AUTO REGRESSIVE TIME SERIES MODELING FOR DEMAND FORECASTING

CHINMAY BAKE

Please note: most cells have not been ran in this Jupyter notebook due to the fact that the graphics could reveal certain aspects of the data, which might be prohibited under a confidentiality agreement. This includes missing values on the y-axis of all the plots and incomplete texts in the printed values. Hence, the below graphics are just screenshots from the actual notebook. This notebook does not represent my complete work, it is just an illustration of the researched methods and their implementation during the capstone project.

In [41]:
```
import pandas as pd
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.stattools import adfuller
from pmdarima.arima import auto_arima
from statsmodels.tsa.statespace.sarimax import SARIMAX
import pandas as pd
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 sklearn.model_selection import TimeSeriesSplit
```

INTRODUCTION TO AUTO REGRESSION

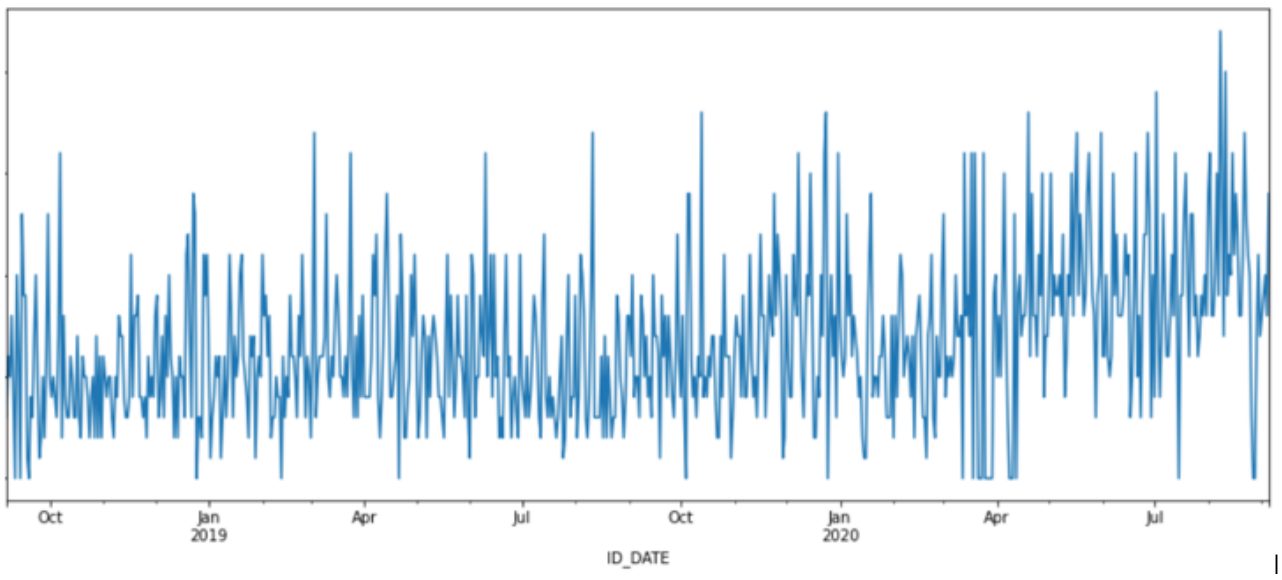
We have a series of data points which are ordered with time, meaning that there is a time dimension associated with each value. Such data could be called as Time Series. One of the very well known ways of modeling Time series is Auto Regression. Lets take a look at the plot below. What if I determine that my sales in March are dependent on my sales in February? Or maybe what if my sale for today would be dependent on my yesterday's sale. An autogregression model works in similar way. A model that depends on one period or one lag in the past is called an Auto Regressive model of order 1.

In [179]:
```
import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter(action='ignore', category=FutureWarning)

def mean_absolute_percentage_error(y_true, y_pred):

    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true))*100

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
```

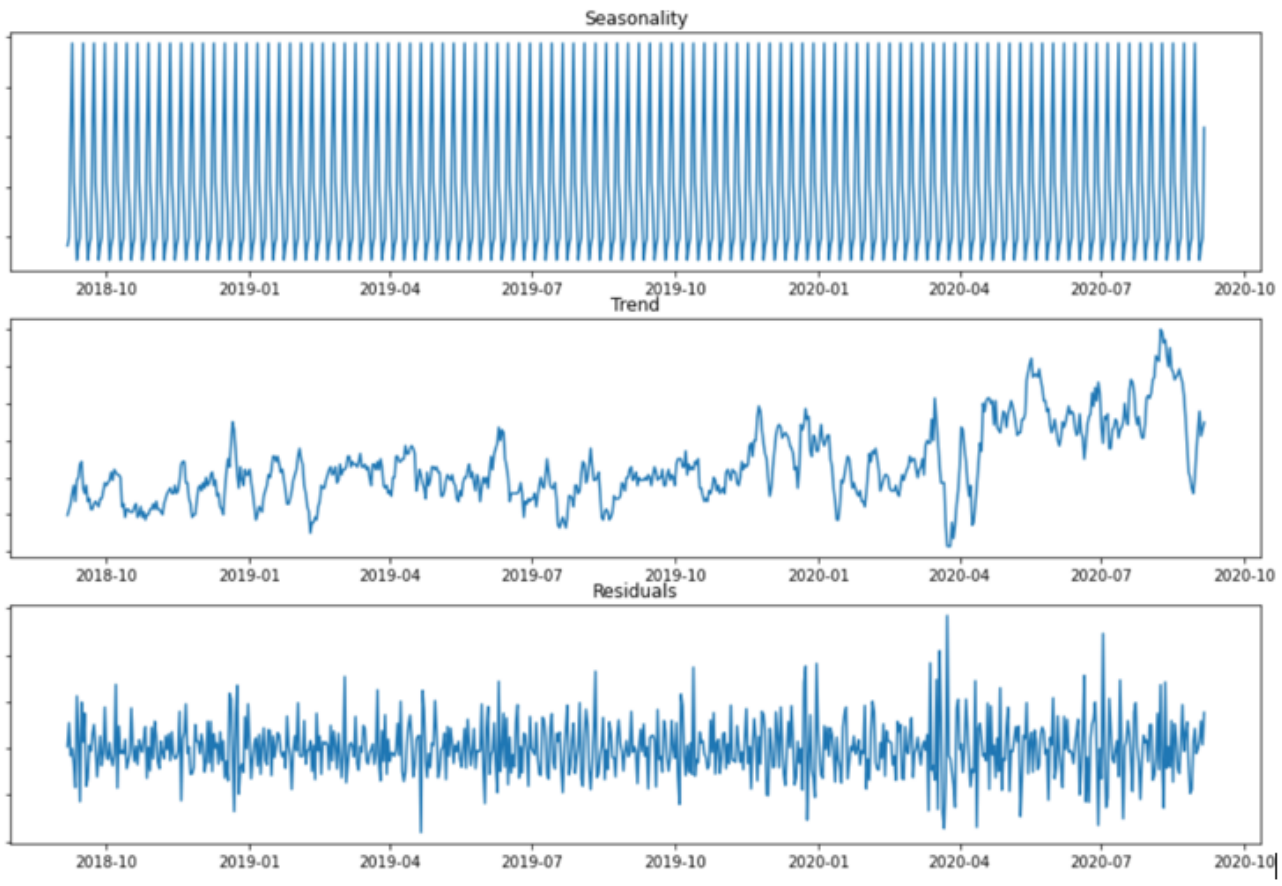
SCALE INDEPENDENT METRIC - MASE

MASE stands for Mean Absolutely Scaled Error. It is a scale free error term. Its best interpretation would be that if its value is lesser than 1, then the model has a good fit, if greater then it doesn't. The interpretation is always in reference to a Naive Forecasting model, with 1 being performance equivalent to it.

In [ ]:
```
x2.plot(figsize=(15,6))
plt.title('Products')
plt.show()
```



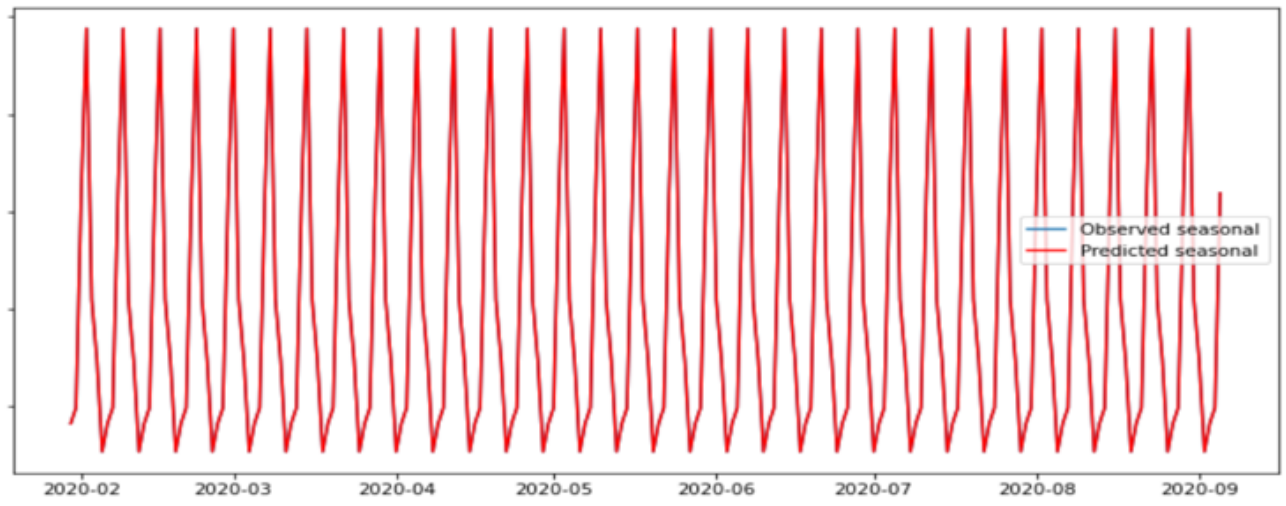
In [ ]:
```
mul = seasonal_decompose(x2,model='additive',extrapolate_trend='freq')
fig, ax = plt.subplots(3,1)
fig.set_figheight(10)
fig.set_figwidth(15)
ax[0].plot(mul.seasonal)
ax[0].set_title('Seasonality')
ax[1].plot(mul.trend)
ax[1].set_title('Trend')
ax[2].plot(mul.resid)
ax[2].set_title('Residuals')
plt.show()
```



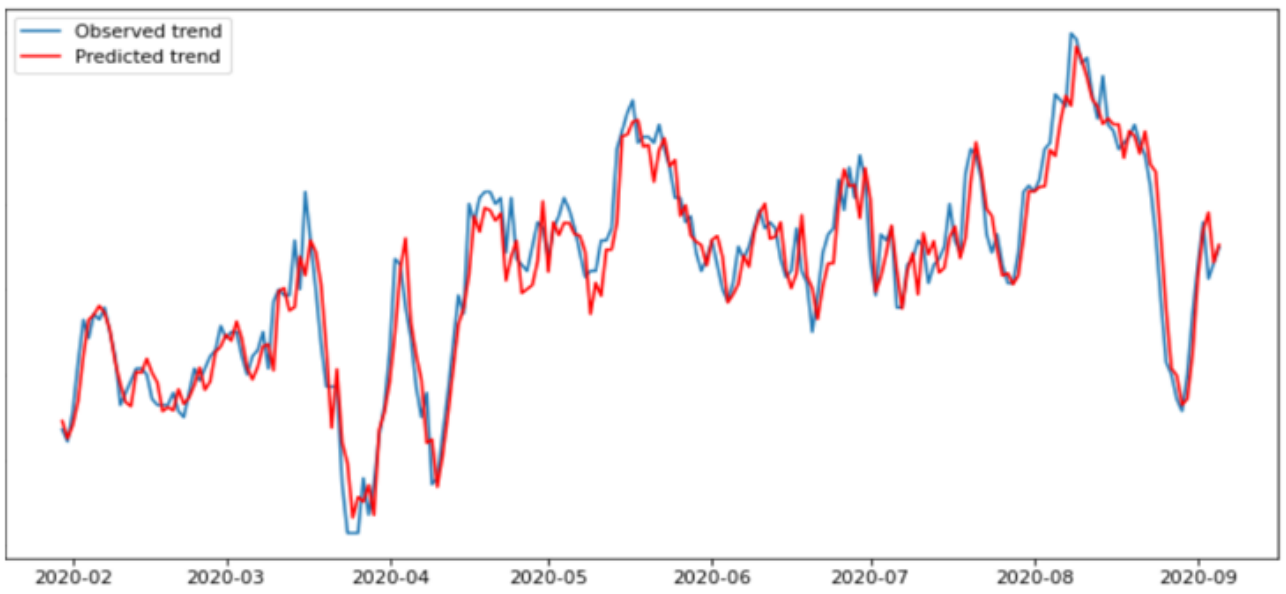
DECOMPOSITION METHODS

We can have two approaches which would best define any time series data point. In an additive time series, the components add together to make the time series. So, if we have an increasing trend, we will still see roughly the same size peaks and troughs throughout the time series. As against that, in a multiplicative time series, the components multiply together to make the time series.

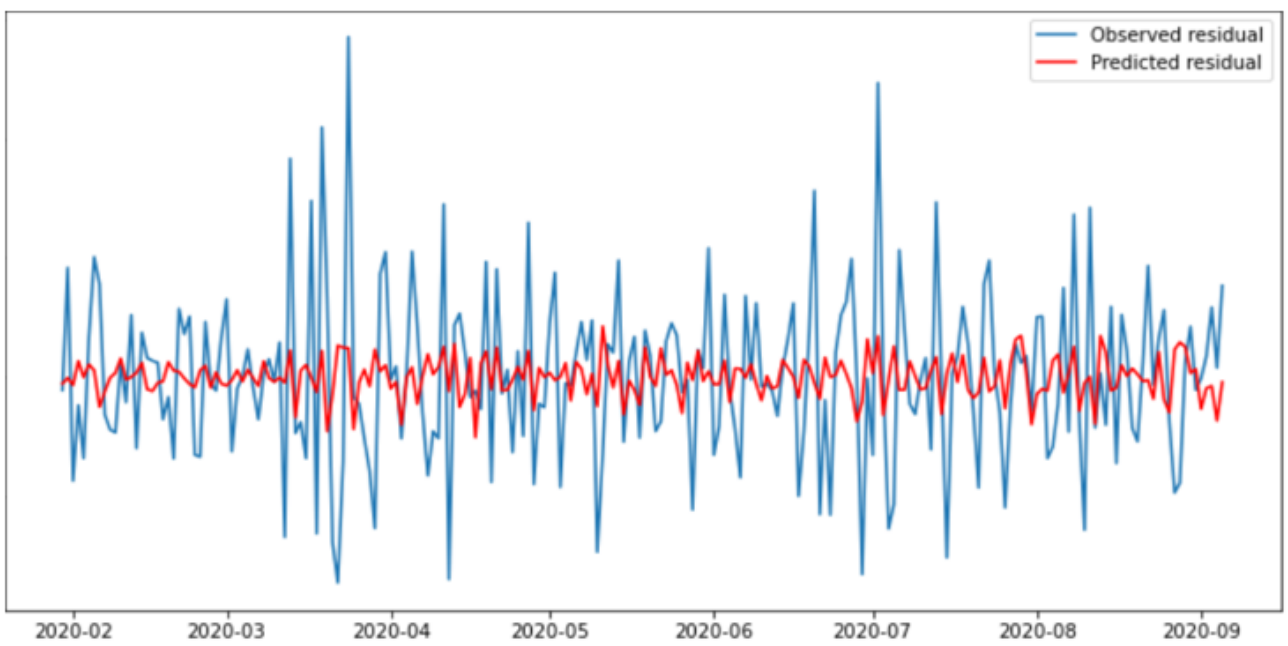
In [ ]:
```
component_dict = {'seasonal': mul.seasonal, 'trend': mul.trend, 'residual': mul.resid}

prediction_results = []

for component in ['seasonal', 'trend', 'residual']:
    historic = component_dict[component].iloc[:int(len(x2) * 0.7)].to_list()
    test = component_dict[component].iloc[int(len(x2) * 0.7):]

    predictions = []

    for i in range(len(test)):

        model = AutoReg(historic,[1,5,6,7,8,14,16,21,24,29])
        model_fit = model.fit()
        pred = model_fit.predict(start = len(historic), end = len(historic), dynamic=False)
        predictions.append(pred[0])
        historic.append(test[i])

    predictions = pd.Series(predictions, index=test.index, name=component)
    prediction_results.append(predictions)
    test_score = np.sqrt(mean_squared_error(test, predictions))
    print(f'Test for {component} MSE: {test_score}')
    # plot results
    plt.figure(figsize=(12,6))
    plt.plot(test.iloc[:], label='Observed '+component)
    plt.plot(predictions.iloc[:], color='red', label='Predicted '+component)
    plt.legend()
    plt.show()
```



t for trend MSE: 0.6871140428912562



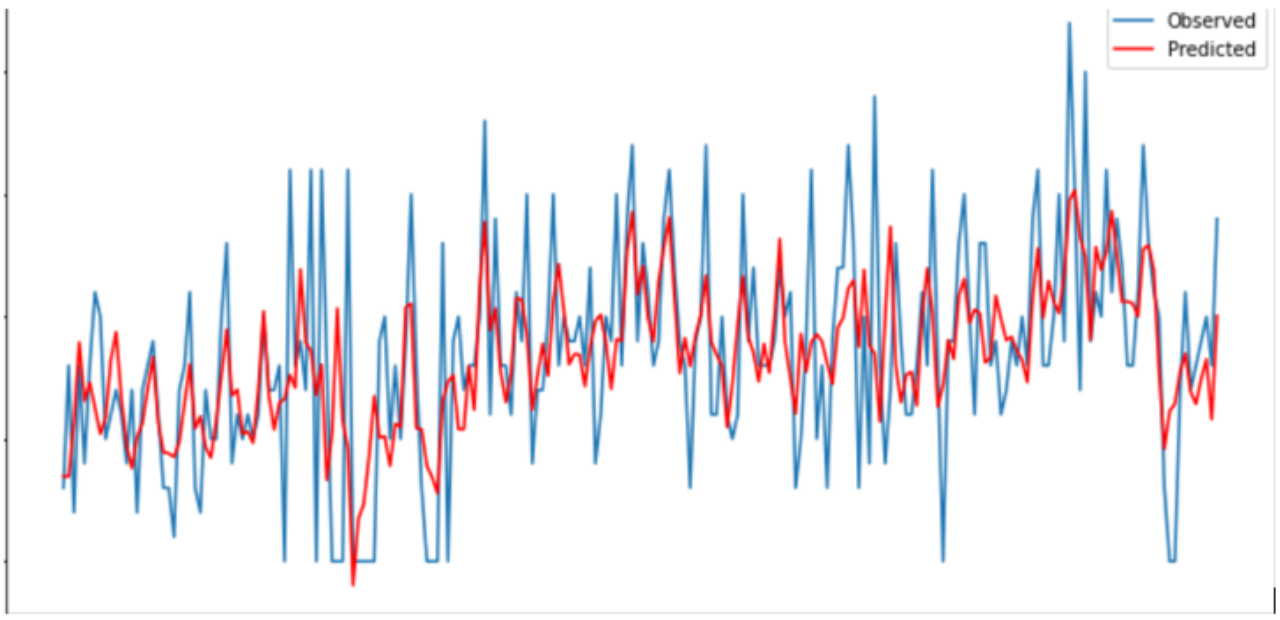
: for residual MSE: 3.531341700646103



RECOMPOSITION & MODELING STRATEGY

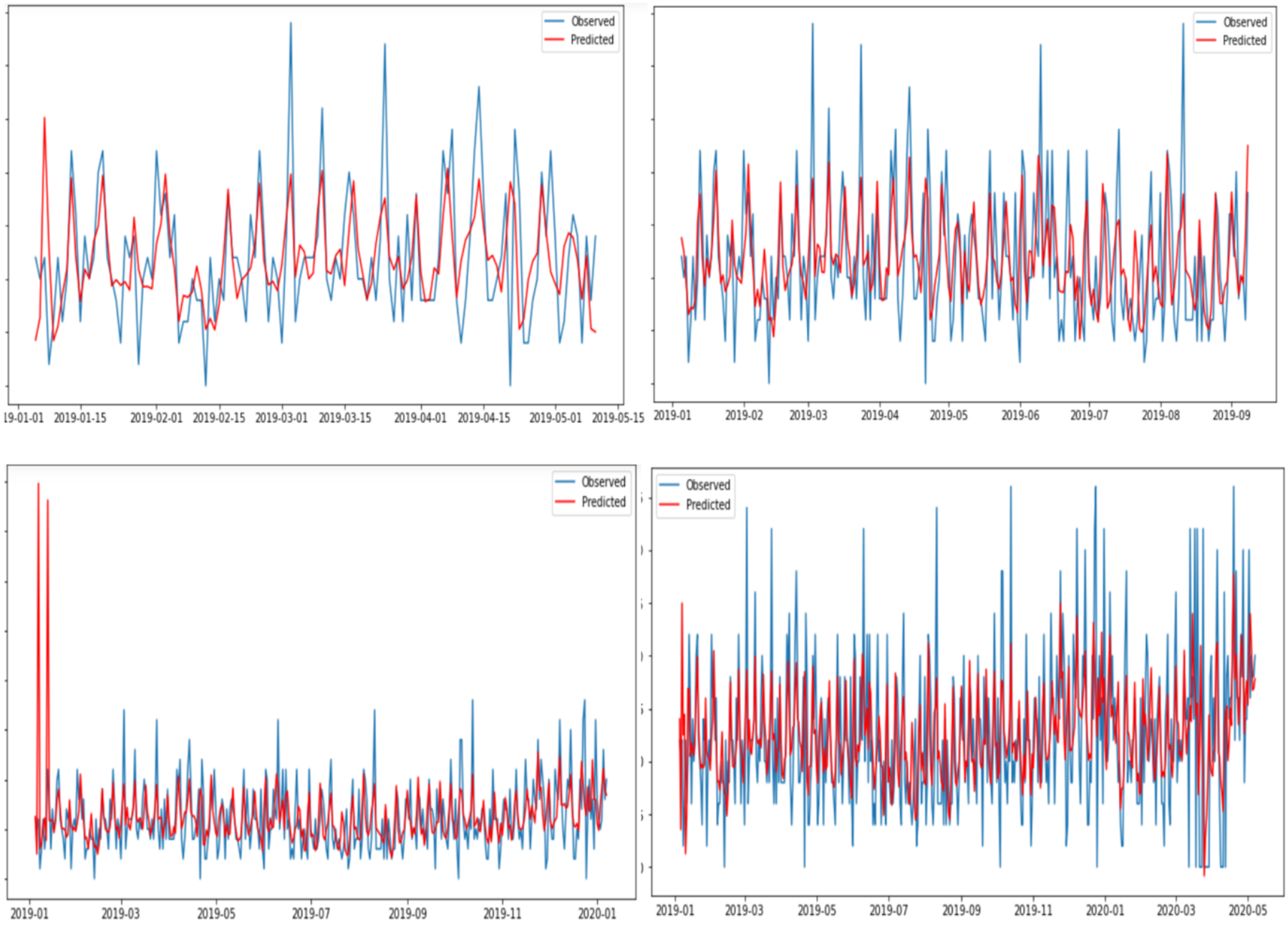
We break that data down into its individual components. Upon that, we Auto Regress each decomposed component and re-compose them together to generate a forecast. The method used for recomposition, could either be additive or multiplicative, depending on what suits best for that product time series.

In [ ]:
```
recomposed_preds = pd.concat(prediction_results,axis=1).sum(axis=1)
recomposed_preds.name = 'recomposed_preds'
plt.figure(figsize=(12,6))
plt.plot(x2.iloc[int(len(x2) * 0.7):], label='Observed')
plt.plot(recomposed_preds, color='red', label='Predicted')
plt.legend()
plt.show()

rmse=np.sqrt(mean_squared_error(x2.iloc[int(len(x2) * 0.7):], recomposed_preds))
mase=MASE(pd.Series(historic),x2.iloc[int(len(x2) * 0.7):],recomposed_preds)
print("RMSE: ", rmse)
print ("MASE: ", mase)
```



CROSS VALIDATION

In [ ]:
```
rmselist = []
maselist = []

splits = TimeSeriesSplit(n_splits=5)

print(splits)

for train_index, test_index in splits.split(x2):
    train = x2[train_index]
    test = x2[test_index]
    s=x2[:len(train)+len(test)]
    print('Observations: %d' % (len(train) + len(test)))
    print('Training Observations: %d' % (len(train)))
    print('Testing Observations: %d' % (len(test)))

for train_index, test_index in splits.split(x2):

    train = x2[train_index]
    testdata = x2[test_index]
    print("Training Size: ",len(train))
    print("Test Size: ",len(testdata))
    print('Observations: %d' % (len(train) + len(testdata)))

    s=x2[:len(train)+len(testdata)]
    mul = seasonal_decompose(s,model='additive',extrapolate_trend='freq')
    prediction_results = []

    component_dict = component_dict = {'seasonal': mul.seasonal, 'trend': mul.trend, 'residual': mul.resid}

    for component in ['seasonal', 'trend', 'residual']:
        historic = component_dict[component].iloc[:len(train)].to_list()
        test = component_dict[component].iloc[len(testdata):]
        predictions = []

        for i in range(len(test)):
            model = AutoReg(historic,[1,5,6,7,8,14,16,21,24])
            model_fit = model.fit()
            pred = model_fit.predict(start=len(historic), end=len(historic), dynamic=False)
            predictions.append(pred[0])
            historic.append(test[i])

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

    recomposed_preds = pd.concat(prediction_results,axis=1).sum(axis=1)
    recomposed_preds.name = 'recomposed_preds'
    plt.figure(figsize=(12,6))
    plt.plot(s.iloc[len(testdata):], label='Observed')
    plt.plot(recomposed_preds, color='red', label='Predicted')
    plt.legend()
    plt.show()

    rmse=np.sqrt(mean_squared_error(s.iloc[len(testdata):], recomposed_preds))
    rmselist.append(rmse)
    mase=MASE(pd.Series(historic),s.iloc[len(testdata):],recomposed_preds)
    maselist.append(mase)
```



In [195]:
```
from statistics import mean

print("Average RMSE: ", mean(rmselist))

print("Average MASE: ", mean(maselist))
```

```
Average RMSE:  2.8353330339700404
Average MASE:  0.6709016871432474
```

AUTOMATIC IDENTIFICATION OF THE DECOMPOSITION METHOD

In order to generalize the recomposition strategy over a wide range of time series patterns, we construct a function which would intelligently determine what would be the appropriate re-composition approach for the entered product, model it using autoregression and finally generate forecasts for us. The function is also capable of modeling seasonal patterns based of the frequency of their occurrence

In [ ]:
```
def acf1(x):
    return np.square(sum(acf(x)))

def ssacf(add,mult):
    return np.where(acf1(add)<acf1(mult),"additive","multiplicative")

mul = seasonal_decompose(x2 + 0.1,model='multiplicative',extrapolate_trend='freq')
add = seasonal_decompose(x2,model='additive',extrapolate_trend='freq')
model_type = ssacf(add.resid,mul.resid)

if model_type=='multiplicative':
    model = seasonal_decompose(x2+0.1,model=str(model_type),extrapolate_trend='freq')
else:
    model = seasonal_decompose(x2,model=str(model_type),extrapolate_trend='freq')
```

FORECASTING AUTOMATION FUNCTION

For detailed code, please refer: https://github.com/chinmaybake/Capstone---Time-series-Modeling (https://github.com/chinmaybake/Capstone---Time-series-Modeling)

OVERVIEW

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/)