CHINMAY BAKE

In [1]:

```
# REQUIRED PYTHON MODULES

import pyodbc
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from pandas.plotting import register_matplotlib_converters
import statsmodels.api as sm
from statsmodels.tsa.seasonal import STL
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.ar_model import AR
from statsmodels.tsa.ar_model import AutoReg,ar_select_order
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import acf
from statsmodels.tsa.stattools import pacf

import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter(action='ignore', category=FutureWarning)
```

OVERVIEW OF THE FORECASTING AUTOMATION FUNCTION

REQUIREMENTS -

1) Required Python Modules - Listed in line of the Notebook

INTRODUCTION -

1) The function - **forecast_func** performs an AutoRegression based on **statsmodels API's AutoReg** function

2) It has four main functions -

- Plucking the right dataframe and from the dataset and performing required **preprocessing** operations on it
- Identifying the correct **decomposition methodology** based on analysing residuals of each decomposition method
- Enumerating or listing the **lags** observed in the data
- And finally, modeling the data with **Autoregressions**

3) The inputs to the function are -

- **Product ID** - Any valid product ID across the store ranges (Integer,Float)
- **Store ID** - Any valid store ID for stores 1 to 15 only (Integer)
- **Seasonality** - Either True or False - based on the seasonal characterstics of the entered product (Boolean)

- **Seasonal Periodicity** - Frequency of seasonal occurences in the given product on a yearly scale (Integer)
- NOTE: The inputs should be enetered in the above sequence while invoking the function.

4) Outputs - Returns below objects in following sequence

- **Forecasted values** over the period of the test data (Pandas Data Series)
- **Plot** showing the goodness of fit (Matplotlib object)
- **Root Mean Squared Error** (float)
- **Mean Absolutely Scaled Error** (float)

In [10]:

```
def forecast_func(PrdId,StrId,Seasonal_Periodicity:bool,Periodicity): # main

    # invoke the error block to check the authencticity of the input
    check_error=error_block(PrdId,StrId)

    if type(check_error)==str:
           # return error for incorrect inputs
        return check_error
    else:
        # data transformation function invoked
        transform=data_transform(check_error)
        if Seasonal_Periodicity==True:
            #Autoregression with seasonal parameters
            AR=func_AR(transform,Seasonal_Periodicity,Periodicity,transform[3
        else:
            #Autoregression without seasonal parameters
            AR=func_AR(transform,Seasonal_Periodicity,0,transform[3])

    # Return a plot of forecasts
    plt.figure(figsize=(12,6))
    plt.plot(transform[2].iloc[int(len(transform[2]) * 0.8):], label='Observe
    plt.plot(AR[0], color='red', label='Predicted')
    plt.xlabel("DATE")
    plt.ylabel("SALES")
    plt.legend()
    plt.show()
    # Return forecasts and error
    print(AR)

def error_block(product,store):

    if store>15 or store<1:
        return "Incorrect StoreID. Please try again."  # Store Id check

    cursor = cnxn.cursor()


    if store in range(1,6,1):
        sales_1_5_df = pd.read_sql_query('',cnxn)
        sales_df_15=sales_1_5_df[''] == store
        sales_df_15=sales_1_5_df[sales_df_15]
    if store in range(6,11,1):
        sales_6_10_df = pd.read_sql_query('',cnxn)
        sales_df_15=sales_6_10_df[''] == store
        sales_df_15=sales_6_10_df[sales_df_15]
    if store in range(11,16,1):
        sales_11_15_df = pd.read_sql_query('',cnxn)
        sales_df_15 =sales_11_15_df[''] == store
        sales_df_15=sales_11_15_df[sales_df_15]

    sales_15_test=sales_df_15.loc[:, ['']]

    # Product ID selection
    x1=sales_15_test[sales_15_test['']==product]
    x1=x1.reset_index(drop=True)
    x1=x1[['','']]
```

```
    x1=x1.set_index('')
    # Exclusion of negative values from the data
    x1=x1.where(x1 > 0,0)

    if len(x1)==0:
        # product ID check
        return "Incorrect ProductID. Please try again."
    else:
        # return product time series
        return x1

def data_transform(dataframe):

    # Daily resampling of given time series to fix any date index inconsisten
    x2=dataframe[''].resample('D').mean()

    # Analysis of residuals for the selection of best decomposition methods
    def acf1(x):
        #sum of squares of autocorrelations
        return np.square(sum(acf(x)))

    def ssacf(add,mult):
        return np.where(acf1(add)<acf1(mult),"additive","multiplicative")
    #Multiplicative decomposition
    mul = seasonal_decompose(x2 + 0.1,model='multiplicative',extrapolate_trer
    #Additive decomposition
    add = seasonal_decompose(x2,model='additive',extrapolate_trend='freq')
    model_type = ssacf(add.resid,mul.resid)

    if model_type=='multiplicative':
        #decomposition after selection of the appropriate decomposition metho
        model = seasonal_decompose(x2+0.1,model=str(model_type),extrapolate_t
    else:
        model = seasonal_decompose(x2,model=str(model_type),extrapolate_trenc

    #Lag order selection using statsmodels' ar_select_order
    lags=ar_select_order(x2,30)
    laglist=list(lags.ar_lags)

    #alternative method of enumerating lags based
    '''ar,ci =pacf(x2,alpha=0.05)
    laglist=list()
    for i in range(0,len(ar),1):
        if ar[i]<(-0.1) or ar[i]>(0.1):
            laglist.append(i)'''

    # removal of zeros from the lag list
    for i in laglist:
        if i==0:
            laglist.remove(i)

    # handling a null laglist and setting default lag at 1
    if len(laglist)<1:
        laglist=[1]

    return (model,laglist,x2,model_type)
```

```
def func_AR(data,seasonal_period,duration,decomp): # Autoregression

    # Function to calculate mean absolutely scaled error
    def MASE(training_series, testing_series, prediction_series):
        n = training_series.shape[0]
        d = np.abs(  np.diff( training_series) ).sum()/(n-1)
        errors = np.abs(testing_series - prediction_series )
        return errors.mean()/d

    # extraction of each decomposed component
    component_dict = {'seasonal': data[0].seasonal, 'trend': data[0].trend, '
    prediction_results = []

    # 80:20 split
    for component in ['seasonal', 'trend', 'residual']:
        historic = component_dict[component].iloc[:int(len(data[2]) * 0.8)].t
        test = component_dict[component].iloc[int(len(data[2]) * 0.8):]

        predictions = []

        # forecast over the period of the test split and fit the AutoReg func
        for i in range(len(test)):
            model = AutoReg(historic,data[1],period=duration,seasonal=seasona
            model_fit = model.fit()
            pred = model_fit.predict(start=len(historic), end=len(historic),d
            predictions.append(pred[0])
            historic.append(test[i])

        predictions = pd.Series(predictions, index=test.index, name=component
        prediction_results.append(predictions)

    if decomp=='multiplicative':
        # Multiplying each decompoosed component forecast to recompose into a
        recomposed_preds = pd.concat(prediction_results,axis=1).prod(axis=1)
        recomposed_preds.name = 'recomposed_preds'
        # removal of negative forecasts
        recomposed_preds = recomposed_preds.where(recomposed_preds > 0,0)
    else:
         # Adding each decompoosed component forecast to recompose into a sin
        recomposed_preds = pd.concat(prediction_results,axis=1).sum(axis=1)
        recomposed_preds.name = 'recomposed_preds'
         # removal of negative forecasts
        recomposed_preds = recomposed_preds.where(recomposed_preds > 0,0)

    # root mean squared error
    rmse = np.sqrt(mean_squared_error(data[2].iloc[int(len(data[2]) * 0.8):],
    # mean absolutely scaled error
    mase=MASE(pd.Series(historic),data[2].iloc[int(len(data[2]) * 0.8):],reco

    return (recomposed_preds,rmse,mase)
```

POSSIBLE LIMITATIONS -

The function in itself **does not perform any data stationarity transformations or evaluations**. This essentially owes to the fact that there is a substantial variation in stationarity behaviors of the products in the given data, and attaining stationarity through transformations for each product might require a varied set of transformation operations. Hence, this step has been descoped from the functionality.

POSSIBLE BENEFITS -

The function automatically identifies the **appropriate decomposition methodology for a given product**. This is estimated by analysing autocorrelations between the residuals of each decomposition method and returning the ones with the least sum of squares.

The **seasonal arguments** in the input allow the user to enforce control over the seasonal magnitude of the given product. For Eg: if a product X showcases yearly seasonality in store Y then the user could call the function as follows -

forecast_func(X,Y,True,360)

where the True parameter stands for the fact that the product showcases seasonality and 360 is the frequency of seasonality. As the data has daily samples, on a yearly scale, yearly seasonal frequency would be equivalent to 360 days. Please note that 360 should be enetered as an Integer without any unit.

Also, the last argument is an optional parameter, if Seasonality is set to FALSE, the function would defualt pass the value for seasonal frequency as 0 days.

REFERENCES

- https://www.statsmodels.org/stable/examples/notebooks/generated/autoregressions.html (https://www.statsmodels.org/stable/examples/notebooks/generated/autoregressions.html)
- https://www.statsmodels.org/stable/generated/statsmodels.tsa.ar_model.AutoReg.html (https://www.statsmodels.org/stable/generated/statsmodels.tsa.ar_model.AutoReg.html)
- https://otexts.com/fpp2/ (https://otexts.com/fpp2/)