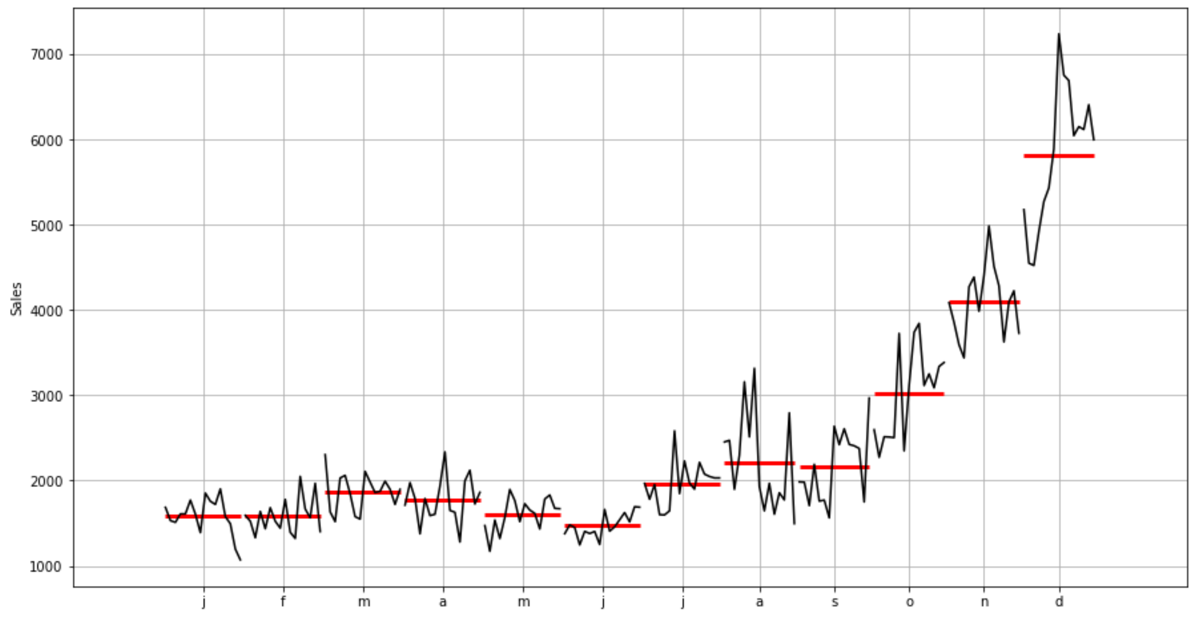
We can see that there are no null values in our dataset. If there was any presence of null value, we would have to use special techniques like interpolation or nearest neighbours to impute them. We cannot impute time series null values just by mean or medians. Additionally , we cannot drop the null values as this would create holes (missing time stamps) in our dataset and the dataset can no longer be passed to time series models for analysis.

After null value check, will look at some boxplots for the time series data. Below is the Boxplot for yearly sales of the Sparkling Wine. We can see outliers in the boxplots but we will not treat them until explicitly specified. The outliers that we see in the plot are clear indicators of the sale spikes due to seasonality.

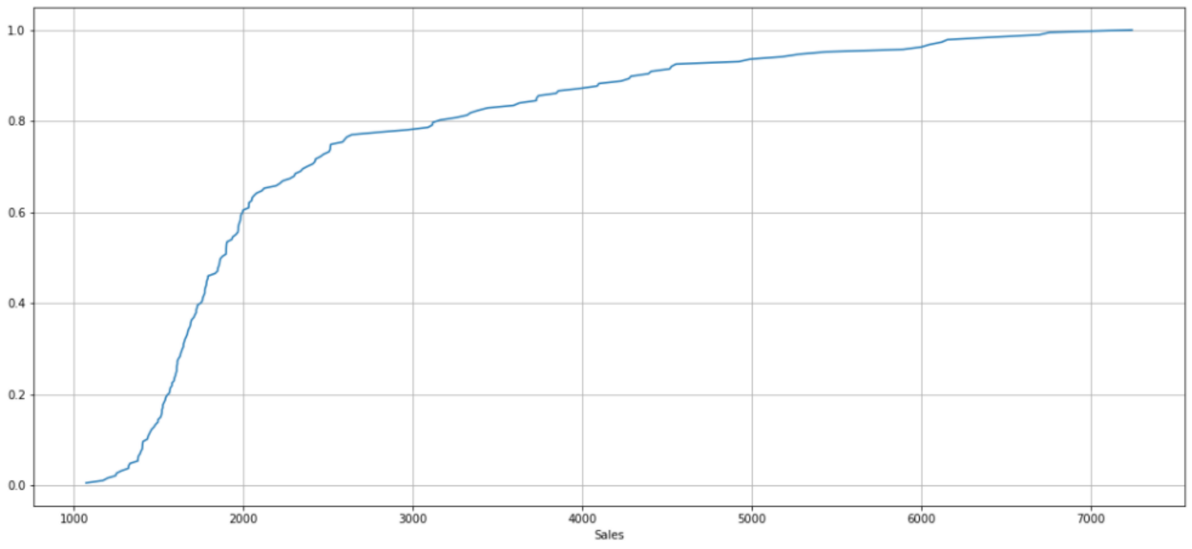


We can get an even more clear information about sales if we take a look at the monthly sales plots for the Sparkling wine. The boxplot and month plot for the same are shown below. We can see clearly from the plots that the sales follow somewhat same trend from January to April, then the depreciate in months of May and June, and then July onwards start to increase again. The sales keep on increasing from July and reach their peak in the month of December. In a nut shell we can say that Sales are highest in December and lowest in June.





We can also plot an ECDF or Empirical Cumulative Distribution Function for our dataset
An ECDF is an estimator of the Cumulative Distribution Function. The ECDF essentially allows us to plot a feature of our data in order from least to greatest and see the whole feature as if is distributed across the data set. An important information that we can see from this dataset is that the Sales range from 1000 to 7000, we can confirm the same if we see the minimum and maximum values using min() and max() function. The ECDF for Sales is shown below:



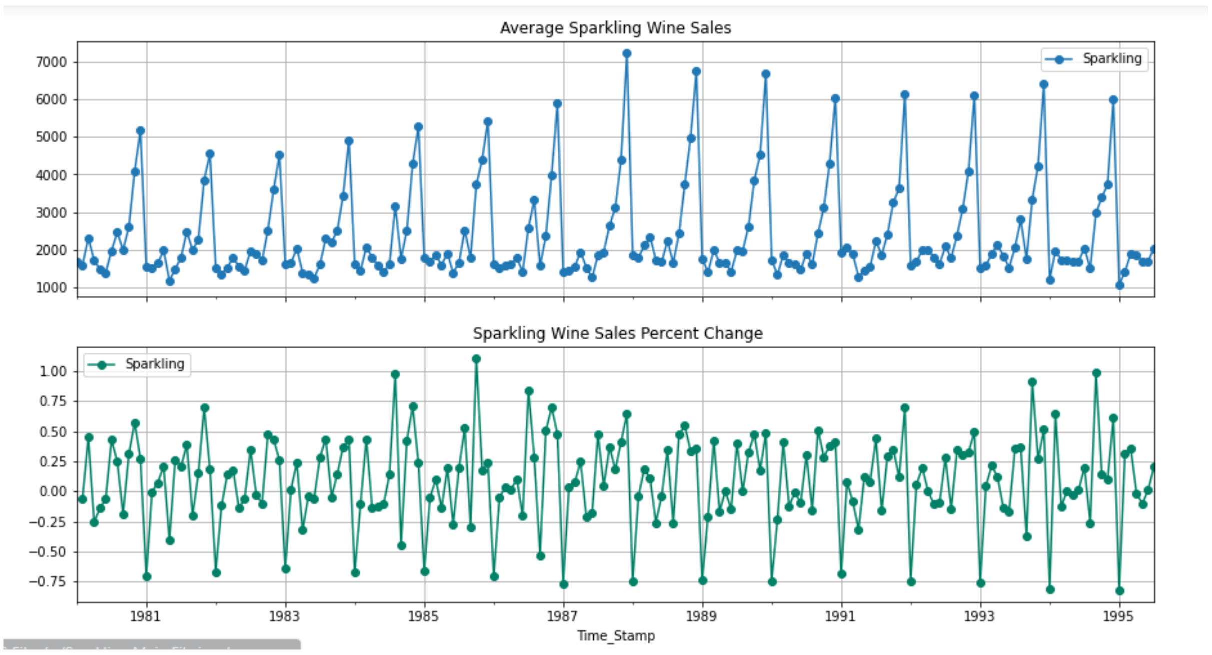
```
df['Sparkling'].min()
```

```
1070
```

```
df['Sparkling'].max()
```

```
7242
```

We can plot the mean Sales and percentage of mean sales in the Sparkling Dataset too. Both these plots are shown below:

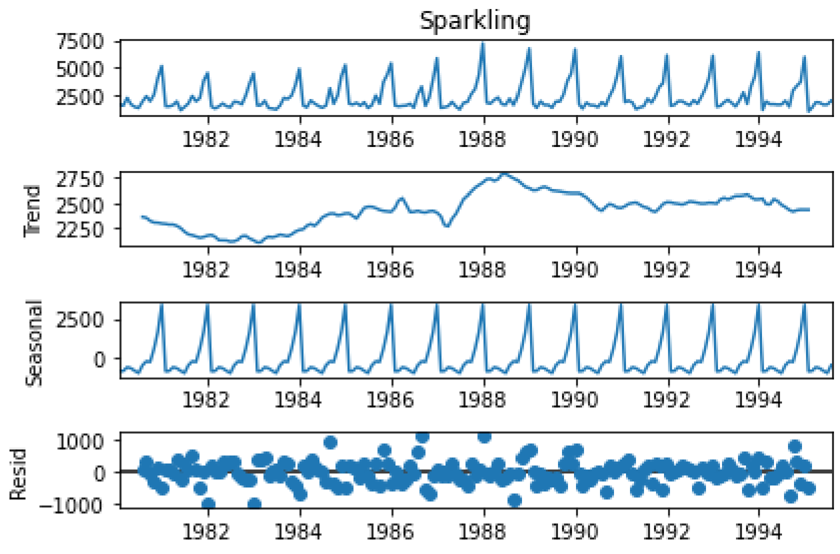


TIME SERIES DECOMPOSTITION:

One of the most important task in the EDA of Time Series Data is performing data decomposition. Decomposition of time series means splitting the time series into Trend, Seasonality and Residuals and visualize each of them. Trend and seasonality are the systematic components of the time series and Residuals are the random component of the time series. The decomposition could be an additive model or a multiplicative model.

```
decomposition = seasonal_decompose(df['Sparkling'],model='additive')
decomposition.plot();
```

```
<Figure size 720x720 with 0 Axes>
```



We can clearly see that the given time series is an additive series as the trend is almost linear with time and the seasonality has almost same pattern, which is typical for an additive series. The residuals all a mean of 0 and standard deviation of 1. If the trend had a curvilinear kind of trend and seasonality had increasing or decreasing peaks with the trend, we would have concluded that the series is multiplicative in nature. We can also see the decomposed time-series' valued for trends, seasonality and residuals as shown below:

```
trend = decomposition.trend
seasonality = decomposition.seasonal
residual = decomposition.resid

print('Trend','\n',trend.head(12),'\n')
print('Seasonality','\n',seasonality.head(12),'\n')
print('Residual','\n',residual.head(12),'\n')
```

```
Trend
 Time_Stamp
1980-01-31            NaN
1980-02-29            NaN
1980-03-31            NaN
1980-04-30            NaN
1980-05-31            NaN
1980-06-30            NaN
1980-07-31    2360.666667
1980-08-31    2351.333333
1980-09-30    2320.541667
1980-10-31    2303.583333
1980-11-30    2302.041667
1980-12-31    2293.791667
Name: trend, dtype: float64



Seasonality
 Time_Stamp
1980-01-31    0.649843
1980-02-29    0.659214
1980-03-31    0.757440
1980-04-30    0.730351
1980-05-31    0.660609
1980-06-30    0.603468
1980-07-31    0.809164
1980-08-31    0.918822
1980-09-30    0.894367
1980-10-31    1.241789
1980-11-30    1.690158
1980-12-31    2.384776
Name: seasonal, dtype: float64
```

```
Residual
 Time_Stamp
1980-01-31          NaN
1980-02-29          NaN
1980-03-31          NaN
1980-04-30          NaN
1980-05-31          NaN
1980-06-30          NaN
1980-07-31     1.029230
1980-08-31     1.135407
1980-09-30     0.955954
1980-10-31     0.907513
1980-11-30     1.050423
1980-12-31     0.946770
Name: resid, dtype: float64
```

3. Split the data into training and test. The test data should start in 1991.

Splitting time series is not like we did for machine learning models because time series has a specific structure that we cannot break. We cannot randomly pick datapoints and split them into train and testing. For splitting time series data, we choose the most recent data as the test dataset and the preceding data as the train data. Also, if the dataset has a seasonality a component, we need to ensure that the seasonality is captured by both the training dataset and testing dataset. For this analysis we have been explicitly told to split the data into train and testing from 1991, so we will do that. The data is split into test and train as shown below. We will also see the head and tail of both test and train dataset. The plot for split up time series is also shown:

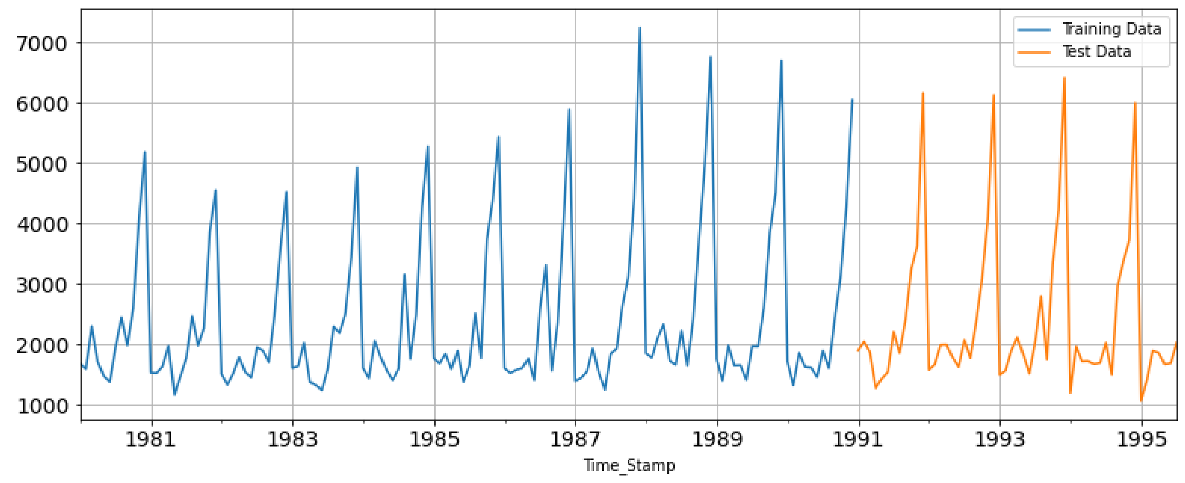
```
train=df[df.index.year < 1991]
test=df[df.index.year >= 1991]
```

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

```
(132, 1)
(55, 1)
```

```
First few rows of Training Data
                Sparkling
Time_Stamp
1980-01-31       1686
1980-02-29       1591
1980-03-31       2304
1980-04-30       1712
1980-05-31       1471

Last few rows of Training Data
                Sparkling
Time_Stamp
1990-08-31       1605
1990-09-30       2424
1990-10-31       3116
1990-11-30       4286
1990-12-31       6047
```

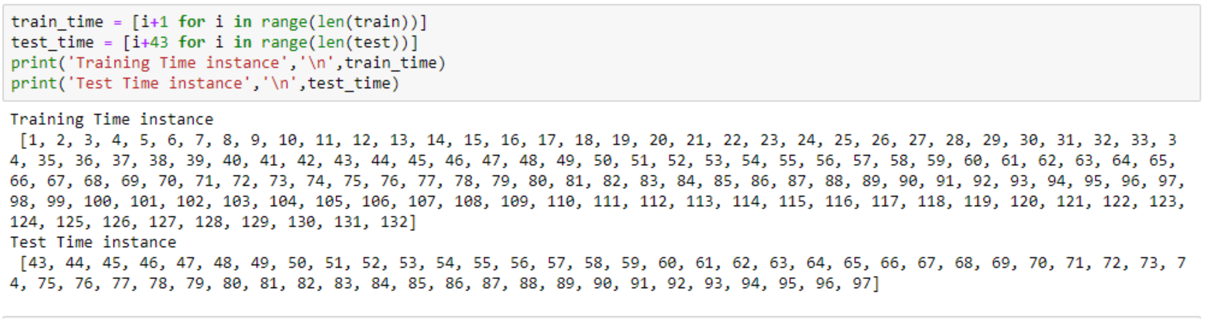
```
First few rows of Test Data
                Sparkling
Time_Stamp
1991-01-31       1902
1991-02-28       2049
1991-03-31       1874
1991-04-30       1279
1991-05-31       1432

Last few rows of Test Data
                Sparkling
Time_Stamp
1995-03-31       1897
1995-04-30       1862
1995-05-31       1670
1995-06-30       1688
1995-07-31       2031
```



4. Build various exponential smoothing models on the training data and evaluate the model using RMSE on the test data.
   Other models such as Regression, naïve forecast models, simple average models etc. should also be built on the training data and check the performance on the test data using RMSE.

(a) <u>REGRESSION MODEL:</u>

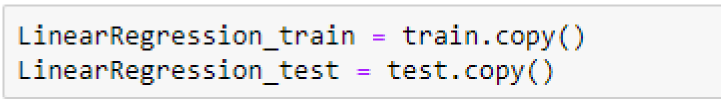
To build a regression model, we are going to regress the 'Sparkling' variable against the order of the occurrence. For this we need to modify our training data before fitting it into a linear regression.



```
train_time = [i+1 for i in range(len(train))]
test_time = [i+43 for i in range(len(test))]
print('Training Time instance','\n',train_time)
print('Test Time instance','\n',test_time)
```

```
Training Time instance
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 3
4, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123,
124, 125, 126, 127, 128, 129, 130, 131, 132]
Test Time instance
 [43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 7
4, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97]
```

We see that we have successfully the generated the numerical time instance order for both the training and test set. Now we will add these values in the training and test set



```
LinearRegression_train = train.copy()
LinearRegression_test = test.copy()
```

```
First few rows of Training Data
            Sparkling  time
Time_Stamp
1980-01-31       1686     1
1980-02-29       1591     2
1980-03-31       2304     3
1980-04-30       1712     4
1980-05-31       1471     5

Last few rows of Training Data
            Sparkling  time
Time_Stamp
1990-08-31       1605   128
1990-09-30       2424   129
1990-10-31       3116   130
1990-11-30       4286   131
1990-12-31       6047   132
```

```
First few rows of Test Data
            Sparkling  time
Time_Stamp
1991-01-31       1902    43
1991-02-28       2049    44
1991-03-31       1874    45
1991-04-30       1279    46
1991-05-31       1432    47

Last few rows of Test Data
            Sparkling  time
Time_Stamp
1995-03-31       1897    93
1995-04-30       1862    94
1995-05-31       1670    95
1995-06-30       1688    96
1995-07-31       2031    97
```

Now we will pass these values to the Regression values as shown below:

```
from sklearn.linear_model import LinearRegression
```

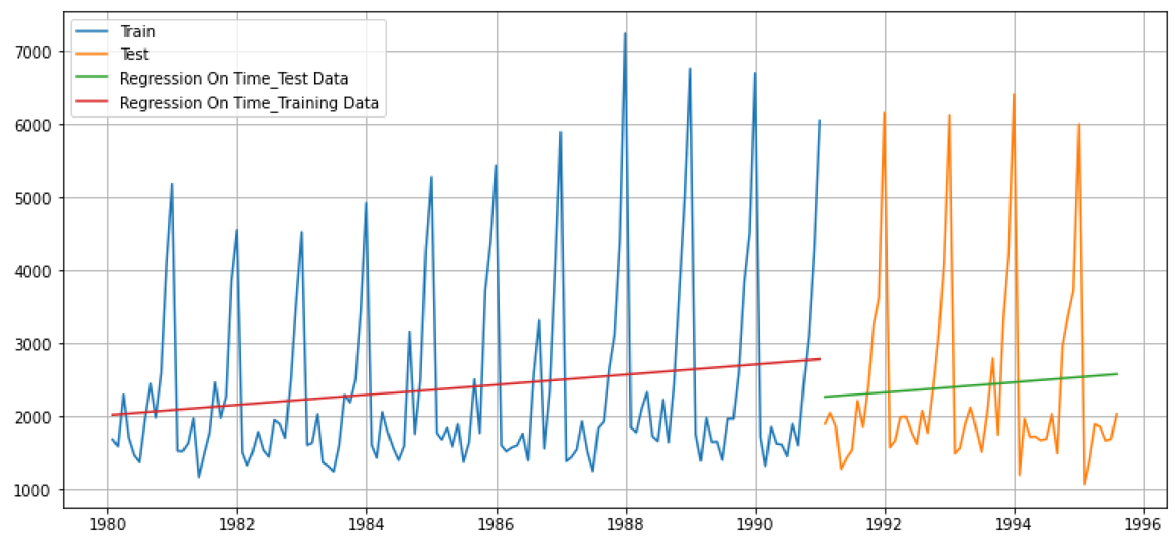
```
lr = LinearRegression()
```

```
lr.fit(LinearRegression_train[['time']],LinearRegression_train['Sparkling'].values)
```

```
LinearRegression()
```

Now we are ready to make predictions by using this model. We will plot the training and testing data too as shown below:

```
#predictions on train data
train_predictions_model1         = lr.predict(LinearRegression_train[['time']])
LinearRegression_train['RegOnTime'] = train_predictions_model1

#predictions on test data
test_predictions_model1         = lr.predict(LinearRegression_test[['time']])
LinearRegression_test['RegOnTime'] = test_predictions_model1
```



We can see that regression model doesn't capture much of the data. We will now get the Root Mean Squared Error values and Mean Absolute Percentage error on dataset. For MAPE we will define a custom function as show below

```
## Mean Absolute Percentage - Function Definition

def MAPE(y, yhat):
    y, yhat = np.array(y), np.array(yhat)
    try:
        mape =  round(np.sum(np.abs(yhat - y)) / np.sum(y) * 100,2)
    except:
        print("Observed values are empty")
        mape = np.nan
    return mape
```

RMSE and MAPE on train and testing data

```
## Training Data - RMSE and MAPE

rmse_model1_train = metrics.mean_squared_error(train['Sparkling'],train_predictions_model1,squared=False)
mape_model1_train = MAPE(train['Sparkling'],train_predictions_model1)
print("For Regression On Time forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model1_train, mape_model1_trai
```

```
For Regression On Time forecast on the Training Data,  RMSE is 1279.322 MAPE is 40.05
```

```
## Test Data - RMSE and MAPE

rmse_model1_test = metrics.mean_squared_error(test['Sparkling'],test_predictions_model1,squared=False)
mape_model1_test = MAPE(test['Sparkling'],test_predictions_model1)
print("For Regression On Time forecast on the Test Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model1_test, mape_model1_test))
```

```
For Regression On Time forecast on the Test Data,  RMSE is 1275.867 MAPE is 39.16
```

(b) NAÏVE FORECAST:

Naïve forecast as the name suggests is a very simple model. It says that the value today will be the same tomorrow. The Naïve Forecast model is built and visualized as shown below:

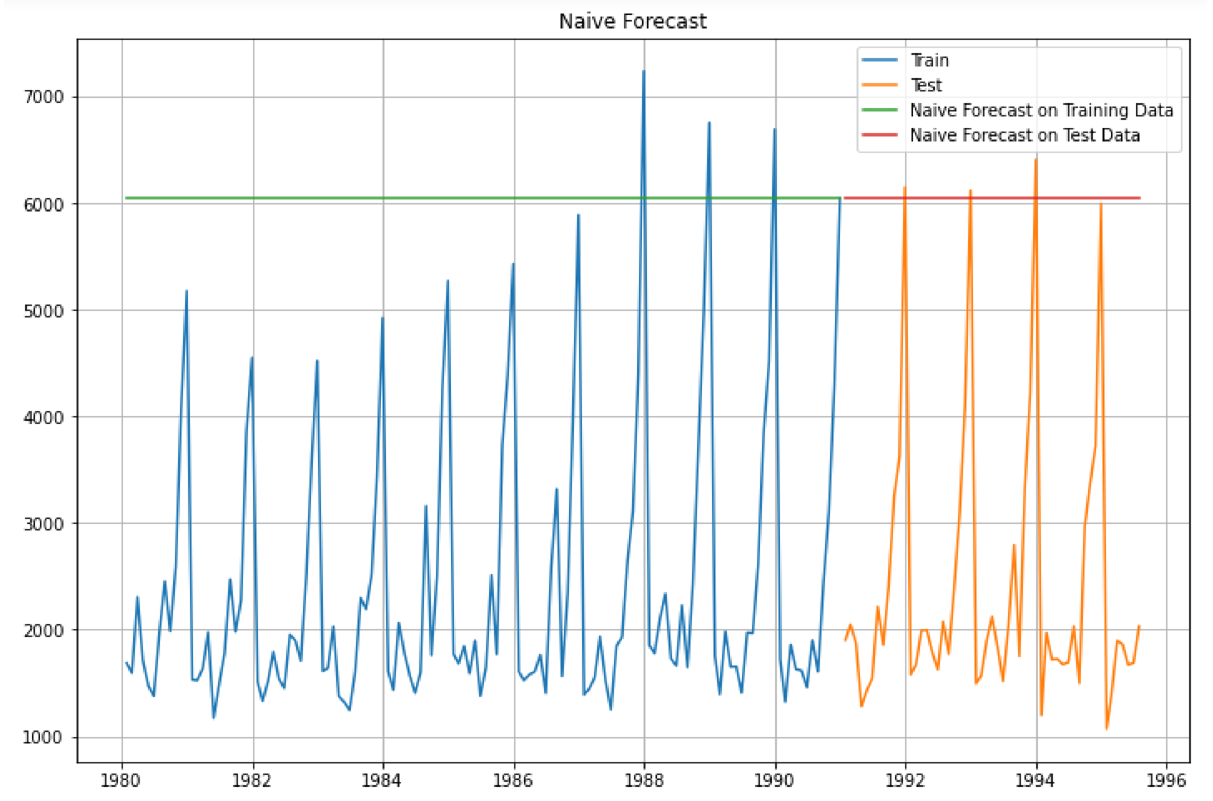
```
NaiveModel_train = train.copy()
NaiveModel_test = test.copy()
```

```
NaiveModel_train['naive'] = np.asarray(train['Sparkling'])[len(np.asarray(train['Sparkling']))-1]
NaiveModel_train['naive'].head()
```

```
Time_Stamp
1980-01-31    6047
1980-02-29    6047
1980-03-31    6047
1980-04-30    6047
1980-05-31    6047
Name: naive, dtype: int64
```

```
NaiveModel_test['naive'] = np.asarray(train['Sparkling'])[len(np.asarray(train['Sparkling']))-1]
NaiveModel_test['naive'].head()
```

```
Time_Stamp
1991-01-31    6047
1991-02-28    6047
1991-03-31    6047
1991-04-30    6047
1991-05-31    6047
Name: naive, dtype: int64
```



We can see that all the future forecasted values are the same as the last value on training dataset.

RMSE and MAPE on Test and Train Dataset

```
## Training Data - RMSE and MAPE

rmse_model2_train = metrics.mean_squared_error(train['Sparkling'],NaiveModel_train['naive'],squared=False)
mape_model2_train = MAPE(train['Sparkling'],NaiveModel_train['naive'])
print("For Naive Model forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model2_train, mape_model2_train))
```

```
For Naive Model forecast on the Training Data,  RMSE is 3867.701 MAPE is 153.17
```

```
## Test Data - RMSE and MAPE

rmse_model2_test = metrics.mean_squared_error(test['Sparkling'],NaiveModel_test['naive'],squared=False)
mape_model2_test = MAPE(test['Sparkling'],NaiveModel_test['naive'])
print("For Naive Model forecast on the Test Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model2_test, mape_model2_test))
```

```
For Naive Model forecast on the Test Data,  RMSE is 3864.279 MAPE is 152.87
```

(c) <u>Simple Average Model:</u>

This is another simple model that says that the future forecasted values will be equal to the mean of the past values. The Simple Model is shown below:

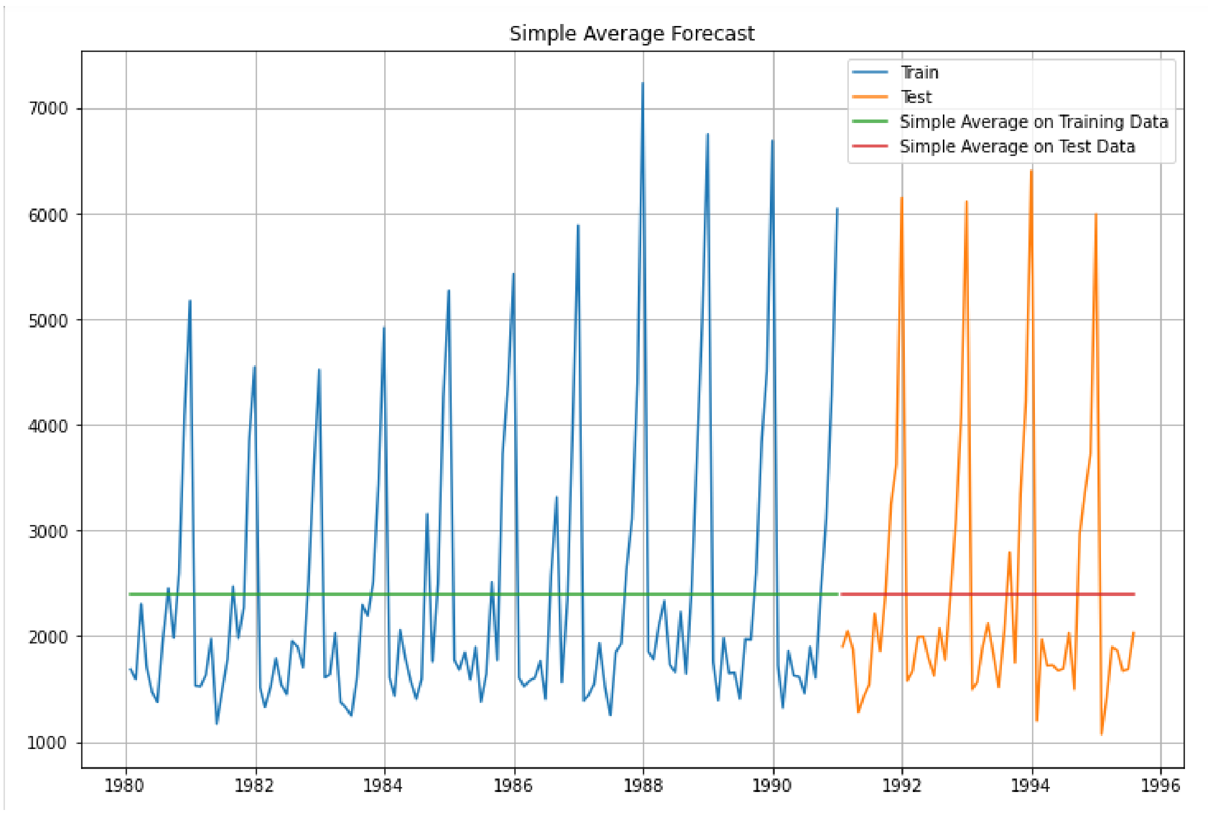
```
SimpleAverage_train = train.copy()
SimpleAverage_test = test.copy()
```

```
SimpleAverage_train['mean_forecast'] = train['Sparkling'].mean()
SimpleAverage_train.head()
```

	Sparkling	mean_forecast
Time_Stamp		
1980-01-31	1686	2403.780303
1980-02-29	1591	2403.780303
1980-03-31	2304	2403.780303
1980-04-30	1712	2403.780303
1980-05-31	1471	2403.780303

```
SimpleAverage_test['mean_forecast'] = train['Sparkling'].mean()
SimpleAverage_test.head()
```

	Sparkling	mean_forecast
Time_Stamp		
1991-01-31	1902	2403.780303
1991-02-28	2049	2403.780303
1991-03-31	1874	2403.780303
1991-04-30	1279	2403.780303
1991-05-31	1432	2403.780303



RMSE and MAPE on training and testing Data

```
## Training Data - RMSE and MAPE

rmse_model3_train = metrics.mean_squared_error(train['Sparkling'],SimpleAverage_train['mean_forecast'],squared=False)
mape_model3_train = MAPE(train['Sparkling'],SimpleAverage_train['mean_forecast'])
print("For Simple Average Model forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model3_train, mape_model3_tra
```

```
For Simple Average Model forecast on the Training Data,  RMSE is 1298.484 MAPE is 40.36
```

```
## Test Data - RMSE and MAPE

rmse_model3_test = metrics.mean_squared_error(test['Sparkling'],SimpleAverage_test['mean_forecast'],squared=False)
mape_model3_test = MAPE(test['Sparkling'],SimpleAverage_test['mean_forecast'])
print("For Simple Average forecast on the Test Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model3_test, mape_model3_test))
```

```
For Simple Average forecast on the Test Data,  RMSE is 1275.082 MAPE is 38.90
```

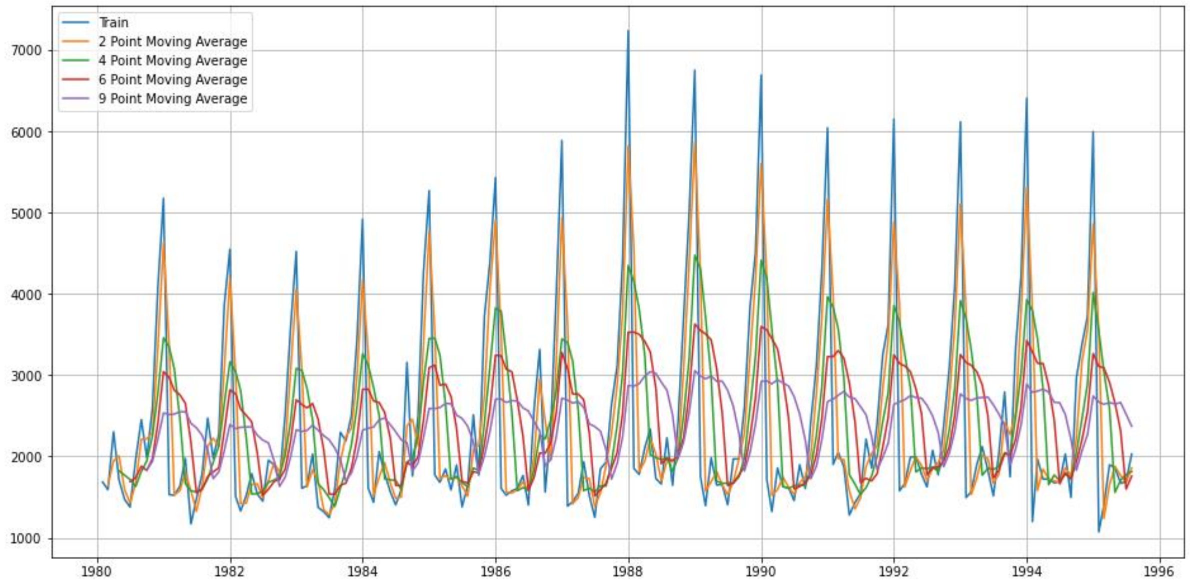
(d) Moving Average Model:

For the moving average model, we are going to calculate rolling means (or moving averages) for different intervals. The best interval can be determined by the maximum accuracy (or the minimum error) over here. For Moving Average, we are going to average over the entire data.

```
MovingAverage['Trailing_2'] = MovingAverage['Sparkling'].rolling(2).mean()
MovingAverage['Trailing_4'] = MovingAverage['Sparkling'].rolling(4).mean()
MovingAverage['Trailing_6'] = MovingAverage['Sparkling'].rolling(6).mean()
MovingAverage['Trailing_9'] = MovingAverage['Sparkling'].rolling(9).mean()

MovingAverage.head()
```

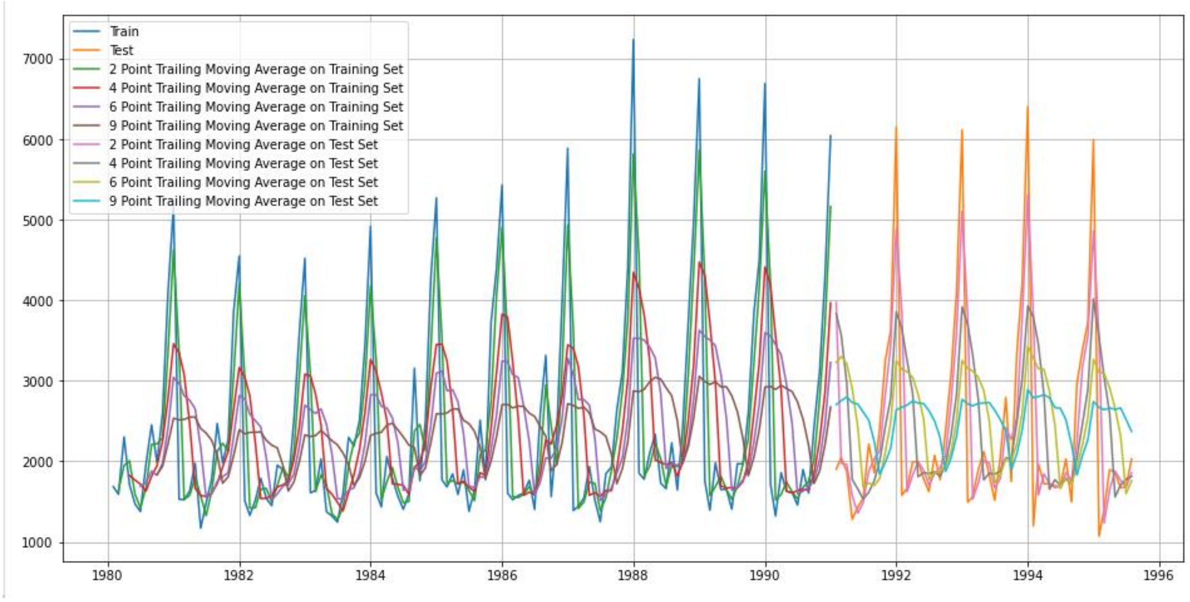
	Sparkling	Trailing_2	Trailing_4	Trailing_6	Trailing_9
Time_Stamp					
1980-01-31	1686	NaN	NaN	NaN	NaN
1980-02-29	1591	1638.5	NaN	NaN	NaN
1980-03-31	2304	1947.5	NaN	NaN	NaN
1980-04-30	1712	2008.0	1823.25	NaN	NaN
1980-05-31	1471	1591.5	1769.50	NaN	NaN

We can visualize the different moving averages over the whole dataset:



```
#Creating train and test set
trailing_MovingAverage_train=MovingAverage[0:int(len(MovingAverage)*0.71)]
trailing_MovingAverage_test=MovingAverage[int(len(MovingAverage)*0.71):]
```

We can visualize the averages on the training and testing data as shown below:



RMSE and MAPE on Testing Dataset:

```
For 2 point Moving Average Model forecast on the Testing Data,  RMSE is 813.401 MAPE is 19.70
For 4 point Moving Average Model forecast on the Testing Data,  RMSE is 1156.590 MAPE is 35.96
For 6 point Moving Average Model forecast on the Testing Data,  RMSE is 1283.927 MAPE is 43.86
For 9 point Moving Average Model forecast on the Testing Data,  RMSE is 1346.278 MAPE is 46.86
```

(e) <u>Simple Exponential Smoothing (automated):</u>

Single Exponential Smoothing, SES for short, also called Simple Exponential Smoothing, is a time series forecasting method for univariate data without a trend or seasonality. It requires a single parameter, called *alpha* (*a*), also called the smoothing factor or smoothing coefficient. This parameter controls the rate at which the influence of the observations at prior time steps decay exponentially. Alpha is often set to a value between 0 and 1.

A value close to 1 indicates fast learning (that is, only the most recent values influence the forecasts), whereas a value close to 0 indicates slow learning (past observations have a large influence on forecasts).

```
model_SES = SimpleExpSmoothing(SES_train['Sparkling'])
```

```
C:\Users\Anmol\AppData\Roaming\Python\Python37\site-packages\sta
information was provided, so inferred frequency M will be used.
  % freq, ValueWarning)
```

```
model_SES_autofit = model_SES.fit(optimized=True,use_brute=True)
```

We can get the model summary for the model. We can see the various parameters like AIC, BIC etc.

```
print(model_SES_autofit.summary())
```

```
                       SimpleExpSmoothing Model Results
==============================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:               SimpleExpSmoothing   SSE                      222559884.633
Optimized:                         True   AIC                           1896.603
Trend:                             None   BIC                           1902.369
Seasonal:                          None   AICC                          1896.918
Seasonal Periods:                  None   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                         05:10:03
Box-Cox Coeff.:                    None
==============================================================================
                       coeff                 code              optimized
------------------------------------------------------------------------------
smoothing_level            0.000000                alpha                 True
initial_level             2403.7856                  1.0                 True
------------------------------------------------------------------------------
```

We can also call the params to show us the model parameters:

```
model_SES_autofit.params
```

```
{'smoothing_level': 0.0,
 'smoothing_slope': nan,
 'smoothing_seasonal': nan,
 'damping_slope': nan,
 'initial_level': 2403.785621547663,
 'initial_slope': nan,
 'initial_seasons': array([], dtype=float64),
 'use_boxcox': False,
 'lamda': None,
 'remove_bias': False}
```

We can see that automated Simple Exponential Smoothing chose alpha as 0. We can predict on training and testing data using this model:

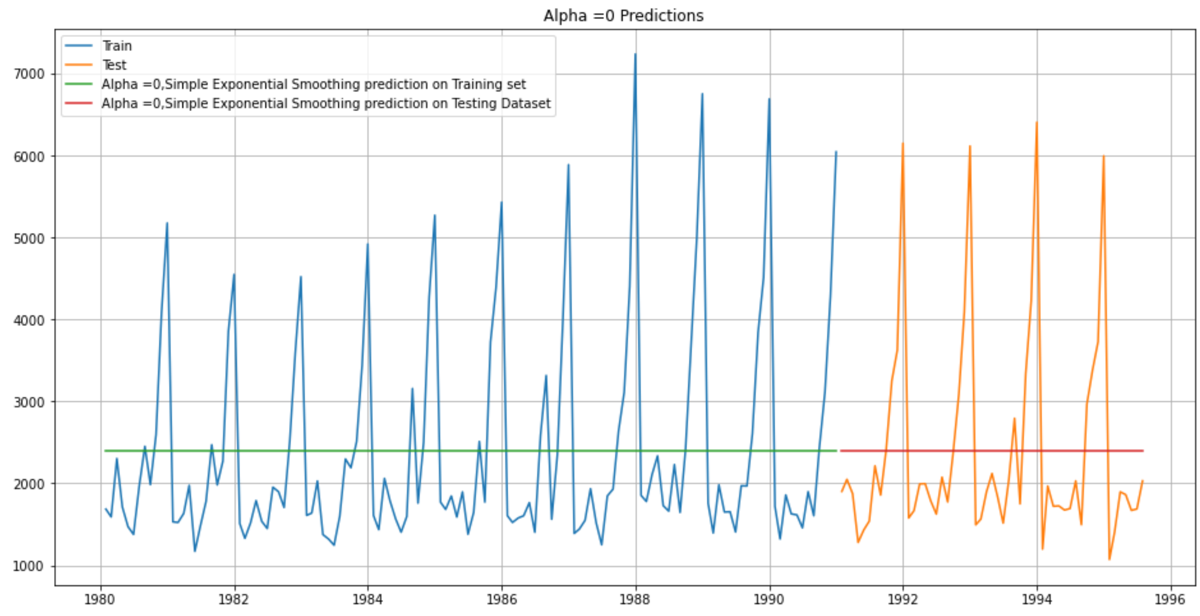
```
SES_train['predict'] = model_SES_autofit.fittedvalues
SES_train.head()
```

	Sparkling	predict
Time_Stamp		
1980-01-31	1686	2403.785622
1980-02-29	1591	2403.785622
1980-03-31	2304	2403.785622
1980-04-30	1712	2403.785622
1980-05-31	1471	2403.785622

```
SES_test['predict'] = model_SES_autofit.forecast(steps=len(SES_test))
SES_test.head()
```

	Sparkling	predict
Time_Stamp		
1991-01-31	1902	2403.785622
1991-02-28	2049	2403.785622
1991-03-31	1874	2403.785622
1991-04-30	1279	2403.785622
1991-05-31	1432	2403.785622

We can visualize our predictions on the training and testing data:



Since, SES model captures only the level, the forecast is a straight line.
We can check the RMSE and MAPE on the training and testing data too:

```
## Training Data

rmse_model5_train = metrics.mean_squared_error(SES_train['Sparkling'],SES_train['predict'],squared=False)
mape_model5_train = MAPE(SES_train['Sparkling'],SES_train['predict'])
print("For Alpha =0 Simple Exponential Smoothing Model forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model5
```

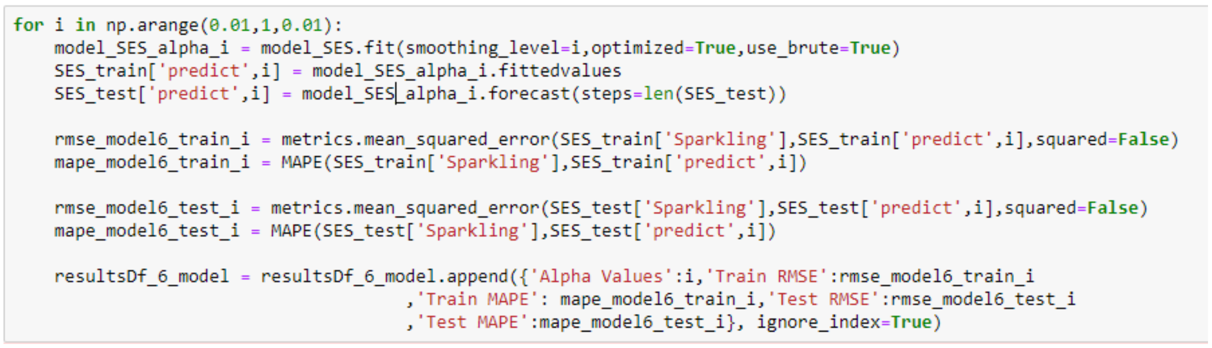
```
For Alpha =0 Simple Exponential Smoothing Model forecast on the Training Data,  RMSE is 1298.484 MAPE is 40.36
```

```
## Test Data

rmse_model5_test = metrics.mean_squared_error(SES_test['Sparkling'],SES_test['predict'],squared=False)
mape_model5_test = MAPE(SES_test['Sparkling'],SES_test['predict'])
print("For Alpha =0 Simple Exponential Smoothing Model forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f" %(rmse_model5
```

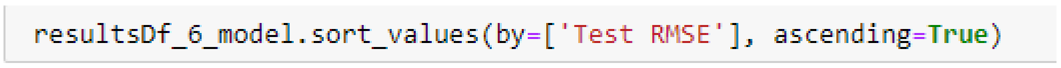
```
For Alpha =0 Simple Exponential Smoothing Model forecast on the Training Data,  RMSE is 1275.082 MAPE is 38.90
```

(f) <u>SIMPLE EXPONENTIAL SMOOTHING with a range of values:</u>

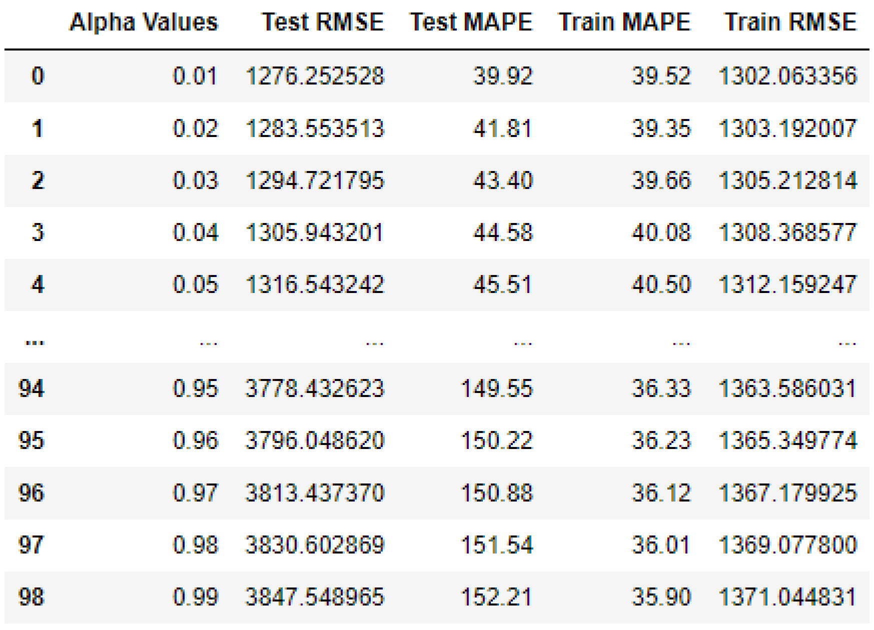
In the earlier model we simply let the model figure out the most suitable values of alpha. Now we will pass a range of alpha values and see which one gives the lowest RMSE on the testing data:



```
for i in np.arange(0.01,1,0.01):
    model_SES_alpha_i = model_SES.fit(smoothing_level=i,optimized=True,use_brute=True)
    SES_train['predict',i] = model_SES_alpha_i.fittedvalues
    SES_test['predict',i] = model_SES_alpha_i.forecast(steps=len(SES_test))

    rmse_model6_train_i = metrics.mean_squared_error(SES_train['Sparkling'],SES_train['predict',i],squared=False)
    mape_model6_train_i = MAPE(SES_train['Sparkling'],SES_train['predict',i])

    rmse_model6_test_i = metrics.mean_squared_error(SES_test['Sparkling'],SES_test['predict',i],squared=False)
    mape_model6_test_i = MAPE(SES_test['Sparkling'],SES_test['predict',i])

    resultsDf_6_model = resultsDf_6_model.append({'Alpha Values':i,'Train RMSE':rmse_model6_train_i
                                  ,'Train MAPE': mape_model6_train_i,'Test RMSE':rmse_model6_test_i
                                  ,'Test MAPE':mape_model6_test_i}, ignore_index=True)
```

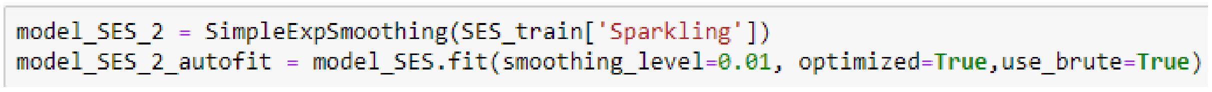


```
resultsDf_6_model.sort_values(by=['Test RMSE'], ascending=True)
```



	Alpha Values	Test RMSE	Test MAPE	Train MAPE	Train RMSE
0	0.01	1276.252528	39.92	39.52	1302.063356
1	0.02	1283.553513	41.81	39.35	1303.192007
2	0.03	1294.721795	43.40	39.66	1305.212814
3	0.04	1305.943201	44.58	40.08	1308.368577
4	0.05	1316.543242	45.51	40.50	1312.159247
...	...	...	...	...	...
94	0.95	3778.432623	149.55	36.33	1363.586031
95	0.96	3796.048620	150.22	36.23	1365.349774
96	0.97	3813.437370	150.88	36.12	1367.179925
97	0.98	3830.602869	151.54	36.01	1369.077800
98	0.99	3847.548965	152.21	35.90	1371.044831

Sorting the results by Test RMSE, we can see that alpha = 0.01 gives the lowest value of RMSE, so we will choose alpha= 0.01. now, we can create a SES model by using these models and het the statistical summary:

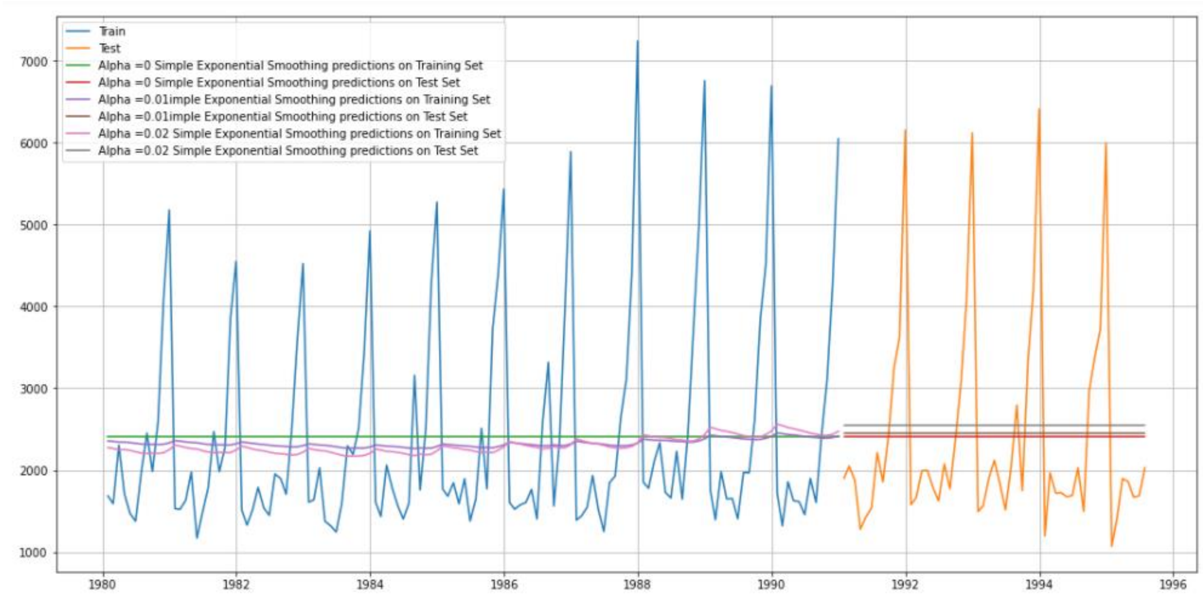


```
model_SES_2 = SimpleExpSmoothing(SES_train['Sparkling'])
model_SES_2_autofit = model_SES.fit(smoothing_level=0.01, optimized=True,use_brute=True)
```

```
print(model_SES_2_autofit.summary())
```

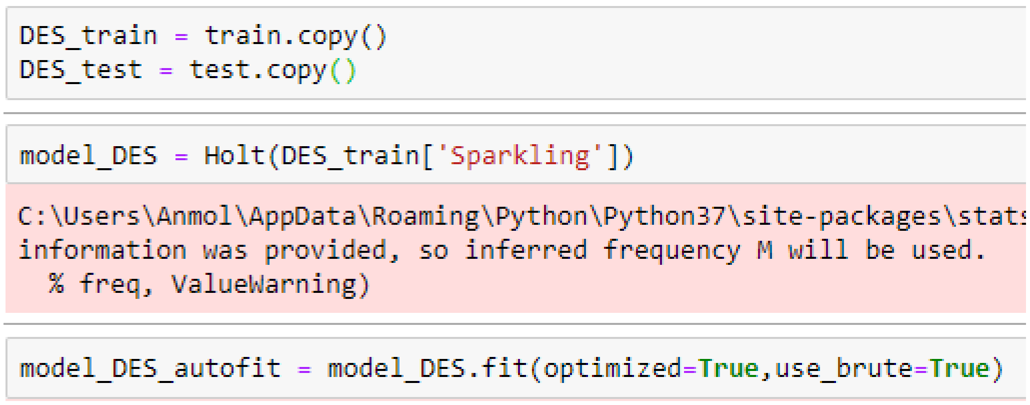
```
                       SimpleExpSmoothing Model Results
==============================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:               SimpleExpSmoothing   SSE                      223788705.861
Optimized:                         True   AIC                           1897.330
Trend:                             None   BIC                           1903.096
Seasonal:                          None   AICC                          1897.645
Seasonal Periods:                  None   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                         05:10:12
Box-Cox Coeff.:                    None
==============================================================================
                       coeff                 code              optimized
------------------------------------------------------------------------------
smoothing_level            0.0100000                alpha                False
initial_level              2357.7864                  1.0                 True
------------------------------------------------------------------------------
```

We can plot the various SES models on test and train data and compare them:



(g) <u>DOUBLE EXPONENTIAL SMOOTHING MODEL (automated)</u>

Double Exponential Smoothing is an extension to Exponential Smoothing that explicitly adds support for trends in the univariate time series. In addition to the *alpha* parameter for controlling smoothing factor for the level, an additional smoothing factor is added to control the decay of the influence of the change in trend called *beta* (*b*). The method supports trends that change in different ways: an additive and a multiplicative, depending on whether the trend is linear or exponential respectively. Double Exponential Smoothing with an additive trend is classically referred to as Holt’s linear trend model.



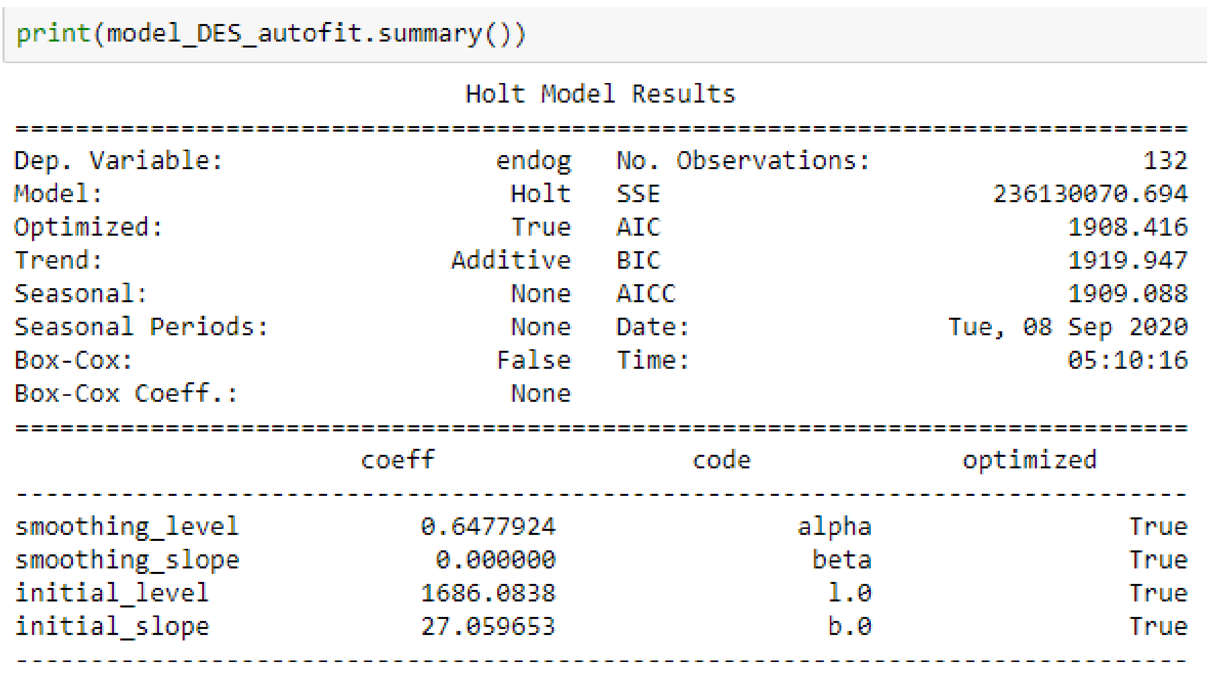
```
DES_train = train.copy()
DES_test = test.copy()
```

```
model_DES = Holt(DES_train['Sparkling'])
```

```
C:\Users\Anmol\AppData\Roaming\Python\Python37\site-packages\stat
information was provided, so inferred frequency M will be used.
  % freq, ValueWarning)
```

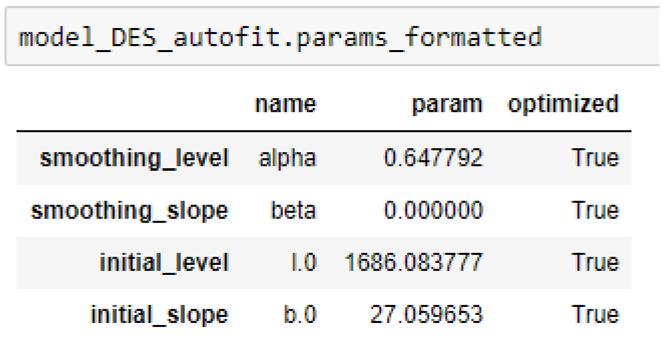
```
model_DES_autofit = model_DES.fit(optimized=True,use_brute=True)
```

we can see the statsmodels summary and the fitted parameters for the Double Exponential Summary:



```
print(model_DES_autofit.summary())
```

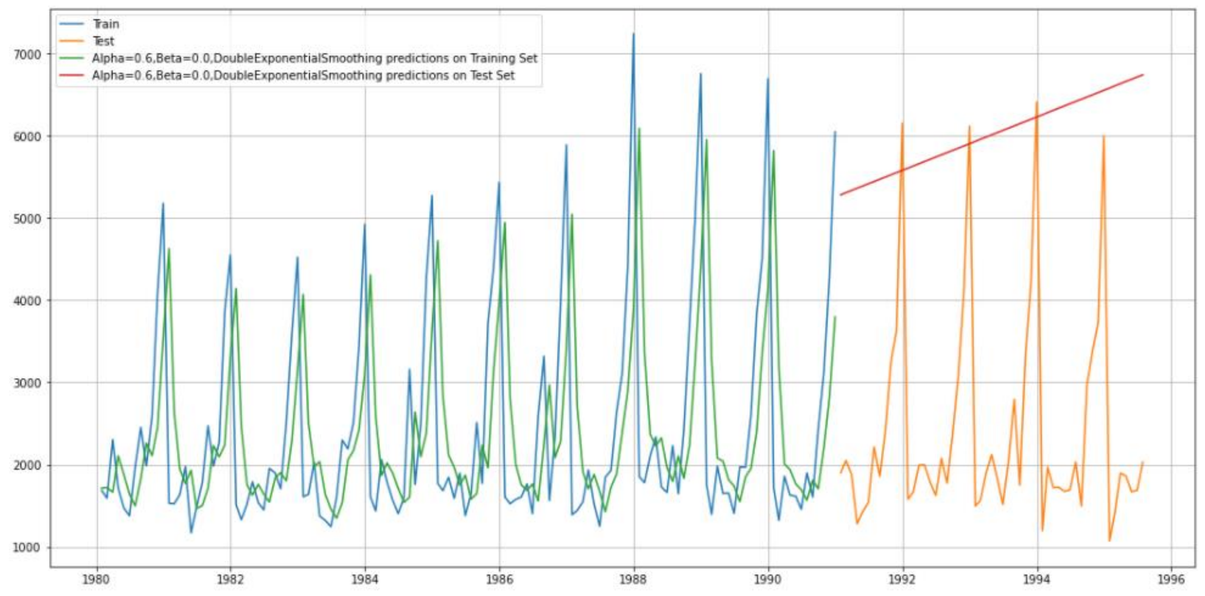
```
                              Holt Model Results
================================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:                             Holt   SSE                      236130070.694
Optimized:                         True   AIC                           1908.416
Trend:                         Additive   BIC                           1919.947
Seasonal:                          None   AICC                          1909.088
Seasonal Periods:                  None   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                         05:10:16
Box-Cox Coeff.:                    None
================================================================================
                       coeff                 code              optimized
--------------------------------------------------------------------------------
smoothing_level            0.6477924                alpha                 True
smoothing_slope             0.000000                 beta                 True
initial_level              1686.0838                  l.0                 True
initial_slope              27.059653                  b.0                 True
--------------------------------------------------------------------------------
```



```
model_DES_autofit.params_formatted
```

	name	param	optimized
smoothing_level	alpha	0.647792	True
smoothing_slope	beta	0.000000	True
initial_level	l.0	1686.083777	True
initial_slope	b.0	27.059653	True

We can use this model to make predictions on the training data and the testing data. After making predictions on the test and train data we can visualize the prediction as shown below:



RMSE and MAPE on test and train data:

```
## Training Data

rmse_model7_train = metrics.mean_squared_error(DES_train['Sparkling'],DES_train['predict',0.64,0.0],squared=False)
mape_model7_train = MAPE(DES_train['Sparkling'],DES_train['predict',0.64,0.0])
print("For Alpha=0.64 and Beta=0 Double Exponential Smoothing Model forecast on the Training Data,  RMSE is %3.3f MAPE is %3.2f"
```

```
For Alpha=0.64 and Beta=0 Double Exponential Smoothing Model forecast on the Training Data,  RMSE is 1337.484 MAPE is 39.11
```

```
## Test Data

rmse_model7_test = metrics.mean_squared_error(DES_test['Sparkling'],DES_test['predict',0.64,0.0],squared=False)
mape_model7_test = MAPE(DES_test['Sparkling'],DES_test['predict',0.64,0.0])
print("For Alpha=0.64 and Beta=0 Double Exponential Smoothing Model forecast on the Testing Data,  RMSE is %3.3f MAPE is %3.2f"
```

(h) <u>DOUBLE EXPONENTIAL SMOOTHING MODEL with automated values:</u>

Again we pass a values in range of values from 0.3 to 1.1 and see which give the lowest value of RMSE.

```
for i in np.arange(0.3,1.1,0.1):
    for j in np.arange(0.3,1.1,0.1):
        model_DES_alpha_i_beta_j = model_DES.fit(smoothing_level=i,smoothing_slope=j,optimized=True,use_brute=True)
        DES_train['predict',i,j] = model_DES_alpha_i_beta_j.fittedvalues
        DES_test['predict',i,j] = model_DES_alpha_i_beta_j.forecast(steps=len(DES_test))

        rmse_model8_train = metrics.mean_squared_error(DES_train['Sparkling'],DES_train['predict',i,j],squared=False)
        mape_model8_train = MAPE(DES_train['Sparkling'],DES_train['predict',i,j])

        rmse_model8_test = metrics.mean_squared_error(DES_test['Sparkling'],DES_test['predict',i,j],squared=False)
        mape_model8_test = MAPE(DES_test['Sparkling'],DES_test['predict',i,j])

        resultsDf_8 = resultsDf_8.append({'Alpha Values':i,'Beta Values':j,'Train RMSE':rmse_model8_train
                                          ,'Train MAPE': mape_model8_train,'Test RMSE':rmse_model8_test
                                          ,'Test MAPE':mape_model8_test}, ignore_index=True)
```

When we sort the obtained result by RMSE, we can see that the best values are equal to alpha = 0.3 and beta = 0.3.

```
resultsDf_8.sort_values(by=['Test RMSE'], ascending=True)
```

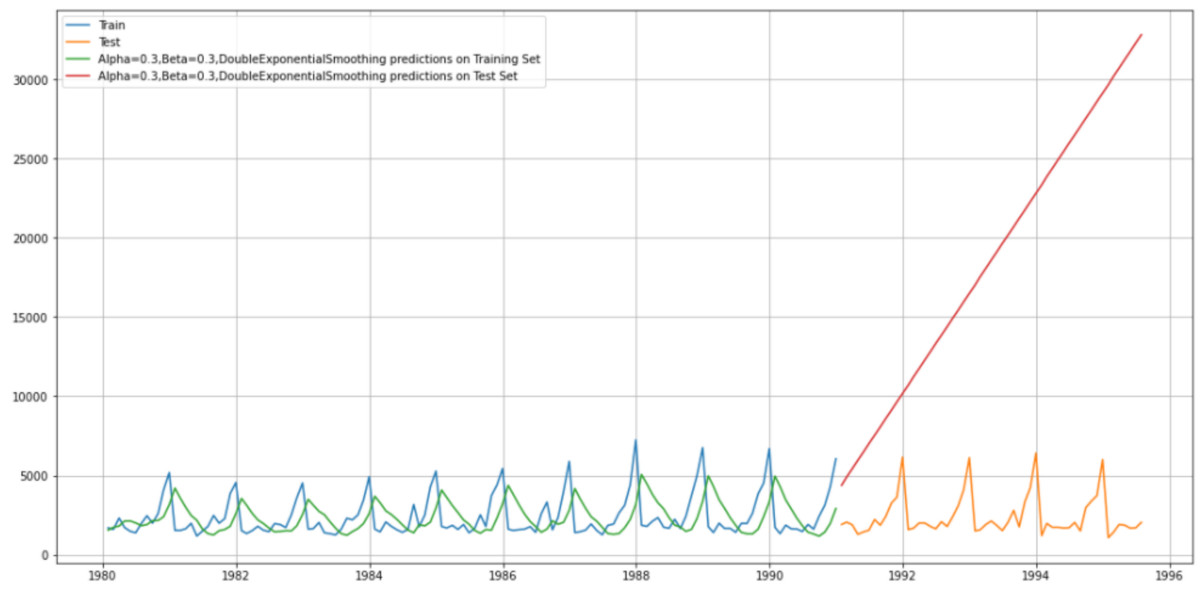
	Alpha Values	Beta Values	Train RMSE	Train MAPE	Test RMSE	Test MAPE
0	0.3	0.3	1590.151688	53.80	18259.110704	675.28
8	0.4	0.3	1568.527728	50.15	23878.496940	886.00
1	0.3	0.4	1681.706138	57.15	26069.841401	960.18
16	0.5	0.3	1530.223987	45.98	27095.532414	1007.39
24	0.6	0.3	1506.223120	42.82	29070.722592	1082.18
...	...	...	...	...	...	...
39	0.7	1.0	1816.954860	47.07	57297.154185	2111.47
62	1.0	0.9	1985.351411	52.00	57823.177011	2132.75
47	0.8	1.0	1872.674084	49.74	57990.117908	2137.46
55	0.9	1.0	1948.013119	52.19	59008.254331	2175.12
63	1.0	1.0	2077.647495	53.78	59877.076519	2207.28

We can create a model based on these values of alpha and beta and see the model summary.

```
print(Holt_model_results.summary())
```

```
                              Holt Model Results                              
==============================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:                             Holt   SSE                      333772875.409
Optimized:                         True   AIC                           1954.098
Trend:                         Additive   BIC                           1965.630
Seasonal:                          None   AICC                          1954.770
Seasonal Periods:                  None   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                         05:10:29
Box-Cox Coeff.:                    None                                         
==============================================================================
                       coeff                 code              optimized      
------------------------------------------------------------------------------
smoothing_level            0.3000000                alpha                False
smoothing_slope            0.3000000                 beta                False
initial_level              1426.2791                  l.0                 True
initial_slope              119.75243                  b.0                 True
------------------------------------------------------------------------------
```

We can also plot predicted values from this model on the training and testing dataset:



(i) <u>TRIPLE EXPONENTIAL SMOOTHING MODEL automated:</u>

Triple Exponential Smoothing is an extension of Exponential Smoothing that explicitly adds support for seasonality to the univariate time series. This method is sometimes called Holt-Winters Exponential Smoothing, named for two contributors to the method: Charles Holt and Peter Winters. In addition to the alpha and beta smoothing factors, a new parameter is added called *gamma* (*g*) that controls the influence on the seasonal component. As with the trend, the seasonality may be modelled as either an additive or multiplicative process for a linear or exponential change in the seasonality. Triple-exponential smoothing is the most advanced variation of exponential smoothing and through configuration, it can also develop double and single exponential smoothing models.

```
TES_train = train.copy()
TES_test = test.copy()
```

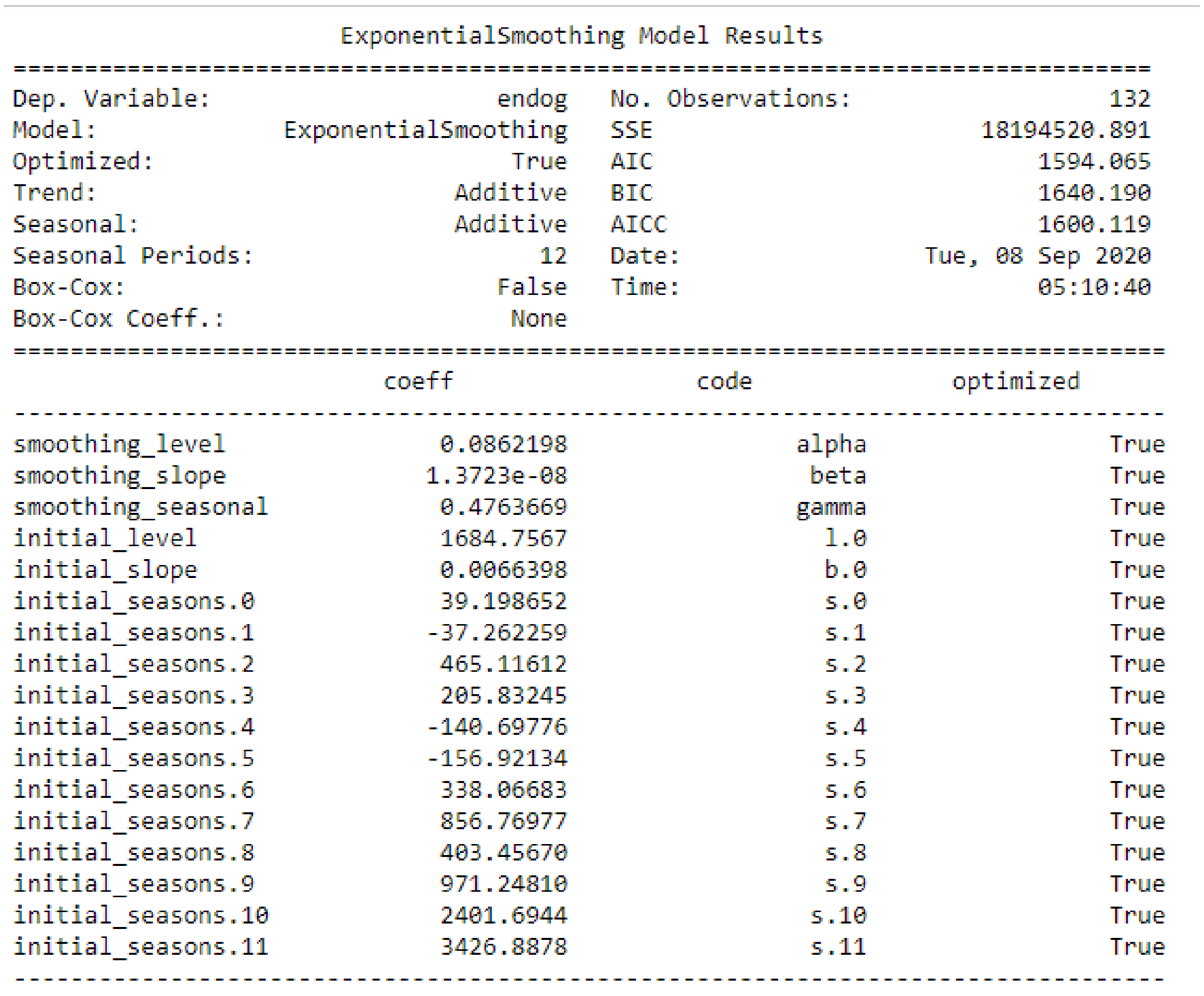
```
model_TES = ExponentialSmoothing(TES_train['Sparkling'], freq='M', trend='additive',seasonal='additive')
```

```
model_TES_autofit = model_TES.fit(optimized=True,use_brute=True)
```

We can see the summary of this model and the fitted parameters:

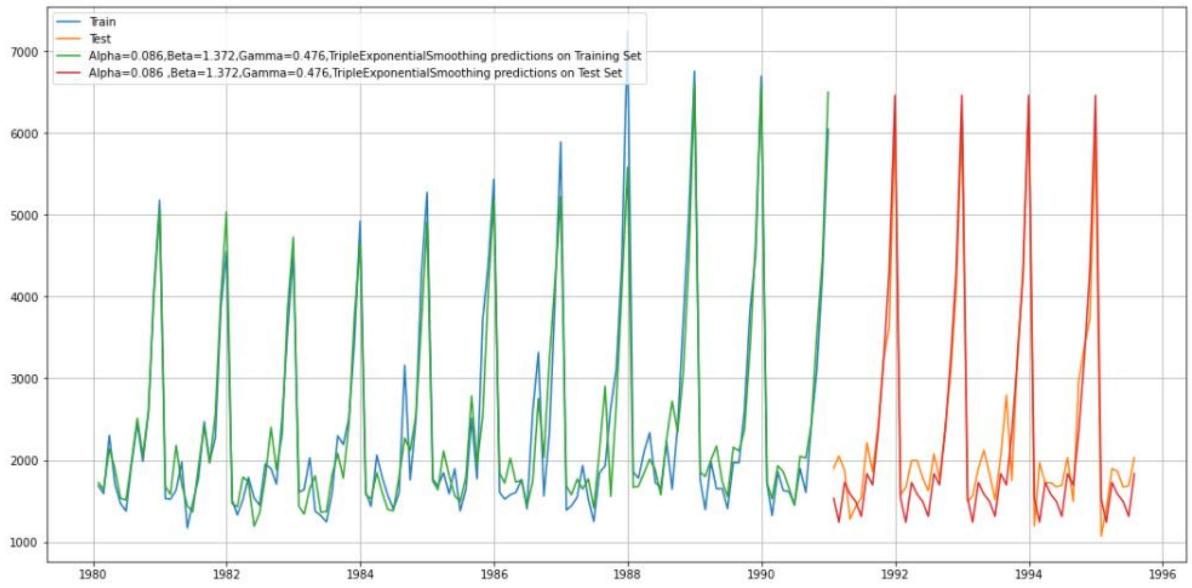
```
model_TES_autofit.params
```

```
{'smoothing_level': 0.08621976457712728,
 'smoothing_slope': 1.3722820308989836e-08,
 'smoothing_seasonal': 0.4763668704627969,
 'damping_slope': nan,
 'initial_level': 1684.7567371537205,
 'initial_slope': 0.00663980720702693,
 'initial_seasons': array([  39.19865194,   -37.26225944,   465.11612145,   205.83244815,
        -140.69775927, -156.92133674,   338.06682948,   856.76977123,
         403.45669741,   971.24810067, 2401.69439277, 3426.88784686]),
 'use_boxcox': False,
 'lamda': None,
 'remove_bias': False}
```

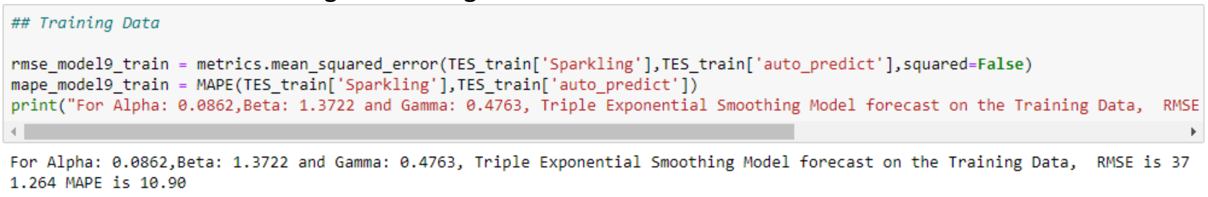


```
                       ExponentialSmoothing Model Results
================================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:             ExponentialSmoothing   SSE                       18194520.891
Optimized:                         True   AIC                           1594.065
Trend:                         Additive   BIC                           1640.190
Seasonal:                      Additive   AICC                          1600.119
Seasonal Periods:                    12   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                         05:10:40
Box-Cox Coeff.:                    None
=================================================================================
                          coeff                 code              optimized
---------------------------------------------------------------------------------
smoothing_level           0.0862198                alpha                 True
smoothing_slope          1.3723e-08                 beta                 True
smoothing_seasonal        0.4763669                gamma                 True
initial_level             1684.7567                  l.0                 True
initial_slope             0.0066398                  b.0                 True
initial_seasons.0         39.198652                  s.0                 True
initial_seasons.1        -37.262259                  s.1                 True
initial_seasons.2         465.11612                  s.2                 True
initial_seasons.3         205.83245                  s.3                 True
initial_seasons.4        -140.69776                  s.4                 True
initial_seasons.5        -156.92134                  s.5                 True
initial_seasons.6         338.06683                  s.6                 True
initial_seasons.7         856.76977                  s.7                 True
initial_seasons.8         403.45670                  s.8                 True
initial_seasons.9         971.24810                  s.9                 True
initial_seasons.10        2401.6944                 s.10                 True
initial_seasons.11        3426.8878                 s.11                 True
---------------------------------------------------------------------------------
```

We can use this model to make predictions on the test and train data. We can plot those predictions on the whole dataset to visualize the predictions:

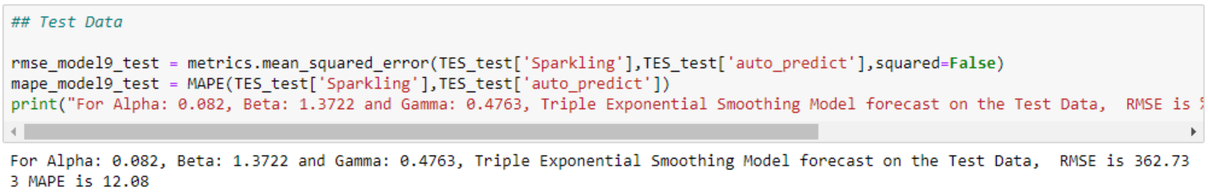


RMSE and MAPE on training and testing data:



```
## Training Data

rmse_model9_train = metrics.mean_squared_error(TES_train['Sparkling'],TES_train['auto_predict'],squared=False)
mape_model9_train = MAPE(TES_train['Sparkling'],TES_train['auto_predict'])
print("For Alpha: 0.0862,Beta: 1.3722 and Gamma: 0.4763, Triple Exponential Smoothing Model forecast on the Training Data,  RMSE
```

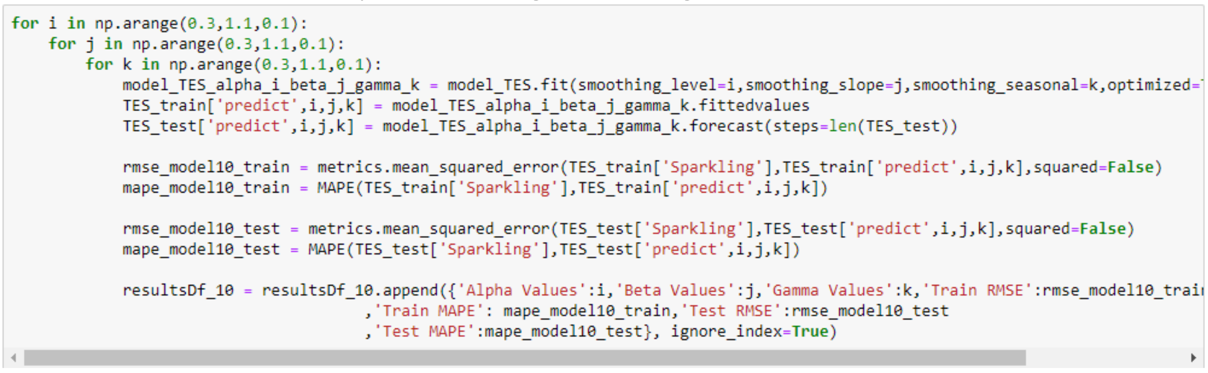
```
For Alpha: 0.0862,Beta: 1.3722 and Gamma: 0.4763, Triple Exponential Smoothing Model forecast on the Training Data,  RMSE is 37
1.264 MAPE is 10.90
```



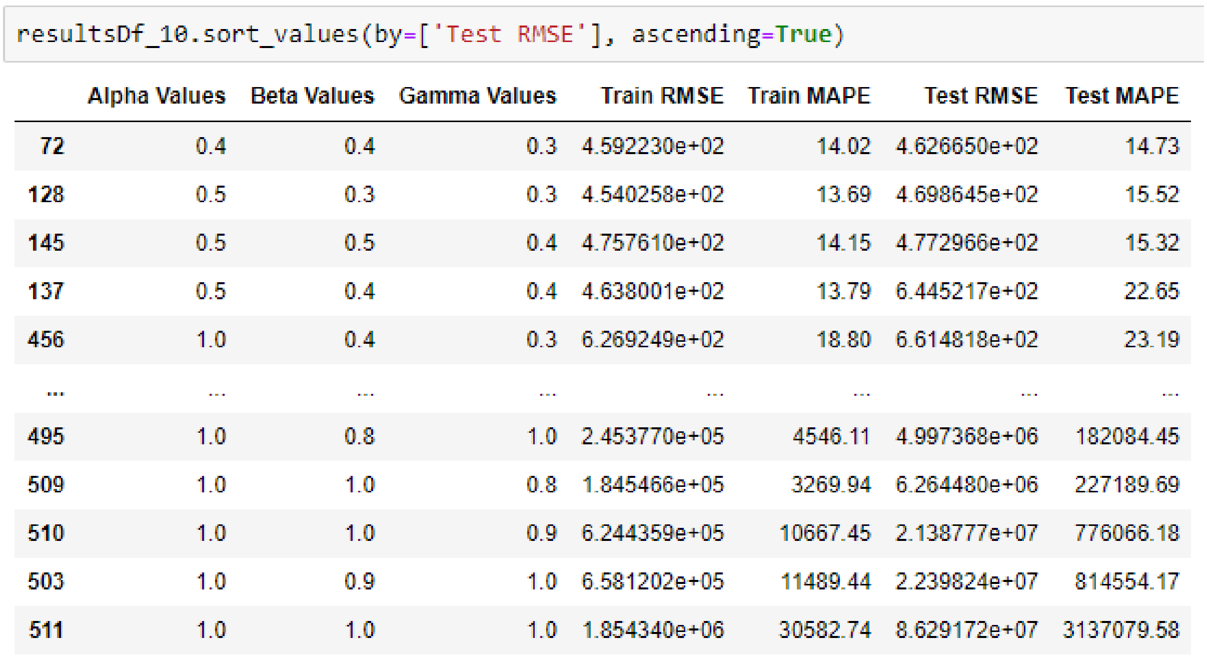
```
## Test Data

rmse_model9_test = metrics.mean_squared_error(TES_test['Sparkling'],TES_test['auto_predict'],squared=False)
mape_model9_test = MAPE(TES_test['Sparkling'],TES_test['auto_predict'])
print("For Alpha: 0.082, Beta: 1.3722 and Gamma: 0.4763, Triple Exponential Smoothing Model forecast on the Test Data,  RMSE is
```

```
For Alpha: 0.082, Beta: 1.3722 and Gamma: 0.4763, Triple Exponential Smoothing Model forecast on the Test Data,  RMSE is 362.73
3 MAPE is 12.08
```

(j) **TRIPLE EXPONENTIAL SMOOTHING MODEL with a range of values:**
Now we will build a TES model where will pass a range of values from 0.3 to 1.1. We will choose the value of alpha, beta and gamma that give the lowest value of RMSE:



```
for i in np.arange(0.3,1.1,0.1):
    for j in np.arange(0.3,1.1,0.1):
        for k in np.arange(0.3,1.1,0.1):
            model_TES_alpha_i_beta_j_gamma_k = model_TES.fit(smoothing_level=i,smoothing_slope=j,smoothing_seasonal=k,optimized=
            TES_train['predict',i,j,k] = model_TES_alpha_i_beta_j_gamma_k.fittedvalues
            TES_test['predict',i,j,k] = model_TES_alpha_i_beta_j_gamma_k.forecast(steps=len(TES_test))

            rmse_model10_train = metrics.mean_squared_error(TES_train['Sparkling'],TES_train['predict',i,j,k],squared=False)
            mape_model10_train = MAPE(TES_train['Sparkling'],TES_train['predict',i,j,k])

            rmse_model10_test = metrics.mean_squared_error(TES_test['Sparkling'],TES_test['predict',i,j,k],squared=False)
            mape_model10_test = MAPE(TES_test['Sparkling'],TES_test['predict',i,j,k])

            resultsDf_10 = resultsDf_10.append({'Alpha Values':i,'Beta Values':j,'Gamma Values':k,'Train RMSE':rmse_model10_trai
                                              ,'Train MAPE': mape_model10_train,'Test RMSE':rmse_model10_test
                                              ,'Test MAPE':mape_model10_test}, ignore_index=True)
```

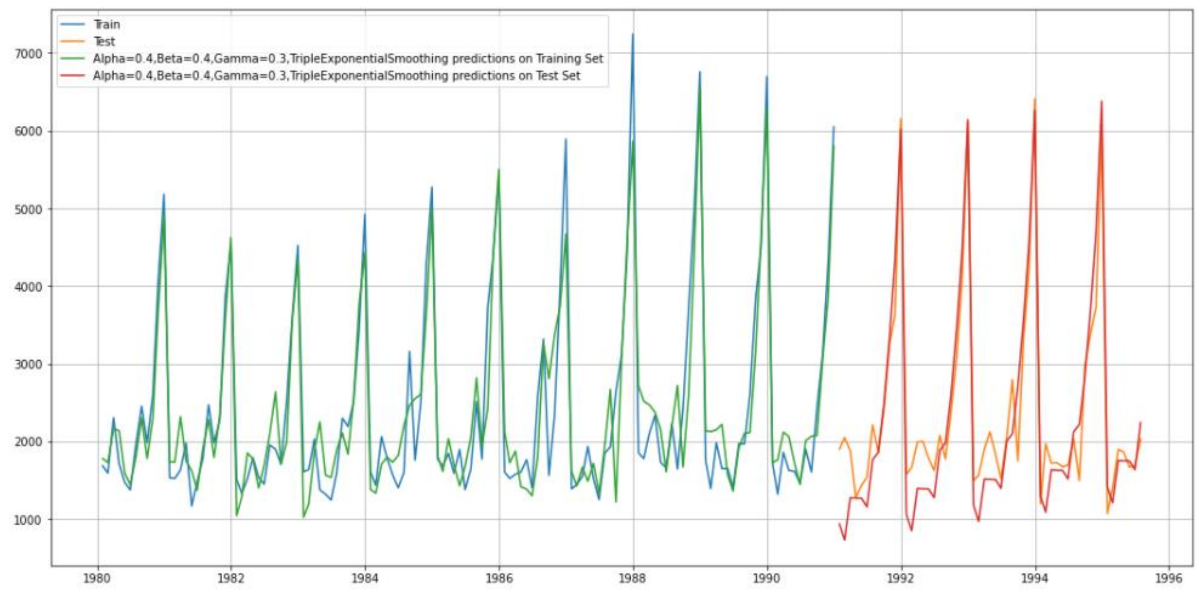


```
resultsDf_10.sort_values(by=['Test RMSE'], ascending=True)
```

	Alpha Values	Beta Values	Gamma Values	Train RMSE	Train MAPE	Test RMSE	Test MAPE
72	0.4	0.4	0.3	4.592230e+02	14.02	4.626650e+02	14.73
128	0.5	0.3	0.3	4.540258e+02	13.69	4.698645e+02	15.52
145	0.5	0.5	0.4	4.757610e+02	14.15	4.772966e+02	15.32
137	0.5	0.4	0.4	4.638001e+02	13.79	6.445217e+02	22.65
456	1.0	0.4	0.3	6.269249e+02	18.80	6.614818e+02	23.19
...	...	...	...	...	...	...	...
495	1.0	0.8	1.0	2.453770e+05	4546.11	4.997368e+06	182084.45
509	1.0	1.0	0.8	1.845466e+05	3269.94	6.264480e+06	227189.69
510	1.0	1.0	0.9	6.244359e+05	10667.45	2.138777e+07	776066.18
503	1.0	0.9	1.0	6.581202e+05	11489.44	2.239824e+07	814554.17
511	1.0	1.0	1.0	1.854340e+06	30582.74	8.629172e+07	3137079.58

When we sort the result by lowest values of TEST RMSE, we can see that the model gave lowest value for error at alpha=0.4, beta=0.4 and gamma=0.3. We can create a model with these values and see the model summary. We can also plot the predicted values from these values of alpha, beta and gamma on the training and testing dataset:

```
                        ExponentialSmoothing Model Results
================================================================================
Dep. Variable:                    endog   No. Observations:                  132
Model:             ExponentialSmoothing   SSE                     27836920.016
Optimized:                         True   AIC                         1650.197
Trend:                         Additive   BIC                         1696.322
Seasonal:                      Additive   AICC                        1656.251
Seasonal Periods:                    12   Date:                 Tue, 08 Sep 2020
Box-Cox:                          False   Time:                       05:11:26
Box-Cox Coeff.:                    None
=================================================================================
                          coeff                 code               optimized
---------------------------------------------------------------------------------
smoothing_level           0.4000000                alpha                False
smoothing_slope           0.4000000                 beta                False
smoothing_seasonal        0.3000000                gamma                False
initial_level             1923.4097                  l.0                 True
initial_slope              0.000000                  b.0                 True
initial_seasons.0        -145.42457                  s.0                 True
initial_seasons.1        -144.84149                  s.1                 True
initial_seasons.2         387.56430                  s.2                 True
initial_seasons.3         314.77666                  s.3                 True
initial_seasons.4         21.124031                  s.4                 True
initial_seasons.5         25.144102                  s.5                 True
initial_seasons.6         522.87689                  s.6                 True
initial_seasons.7         1043.6841                  s.7                 True
initial_seasons.8         523.78514                  s.8                 True
initial_seasons.9         987.48549                  s.9                 True
initial_seasons.10        2256.0583                 s.10                 True
initial_seasons.11        3230.1024                 s.11                 True
---------------------------------------------------------------------------------
```



5. Check for the stationarity of the data on which the model is being built on using appropriate statistical tests and also mention the hypothesis for the statistical test. If the data is found to be non-stationary, take appropriate steps to make it stationary. Check the new data for stationarity and comment.
   Note: Stationarity should be checked at alpha = 0.05.

A stationary time series is one whose properties do not depend on the time at which the series is observed. The series will not have any predictable pattern. A stationary time series is not constant, but has a constant mean and variance across time. A stationary series can also be thought of as one which looks the same whether we look in forward direction of time or backward direction of time. It has no directionality to it.
We want a stationary time series because we want to use statistical methods to estimate parameters and we can only do so if we have observations that are similar or can be thought of as coming from the same process. We want to get a stationary time series because we want to use the power of repetitiveness to make predictions.
To check for stationarity, we will use the Augmented Dickey fuller Test or ADF test. The null hypothesis and alternate hypothesis for this test are as mentioned below:

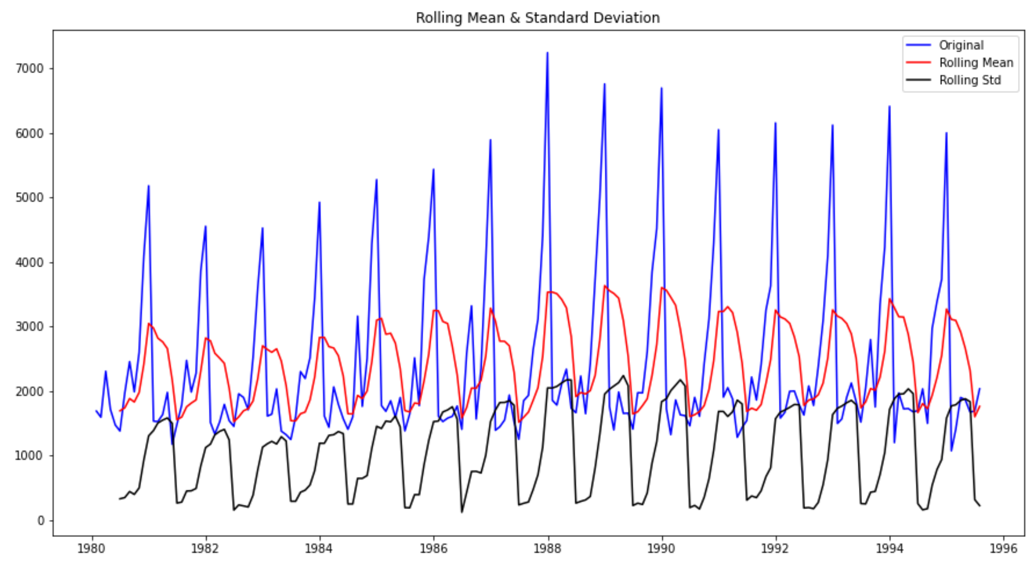
- The null hypothesis for ADF test (H0) is that the time series is non-stationary
- The alternate hypothesis for ADF test (H1) is that time series is stationary

To perform the stationarity test, we will import the adfuller from statstools. We will define a custom function that plots the time series, rolling mean, standard deviation and also give test statistics:

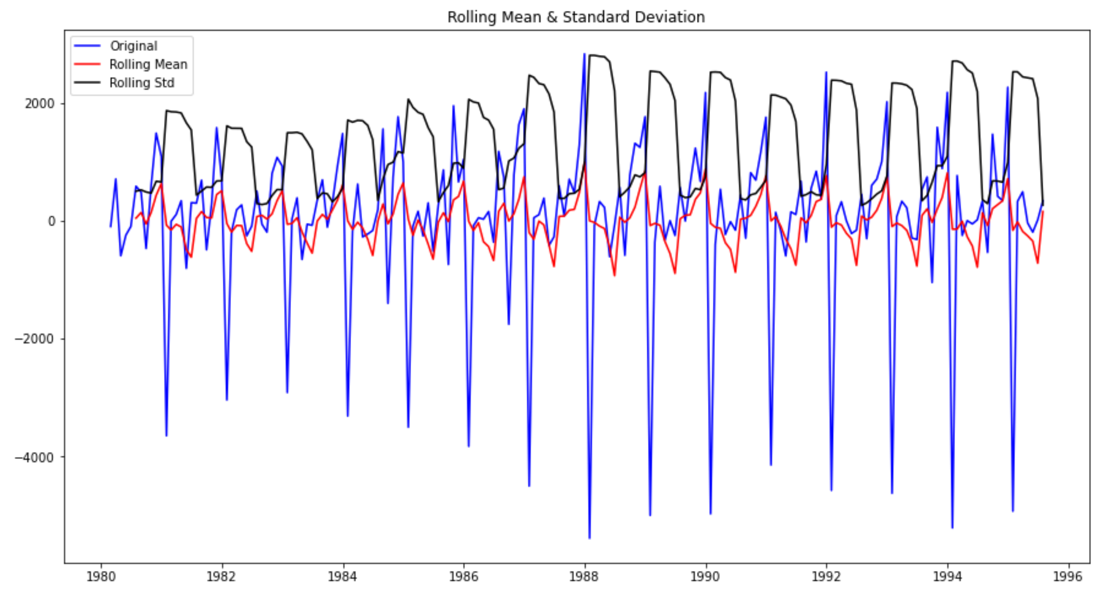
```
## Test for stationarity of the series - Dicky Fuller test

from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = timeseries.rolling(window=6).mean() #determining the rolling mean
    rolstd = timeseries.rolling(window=6).std()   #determining the rolling standard deviation

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test:
    print ('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput,'\n')
```

The test statistics and plot from the above code is shown below:

```
Results of Dickey-Fuller Test:
Test Statistic                  -1.360497
p-value                          0.601061
#Lags Used                      11.000000
Number of Observations Used    175.000000
Critical Value (1%)             -3.468280
Critical Value (5%)             -2.878202
Critical Value (10%)            -2.575653
dtype: float64
```



We can see from the above results that the p-value is equal to 0.601 and test statistic is equal to -1.360. Since, the p-values is greater than 0.05 (alpha value) and test statistic is greater than the critical value at 0.05 so we fail reject the null hypothesis and thus the given time series is non-stationary. To make this stationary, we will take difference of order 1 and reperform the test. The results for that are shown below:



```
Results of Dickey-Fuller Test:
Test Statistic                 -45.050301
p-value                          0.000000
#Lags Used                      10.000000
Number of Observations Used    175.000000
Critical Value (1%)             -3.468280
Critical Value (5%)             -2.878202
Critical Value (10%)            -2.575653
dtype: float64
```

We can see that the p-value is now less than 0.05 and the test statistics is lower than the critical value. Thus, the time series becomes stationary after taking differencing of order 1.

6. Build an automated version of the ARIMA/SARIMA model in which the parameters are selected using the lowest Akaike Information Criteria (AIC) on the training data and evaluate this model on the test data using RMSE.

ARIMA MODEL:

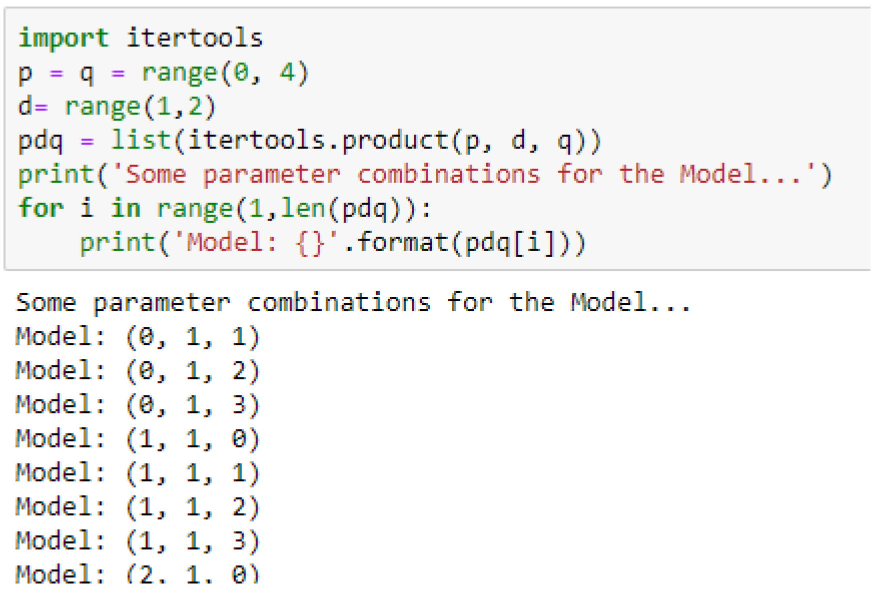
ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: Autoregression. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I**: Integrated. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: Moving Average. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.
  Each of these components are explicitly specified in the model as a parameter. A standard notation is used of ARIMA(p,d,q) where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

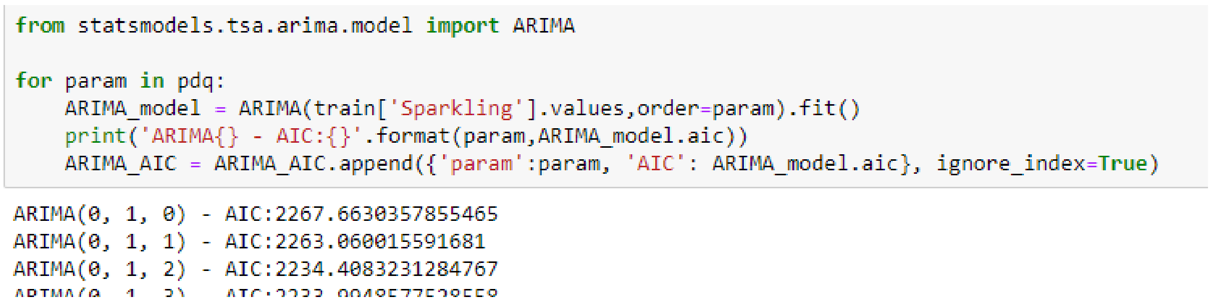
- **p**: The number of lag observations included in the model, also called the lag order.
- **d**: The number of times that the raw observations are differenced, also called the degree of differencing.
- **q**: The size of the moving average window, also called the order of moving average.
  A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

To build an automated version of ARIMA we will pass a range of values for p, d and q. We will construct various models and also compute the AIC of each model. The values of p, d & q which give the lowest value of AIC will be chosen as the final model.



```python
import itertools
p = q = range(0, 4)
d= range(1,2)
pdq = list(itertools.product(p, d, q))
print('Some parameter combinations for the Model...')
for i in range(1,len(pdq)):
    print('Model: {}'.format(pdq[i]))
```

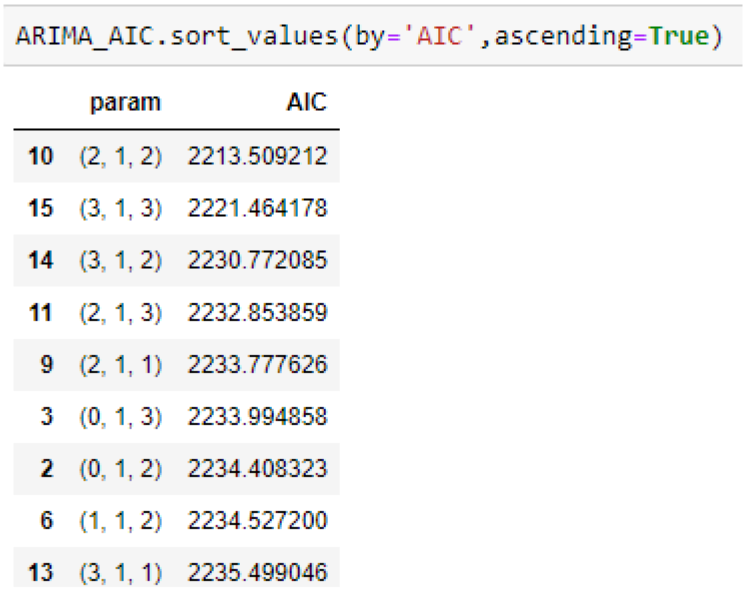
```
Some parameter combinations for the Model...
Model: (0, 1, 1)
Model: (0, 1, 2)
Model: (0, 1, 3)
Model: (1, 1, 0)
Model: (1, 1, 1)
Model: (1, 1, 2)
Model: (1, 1, 3)
Model: (2, 1, 0)
```



```
from statsmodels.tsa.arima.model import ARIMA

for param in pdq:
    ARIMA_model = ARIMA(train['Sparkling'].values,order=param).fit()
    print('ARIMA{} - AIC:{}'.format(param,ARIMA_model.aic))
    ARIMA_AIC = ARIMA_AIC.append({'param':param, 'AIC': ARIMA_model.aic}, ignore_index=True)
```

```
ARIMA(0, 1, 0) - AIC:2267.6630357855465
ARIMA(0, 1, 1) - AIC:2263.060015591681
ARIMA(0, 1, 2) - AIC:2234.4083231284767
ARIMA(0, 1, 3) - AIC:2233.9948577528558
```

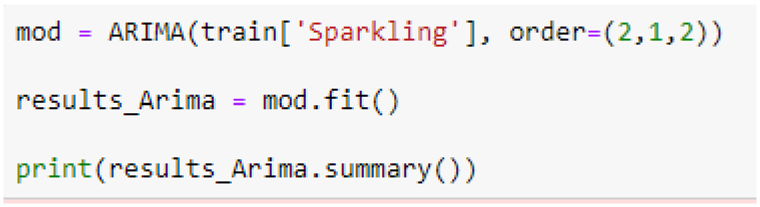
The above piece of code will yield a series of parameter values and their AIC, we are going to sort the results by AIC as shown below:



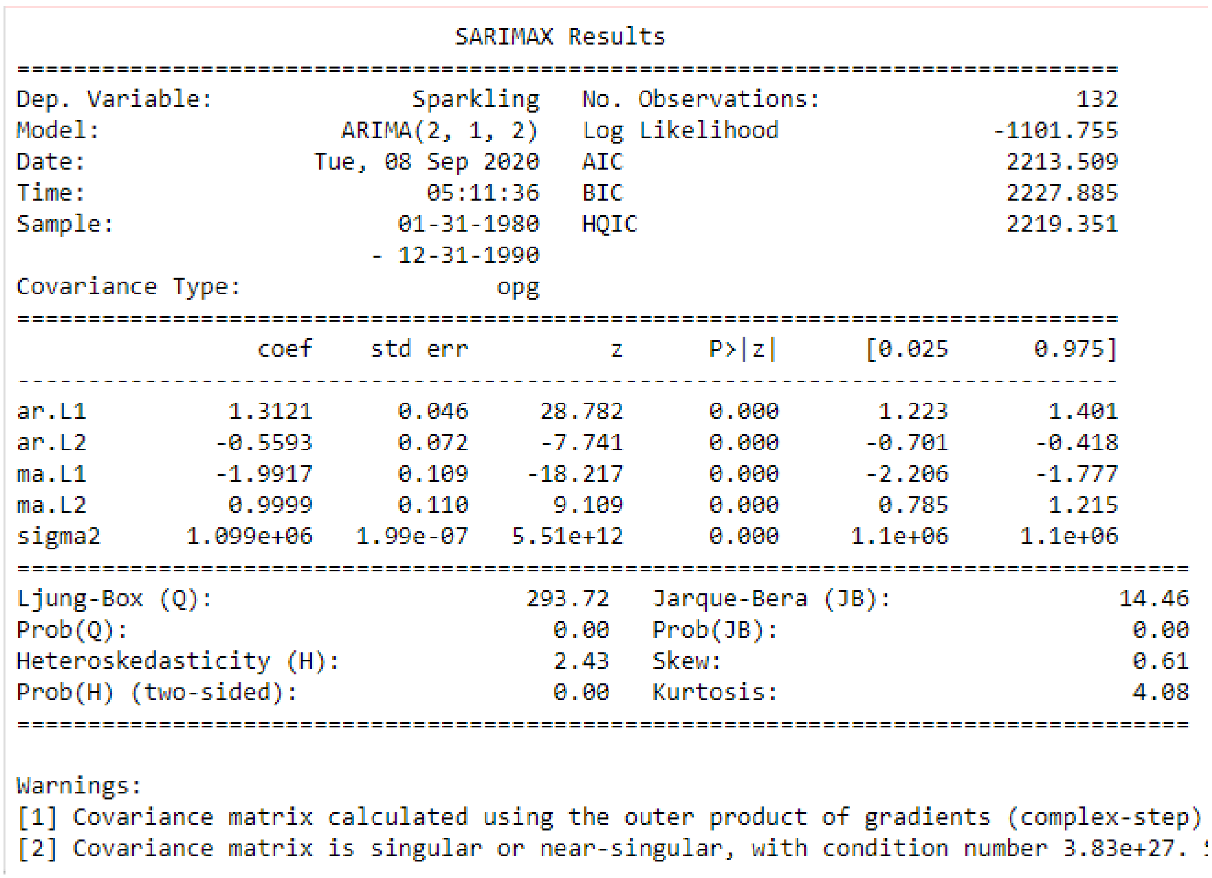
```
ARIMA_AIC.sort_values(by='AIC',ascending=True)
```

	param	AIC
10	(2, 1, 2)	2213.509212
15	(3, 1, 3)	2221.464178
14	(3, 1, 2)	2230.772085
11	(2, 1, 3)	2232.853859
9	(2, 1, 1)	2233.777626
3	(0, 1, 3)	2233.994858
2	(0, 1, 2)	2234.408323
6	(1, 1, 2)	2234.527200
13	(3, 1, 1)	2235.499046

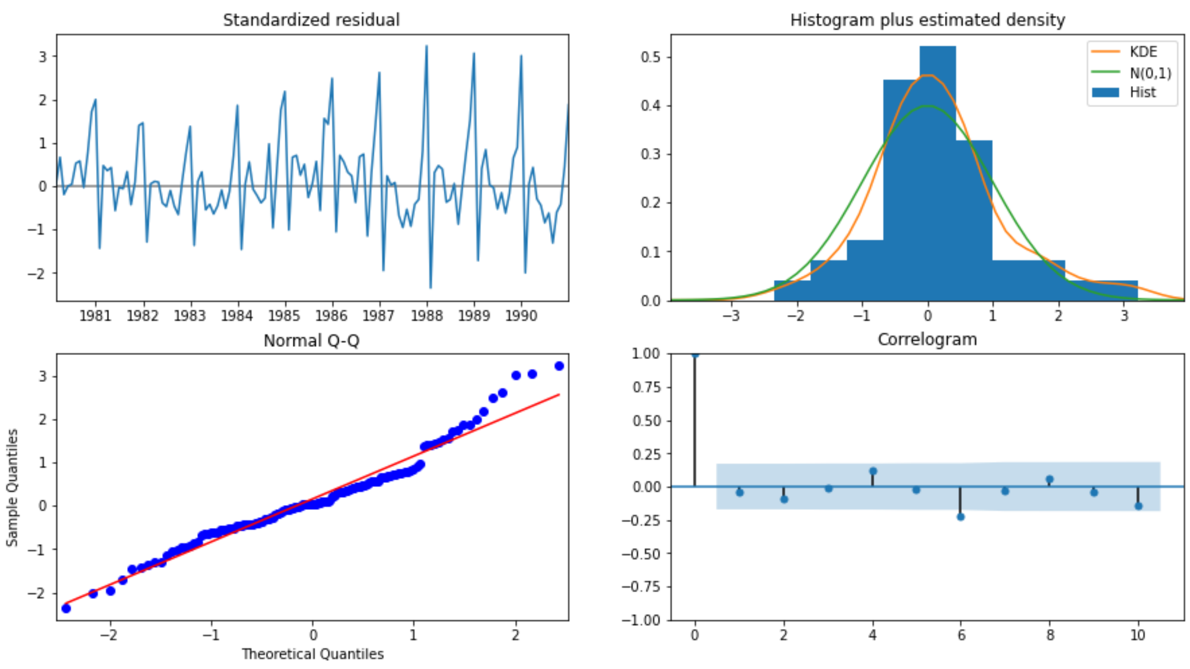
We can see that we get lowest value of AIC for the ARIMA model which was built using p=2, d=1 and q=2. So, now we will build the ARIMA model again with these parameters to get the model summary and the diagnostic plot:



```
mod = ARIMA(train['Sparkling'], order=(2,1,2))

results_Arima = mod.fit()

print(results_Arima.summary())
```



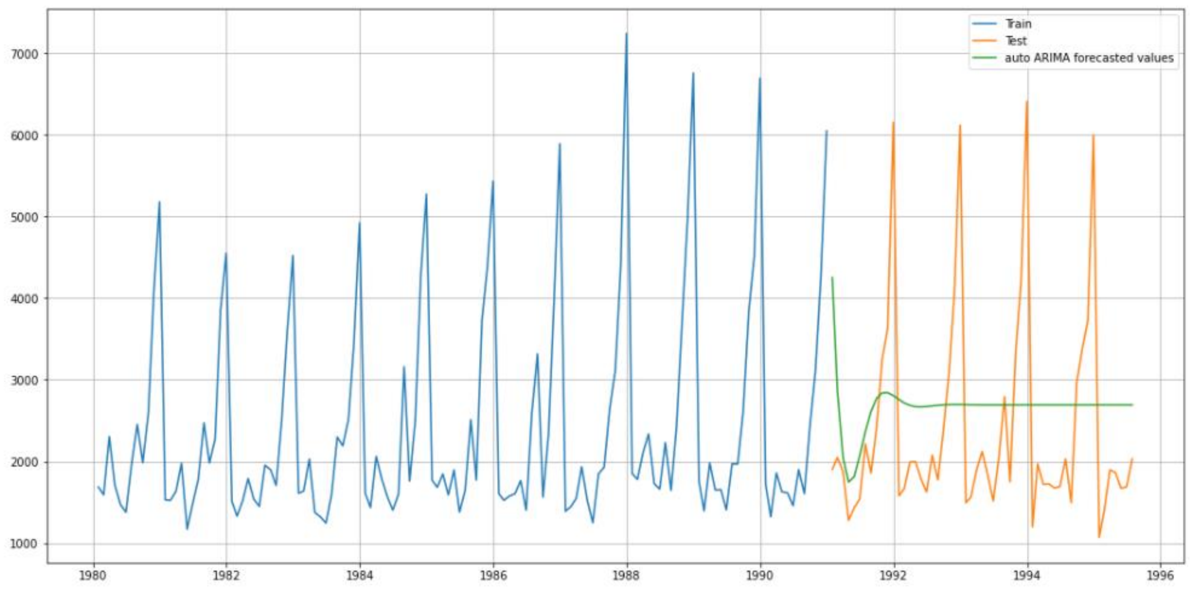
```
                               SARIMAX Results
==============================================================================
Dep. Variable:              Sparkling   No. Observations:                  132
Model:                 ARIMA(2, 1, 2)   Log Likelihood               -1101.755
Date:                Tue, 08 Sep 2020   AIC                           2213.509
Time:                        05:11:36   BIC                           2227.885
Sample:                    01-31-1980   HQIC                          2219.351
                         - 12-31-1990
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.3121      0.046     28.782      0.000       1.223       1.401
ar.L2         -0.5593      0.072     -7.741      0.000      -0.701      -0.418
ma.L1         -1.9917      0.109    -18.217      0.000      -2.206      -1.777
ma.L2          0.9999      0.110      9.109      0.000       0.785       1.215
sigma2      1.099e+06   1.99e-07   5.51e+12      0.000     1.1e+06     1.1e+06
===================================================================================
Ljung-Box (Q):                      293.72   Jarque-Bera (JB):                14.46
Prob(Q):                              0.00   Prob(JB):                         0.00
Heteroskedasticity (H):               2.43   Skew:                             0.61
Prob(H) (two-sided):                  0.00   Kurtosis:                         4.08
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step)
[2] Covariance matrix is singular or near-singular, with condition number 3.83e+27.
```



We can see from the diagnostic plots that the residuals are almost normally distributed and don't have a specific pattern. After this, we will use the forecast() method to get the forecast on the testing data. We can then plot these forecasted values on the training and testing data as shown below:

```
predicted_autoARIMA = results_Arima.forecast(steps=len(test))
predicted_autoARIMA
```

```
1991-01-31    4252.346471
1991-02-28    2863.094190
1991-03-31    2043.981113
1991-04-30    1746.214744
1991-05-31    1813.637173
1991-06-30    2068.641485
1991-07-31    2365.526078
1991-08-31    2612.448890
1991-09-30    2770.393033
1991-10-31    2839.530730
1991-11-30    2841.909845
1991-12-31    2806.363256
1992-01-31    2758.391627
1992-02-29    2715.328578
1992-03-31    2685.655365
1992-04-30    2670.805741
```



RMSE and MAPE on testing Data:

```
RMSE_autoARIMA = mean_squared_error(test['Sparkling'],predicted_autoARIMA,squared=False)
MAPE_autoARIMA = MAPE(test['Sparkling'],predicted_autoARIMA)

print('RMSE for the autofit ARIMA model:',RMSE_autoARIMA,'\nMAPE for the autofit ARIMA model:',MAPE_autoARIMA)
```

```
RMSE for the autofit ARIMA model: 1299.9802037470515
MAPE for the autofit ARIMA model: 43.2
```

SARIMA MODEL:

A problem with ARIMA is that it does not support seasonal data. That is a time series with a repeating cycle.
ARIMA expects data that is either not seasonal or has the seasonal component removed, e.g. seasonally adjusted via methods such as seasonal differencing.
Seasonal Autoregressive Integrated Moving Average, SARIMA or Seasonal ARIMA, is an extension of ARIMA that explicitly supports univariate time series data with a seasonal component.
It adds three new hyperparameters to specify the autoregression (AR), differencing (I) and moving average (MA) for the seasonal component of the series, as well as an additional parameter for the period of the seasonality.

- **P**: Seasonal autoregressive order.
- **D**: Seasonal difference order.
- **Q**: Seasonal moving average order.
- **m**: The number of time steps for a single seasonal period.

To build an automated SARIMA model we will follow the same steps as ARIMA model. We will pass a range of additional values for p, d, q, m for the seasonality components of the SARIMA model. We will run a loop like before that will create SARIMA model in each iteration using different values of p, d, q and also calculate the AIC for each model as shown below:

```
import itertools
p = q = range(0, 3)
d= range(1,2)
D = range(0,1)
pdq = list(itertools.product(p, d, q))
model_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, D, q))]
print('Examples of some parameter combinations for Model...')
for i in range(1,len(pdq)):
    print('Model: {}{}'.format(pdq[i], model_pdq[i]))
```

```
Examples of some parameter combinations for Model...
Model: (0, 1, 1)(0, 0, 1, 12)
Model: (0, 1, 2)(0, 0, 2, 12)
Model: (1, 1, 0)(1, 0, 0, 12)
```

```
import statsmodels.api as sm

for param in pdq:
    for param_seasonal in model_pdq:
        SARIMA_model = sm.tsa.statespace.SARIMAX(train['Sparkling'].values,
                                                  order=param,
                                                  seasonal_order=param_seasonal,
                                                  enforce_stationarity=False,
                                                  enforce_invertibility=False)

        results_SARIMA = SARIMA_model.fit(maxiter=1000)
        print('SARIMA{}x{}7 - AIC:{}'.format(param, param_seasonal, results_SARIMA.aic))
        SARIMA_AIC = SARIMA_AIC.append({'param':param,'seasonal':param_seasonal ,'AIC': results_SARIMA.aic}, ignore_index=True)
```

After this step will sort the obtained values by the AIC values and then get the parameters that give the lowest AIC as shown below:

```
SARIMA_AIC.sort_values(by=['AIC'], ascending=True).head()
```

	param	seasonal	AIC
50	(1, 1, 2)	(1, 0, 2, 12)	1555.584247
53	(1, 1, 2)	(2, 0, 2, 12)	1555.934565
26	(0, 1, 2)	(2, 0, 2, 12)	1557.121563
23	(0, 1, 2)	(1, 0, 2, 12)	1557.160507
77	(2, 1, 2)	(1, 0, 2, 12)	1557.340402

We can see from the above results that p=1, d=1, q=2, P=1, D=0, Q=2, M=12 give the lowest value of AIC so we will choose them and create the SARIMA model.

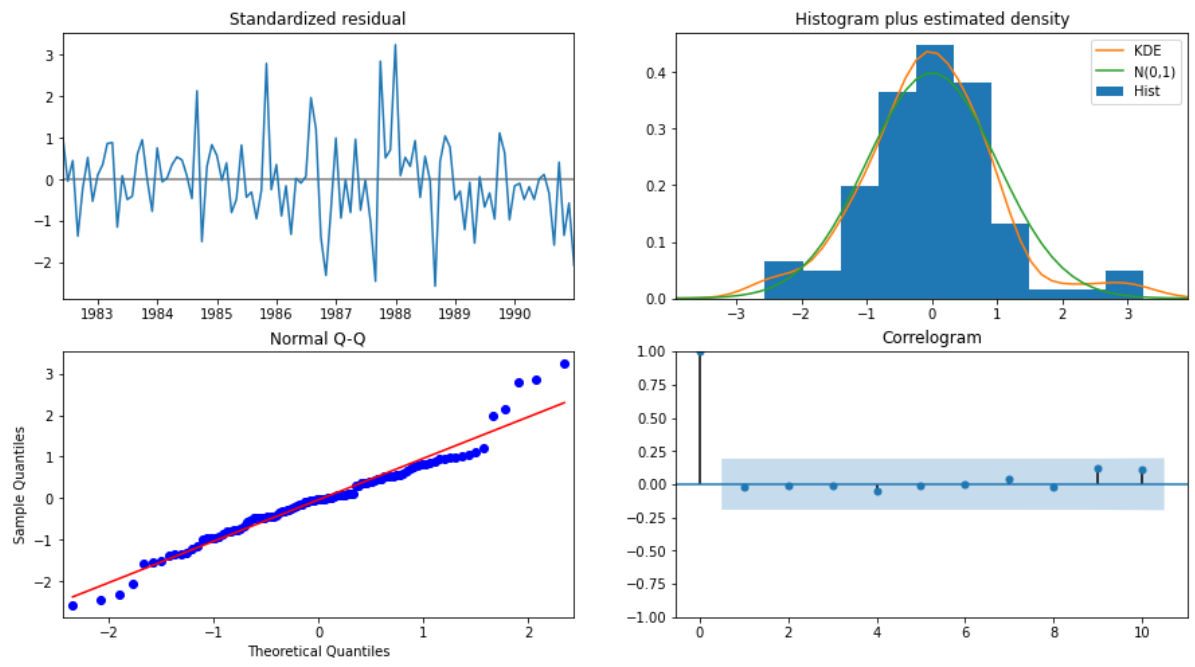
```
import statsmodels.api as sm

mod = sm.tsa.statespace.SARIMAX(train['Sparkling'],
                                order=(1,1,2),
                                seasonal_order=(1, 0, 2, 12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results_SARIMA = mod.fit()
print(results_SARIMA.summary())
```

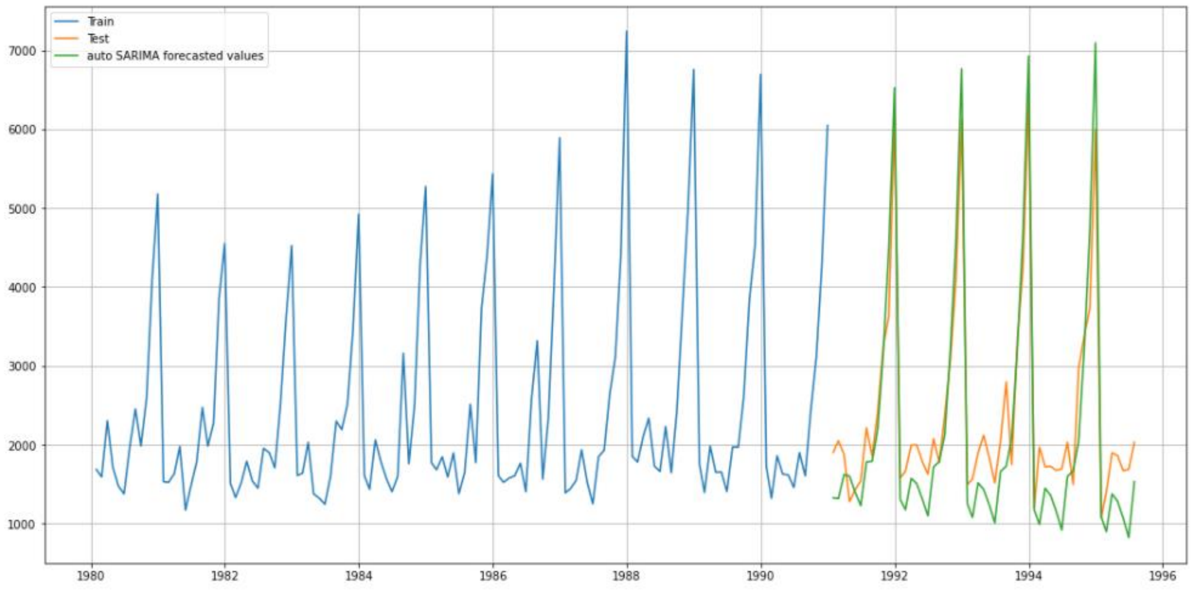
We are going to see the summary of this model and make the forecast. We are also going to see the diagnostic plot of the residuals and plot the forecasted values on the complete dataset.

```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                            Sparkling   No. Observations:                  132
Model:             SARIMAX(1, 1, 2)x(1, 0, 2, 12)   Log Likelihood                -770.792
Date:                              Tue, 08 Sep 2020   AIC                           1555.585
Time:                                      05:13:46   BIC                           1574.096
Sample:                                  01-31-1980   HQIC                          1563.084
                                       - 12-31-1990
Covariance Type:                                opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.6283      0.253     -2.483      0.013      -1.124      -0.132
ma.L1         -0.1030      0.223     -0.463      0.643      -0.539       0.333
ma.L2         -0.7291      0.151     -4.813      0.000      -1.026      -0.432
ar.S.L12       1.0438      0.014     72.785      0.000       1.016       1.072
ma.S.L12      -0.5552      0.098     -5.661      0.000      -0.747      -0.363
ma.S.L24      -0.1352      0.120     -1.131      0.258      -0.369       0.099
sigma2      1.505e+05   2.03e+04      7.410      0.000    1.11e+05     1.9e+05
===================================================================================
Ljung-Box (Q):                       23.01   Jarque-Bera (JB):                11.63
Prob(Q):                              0.99   Prob(JB):                         0.00
Heteroskedasticity (H):               1.47   Skew:                             0.36
Prob(H) (two-sided):                  0.26   Kurtosis:                         4.47
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

```
predicted_autoSARIMA = results_SARIMA.forecast(steps=len(test))
predicted_autoSARIMA
```

```
1991-01-31    1326.800382
1991-02-28    1317.008560
1991-03-31    1621.845395
1991-04-30    1600.134859
1991-05-31    1393.357369
1991-06-30    1225.908188
1991-07-31    1781.184512
1991-08-31    1788.738808
```





RMSE and MAPE on testing data:

```
RMSE_autoSARIMA = mean_squared_error(test['Sparkling'],predicted_autoSARIMA,squared=False)
MAPE_autoSARIMA = MAPE(test['Sparkling'],predicted_autoSARIMA)

print('RMSE for the autofit SARIMA model:',RMSE_autoSARIMA,'\nMAPE for the autofit SARIMA model:',MAPE_autoSARIMA)
```

```
RMSE for the autofit SARIMA model: 527.5713423575794
MAPE for the autofit SARIMA model: 18.85
```

7. Build ARIMA/SARIMA models based on the cut-off points of ACF and PACF on the training data and evaluate this model on the test data using RMSE.

   In building the automated ARIMA and SARIMA model we passed a range of valued from p, d, q and then we chose the best parameter based on the model which gave the lowest value for AIC. To find the values for p, d, q manually we need to use the autocorrelation function plot and partial autocorrelation function plot.

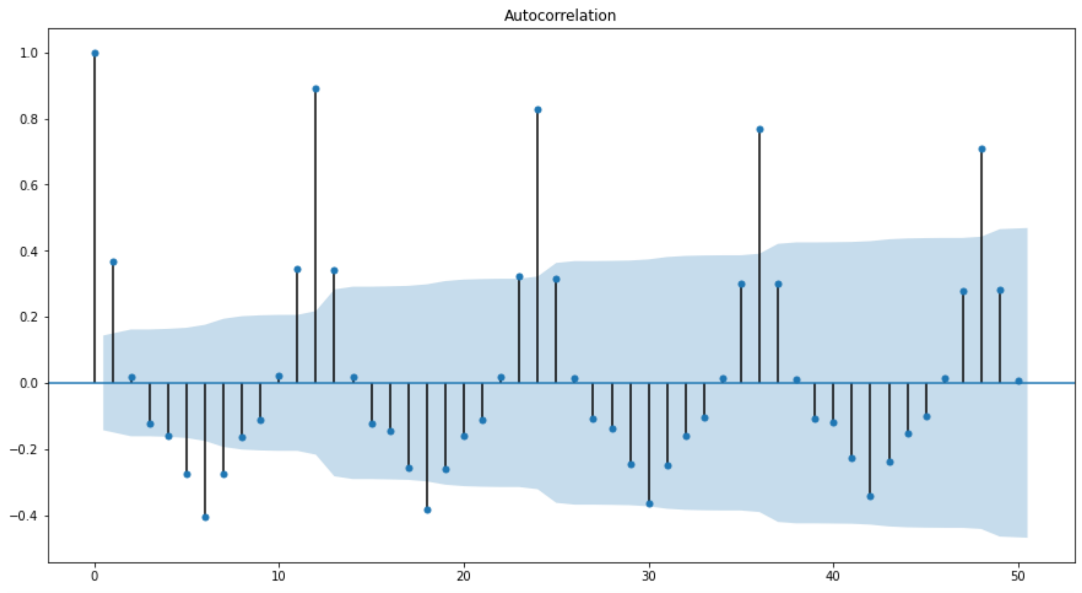
   Autocorrelation Function (ACF):

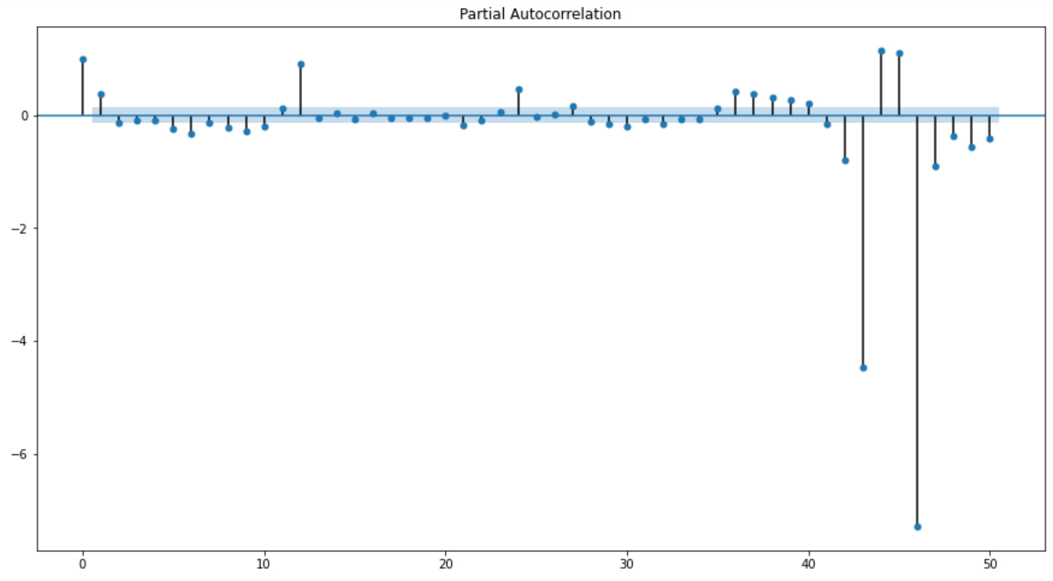
   ACF tells us how related the series is to itself. It is the coefficient of correlation between two values in a time series. Since, the correlation is calculated between variable with itself at a previous time, hence the name autocorrelation. We get the order of q from the ACF plot

   Partial Autocorrelation Function (PACF):

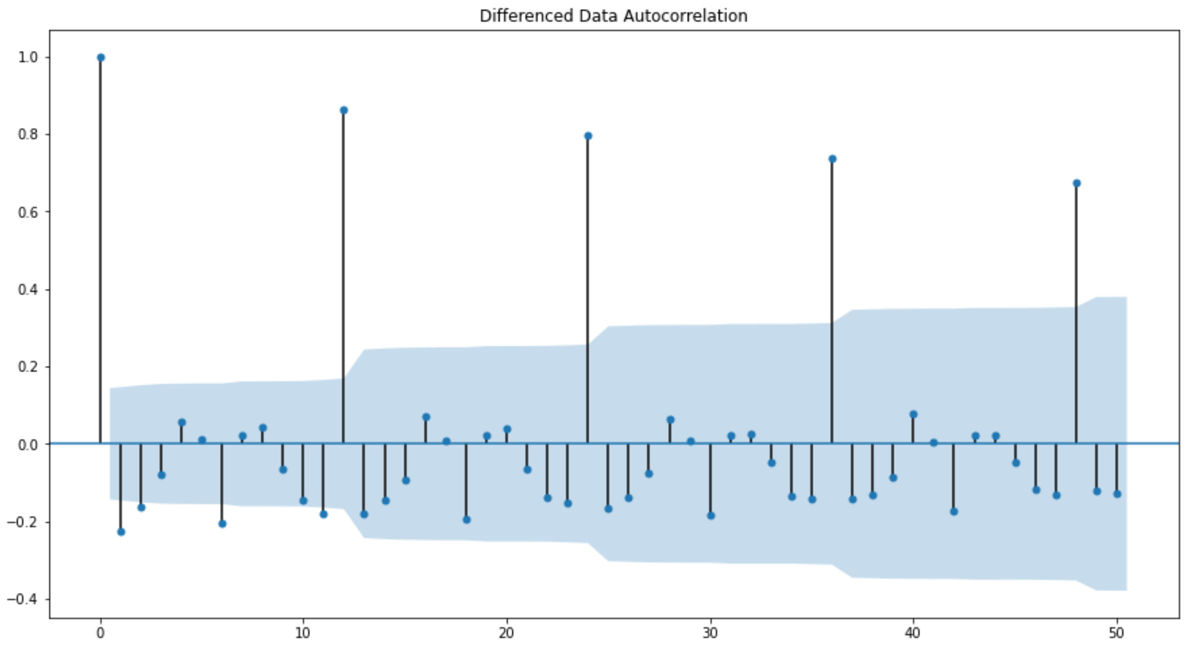
   A Partial autocorrelation is a summary of the relationship between an observation in a time series with observation at a prior time step, with the relationship of the intervening observations removed. We get the order of p from PACF plot.

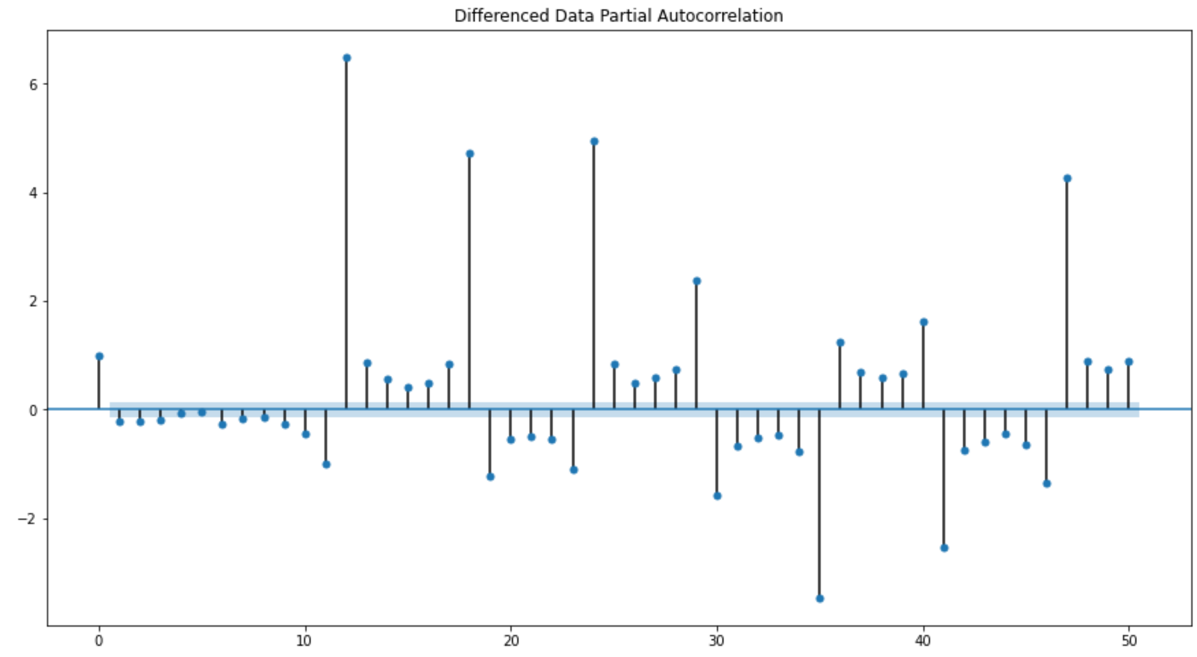
   The ACF and PACF plots for the time series are shown below:





While performing the stationarity test, we saw that the time series becomes stationary after taking difference of order 1, so we will plot ACF and PACF for first order difference.

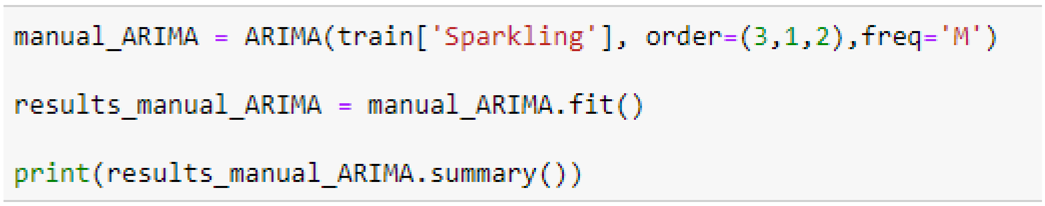




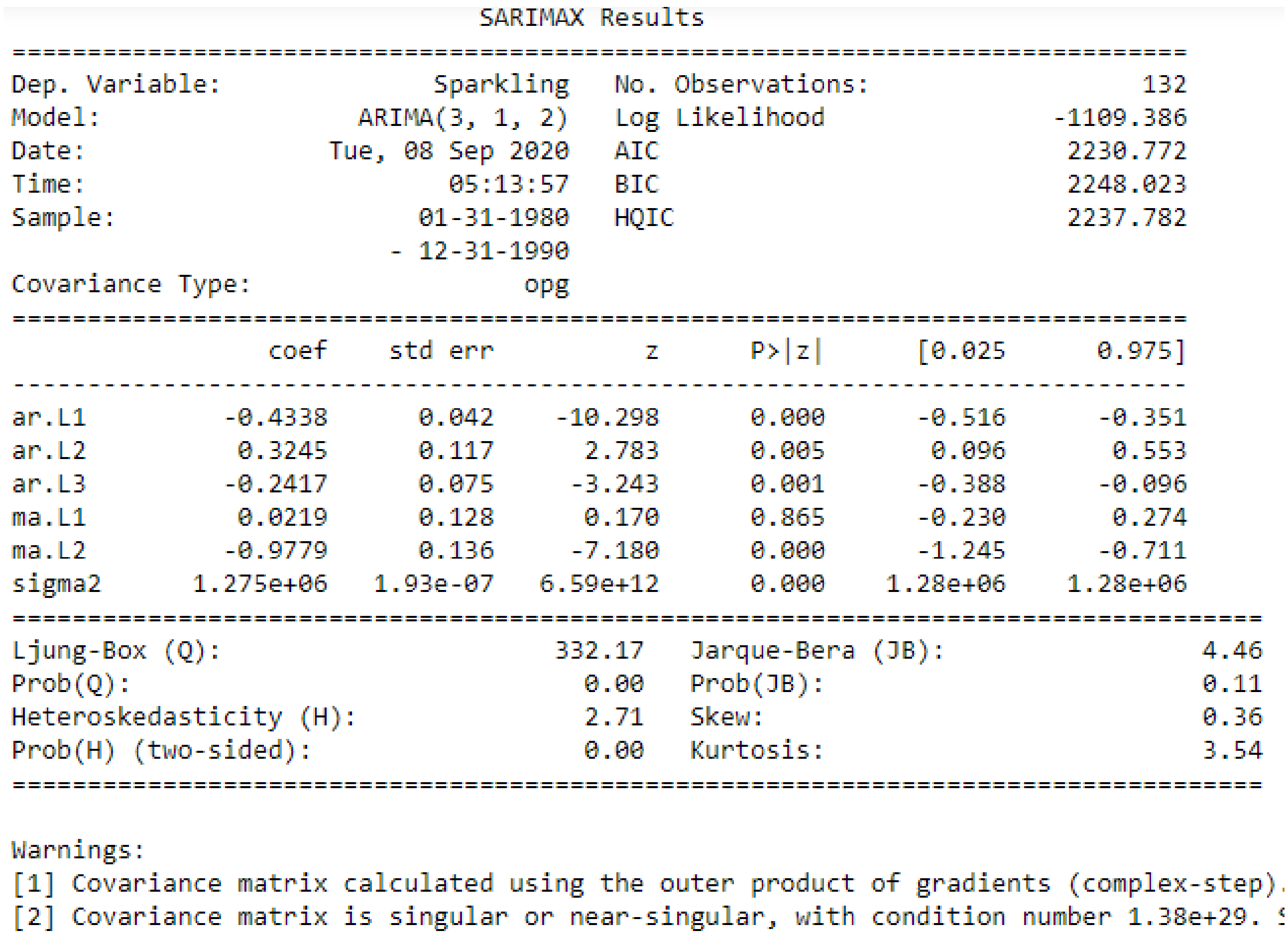
To get the order of p & q we need to see the number of significant values above the cut off before they drop to insignificant region

- the p value from differenced PACF is 3 as there are 3 significant values above the cut-off
- the q value from differenced ACF is 2 as there are 2 significant values above the cut-off
- the d values is 1 as we need single order differencing to make the series stationary

ARIMA MODEL:
We will build ARIMA model using the values of p, d and q obtained from above. We will check the summary statistics and diagnostic plots for the model. We will also forecast future values from this model and plot them on the test and train data.

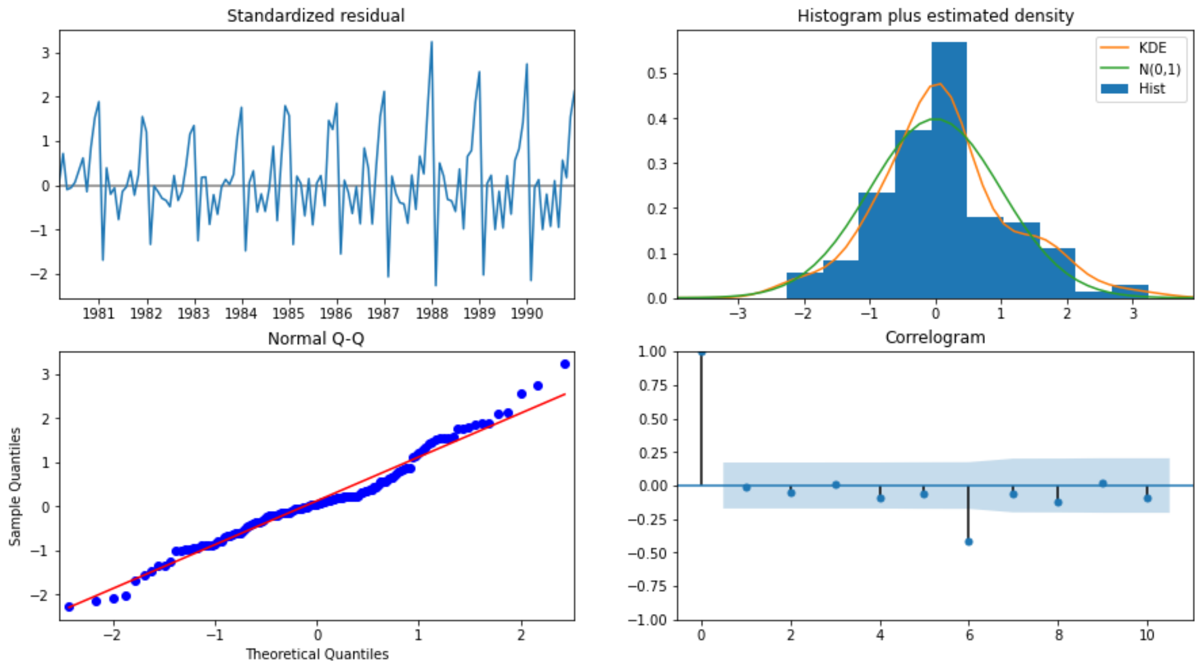


```
manual_ARIMA = ARIMA(train['Sparkling'], order=(3,1,2),freq='M')

results_manual_ARIMA = manual_ARIMA.fit()

print(results_manual_ARIMA.summary())
```



```
                              SARIMAX Results
==============================================================================
Dep. Variable:              Sparkling   No. Observations:                  132
Model:                 ARIMA(3, 1, 2)   Log Likelihood               -1109.386
Date:                Tue, 08 Sep 2020   AIC                           2230.772
Time:                        05:13:57   BIC                           2248.023
Sample:                    01-31-1980   HQIC                          2237.782
                         - 12-31-1990
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.4338      0.042    -10.298      0.000      -0.516      -0.351
ar.L2          0.3245      0.117      2.783      0.005       0.096       0.553
ar.L3         -0.2417      0.075     -3.243      0.001      -0.388      -0.096
ma.L1          0.0219      0.128      0.170      0.865      -0.230       0.274
ma.L2         -0.9779      0.136     -7.180      0.000      -1.245      -0.711
sigma2      1.275e+06   1.93e-07   6.59e+12      0.000    1.28e+06    1.28e+06
===================================================================================
Ljung-Box (Q):                      332.17   Jarque-Bera (JB):                 4.46
Prob(Q):                              0.00   Prob(JB):                         0.11
Heteroskedasticity (H):               2.71   Skew:                             0.36
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.54
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 1.38e+29.
```

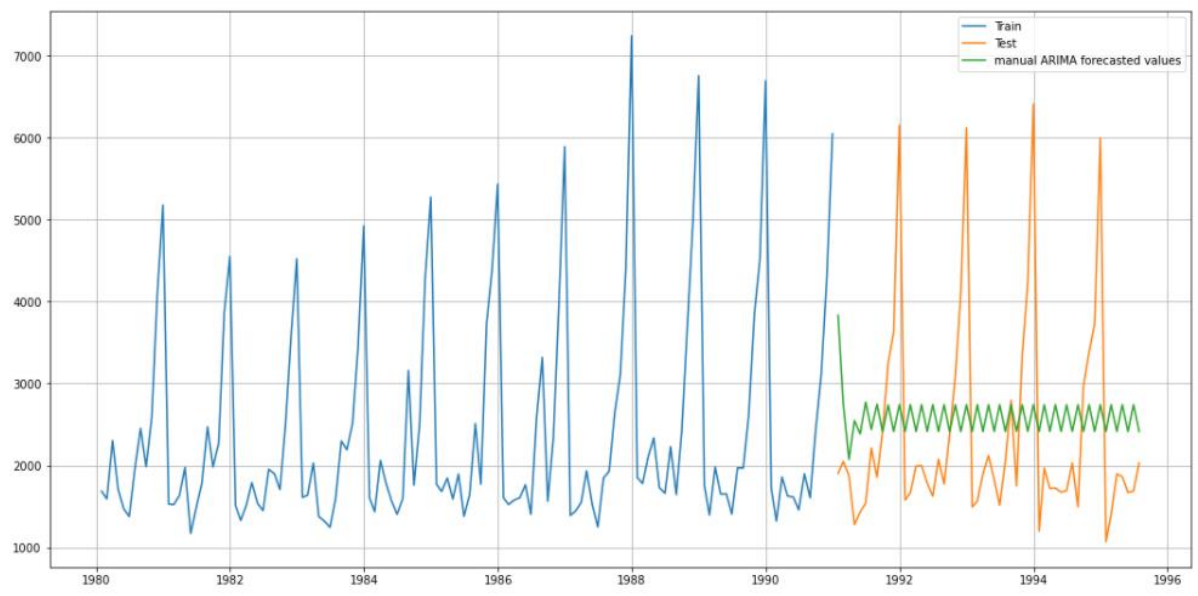
Diagnostics plot:



Predictions:

```
predicted_manual_ARIMA = results_manual_ARIMA.forecast(steps=len(test))
predicted_manual_ARIMA
```

```
1991-01-31    3832.793792
1991-02-28    2746.571295
1991-03-31    2073.508340
1991-04-30    2548.151578
1991-05-31    2386.397442
1991-06-30    2773.282144
1991-07-31    2438.245840
1991-08-31    2748.223716
1991-09-30    2411.524396
1991-10-31    2739.149898
1991-11-30    2412.845407
1991-12-31    2742.091123
1992-01-31    2414.191580
1992-02-29    2742.142404
1992-03-31    2413.895272
1992-04-30    2741.962186
1992-05-31    2413.864889
1992-06-30    2741.988501
1992-07-31    2413.887176
1992-08-31    2741.994717
1992-09-30    2413.885351
1992-10-31    2741.992140
1992-11-30    2413.884374
1992-12-31    2741.992168
```



TEST RMSE and MAPE:

```
RMSE_manual_ARIMA = mean_squared_error(test['Sparkling'],predicted_manual_ARIMA,squared=False)
MAPE_manual_ARIMA = MAPE(test['Sparkling'],predicted_manual_ARIMA)

print('RMSE for the manual ARIMA model:',RMSE_manual_ARIMA,'\nMAPE for the manual ARIMA model:',MAPE_manual_ARIMA)
```

```
RMSE for the manual ARIMA model: 1286.4358401866896
MAPE for the manual ARIMA model: 42.04
```

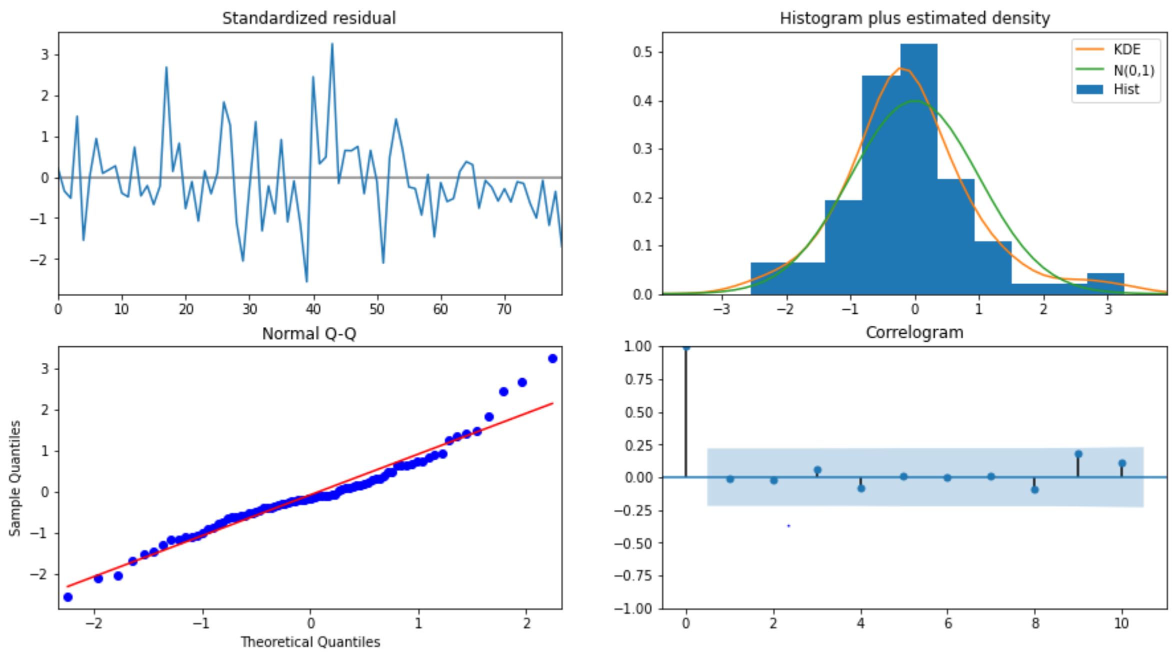
SARIMA MODEL:

We will build SARIMA model using the values of p, d and q obtained from above plots. We will check the summary statistics and diagnostic plots for the model. We will also forecast future values from this model and plot them on the test and train data

```
import statsmodels.api as sm

manual_SARIMA = sm.tsa.statespace.SARIMAX(train['Sparkling'].values,
                                order=(3,1,2),
                                seasonal_order=(3,1,2,12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results_manual_SARIMA = manual_SARIMA.fit(maxiter=1000)
print(results_manual_SARIMA.summary())
```

```
                                      SARIMAX Results
==========================================================================================
Dep. Variable:                                  y   No. Observations:                  132
Model:             SARIMAX(3, 1, 2)x(3, 1, 2, 12)   Log Likelihood                -598.630
Date:                            Tue, 08 Sep 2020   AIC                           1219.260
Time:                                    05:14:17   BIC                           1245.462
Sample:                                         0   HQIC                          1229.765
                                            - 132
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.7556      0.151     -5.013      0.000      -1.051      -0.460
ar.L2          0.1169      0.185      0.633      0.527      -0.245       0.479
ar.L3         -0.0520      0.143     -0.365      0.715      -0.332       0.228
ma.L1          0.0330      0.191      0.173      0.863      -0.341       0.407
ma.L2         -0.9670      0.156     -6.197      0.000      -1.273      -0.661
ar.S.L12      -0.7538      0.496     -1.520      0.128      -1.725       0.218
ar.S.L24      -0.6371      0.351     -1.818      0.069      -1.324       0.050
ar.S.L36      -0.2469      0.151     -1.641      0.101      -0.542       0.048
ma.S.L12       0.3719      0.491      0.758      0.448      -0.590       1.334
ma.S.L24       0.3466      0.365      0.949      0.343      -0.370       1.063
sigma2       1.79e+05   1.67e-06   1.07e+11      0.000    1.79e+05    1.79e+05
===================================================================================
Ljung-Box (Q):                       29.02   Jarque-Bera (JB):                13.16
Prob(Q):                              0.90   Prob(JB):                         0.00
Heteroskedasticity (H):               0.66   Skew:                             0.62
Prob(H) (two-sided):                  0.29   Kurtosis:                         4.55
===================================================================================
```

Diagnostics Plot



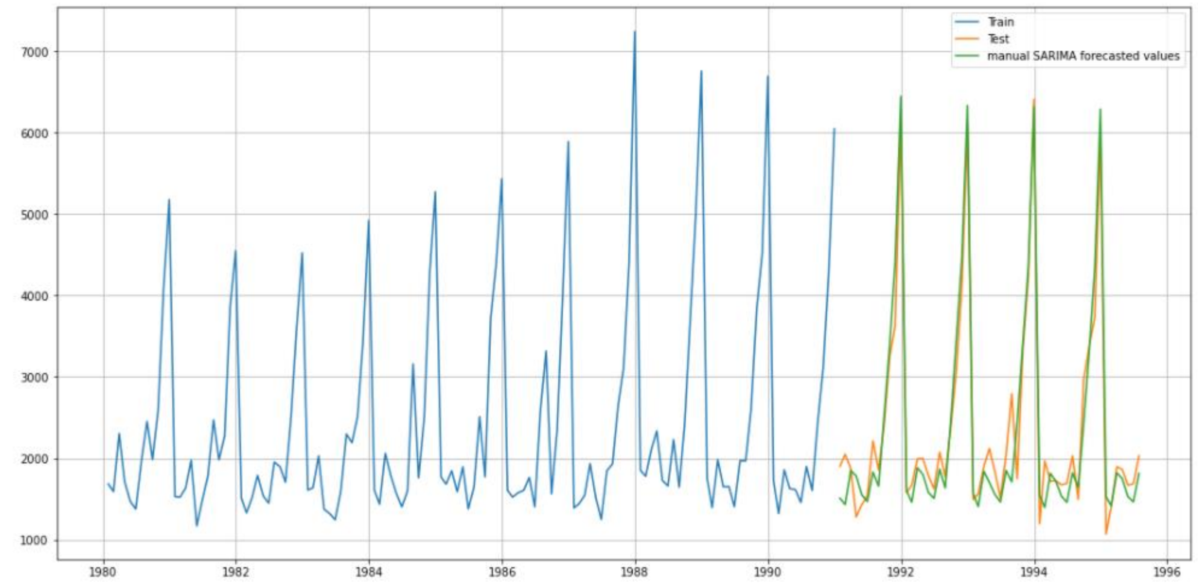
Prediction:

RMSE and MAPE on testing data

```
RMSE_manual_SARIMA = mean_squared_error(test['Sparkling'],predicted_manual_SARIMA,squared=False)
MAPE_manual_SARIMA = MAPE(test['Sparkling'],predicted_manual_SARIMA)

print('RMSE for the manual SARIMA model:',RMSE_manual_SARIMA,'\nMAPE for the autofit SARIMA model:',MAPE_manual_SARIMA)
```

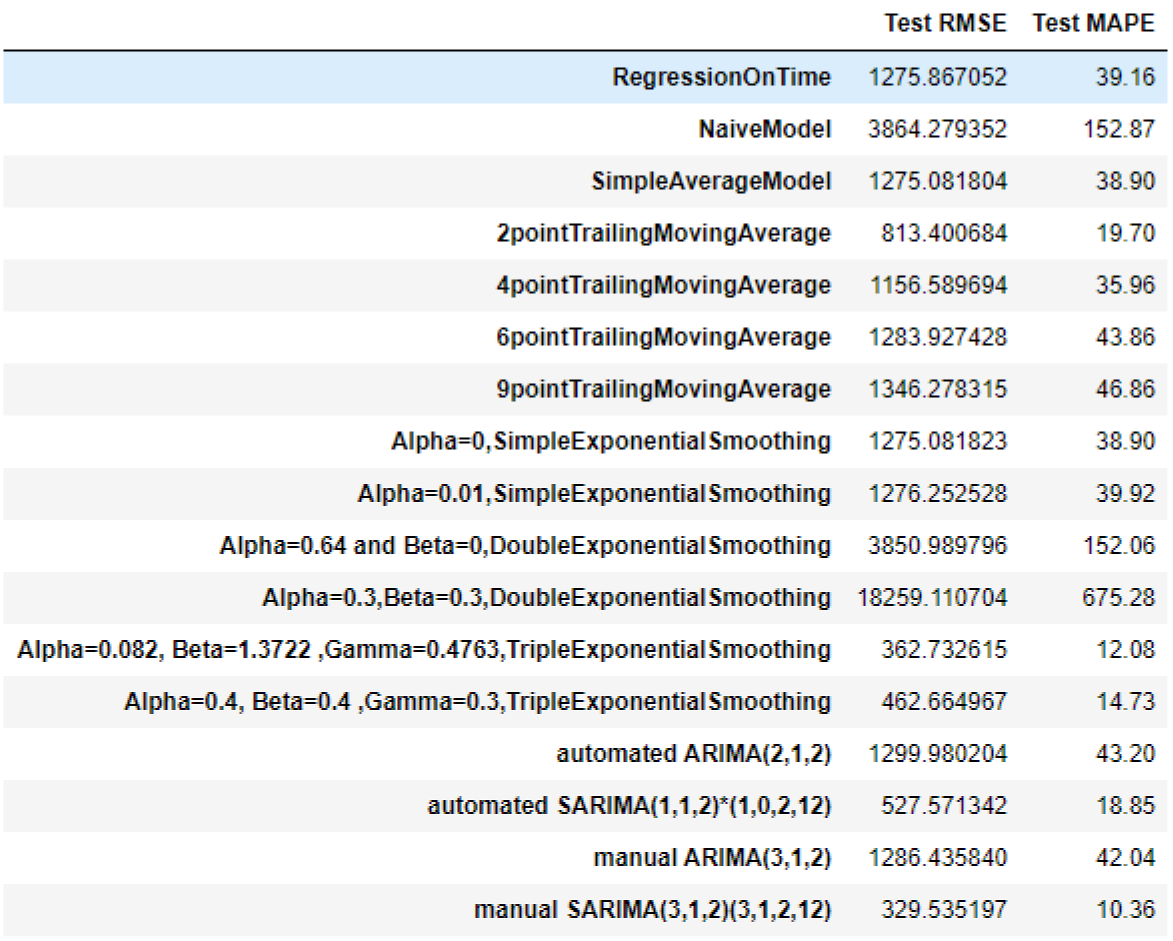
```
RMSE for the manual SARIMA model: 329.5351968729145
MAPE for the autofit SARIMA model: 10.36
```

Plotting on the predictions on training and testing data



8. Build a table with all the models built along with their corresponding parameters and the respective RMSE values on the test data.

Below is shown a table that contains all the Test RMSE and TEST MAPE for all the models build until now in the specific order.



	Test RMSE	Test MAPE
RegressionOnTime	1275.867052	39.16
NaiveModel	3864.279352	152.87
SimpleAverageModel	1275.081804	38.90
2pointTrailingMovingAverage	813.400684	19.70
4pointTrailingMovingAverage	1156.589694	35.96
6pointTrailingMovingAverage	1283.927428	43.86
9pointTrailingMovingAverage	1346.278315	46.86
Alpha=0,SimpleExponentialSmoothing	1275.081823	38.90
Alpha=0.01,SimpleExponentialSmoothing	1276.252528	39.92
Alpha=0.64 and Beta=0,DoubleExponentialSmoothing	3850.989796	152.06
Alpha=0.3,Beta=0.3,DoubleExponentialSmoothing	18259.110704	675.28
Alpha=0.082, Beta=1.3722 ,Gamma=0.4763,TripleExponentialSmoothing	362.732615	12.08
Alpha=0.4, Beta=0.4 ,Gamma=0.3,TripleExponentialSmoothing	462.664967	14.73
automated ARIMA(2,1,2)	1299.980204	43.20
automated SARIMA(1,1,2)*(1,0,2,12)	527.571342	18.85
manual ARIMA(3,1,2)	1286.435840	42.04
manual SARIMA(3,1,2)(3,1,2,12)	329.535197	10.36

If we will sort them by the RMSE (since it is explicitly mentioned in the question), we can see that manual SARIMA has the least RMSE and MAPE on testing data. Triple exponential Smoothing is also giving pretty good results. It is clearly evident that the models which took seasonality into account (SARIMA, Triple Exponential Smoothing) are giving the best results.

Manual SARIMA has the lowest value of RMSE and MAPE, so we will choose it as the most optimum model.

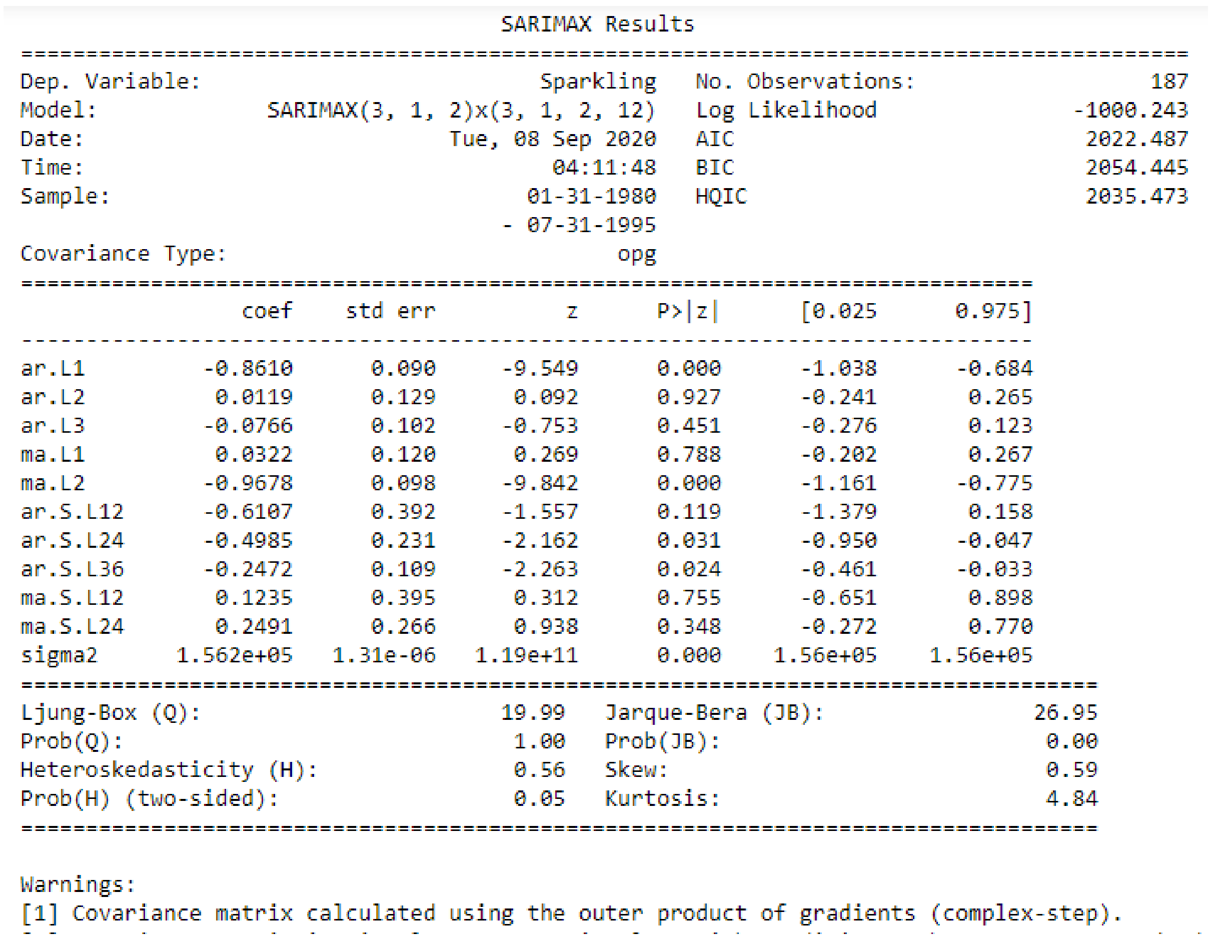
	Test RMSE	Test MAPE
manual SARIMA(3,1,2)(3,1,2,12)	329.535197	10.36
Alpha=0.082, Beta=1.3722 ,Gamma=0.4763,TripleExponentialSmoothing	362.732615	12.08
Alpha=0.4, Beta=0.4 ,Gamma=0.3,TripleExponentialSmoothing	462.664967	14.73
automated SARIMA(1,1,2)*(1,0,2,12)	527.571342	18.85
2pointTrailingMovingAverage	813.400684	19.70
4pointTrailingMovingAverage	1156.589694	35.96
SimpleAverageModel	1275.081804	38.90
Alpha=0,SimpleExponentialSmoothing	1275.081823	38.90
RegressionOnTime	1275.867052	39.16
Alpha=0.01,SimpleExponentialSmoothing	1276.252528	39.92
manual ARIMA(3,1,2)	1286.435840	42.04
automated ARIMA(2,1,2)	1299.980204	43.20
6pointTrailingMovingAverage	1283.927428	43.86
9pointTrailingMovingAverage	1346.278315	46.86
Alpha=0.64 and Beta=0,DoubleExponentialSmoothing	3850.989796	152.06
NaiveModel	3864.279352	152.87
Alpha=0.3,Beta=0.3,DoubleExponentialSmoothing	18259.110704	675.28

9. Based on the model-building exercise, build the most optimum model(s) on the complete data and predict 12 months into the future with appropriate confidence intervals/bands.

Now we will build the final model on the complete dataset. The SARIMA model is built as shown below:

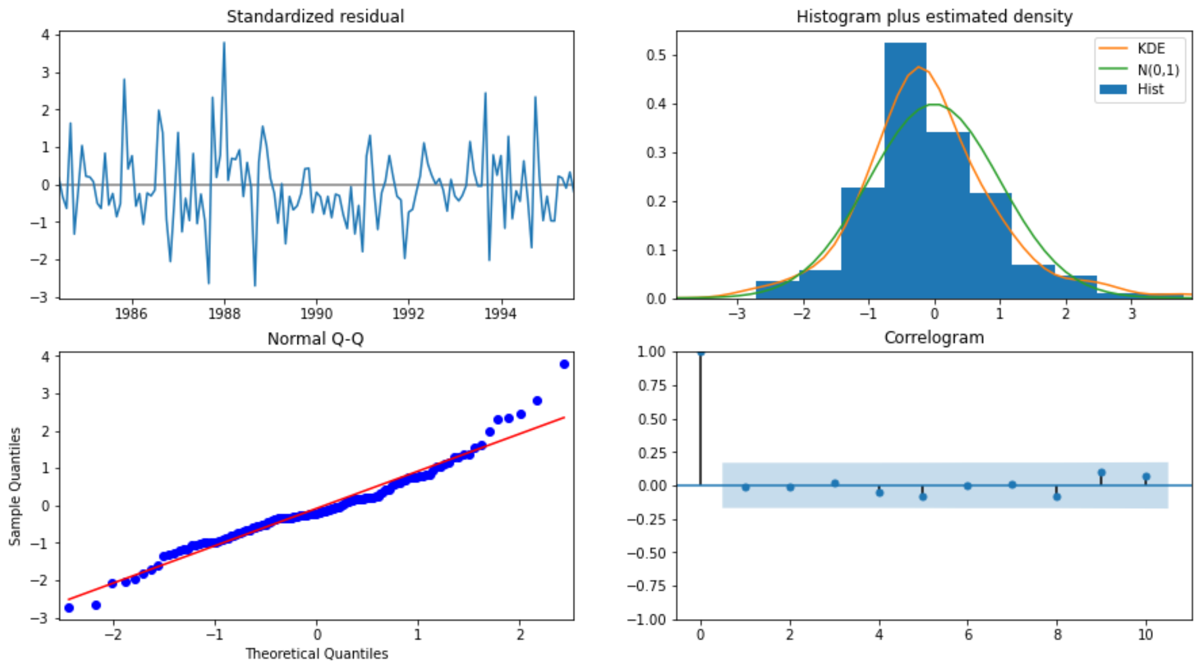
```
full_data_model = sm.tsa.statespace.SARIMAX(df['Sparkling'],
                                order=(3,1,2),
                                seasonal_order=(3, 1, 2, 12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results_full_data_model = full_data_model.fit(maxiter=1000)
print(results_full_data_model.summary())
```

We can see the model summary and the diagnostics plot of the residuals:



```
                                    SARIMAX Results
==========================================================================================
Dep. Variable:                            Sparkling   No. Observations:                187
Model:             SARIMAX(3, 1, 2)x(3, 1, 2, 12)   Log Likelihood               -1000.243
Date:                              Tue, 08 Sep 2020   AIC                           2022.487
Time:                                      04:11:48   BIC                           2054.445
Sample:                                  01-31-1980   HQIC                          2035.473
                                       - 07-31-1995
Covariance Type:                                opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.8610      0.090     -9.549      0.000      -1.038      -0.684
ar.L2          0.0119      0.129      0.092      0.927      -0.241       0.265
ar.L3         -0.0766      0.102     -0.753      0.451      -0.276       0.123
ma.L1          0.0322      0.120      0.269      0.788      -0.202       0.267
ma.L2         -0.9678      0.098     -9.842      0.000      -1.161      -0.775
ar.S.L12      -0.6107      0.392     -1.557      0.119      -1.379       0.158
ar.S.L24      -0.4985      0.231     -2.162      0.031      -0.950      -0.047
ar.S.L36      -0.2472      0.109     -2.263      0.024      -0.461      -0.033
ma.S.L12       0.1235      0.395      0.312      0.755      -0.651       0.898
ma.S.L24       0.2491      0.266      0.938      0.348      -0.272       0.770
sigma2      1.562e+05   1.31e-06   1.19e+11      0.000    1.56e+05    1.56e+05
===================================================================================
Ljung-Box (Q):                       19.99   Jarque-Bera (JB):                26.95
Prob(Q):                              1.00   Prob(JB):                         0.00
Heteroskedasticity (H):               0.56   Skew:                             0.59
Prob(H) (two-sided):                  0.05   Kurtosis:                         4.84
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Diagnostic plot



We can see that the residuals don't have any specific pattern to them, are almost normally distributed.

<u>FORECASTING:</u>

We will use the get_forecast method this time to forecast 12 months into the future. The advantage of the get_forecast method over forecast method is that we can get confidence intervals for our predictions. We will set alpha as 0.05 (95% confidence) for our predictions. We can print the complete information by using the summary function. The summary table is shown below:

```
pred_full_manual_SARIMA_date = predicted_manual_SARIMA_full_data.summary_frame(alpha=0.05).s
```

```
pred_full_manual_SARIMA_date
```

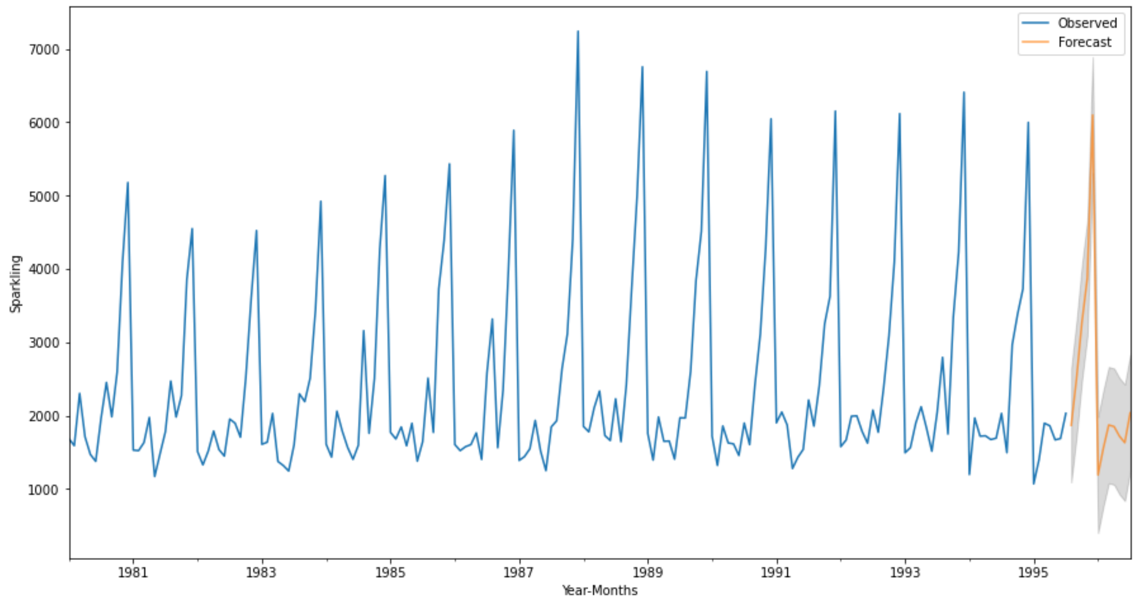
Sparkling	mean	mean_se	mean_ci_lower	mean_ci_upper
1995-08-31	1868.881794	396.433151	1091.887095	2645.876493
1995-09-30	2511.263903	401.786376	1723.777077	3298.750730
1995-10-31	3272.775604	402.629455	2483.636373	4061.914836
1995-11-30	3874.562352	403.044638	3084.609377	4664.515326
1995-12-31	6099.073837	403.064212	5309.082498	6889.065175
1996-01-31	1191.733252	403.772626	400.353447	1983.113056
1996-02-29	1557.109345	403.784457	765.706351	2348.512339
1996-03-31	1872.430762	404.404055	1079.813379	2665.048146
1996-04-30	1851.414350	404.425127	1058.755666	2644.073035
1996-05-31	1719.872671	405.005216	926.077033	2513.668309
1996-06-30	1631.740976	405.035370	837.886237	2425.595714
1996-07-31	2038.437772	405.577226	1243.521016	2833.354528

RMSE for this model

```
rmse = mean_squared_error(df['Sparkling'],results_full_data_model.fittedvalues,squared=False)
print('RMSE of the Full Model',rmse)
```

```
RMSE of the Full Model 578.9671949347013
```

Plotting the forecasted values with confidence intervals:

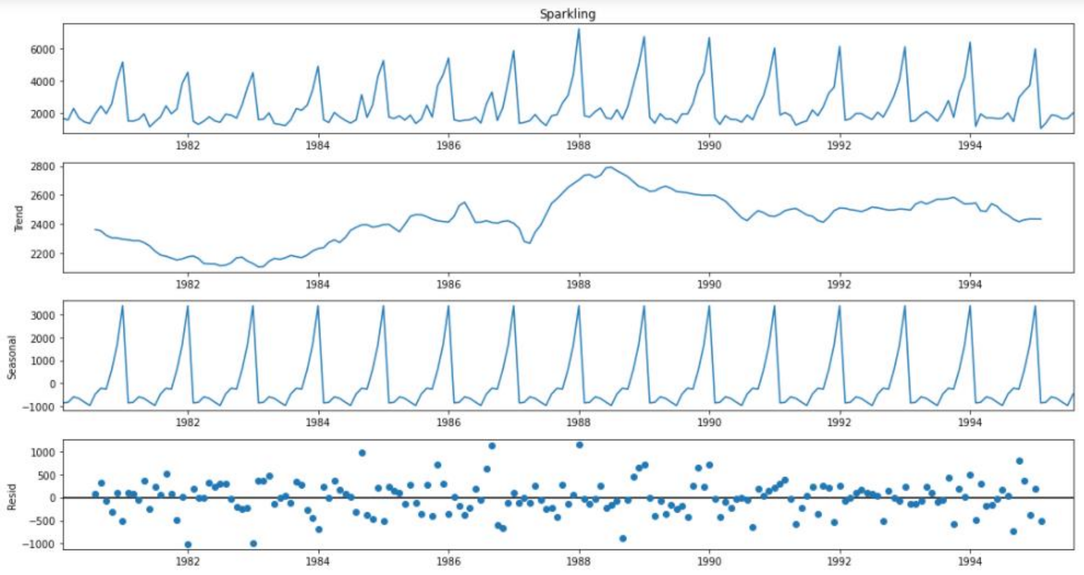


10. Comment on the model thus built and report your findings and suggest the measures that the company should be taking for future sales.

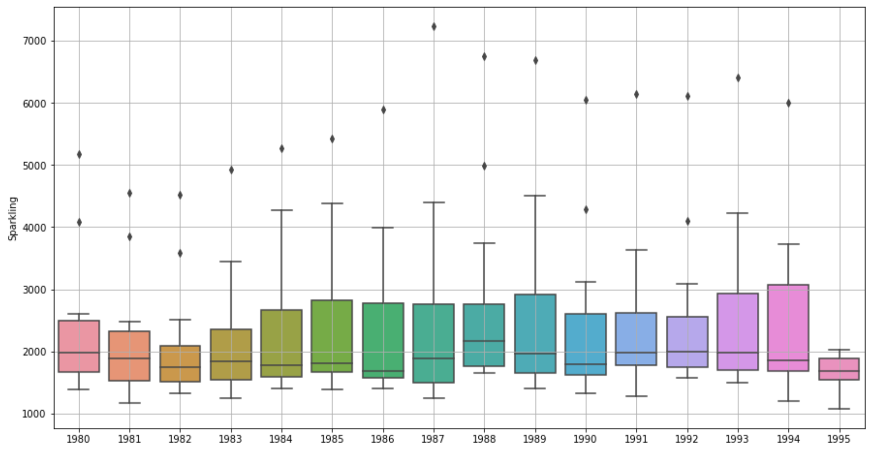
Comments:

The time series data that was provided to us contained the data for the sale of Sparkling Wine from the year 1980 to 1995 and were to analyse this time series and forecast 12 months into the future using a model.

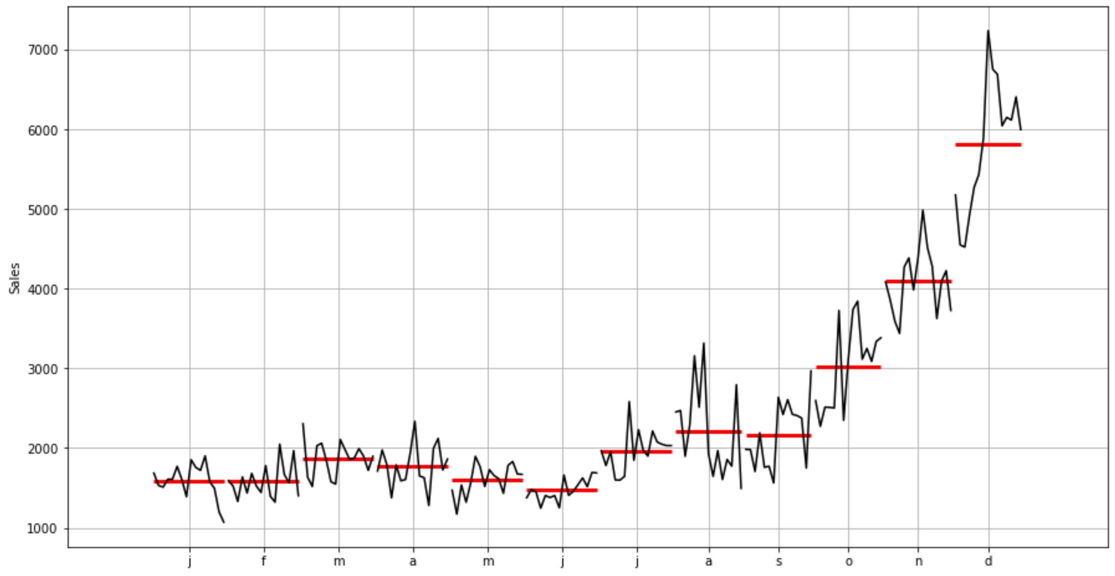
- When we plot the time series, the first graphical inspection of the time series gives us the intuition that this is an additive time series. This is due to the fact that the long-term Trend of the data is almost straight line and does not show an exponential growth of any degree. The data also has a very strong yearly seasonality component. This seasonality is additive in nature because the amplitudes of the seasonality waves are almost same for all the years. These properties indicate strongly towards additive time series. If we also see the residuals of the decomposition of the time series, they all have a mean of 0.



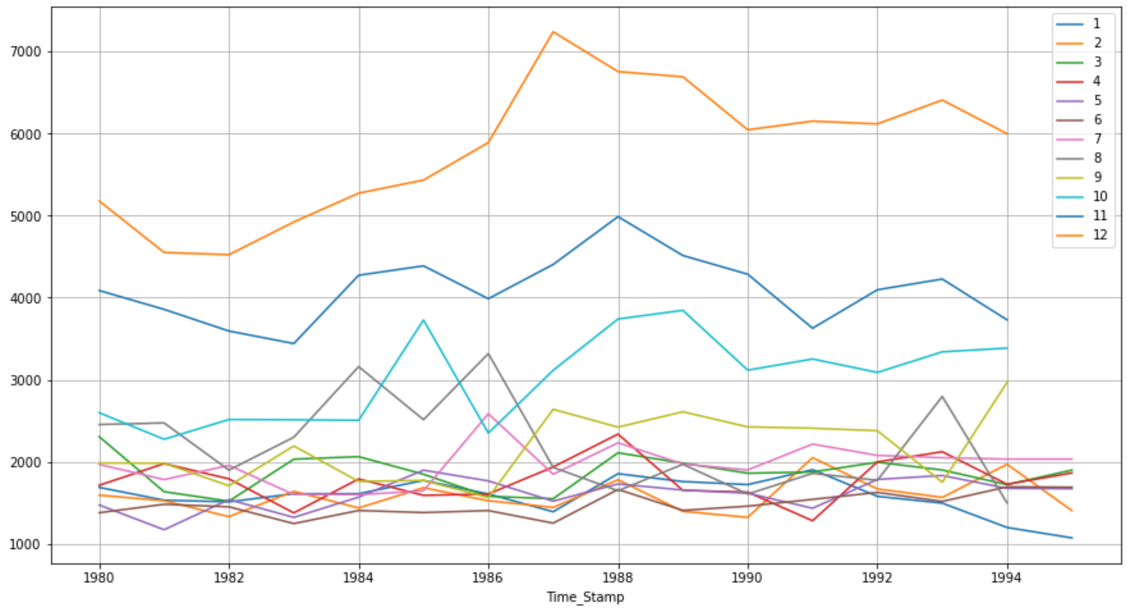
- If we take a close look at the yearly boxplot , we get some very interesting intuitions. All the yearly boxplots have one or two outliers. These outliers are due to the sales peaks in the months of November and December. Contrary to the traditional machine learning, these data points capture very import seasonality information and hence will be kept as such without any treatment.



- If we look at the monthly plot of the sales, we get some straightforward intuitions. The sales are low in the first half of the year, with lowest sales in the months of January, February and June. In the second half of the year, the sales start to pick up and reach peaks in November and December.



- We can also see the line plot made from the pivot table. The lines indicate the pattern of wine sales for a particular month across all the years. We can see from this graph that sales for months of November and December have been at the top consistently across all the years, with December of 1987 recording highest sales ever. The months from January to august al have low trends of sales across all the years as we can see from the clutters of the lines.

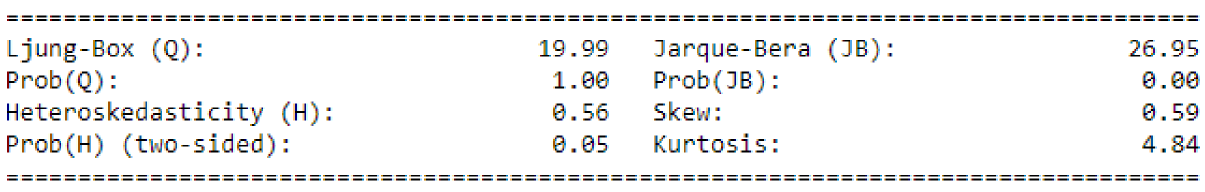


- We have chosen the final model to be the manual SARIMA out of all the other models based on the RMSEon the test data and AIC on training data. **Root Mean Square Error (RMSE)** is the standard deviation of the residuals (prediction errors). It is a measure of how spread out

these residuals are. The mean **absolute percentage error (MAPE**) is a statistical measure of how accurate a forecast system is. It measures this accuracy as a percentage, and can be calculated as the average absolute percent error for each time period minus actual values divided by actual values.
We could have simply used the AIC criteria of all the models to see the most optimum model but we used RMSE and MAPE because AIC has a flaw. The AIC can only measure relative quality of the models; this means that the model tested may still fit poorly that is why we need other measures like RMSE and MAPE to be sure that the model is indeed a good fit.

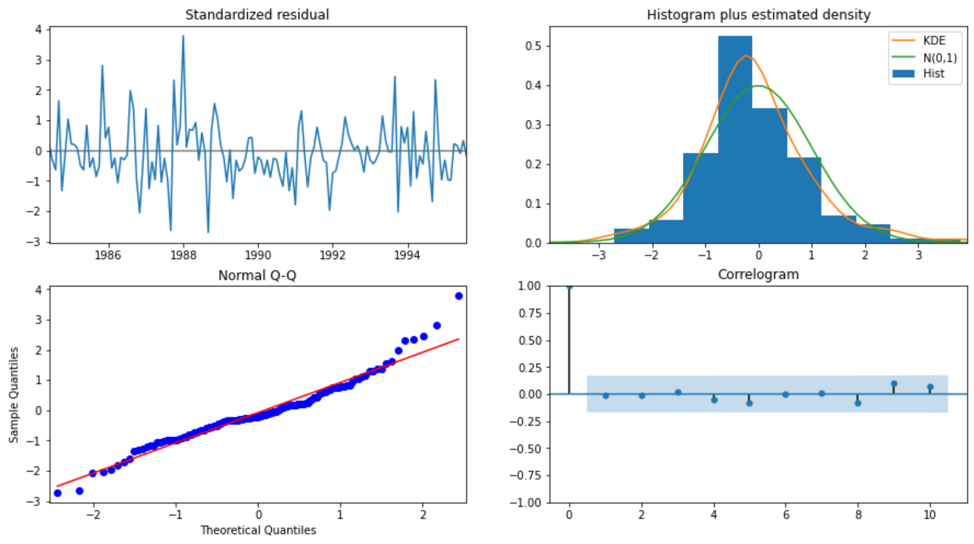
- If we look at the model summary of the SARIMAX model built on the complete data, we can see that it gives us 11 coefficients. The ar.L1, ar.L2, ar.L3, ma.L1, ma.L2 all represent the ARIMA part of the model and ar.S.L12, ar.S.L24, ar.S.L36, ma.S.L12, ma.S.L2 represent the seasonal part and sigma2 represents the residuals. If we take a look at the p-value of all these estimated parameters, we can see that ar.L2, ar.L3, ma.L1, ar.S.L12, ma.S.L12, ma.S.L 24 are not significant because their p-values are greater than 0.05 (95% confidence level). All the other parameters are in the significant region.

- Lastly, we will look at the residual statistics and the diagnostic plots.



```
===================================================================================
Ljung-Box (Q):                      19.99   Jarque-Bera (JB):                 26.95
Prob(Q):                             1.00   Prob(JB):                          0.00
Heteroskedasticity (H):              0.56   Skew:                              0.59
Prob(H) (two-sided):                 0.05   Kurtosis:                          4.84
===================================================================================
```

1. The Ljung-Box test is test for the absence of serial autocorrelations up to a specified lag k. This test determines whether or not the residuals are white noise. This is a test of Lack of fit. If the autocorrelation of the residuals is very small, we say that the model does not show Lack of fit. The null hypothesis is that residuals are independently distributed and the alternate hypothesis is that residuals are not independently distributed.
   Interestingly, the test says that the p-value is greater than critical value so, we reject the null hypothesis. This mean that the residuals have some autocorrelation and are not white noise. Although this is not the desired result from a statistics point of view, real world data is far from perfect and so rarely we encounter a data that can give perfect result.
2. Jarque-Bera test is a test of normality. It is used generally used for a large dataset, because other tests are not reliable when n is large. This test matches the skewness and kurtosis of the data to see if it matched a normal distribution. A normal distribution has skewness=0 and kurtosis=3. Skewness is a measure of symmetry in a distribution and kurtosis tells us how heavy is the tail of the data.
   The residuals have a skew of 0.59 and kurtosis of 4.48 (excess kurt of 1.48) which is very close to normal distribution. We can see these same results in the QQ plot and histogram of residuals.
3. We can also look at the correlogram. There is no significant autocorrelation in the residuals at any lag.



Suggestions to the Company:

- The sale of the sparkling wine has been good but the long-term trend isn't very encouraging. The company's strategy team needs to take measures to increase the sales.
- The company sells a lot a wine in the second half of the year and peaks in the month of November and December. The peaks in November and December seem to be due to the festivities. But since 1987, the sales in December and November are showing a gradual decrease. The company's marketing and sales teams need to make efforts to capture the maximum amount of markets. If the company feels a lack of labour, they could hire more people temporarily.
- The wine sales are very low in the first half of the year. The company must try to capture new and international markets where there is demand in the first half of the year. A good example would be to try and capture the Indian markets in the summer wedding seasons.
- The company must setup a social media department to capture the sentiment of the public. It must make an effort to work on areas where the customers feel it Is lacking.
- The Sales forecast show similar trend like the past, but the company must try to work on all the above points and try to outperform the sales forecast in the coming seasons.