Time Series

Tan Wen Tao Bryan
2214449
DAAA/FT/2A/01

Project Objective

- To build a time-series model to forecast the gas consumption, electricity consumption and water consumption in the future and obtain the most accurate prediction

Background Info

Forecasting is a method of examining past and current statistics movements and patterns in order to gain some insight or hints about future trends and business movements. Forecasting is looking into the future to help beneficials prepare for it accordingly.

Energy consumption is the total amount of energy required for a given process. This includes the use of electricity, gas, water, oil and biomass.

Various Factors Affecting Rate of Energy Consumption

- **Natural Factors**:
  - Rising temperature
    - Warmer winters will decrease the need for heating, reducing energy consumption. Hotter summers will increase the demand for cooling.
  - Change in energy sources
    - Mix of energy sources used in each country and state might differ with the increased temperature. Winter heating is powered by a mixture of electricity, fuel oil, and natural gas, whereas summer cooling is powered by electricity. The expected consequence of this situation is reduced demand for fuel oil and natural gas and increased demand for electricity. Using wind and solar to generate electricity to meet this demand may shift consumption of fossil fuels to more renewable sources of energy.
- **Human Factors**:
  - Agriculture
    - As a country becomes more developed, there is a higher demand for food, leading to more intensive farming techniques which require additional energy to power machinery, provide lighting and heating. The processing, manufacturing and transport of food also lead to increased energy demands. Commercial farming as a country also increase the consumption of energy. Certain times of the year like special holidays will also have a higher demand of certain agricultural products which means more energy is used to increase the supply.
  - Industry
    - Rapid industralisation leads to the development of manufacturing and processing industries which consume large levels of energy especially due to the rise in technological advancements. This increase the huge demand for many products and services which causes the industries to consume energy to build up the supply to meet the demand.
  - Economical Development
    - As a country GDP increases, more citizens in the country will rise to a higher SES. The demand for energy grows with the increased purchases of domestic appliances, leisure and recreation activities.
  - Global Crisis
    - This includes many crisis that can affect how people consume energy at home and in industries such as: Pandemics, Geopolitical Developments, Wars
  - Political Development
    - Certain countries like Singapore have passed some energy-saving policies that can reduce the consumption of energy like establishing energy efficiency sectors, encouraging the adoption of best practices and energy-saving technologies.

Advantages of Energy Consumption Forecasting:

- **Helps in Scheduling**: Help to prepare the organisation/authority for the future. Forecasting helps them to prepare of what the future may behold for the society/business which allows them to plan out measures in place for them.
- **Weak Spots Detection**: Another benefit of forecasting is that it can help the manager find any weak points that the company may have or overlooked areas. When attention has been drawn to these areas, successful controls and preparation strategies to fix them can be put into practice by the manager.
- **Enhances Coordination and Control**: Information and data from a lot of external and internal sources are needed for forecasting. This knowledge is obtained from different internal sources by the various managers and employees. Thus, nearly all of the organization's divisions and verticals are involved in the forecasting process. This facilitates greater cooperation and communication between them.

Step 1: EDA (Distribution)

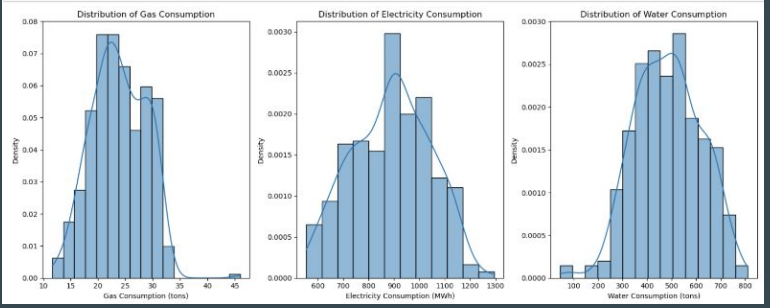
```
print(energyConsumption_ds.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 397 entries, 0 to 396
Data columns (total 4 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   DATE                            397 non-null    object
 1   Gas Consumption (tons)          397 non-null    float64
 2   Electricity Consumption (MWh)   397 non-null    float64
 3   Water Consumption (tons)        397 non-null    float64
dtypes: float64(3), object(1)
memory usage: 12.5+ KB
None
```

Observations:

- 397 rows and 4 columns
- 1 categorical column, 3 numerical columns
- All columns has an equal number of observations
- Date needs to be in date type and not object type
- No missing values in every row

```
#Plot density based histograms to show distribution
fig, ax = plt.subplots(1, 3, figsize=(15,6))
sns.histplot(data=energyConsumption_ds, x='Gas Consumption (tons)', kde=True, stat="density", ax=ax[0])
ax[0].set_title("Distribution of Gas Consumption")
ax[0].set_yticks(np.arange(0,0.081, 0.01))

sns.histplot(data=energyConsumption_ds, x='Electricity Consumption (MWh)', kde=True, stat="density", ax=ax[1])
ax[1].set_title("Distribution of Electricity Consumption")

sns.histplot(data=energyConsumption_ds, x='Water Consumption (tons)', kde=True, stat="density", ax=ax[2])
ax[2].set_title("Distribution of Water Consumption")

plt.tight_layout()
plt.show()
```



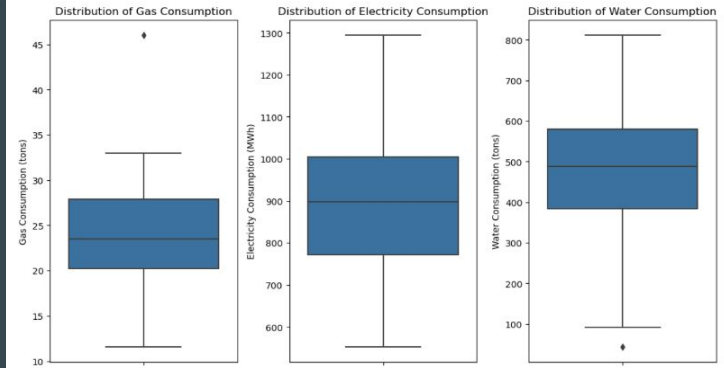

```
#Descriptive Stats
energy_stats=energyConsumption_ds.describe(include="all").T
display(energy_stats)
```

	count	mean	std	min	25%	50%	75%	max
Gas Consumption (tons)	397.0	23.785139	4.903452	11.6	20.2	23.5	27.9	46.0
Electricity Consumption (MWh)	397.0	888.472544	153.877594	553.2	771.1	897.8	1005.2	1294.0
Water Consumption (tons)	397.0	484.953652	133.908863	44.4	384.4	487.4	580.2	811.0

Observations:

- Between 75% and max value for Gas Consumption (tons), there was a huge jump in values compared to the other first 75% of the values, suggesting that there is an outlier in the last 25% of the data

```
#Boxplots to show median and distribution of data
fig, ax = plt.subplots(1, 3, figsize=(11,6))
sns.boxplot(data=energyConsumption_ds, y='Gas Consumption (tons)',  ax=ax[0])
ax[0].set_title("Distribution of Gas Consumption")

sns.boxplot(data=energyConsumption_ds, y='Electricity Consumption (MWh)', ax=ax[1])
ax[1].set_title("Distribution of Electricity Consumption")

sns.boxplot(data=energyConsumption_ds, y='Water Consumption (tons)', ax=ax[2])
ax[2].set_title("Distribution of Water Consumption")

plt.tight_layout()
plt.show()
```



Observations:

- Gas consumption is slightly positively skewed but it is generally normally distributed apart from the outlier.
- Generally, the electricity and water consumption seem to be normally distributed.
- Gas consumption & Water consumption has 1 outlier each that do not really seem to affect the normal distribution.

Step 1: EDA (TSD)

Time Series Decomposition

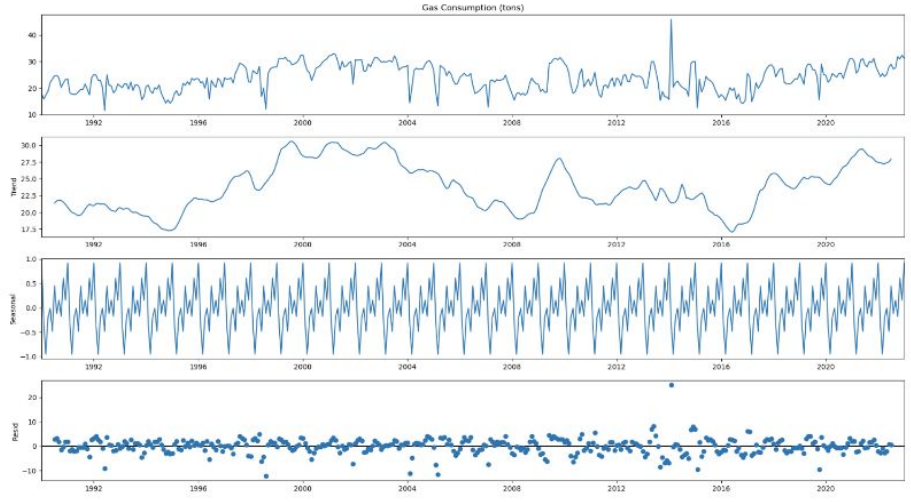
- Breaking down the time series into its underlying components to further understand the patterns, fluctuations and trend present
- Some tests that I will perform:
  1. Seasonal Decomposition
  2. Test for Stationarity
     - ADF test
     - KPSS test
  3. Identification of orders (p,d,q),(P,D,Q,S)
     - Differencing
     - Autocorrelation Analysis
  4. Test for Causality
     - Granger's Causality Test
  5. Test for Cointegration
     - Engle Granger Cointegration Test
     - Johansen's Cointegration Test

Seasonal Decomposition

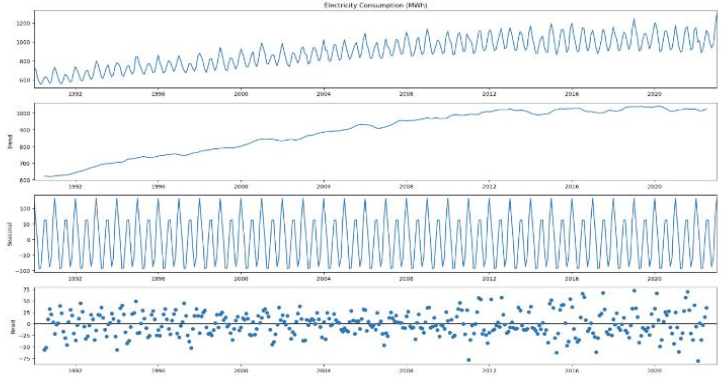
Time series is being represented by either a sum or a product of three components - (Trend, Seasonality & Random Component)

```
# Perform seasonal decomposition
for i in energyConsumption_ds.columns:
    print(f'Column: {i}')
    plt.rc("figure",figsize=(18,10))
    decomposition = seasonal_decompose(energyConsumption_ds[i])
    decomposition.plot()
    plt.show()
```

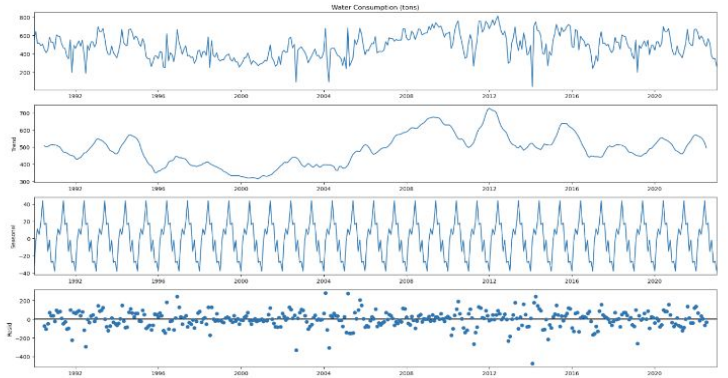
Column: Gas Consumption (tons)



Column: Electricity Consumption (MWh)



Column: Water Consumption (tons)



Observations:

- There is a slight upward linear in the trend for Electricity Consumption but non-linear trend for Gas Consumption & Water Consumption
- Seasonality seems to be occurring repeatedly similarly in a cyclic manner for all the features
- Pattern repeats once every year ((1996-1992)/4 cyclic patterns) = 1 cycle per year
- Residuals seem to be quite generally constant for Electricity Consumption
- Residuals seem to show a sharp spike around 2014 and anomal decrease for other years for Gas Consumption
- Residuals seem to show a very sharp decrease around 2014 for Water Consumption

Step 1: EDA (ADF Test)

Test for Stationarity (Augmented Dickey-Fuller Test)

A stationary series is one where the properties of mean, variance and covariance do not vary with time, whereas a non-stationary series has properites which vary with time. ADF test determines how strongly a time series is defined by a trend. It can be determined if the series is stationary or not.

- H_0: The series is non-stationary/The series has a unit root.
- H_1: The series is stationary/The series has no unit root.

```python
#ADF test
def adf_test(time_series, significant_value=0.05, name="", verbose=False):
    unit_root=adfuller(time_series, autolag="BIC")
    test_statistic = round(unit_root[0],4)
    p_value = round(unit_root[1],4)
    n_lags = round(unit_root[2], 4)

    #Print summary
    print(f'ADF Test on {name}')
    print('-'*50)
    print(f'Significance level = {significant_value}')
    print(f'Test Statistic = {test_statistic}')
    print(f'No of Lags = {n_lags}')

    #Critical values for each percentage
    for key, val in unit_root[4].items():
        print(f"Critical Value ({key})= {round(val, 3)}")

    #Find out if p-value is smaller than or larger than the significance value
    if p_value<=significant_value:
        print(f'P-Value = {p_value} Null hypothesis is rejected.')
        print(f'The series is stationary.')
    else:
        print(f'P-Value = {p_value} Insufficient evidence to reject the null hypothesis.')
        print(f'The series is non-stationary.')

#Initiating functions
for column in energyConsumption_ds.columns:
    adf_test(energyConsumption_ds[column], name=column)
    print()
```

```
ADF Test on Gas Consumption (tons)
--------------------------------------------------
Significance level = 0.05
Test Statistic = -6.5666
No of Lags = 1
Critical Value (1%)= -3.447
Critical Value (5%)= -2.869
Critical Value (10%)= -2.571
P-Value = 0.0 Null hypothesis is rejected.
The series is stationary.

ADF Test on Electricity Consumption (MWh)
--------------------------------------------------
Significance level = 0.05
Test Statistic = -2.0912
No of Lags = 12
Critical Value (1%)= -3.447
Critical Value (5%)= -2.869
Critical Value (10%)= -2.571
P-Value = 0.2481 Insufficient evidence to reject the null hypothesis.
The series is non-stationary.

ADF Test on Water Consumption (tons)
--------------------------------------------------
Significance level = 0.05
Test Statistic = -7.0564
No of Lags = 1
Critical Value (1%)= -3.447
Critical Value (5%)= -2.869
Critical Value (10%)= -2.571
P-Value = 0.0 Null hypothesis is rejected.
The series is stationary.
```

Observations:

- The series for Gas Consumption & Water Consumption has a p-value of 0 which means that those time series data are stationary.
- Test statistic of -6.5666 for Gas Consumption & test statistic of -7.0564 for Water Consumption are lower than all the critical values, which further shows that null hypothesis is rejected and both series are stationary.
- The series for Electricity Consumption has a p-value of 0.248 which means that time series data is non-stationary.
- Test statistic of -2.0912 for Electricity Consumption is greater than all the critical values, which further shows that there are insufficient evidence to reject the null hypothesis and the series is non-stationary.

Step 1: EDA (KPSS Test)

Test for Stationarity (Kwiatkowski-Phillips-Schmidt-Shin Test)

KPSS test is a type of unit root test that tests for the stationarity of a given series around a deterministic trend.

- H_0: The series is trend stationary or the series has no unit root.
- H_1: The series is non-stationary or series has a unit root.

```
#KPSS test
warnings.filterwarnings('ignore')
def kpss_test(time_series, significant_value=0.05, name="", verbose=False):
    unit_root=kpss(time_series)
    test_statistic = round(unit_root[0],4)
    p_value = round(unit_root[1],4)
    n_lags = round(unit_root[2], 4)

    #Print summary
    print(f'KPSS Test on {name}')
    print('-'*50)
    print(f'Significance level = {significant_value}')
    print(f'Test Statistic = {test_statistic}')
    print(f'No of Lags = {n_lags}')

    #Critical values for each percentage
    for key, val in unit_root[3].items():
        print(f"Critical Value ({key})= {round(val, 3)}")

    #Find out if p-value is smaller than or larger than the significance value
    if p_value<=significant_value:
        print(f'P-Value = {p_value} Null hypothesis is rejected.')
        print(f'The series is non-stationary.')
    else:
        print(f'P-Value = {p_value} Insufficient evidence to reject the null hypothesis.')
        print(f'The series is trend stationary.')

#Initiating functions
for column in energyConsumption_ds.columns:
    kpss_test(energyConsumption_ds[column], name=column)
    print()
```

```
KPSS Test on Gas Consumption (tons)
--------------------------------------------------
Significance level = 0.05
Test Statistic = 0.3402
No of Lags = 10
Critical Value (10%)= 0.347
Critical Value (5%)= 0.463
Critical Value (2.5%)= 0.574
Critical Value (1%)= 0.739
P-Value = 0.1 Insufficient evidence to reject the null hypothesis.
The series is trend stationary.

KPSS Test on Electricity Consumption (MWh)
--------------------------------------------------
Significance level = 0.05
Test Statistic = 3.5316
No of Lags = 10
Critical Value (10%)= 0.347
Critical Value (5%)= 0.463
Critical Value (2.5%)= 0.574
Critical Value (1%)= 0.739
P-Value = 0.01 Null hypothesis is rejected.
The series is non-stationary.

KPSS Test on Water Consumption (tons)
--------------------------------------------------
Significance level = 0.05
Test Statistic = 0.8388
No of Lags = 10
Critical Value (10%)= 0.347
Critical Value (5%)= 0.463
Critical Value (2.5%)= 0.574
Critical Value (1%)= 0.739
P-Value = 0.01 Null hypothesis is rejected.
The series is non-stationary.
```

Stationarity Observations

- For Gas Consumption, ADF test & KPSS test shows that the the series is stationary. Hence, the series is stationary.
- For Electricity Consumption, ADF test & KPSS shows that the series are non-stationary. Hence, the series is non-stationary.
- For Water Consumption, ADF test shows that the series is stationary while KPSS test shows that the series is non-stationary. Hence, the series is difference stationary.

Observations:

- For Gas Consumption, test statistics is lower than their individual critical values which means that there is insufficient evidence to reject the null hypothesis, implying that the series are trend stationary.
- For Electricity Consumption & Water Consumption, both test statistic are larger than the individual critical values which means that the null hypothesis is rejected and both time series are non-stationary.

Possible Outcomes to Truly Stationary Series

- Case 1: Both tests conclude that the given series is stationary – The series is stationary
- Case 2: Both tests conclude that the given series is non-stationary – The series is non-stationary
- Case 3: ADF concludes non-stationary, and KPSS concludes stationary – The series is trend stationary. To make the series strictly stationary, the trend needs to be removed in this case. Then the detrended series is checked for stationarity.
- Case 4: ADF concludes stationary, and KPSS concludes non-stationary – The series is difference stationary. Differencing is to be used to make series stationary. Then the differenced series is checked for stationarity

Step 1: EDA (Differencing)

Identification of Orders (p,d,q)

- Before we identify the orders for the time series models, we have to ensure that the time series is stationary first by applying differencing on the non-stationary time series.
- Next, we will use the ACF & PACF plot to determine the orders to be used for ARIMA & SARIMA.

Differencing (1st Round)

- In this case, we need to make sure that Electricity Consumption is stationary first.
- Water Consumption has to be differenced first and check whether it is stationary, since it is difference stationary.
- First-Order Differencing:

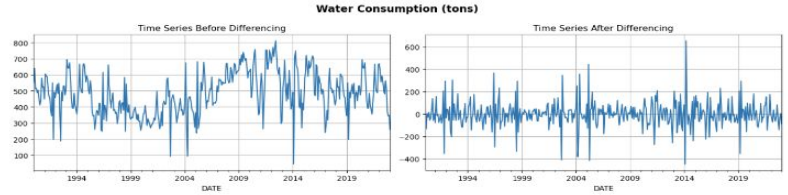
$$y'_t = y_t - y_{t-1}$$

- y_t: value with time

Water Consumption

```
#Perform differencing on Water Consumption (order=1)
water_diff=diff(energyConsumption_ds["Water Consumption (tons)"], k_diff=1)
adf_result1=adfuller(energyConsumption_ds["Water Consumption (tons)"], autolag="BIC")
adf_result2=adfuller(water_diff, autolag="BIC")
kpss_result1=kpss(energyConsumption_ds["Water Consumption (tons)"])
kpss_result2=kpss(water_diff)

print('ADF-test (p-value before differencing): %f' % adf_result1[1])
print('ADF-test (p-value after differencing): %f' % adf_result2[1])
print('KPSS-test (p-value before differencing): %f' % kpss_result1[1])
print('KPSS-test (p-value after differencing): %f' % kpss_result2[1])

fig, ax = plt.subplots(1, 2, figsize=(14, 4))
energyConsumption_ds["Water Consumption (tons)"].plot(ax=ax[0])
ax[0].grid(True)
water_diff.plot(ax=ax[1])
ax[1].grid(True)
ax[0].set_title('Time Series Before Differencing')
ax[1].set_title('Time Series After Differencing')
fig.suptitle("Water Consumption (tons)", fontsize=14, fontweight="bold")
plt.tight_layout()
plt.show()
```

```
ADF-test (p-value before differencing): 0.000000
ADF-test (p-value after differencing): 0.000000
KPSS-test (p-value before differencing): 0.010000
KPSS-test (p-value after differencing): 0.100000
```



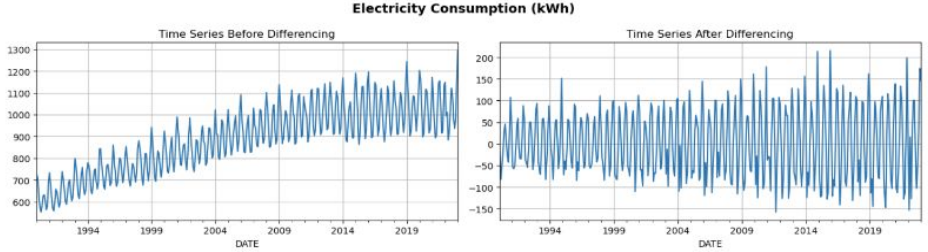
Observations:

- ADF test shows a p-value smaller than the significance value of 0.05, null hypothesis is rejected.
- KPSS test shows a p-value larger than the significance value of 0.05, little evidence to show that null hypothesis is rejected.
- This show that for Water Consumption, the series differenced order of 1 is stationary.

Electricity Consumption

```
#Perform differencing on Electricity Consumption (order=1)
electricity_diff=diff(energyConsumption_ds["Electricity Consumption (MWh)"], k_diff=1)
adf_result1=adfuller(energyConsumption_ds["Electricity Consumption (MWh)"], autolag="BIC")
adf_result2=adfuller(electricity_diff, autolag="BIC")
kpss_result1=kpss(energyConsumption_ds["Electricity Consumption (MWh)"])
kpss_result2=kpss(electricity_diff)


print('ADF-test (p-value before differencing): %f' % adf_result1[1])
print('ADF-test (p-value after differencing): %f' % adf_result2[1])
print('KPSS-test (p-value before differencing): %f' % kpss_result1[1])
print('KPSS-test (p-value after differencing): %f' % kpss_result2[1])

fig, ax = plt.subplots(1, 2, figsize=(14, 4))
energyConsumption_ds["Electricity Consumption (MWh)"].plot(ax=ax[0])
ax[0].grid(True)
electricity_diff.plot(ax=ax[1])
ax[1].grid(True)
ax[0].set_title('Time Series Before Differencing')
ax[1].set_title('Time Series After Differencing')
fig.suptitle("Electricity Consumption (kWh)", fontsize=14, fontweight="bold")
plt.tight_layout()
plt.show()
```

```
ADF-test (p-value before differencing): 0.248052
ADF-test (p-value after differencing): 0.000000
KPSS-test (p-value before differencing): 0.010000
KPSS-test (p-value after differencing): 0.100000
```

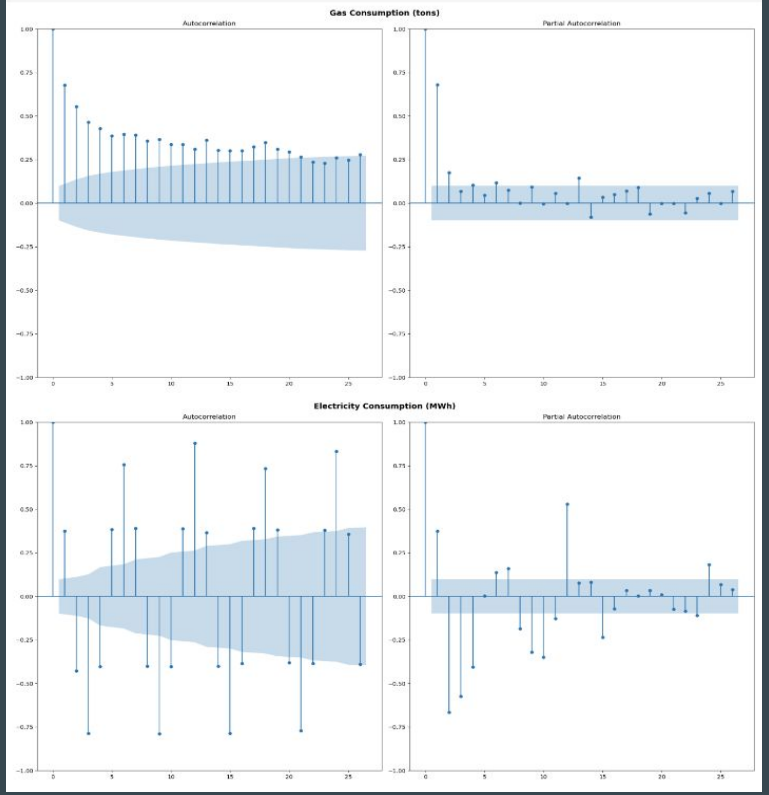


Observations:

- ADF test shows a p-value smaller than the significance value of 0.05, null hypothesis is rejected.
- KPSS test shows a p-value larger than the significance value of 0.05, little evidence to show that null hypothesis is rejected.
- This show that for Electricity Consumption, the series differenced order of 1 is stationary.

Step 1: EDA (ACF/PACF)

```
#Plot acf and pacf graphs
for i in energyConsumption_ds.columns:
    fig,ax=plt.subplots(1,2, tight_layout=True)
    plot_acf(energyConsumption_ds[i], ax=ax[0])
    plot_pacf(energyConsumption_ds[i], ax=ax[1])
    fig.suptitle(i, fontsize=14, fontweight="bold")
    plt.show()
```



Autocorrelation Analysis

Autocorrelation is the correlation of a time series and its lagged version over time.

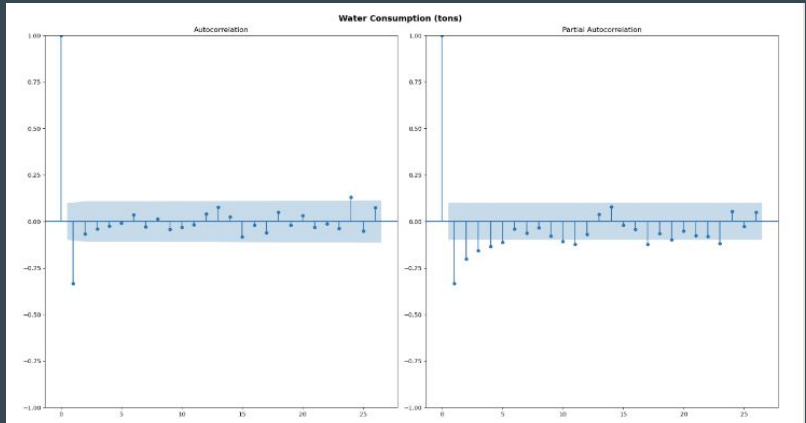
- **ACF**: Measures the degree of similarity between a given time series and the lagged version of that time series at the various intervals observed
- **PACF**: Measures the degree of similarity between a given time series and the lagged version of that time series at the various intervals observed in which only the direct effect has been shown, and all other intermediary effects are removed from the given time series

AR & MA Process (ARIMA)

- ARIMA will be one of the models commonly used for time series, made up of Auto Regressive (AR), Integrated (I) & Moving Average (MA) process
- AR(p) process: Current values depend on its own p-previous values
- MA (q) process: The current deviation from mean depends on q-previous deviations

How to identify orders of p and q from ACF & PACF

- Model is AR if the ACF trails off after a lag and has a hard cut-off in the PACF after a lag. This lag is taken as the value for p.
- Model is MA if the PACF trails off after a lag and has a hard cut-off in the ACF after a lag. This lag is taken as the value for q.
- Model is a mix of AR & MA if both the ACF & PACF trail off.



Observations:

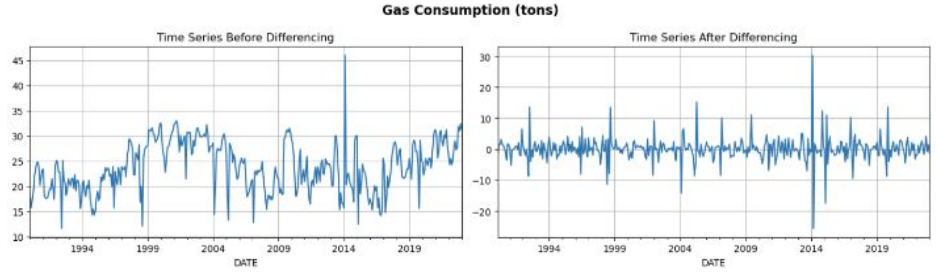
- For Gas Consumption, even though there is a tailoff in the ACF plot, it has a relatively high number of lags (>10) hence an order of differencing is needed.
- For Electricity Consumption, there is a sine-like curve which shows a noticeable seasonal pattern after the first order of differencing, hence seasonal differencing is needed. A significant lag spike is noticed to happen every 6 lags and 12 lags.
- For Water Consumption, there is a significant lag spike at lag 1 in ACF plot & there is a gradual decrease in the PACF plot, so we shall use Moving Average model with q parameter of 1.

Step 1: EDA (Differencing)

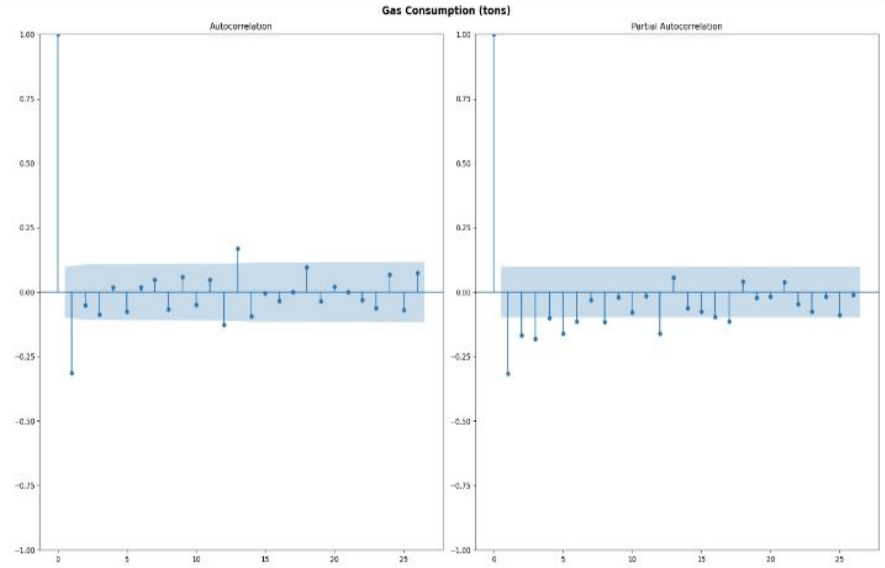
Gas Consumption

```
#Perform differencing on Gas Consumption (order=1)
gas_diff=diff(energyConsumption_ds["Gas Consumption (tons)"], k_diff=1)
adf_result1=adfuller(energyConsumption_ds["Gas Consumption (tons)"], autolag="BIC")
adf_result2=adfuller(gas_diff, autolag="BIC")
kpss_result1=kpss(energyConsumption_ds["Gas Consumption (tons)"])
kpss_result2=kpss(gas_diff)


print('ADF-test (p-value before differencing): %f' % adf_result1[1])
print('ADF-test (p-value after differencing): %f' % adf_result2[1])
print('KPSS-test (p-value before differencing): %f' % kpss_result1[1])
print('KPSS test (p-value after differencing): %f' % kpss_result2[1])

fig, ax = plt.subplots(1, 2, figsize=(14, 4))
energyConsumption_ds["Gas Consumption (tons)"].plot(ax=ax[0])
ax[0].grid(True)
gas_diff.plot(ax=ax[1])
ax[1].grid(True)
ax[0].set_title('Time Series Before Differencing')
ax[1].set_title('Time Series After Differencing')
fig.suptitle("Gas Consumption (tons)", fontsize=14, fontweight="bold")
plt.tight_layout()
plt.show()
```

```
ADF-test (p-value before differencing): 0.000000
ADF-test (p-value after differencing): 0.000000
KPSS-test (p-value before differencing): 0.100000
KPSS-test (p-value after differencing): 0.100000
```




```
#Plot acf & pacf plot for Gas Consumption
fig,ax=plt.subplots(1,2, tight_layout=True)
plot_acf(energyConsumption_ds["Gas Consumption (tons)"], ax=ax[0])
plot_pacf(energyConsumption_ds["Gas Consumption (tons)"], ax=ax[1])
fig.suptitle("Gas Consumption (tons)", fontsize=14, fontweight="bold")
plt.show()
```



Observations:

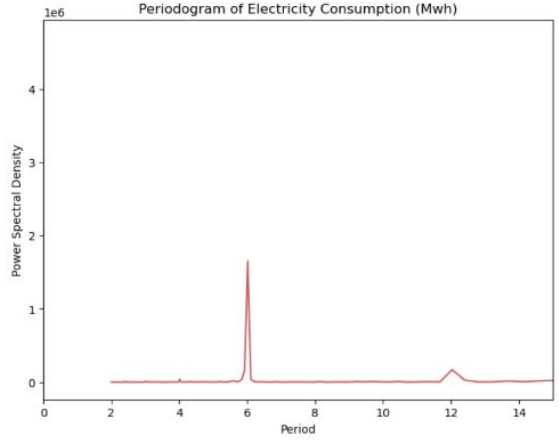
- For Gas Consumption, there is a significant lag spike at lag 1 in ACF plot & there is a gradual decrease in the PACF plot, so we shall use Moving Average model with q parameter of 1.

Step 1: EDA (Seasonality)

Electricity Consumption (Periodogram)

- Used to reveal underlying periodic components present in a time series dataset.
- Helps identify seasonal patterns in time series data
- Identify the most dominant frequency

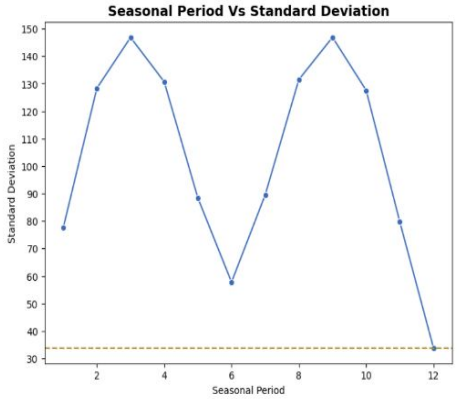
```
#Plot Periodogram
freq, power = periodogram(energyConsumption_df["Electricity Consumption (MWh)"])
plt.figure(figsize=(8, 6))

period = 1/freq
plt.plot(period, power, color='indianred')

plt.xlim(0, 15)
plt.xticks(np.arange(0, 15, 2))
plt.xlabel('Period')
plt.ylabel('Power Spectral Density')
plt.title('Periodogram of Electricity Consumption (Mwh)')
plt.show()
```



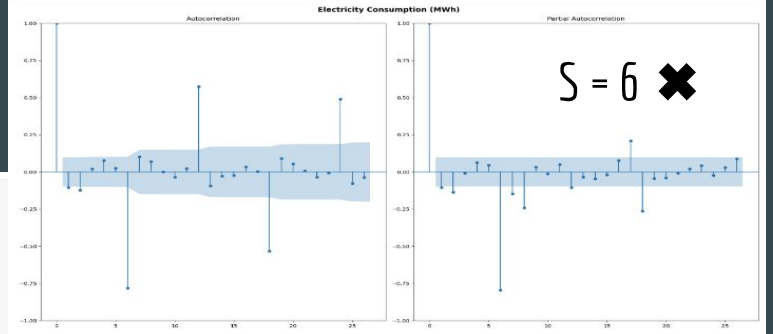
Observations:

- Even though the periodogram clearly shows that 6th month is the dominant frequency, 12th month also has a potential frequency, so these 2 can be the potential seasonality parameters for seasonality component.
- We shall look at the acf & pacf plot to decide whether to use S=6 or S=12 as the seasonality.

```
#Calculate the seasonal period with the lowest standard deviation
S = np.arange(1, 13)
season_std = []
for season in S:
    data = energyConsumption_ds["Electricity Consumption (MWh)"]
    seasonality = data.diff(periods=season).fillna(0)
    std = seasonality.std()
    season_std.append(std)

#Plot a graph (Seasonal Period Vs Standard Deviation)
plt.figure(figsize=(8,6))
ax=plt.axes()
ax=sns.lineplot(x=S, y=season_std, marker="o", color="#316FBE")
#Get the lowest standard deviation
lowest_std = min(season_std)
plt.axhline(y=lowest_std, color="#b58900", linestyle="--")
plt.yticks(np.arange(30, 151, 10))

plt.xlabel("Seasonal Period")
plt.ylabel("Standard Deviation")
plt.title("Seasonal Period Vs Standard Deviation", fontweight="bold", fontsize=14)
plt.show()
```



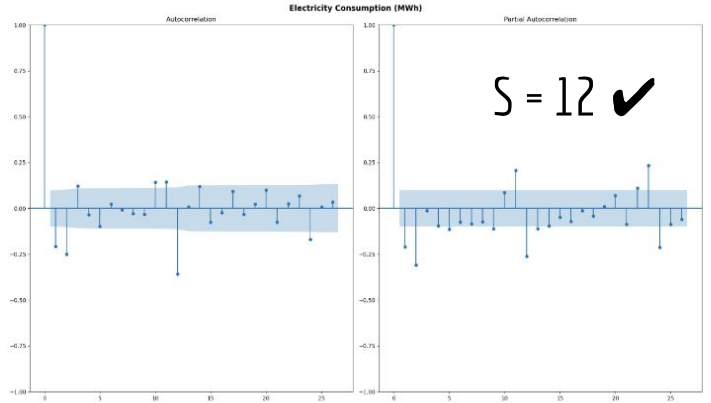

```
#Plot acf & pacf for Electricity Consumption
fig,ax=plt.subplots(1,2, tight_layout=True)
plot_acf(energyConsumption_ds["Electricity Consumption (MWh)"], ax=ax[0])
plot_pacf(energyConsumption_ds["Electricity Consumption (MWh)"], ax=ax[1])
fig.suptitle("Electricity Consumption (MWh)", fontsize=14, fontweight="bold")
plt.show()
```



S = 6 ✖

Observations:

- For S=6, we can see that there is still an underlying seasonality (sin-curve) that can be seen in the acf plot and there is an increasing threshold which suggest that an additional order of differencing may be required.
- Furthermore, in the case of strong seasonality, as observed in our time series, D=1. A rule of thumb is that d + D should not be greater than 2.
- So, we have to look at another seasonal period.

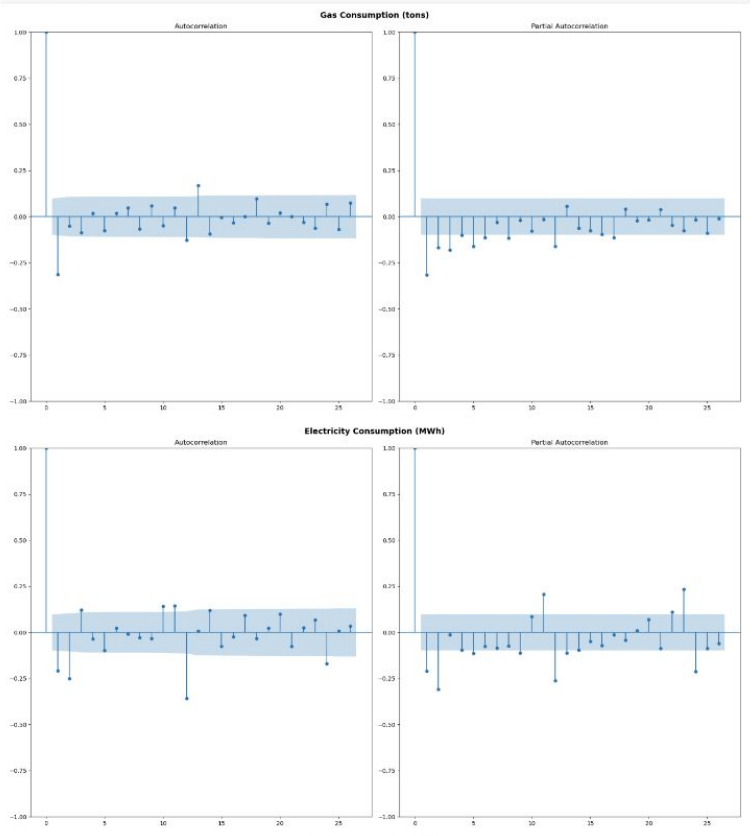


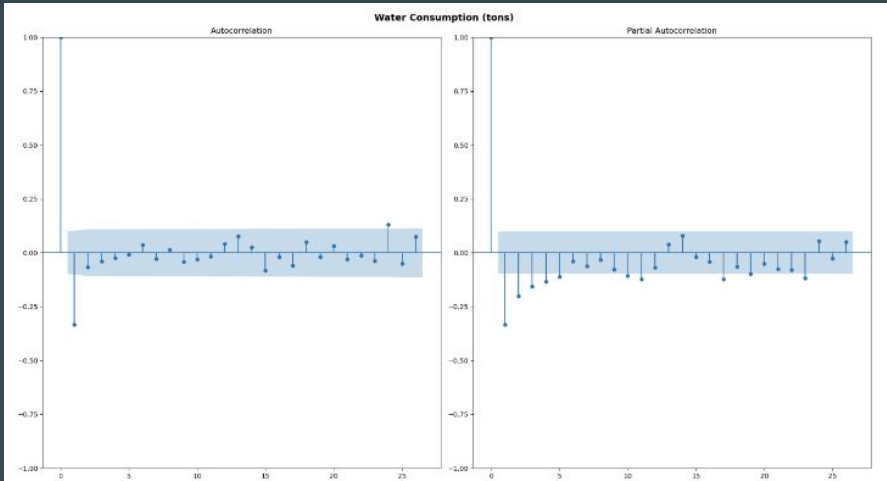
S = 12 ✔

Observations:

- For Electricity Consumption, there is a significant lag spike at lag 2 in PACF plot & a significant lag at lag 3 in the ACF plot before the threshold level cuts it off, so we shall use a mix of Auto Regressive & Moving Average model with p parameter of 2 and a q parameter of 3.
- According to the Duke University article, if the autocorrelation of the appropriately differenced series is positive at lag s, where s is the number of periods in a season, then consider adding an SAR term to the model. If the autocorrelation of the differenced series is negative at lag s, consider adding an SMA term to the model.
- You should try to avoid using more than one or two seasonal parameters (SAR+SMA) in the same model, as this is likely to lead to overfitting of the data and/or problems in estimation. Hence, I will consider P=0, Q=1.

Step 1: EDA (Interpreting Components)





Observations:

- For Gas Consumption, there is a significant lag spike at lag 1 in ACF plot & there is a gradual decrease in the PACF plot, so we shall use Moving Average model with q parameter of 1.
- For Electricity Consumption, there is a significant lag spike at lag 2 in PACF plot & a significant lag at lag 3 in the ACF plot before the threshold level cuts it off, so we shall use a mix of Auto Regressive & Moving Average model with p parameter of 2 and a q parameter of 3.
- You should try to avoid using more than one or two seasonal parameters (SAR+SMA) in the same model, as this is likely to lead to overfitting of the data and/or problems in estimation. Hence, I will consider P=0, Q=1.
- For Water Consumption, there is a significant lag spike at lag 1 in ACF plot & there is a gradual decrease in the PACF plot, so we shall use Moving Average model with q parameter of 1.

Order of (p,d,q) and (P,D,Q,S)

- Gas Consumption = (0,1,1)
- Electricity Consumption = (2,1,3),(0,1,1,12)
- Water Consumption = (0,1,1)

Step 1: EDA (Test for Causality)

Test for Causality (Grangers Causality Test)

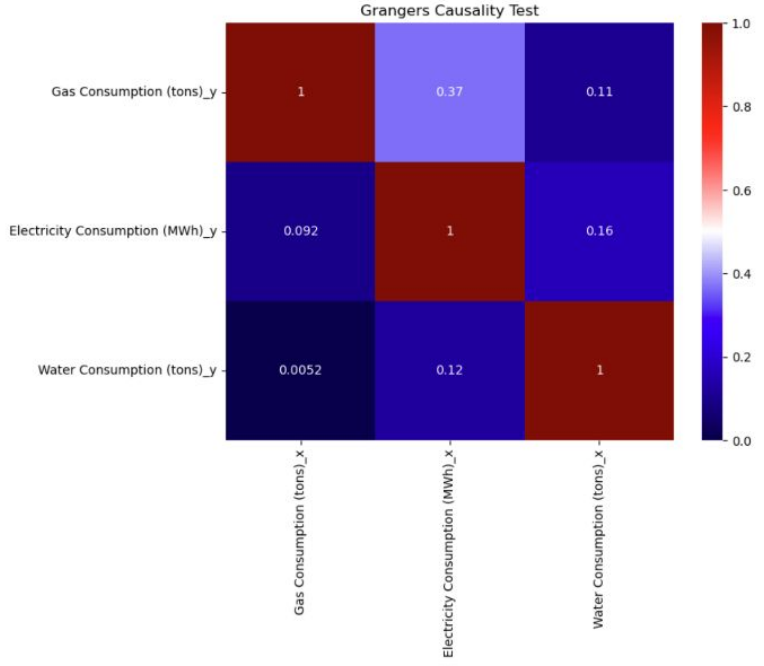
Grangers Causality Test determines whether one time series is useful in forecasting another. Focuses on a rather short term causality and we can consider a Multivariate Model for time series

- H_0: X does not cause the time series in Y.
- H_1: X cause the time series in Y.

```
#Function for Grangers Causality Test
def grangers_causality_matrix(data, variables, maxlag=13, test = 'ssr_chi2test', verbose=False):

    dataset = pd.DataFrame(np.zeros((len(variables), len(variables))), columns=variables, index=variables)

    for c in dataset.columns:
        for r in dataset.index:
            test_result = grangercausalitytests(data[[r,c]], maxlag=maxlag, verbose=False)
            p_values = [round(test_result[i+1][0][test][1],4) for i in range(maxlag)]
            if verbose:
                print(f'Y = {r}, X = {c}, P Values = {p_values}')

            min_p_value = np.min(p_values)
            dataset.loc[r,c] = min_p_value

    dataset.columns = [var + '_x' for var in variables]
    dataset.index = [var + '_y' for var in variables]
    return dataset
```

```
granger_df = grangers_causality_matrix(energyConsumption_ds, variables = energyConsumption_ds.columns)

#Print out significant pairs that show causality
pairs = granger_df.unstack().sort_values(kind="quicksort")
mask = pairs < 0.05
print("Significant Pairs:")
print(pairs[mask])

plt.figure(figsize=(8,6))
sns.heatmap(granger_df, annot=True, cmap='seismic', vmin=0)
plt.title("Grangers Causality Test")
plt.show()
```

```
Significant Pairs:
Gas Consumption (tons)_x  Water Consumption (tons)_y    0.0052
dtype: float64
```



Observations:

- If p-value is less than 0.05, we would reject null hypothesis and it means X cause the time series in Y.
- Gas Consumption causes the time series in Water Consumption.

Step 1: EDA (Test for Cointegration)

Test for Cointegration (Engle Granger Cointegration Test)

Cointegration refers to long-term equilibrium relationship between non-stationary variables, means they move together in the long run despite having individual trends.

- H_0: There is no cointegration.
- H_1: There is a cointegrating relationship.

Limitation: Assumes that the cointegrating relationship is linear and there is only one linear relationship among the variables is being tested.

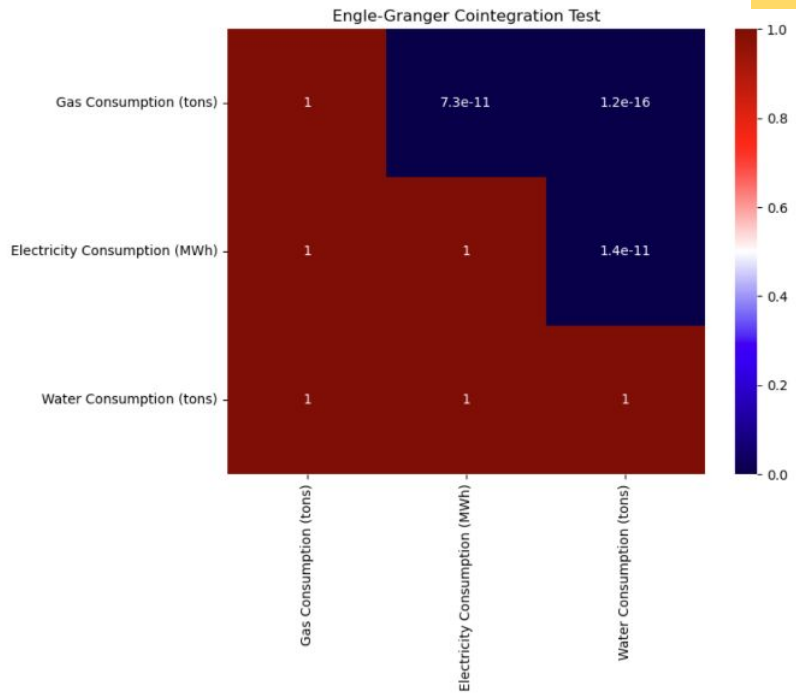
Step 1

```
#Function for Engle-Granger two-step cointegration test
def EngleGranger_coint_test(dataframe, critical_level = 0.05):
    n = dataframe.shape[1]
    pvalue_matrix = np.ones((n, n))
    keys = dataframe.columns
    pairs = []
    for i in range(n):
        for j in range(i+1, n):
            series1 = dataframe[keys[i]]
            series2 = dataframe[keys[j]]
            result = coint(series1, series2)
            pvalue = result[1]
            pvalue_matrix[i, j] = pvalue
            if pvalue < critical_level:
                pairs.append((keys[i], keys[j], pvalue))
    return pvalue_matrix, pairs
```

```
#Create dataframe to store the matrix
pvalue_matrix, pairs = EngleGranger_coint_test(energyConsumption_ds)
coint_pvalue_matrix_df = pd.DataFrame(pvalue_matrix)

for pair in pairs:
    print(pair)

plt.figure(figsize=(8,6))
sns.heatmap(coint_pvalue_matrix_df, xticklabels=energyConsumption_ds.columns,
                yticklabels=energyConsumption_ds.columns, annot=True, cmap='seismic')
plt.title('Engle-Granger Cointegration Test')
plt.show()
```

```
('Gas Consumption (tons)', 'Electricity Consumption (MWh)', 7.310878998998921e-11)
('Gas Consumption (tons)', 'Water Consumption (tons)', 1.2408872926483798e-16)
('Electricity Consumption (MWh)', 'Water Consumption (tons)', 1.356676425276096e-11)
```

Step 2



Observations:

- It looks like in the long run, every feature has a cointegrating relationship with all the other features as they have p-values of less than 0.05 which means null hypothesis is rejected.

Test for Cointegration (Johansen's Cointegration Test)

An extension of Engle-Granger two-step cointegration test and can handle cases when there are more than two stationary variables involved.

- H_0: There is no cointegration relationship among the variables.
- H_1: There are one or more cointegrating relationships among the variables.

Step 3

```
coint_johansen = coint_johansen(energyConsumption_ds, 1, 1)

critical_vals = {"0.90": 0, "0.95": 1, "0.99": 2}
#Trace statistic
trace_stat = coint_johansen.lr1
cvt = coint_johansen.cvt[:, critical_vals[str(1 - 0.05)]]

# Summary
print("Column  : Test Statistic > CI(95%)    =>   Significant")
for col, trace, ci in zip(energyConsumption_ds.columns, trace_stat, cvt):
    print(col, ":", round(trace, 2), ">", ci, " => ", trace > ci)
```

```
Column  : Test Statistic > CI(95%)    =>   Significant
Gas Consumption (tons) : 813.96 > 35.0116  =>  True
Electricity Consumption (MWh) : 510.52 > 18.3985  =>  True
Water Consumption (tons) : 230.69 > 3.8415  =>  True
```

Observations:

- Gas Consumption, Electricity Consumption, Water Consumption have a significant cointegration which suggests that there is a correlation between the 3 values in the long run.

Step 2: Data Cleaning/Feature Engineering

Setting Index

```
#Convert date to date type by formatting in the appropriate Date format
energyConsumption_df["DATE"] = pd.to_datetime(energyConsumption_df["DATE"], format='%d/%m/%Y')
energyConsumption_df.set_index('DATE', inplace=True)
display(energyConsumption_df.head())
```

DATE	Gas Consumption (tons)	Electricity Consumption (MWh)	Water Consumption (tons)
1990-01-01	18.0	725.1	548.8
1990-02-01	15.8	706.7	640.7
1990-03-01	17.3	624.5	511.1
1990-04-01	18.9	574.7	515.3
1990-05-01	22.0	553.2	488.4

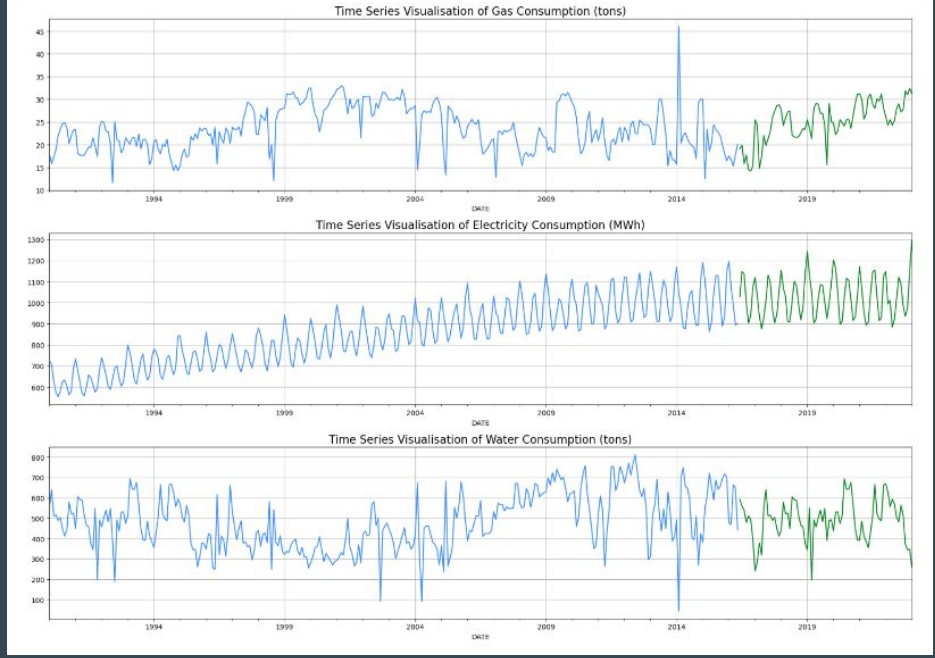
Splitting Train & Test Dataset

```
#Reason why this date was chosen to split: 80% data before this date & 20% data during and after this date
split_date = '2016-06-01'
train_set = energyConsumption_df[energyConsumption_df.index < split_date]
test_set = energyConsumption_df[energyConsumption_df.index >= split_date]
print(f"train.shape = {train_set.shape}, test.shape = {test_set.shape}")
print(type(train_set.index))
```

```
train.shape = (317, 3), test.shape = (80, 3)
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Tried out transformations like **Box-Cox & Log Transform** but it proved to have **little to no effect** so they were **removed**

```
#Visualise train and test time series in a time series plot
fig = plt.figure(figsize=(18,13), tight_layout=True)
for i, column in enumerate(energyConsumption_ds.columns):
    ax = fig.add_subplot(len(energyConsumption_ds.columns), 1, i+1)
    train_set[column].plot(ax=ax, color="dodgerblue")
    test_set[column].plot(ax=ax, color="green")
    ax.set_title(f"Time Series Visualisation of {column}", fontsize=16)
    ax.grid(True)
plt.show()
```



Step 3: Model Selection

3) Model Selection

The following Time-Series Models are used:

1. Auto ARIMA (part of ARIMA)

- Automated version of the ARIMA
- Automatically searches for the best combination of ARIMA hyperparameters (p,d,q) using algorithms like grid search or stepwise search
- Evaluate different parameter combinations using BIC to choose the best fitting model
- Quickly identify suitable ARIMA models for different time series without manual parameter tuning

2. ARIMA (Auto Regressive Integrated Moving Average Model)

- Time series forecasting model that combines autoregressive (AR) and moving average (MA) components
- (AR) component: Models the dependency of the current value on its past values (lags).
- (MA) component: Models the dependency of the current value on past forecast errors (residuals).
- Uses (p,d,q) where p is order of AR component, d is the degree of differencing, q is the order of the MA component

3. SARIMA (Seasonal Auto Regressive Integrated Moving Average Model)

- Extends the basic ARIMA model to include seasonality patterns
- Introduces seasonal autoregressive (SAR) and seasonal moving average (SMA) components to capture the periodic patterns in the data
- Uses (p,d,q),(P,D,Q,s) where the lowercase letters correspond to the non-seasonal components, uppercase letters correspond to the seasonal components and s represents the seasonality period

4. VARMA (Vector Auto Regressive Moving Average Model)

- Multivariate extension of the ARIMA model designed to handle multiple time series variables that can influence each other
- Each variable is regressed on its own past values as well as the past values of other variables (AR)
- Model uses lagged forecast errors (residuals) of the same variable and other variables to predict future values (MA)
- Uses (p,q) where p is the order of the autoregressive component and q is the order of the moving average component
- Useful when several related time series variables interact with one another

Step 4: Model Evaluation

- More Info is on Jupyter Report.
- **Slides** will only show Model Evaluation for the orders that were deemed based on **score metrics** in **Model Improvement**.
- **Jupyter Report** shows how the models evaluated the orders **interpreted by the acf/pacf plot**

4) Model Evaluation

A few methods I used to evaluate how each model would handle each column:

- Walk-Forward Validation
  - A special technique of Cross Validation.
  - Process involving dividing time series data into training & testing set and updating the model at each step, incorporating the latest data and making predictions for the next time step.
  - Assess the model ability to adapt and changing patterns in time series.

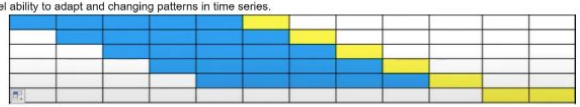


Image Source: Miro Medium

  - Use graphical visualization to see how accurate predictions are to actual data

- Summaries of Models
  - $P > |z|$ column: shows whether each coefficient is significant or not
    - H_0: Each coefficient is not statistically significant.
    - H_1: Each coefficient is statistically significant.
  - Ljung Box Test: checks if an autocorrelation exists in a time series
    - H_0: The residuals are independently distributed. (white noise)
    - H_1: The residuals are not independently distributed. (not white distributed)
  - Heteroscedasticity Test: checks if error residuals are homoscedastic or have the same variance
    - H_0: The residuals show variance.
    - H_1: The residuals do not show variance.
  - Jarque-Bera Test: checks for normality of errors
    - H_0: The data is normally distributed against an alternative of another distribution.
    - H_1: The data is not normally distributed against an alternative of another distribution.
  - Log-Likelihood: identifies a distribution that fits best with the sampled data
  - AIC: determines the strength of the linear regression model. The AIC penalizes a model for adding parameters since adding more parameters will always increase the maximum likelihood value.
  - BIC: punishes a model for complexity, but it also incorporates the number of rows in the data. (More Info Below)
  - HQIC: another criterion for model selection, not commonly used in practice
- Diagnostics Plot
  - Made up of standardized residuals plot, histogram with kde plot, Normal Q-Q plot and Correlogram
- Metrics:
  - **MAPE (Mean Absolute Percentage Error)**
    - Measures the percentage difference between the predicted values and the actual values in the time series

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \times 100$$

  - **RMSE (Root Mean Square Error)**
    - Measures the average magnitude of the errors between forecasted values and actual values in the time series

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

  - **SMAPE (Symmetric Mean Absolute Percentage Error)**
    - Measures the absolute percentage error for each individual forecasted value and take the mean of these absolute percentage errors across all time steps in the test set
    - Does not distinguish between over-predictions and under-predictions and give equal weight to both

$$SMAPE = \frac{100}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{(|y_i| + |\hat{y}_i|)/2}$$

Step 4: Model Evaluation (Auto ARIMA)

Auto ARIMA

I will be using this metrics to see which is the best order of parameters. I will just check the summary to see how well the ARIMA model do with the order Auto ARIMA deem the best.

BIC (Bayesian Information Criterian)

- Entails a calculation of maximum log-likelihood and a higher penalty term for a higher number of parameters

$$BIC_i = -2\log L_i + p_i \log n$$

- L: Max likelihood - parameter with the highest probability of correctly representing the relationship between input & output
- p: Number of parameters
- n: Number of values in the dataset
- The lower the value is, the better the model is

Gas Consumption (Auto ARIMA)

```
#Stepwise set to True to see the BIC for every parameter searched
arima_modelGas=auto_arima(train_set["Gas Consumption (tons)"], start_p=1, start_q=1,
                          test='adf', max_p=5, max_q=5, m=1, information_criterion='bic',
                          seasonal=False, start_P=0, D=None, trace=True,
                          error_action="ignore", suppress_warnings=True, stepwise=True)
print(arima_modelGas)
```

```
Performing stepwise search to minimize bic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : BIC=1726.744, Time=0.18 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : BIC=1794.151, Time=0.05 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : BIC=1763.273, Time=0.06 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : BIC=1745.416, Time=0.07 sec
 ARIMA(0,1,0)(0,0,0)[0]             : BIC=1788.396, Time=0.03 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : BIC=1730.887, Time=0.43 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : BIC=1731.259, Time=0.21 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : BIC=1736.710, Time=0.13 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : BIC=1762.028, Time=0.10 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : BIC=1735.755, Time=0.24 sec
 ARIMA(1,1,1)(0,0,0)[0]             : BIC=1720.989, Time=0.07 sec
 ARIMA(0,1,1)(0,0,0)[0]             : BIC=1739.661, Time=0.04 sec
 ARIMA(1,1,0)(0,0,0)[0]             : BIC=1757.519, Time=0.04 sec
 ARIMA(2,1,1)(0,0,0)[0]             : BIC=1725.131, Time=0.22 sec
 ARIMA(1,1,2)(0,0,0)[0]             : BIC=1725.504, Time=0.09 sec
 ARIMA(0,1,2)(0,0,0)[0]             : BIC=1730.958, Time=0.05 sec
 ARIMA(2,1,0)(0,0,0)[0]             : BIC=1756.274, Time=0.04 sec
 ARIMA(2,1,2)(0,0,0)[0]             : BIC=1729.999, Time=0.12 sec

Best model:  ARIMA(1,1,1)(0,0,0)[0]
Total fit time: 2.192 seconds
 ARIMA(1,1,1)(0,0,0)[0]
```

Water Consumption (Auto ARIMA)

```
#Stepwise set to True to see the BIC for every parameter searched
arima_modelWater=auto_arima(train_set["Water Consumption (tons)"], start_p=1, start_q=1,
                            test='adf', max_p=5, max_q=5, m=1, d=1, information_criterion='bic',
                            seasonal=False, start_P=0, D=None, trace=True,
                            error_action="ignore", suppress_warnings=True, stepwise=True)
print(arima_modelWater)
```

```
Performing stepwise search to minimize bic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : BIC=3854.921, Time=0.17 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : BIC=3930.713, Time=0.03 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : BIC=3899.250, Time=0.06 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : BIC=3869.609, Time=0.11 sec
 ARIMA(0,1,0)(0,0,0)[0]             : BIC=3924.959, Time=0.03 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : BIC=3859.638, Time=0.22 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : BIC=3859.513, Time=0.21 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : BIC=3861.051, Time=0.16 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : BIC=3890.107, Time=0.06 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : BIC=3865.002, Time=0.39 sec
 ARIMA(1,1,1)(0,0,0)[0]             : BIC=3849.228, Time=0.08 sec
 ARIMA(0,1,1)(0,0,0)[0]             : BIC=3863.853, Time=0.05 sec
 ARIMA(1,1,0)(0,0,0)[0]             : BIC=3893.496, Time=0.05 sec
 ARIMA(2,1,1)(0,0,0)[0]             : BIC=3853.953, Time=0.12 sec
 ARIMA(1,1,2)(0,0,0)[0]             : BIC=3853.831, Time=0.13 sec
 ARIMA(0,1,2)(0,0,0)[0]             : BIC=3855.314, Time=0.07 sec
 ARIMA(2,1,0)(0,0,0)[0]             : BIC=3884.352, Time=0.04 sec
 ARIMA(2,1,2)(0,0,0)[0]             : BIC=3859.325, Time=0.22 sec

Best model:  ARIMA(1,1,1)(0,0,0)[0]
Total fit time: 2.191 seconds
 ARIMA(1,1,1)(0,0,0)[0]
```

Electricity Consumption (Auto ARIMA)

```
#Stepwise set to True to see the BIC for every parameter searched
#Does not account for seasonality
arima_modelElectricity=auto_arima(train_set["Electricity Consumption (MWh)"], start_p=1, start_q=1,
                          test='adf', max_p=5, max_q=5, information_criterion='bic',
                          seasonal=False, start_P=0, D=None, trace=True,
                          error_action="ignore", suppress_warnings=True, stepwise=True)
print(arima_modelElectricity)
```

```
Performing stepwise search to minimize bic
 ARIMA(1,0,1)(0,0,0)[0]             : BIC=3533.438, Time=0.08 sec
 ARIMA(0,0,0)(0,0,0)[0]             : BIC=5193.516, Time=0.03 sec
 ARIMA(1,0,0)(0,0,0)[0]             : BIC=inf, Time=0.03 sec
 ARIMA(0,0,1)(0,0,0)[0]             : BIC=inf, Time=0.08 sec
 ARIMA(2,0,1)(0,0,0)[0]             : BIC=3538.824, Time=0.16 sec
 ARIMA(1,0,2)(0,0,0)[0]             : BIC=3470.074, Time=0.16 sec
 ARIMA(0,0,2)(0,0,0)[0]             : BIC=inf, Time=0.17 sec
 ARIMA(2,0,2)(0,0,0)[0]             : BIC=3447.339, Time=0.29 sec
 ARIMA(3,0,2)(0,0,0)[0]             : BIC=inf, Time=0.39 sec
 ARIMA(2,0,3)(0,0,0)[0]             : BIC=3421.661, Time=0.29 sec
 ARIMA(1,0,3)(0,0,0)[0]             : BIC=3417.635, Time=0.25 sec
 ARIMA(0,0,3)(0,0,0)[0]             : BIC=inf, Time=0.38 sec
 ARIMA(1,0,4)(0,0,0)[0]             : BIC=3404.362, Time=0.34 sec
 ARIMA(0,0,4)(0,0,0)[0]             : BIC=inf, Time=0.50 sec
 ARIMA(2,0,4)(0,0,0)[0]             : BIC=3382.828, Time=0.55 sec
 ARIMA(3,0,4)(0,0,0)[0]             : BIC=3426.799, Time=0.67 sec
 ARIMA(2,0,5)(0,0,0)[0]             : BIC=3342.279, Time=0.51 sec
 ARIMA(1,0,5)(0,0,0)[0]             : BIC=3350.089, Time=0.46 sec
 ARIMA(3,0,5)(0,0,0)[0]             : BIC=inf, Time=0.79 sec
 ARIMA(2,0,5)(0,0,0)[0] intercept   : BIC=3456.562, Time=0.71 sec

Best model:  ARIMA(2,0,5)(0,0,0)[0]
Total fit time: 6.840 seconds
 ARIMA(2,0,5)(0,0,0)[0]
```

```
#Stepwise set to True to see the BIC for every parameter searched
#Account for seasonality
arima_modelElectricity=auto_arima(train_set["Electricity Consumption (MWh)"], start_p=1, start_q=1,
                          test='adf', max_p=5, max_q=5, m=12, d=1, max_P=3,max_Q=3, information_criterion='bic',
                          seasonal=True, start_P=0, D=None, trace=True,
                          error_action="ignore", suppress_warnings=True, stepwise=True)
print(arima_modelElectricity)
```

```
Performing stepwise search to minimize bic
 ARIMA(1,1,1)(0,1,1)[12]            : BIC=2772.885, Time=0.75 sec
 ARIMA(0,1,0)(0,1,0)[12]            : BIC=2938.015, Time=0.03 sec
 ARIMA(1,1,0)(1,1,0)[12]            : BIC=2883.919, Time=0.13 sec
 ARIMA(0,1,1)(0,1,1)[12]            : BIC=2809.733, Time=0.31 sec
 ARIMA(1,1,1)(0,1,0)[12]            : BIC=inf, Time=0.20 sec
 ARIMA(1,1,1)(1,1,1)[12]            : BIC=2778.554, Time=0.96 sec
 ARIMA(1,1,1)(0,1,2)[12]            : BIC=2778.528, Time=1.64 sec
 ARIMA(1,1,1)(1,1,0)[12]            : BIC=2827.936, Time=0.46 sec
 ARIMA(1,1,1)(1,1,2)[12]            : BIC=2782.610, Time=2.83 sec
 ARIMA(1,1,0)(0,1,1)[12]            : BIC=2826.664, Time=0.28 sec
 ARIMA(2,1,1)(0,1,1)[12]            : BIC=2776.543, Time=0.68 sec
 ARIMA(1,1,2)(0,1,1)[12]            : BIC=2775.498, Time=0.75 sec
 ARIMA(0,1,0)(0,1,1)[12]            : BIC=2831.958, Time=0.17 sec
 ARIMA(0,1,2)(0,1,1)[12]            : BIC=2773.425, Time=0.50 sec
 ARIMA(2,1,0)(0,1,1)[12]            : BIC=2805.775, Time=0.38 sec
 ARIMA(2,1,2)(0,1,1)[12]            : BIC=2780.005, Time=1.05 sec
 ARIMA(1,1,1)(0,1,1)[12] intercept  : BIC=inf, Time=0.68 sec

Best model:  ARIMA(1,1,1)(0,1,1)[12]
Total fit time: 11.845 seconds
 ARIMA(1,1,1)(0,1,1)[12]
```

Step 4: Model Evaluation (Walk Forward Validation)

- First, I tried out the orders interpreted from the **acf & pacf plot**
- Next, I tried out the orders that were deemed best based on **score metrics**

For pmdARIMA models

Walk-Forward Validation + Score Metrics

- Trained on historical data up to a certain point and then used to make one-step predictions using test data, repeat iteratively
- Model is retrained with the updated historical data and makes predictions for each subsequent time step in the test set

```
def walk_forward_validate(column, train_set, test_set, model, original_data):
    #store predictions
    predictions=[]
    data=train_set[column].values
    model_selected=model.fit(data)

    #making in-sample predictions of training data
    pred_in_sample = model_selected.predict_in_sample()
    trainPredicted = pd.Series(pred_in_sample, index=train_set[column].index)

    for t in test_set[column].values:
        #predicting the next time step
        y=model_selected.predict(start=len(data), end=len(data))

        #appends one step ahead prediction
        predictions.append(y[0])

        #adds current value from test set to data array which include actual value from the history
        #to predict the next iteration prediction
        data = np.append(data, t)

        #update ARIMA model with new observation, ensure model uses updated history for next prediction
        model_selected.update(t)
        model_selected=model.fit(data)

        #calculate the aic & bic scores based on the order
        aic=model_selected.aic()
        bic=model_selected.bic()

    testPredicted = pd.Series(predictions, index=test_set[column].index)
    pred_full = np.concatenate((pred_in_sample, predictions))
    pred_full_Energy = pd.Series(pred_full, index=original_data[column].index)

    #calculate the MAPE & RMSE scores for train set and validaion set based on the order
    train_MAPE = round(mean_absolute_percentage_error(train_set[column], trainPredicted)*100,3)
    train_RMSE = round(mean_squared_error(train_set[column], trainPredicted, squared=False),3)
    validate_MAPE = round(mean_absolute_percentage_error(test_set[column], testPredicted)*100, 3)
    validate_RMSE = round(mean_squared_error(test_set[column], testPredicted, squared=False),3)

    return(
        trainPredicted, testPredicted, pred_full_Energy, model_selected, aic, bic,
        train_MAPE, train_RMSE, validate_MAPE, validate_RMSE)
```

For statsmodels.SARIMAX

Electricity Consumption (Walk Forward Validation Using statsmodel SARIMA)

Now I will try to run the same model but from a different library just to see whether a different result was obtained.

```
#New walk forward cross validation function since a different library was used which means it has different methods
def walk_forward_validate_sarimax(column, train_set, test_set, model, order, seasonal_order):
    #store predictions
    trainPredictions=[]
    testPredictions=[]
    fullPredictions=[]
    data=train_set[column].values

    model_fit = model
    trainPredictions = model_fit.get_prediction(start='1990-01-01', end='2016-05-01').predicted_mean
    for t in test_set[column].values:
        model = SARIMAX(data, order=order, seasonal_order=seasonal_order)
        model_fit = model.fit()

        #predicting the next time step
        y_test = model_fit.forecast(steps=1)
        #appends one step ahead prediction
        testPredictions.append(y_test[0])

        #adds current value from test set to data array which include actual value from the history
        #to predict the next iteration prediction
        data = np.append(data, t)

    #Concatenate full predictions
    fullPredictions=np.concatenate((trainPredictions, testPredictions))

    #calculate the aic & bic scores based on the order
    aic=model_fit.aic
    bic=model_fit.bic

    #convert everything to dataframe
    trainPredicted = pd.Series(data=trainPredictions, index=train_set.index)
    testPredicted = pd.Series(data=testPredictions, index=test_set.index)
    pred_full_energy = pd.Series(data=fullPredictions, index=energyConsumption_df[column].index)

    #calculate the MAPE & RMSE scores for train set and validaion set based on the order
    train_MAPE = round(mean_absolute_percentage_error(train_set[column], trainPredicted)*100,3)
    train_RMSE = round(mean_squared_error(train_set[column], trainPredicted, squared=False),3)
    validate_MAPE = round(mean_absolute_percentage_error(test_set[column], testPredicted)*100, 3)
    validate_RMSE = round(mean_squared_error(test_set[column], testPredicted, squared=False),3)

    return(
        trainPredicted, testPredicted, pred_full_energy, aic, bic,
        train_MAPE, train_RMSE, validate_MAPE, validate_RMSE)
```

For statsmodels.VARMAX

```
# Walk forward cross validation for VARMA
def walk_forward_validate_varma(columns, train_set, test_set, model, order):
    #store predictions
    trainPredictions=[]
    testPredictions=[]
    fullPredictions=[]
    data=train_set[columns].values

    model_fit = model
    trainPredictions = model_fit.get_prediction(start='1990-01-01', end='2016-05-01').predicted_mean
    for t in test_set[columns].values:
        model = VARMAX(endog=data, order=order, enforce_stationary=False, enforce_invertibility=False)
        model_fit = model.fit(disp=False)

        #predicting the next time step
        y_test = model_fit.forecast(steps=1)
        #appends one step ahead prediction
        testPredictions.append(y_test[0])

        #adds current value from test set to data array which include actual value from the history
        #to predict the next iteration prediction
        data = np.append(data, [t], axis=0)

    #Concatenate full predictions
    fullPredictions=np.concatenate((trainPredictions, testPredictions))

    #calculate the aic & bic scores based on the order
    aic=model_fit.aic
    bic=model_fit.bic

    #convert everything to dataframe
    trainPredicted = pd.DataFrame(data=trainPredictions, columns=columns, index=train_set.index)
    testPredicted = pd.DataFrame(data=testPredictions, columns=columns, index=test_set.index)
    pred_full_energy = pd.DataFrame(data=fullPredictions, columns=columns, index=energyConsumption_df.index)

    #calculate the MAPE & RMSE scores for train set and validaion set based on the order
    train_MAPE = round(mean_absolute_percentage_error(train_set[columns], trainPredicted)*100,3)
    train_RMSE = round(mean_squared_error(train_set[columns], trainPredicted, squared=False),3)
    validate_MAPE = round(mean_absolute_percentage_error(test_set[columns], testPredicted)*100, 3)
    validate_RMSE = round(mean_squared_error(test_set[columns], testPredicted, squared=False),3)

    return(
        trainPredicted, testPredicted, pred_full_energy, model, aic, bic,
        train_MAPE, train_RMSE, validate_MAPE, validate_RMSE)
```

Step 5: Model Improvement (ARIMA)

Gas Consumption

```
scoreMetrics_values={}

#Values of p & q
p_values = np.arange(0, 3)
q_values = np.arange(0, 8)

for i in list(product(p_values,q_values)):
    order=(i[0], 1, i[1])
    model = ARIMA(order=order)
    (gas_trainPredicted, gas, pred_full_Gas, arima_modelGas, gas_aic, gas_bic,
     gas_train_MAPE, gas_train_RMSE, gas_validate_MAPE, gas_validate_RMSE) = walk_forward_validate(
        "Gas Consumption (tons)", train_set, test_set, model, energyConsumption_df
    )

    #concatenate order as a string
    order_string=f'({i[0]},1,{i[1]})'
    #store the score metrics in the dictionaries
    scoreMetrics_values[order_string] = {'AIC': round(gas_aic,3), 'BIC': round(gas_bic,3), 'Train MAPE': gas_train_MAPE,
                                         'Train RMSE': gas_train_RMSE, 'Validate MAPE': gas_validate_MAPE,
                                         'Validate RMSE': gas_validate_RMSE}

#Convert the dictionary to dataframe
scoreMetrics_Gas_df = pd.DataFrame.from_dict(scoreMetrics_values, orient='index').sort_values(
    by=["BIC","Validate MAPE", "Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
display(scoreMetrics_Gas_df)
```

Gas

	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(1,1,1)	2116.039	2131.965	11.345	3.716	9.344	2.961
(2,1,1)	2116.763	2136.671	11.344	3.708	9.426	2.969
(1,1,2)	2116.935	2136.843	11.329	3.710	9.319	2.968
(0,1,3)	2118.446	2138.353	11.598	3.713	9.605	2.991
(2,1,2)	2118.041	2141.930	11.402	3.703	9.395	2.978

Water Consumption

```
scoreMetrics_values={}

#Values of p & q
p_values = np.arange(0, 4)
q_values = np.arange(0, 5)

for i in list(product(p_values,q_values)):
    order=(i[0], 1, i[1])
    model = ARIMA(order=order)
    (water_trainPredicted, water, pred_full_Water, arima_modelWater, water_aic, water_bic,
     water_train_MAPE, water_train_RMSE, water_validate_MAPE, water_validate_RMSE) = walk_forward_validate(
        "Water Consumption (tons)", train_set, test_set, model, energyConsumption_df
    )

    #concatenate order as a string
    order_string=f'({i[0]},1,{i[1]})'
    #store the score metrics in the dictionaries
    scoreMetrics_values[order_string] = {'AIC': round(water_aic,3), 'BIC': round(water_bic,3),
                                         'Train MAPE': water_train_MAPE, 'Train RMSE': water_train_RMSE,
                                         'Validate MAPE': water_validate_MAPE, 'Validate RMSE': water_validate_RMSE}

#Convert the dictionary to dataframe
scoreMetrics_Water_df = pd.DataFrame.from_dict(scoreMetrics_values, orient='index').sort_values(
    by=["BIC","Validate MAPE", "Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
display(scoreMetrics_Water_df)
```

Water

	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(1,1,1)	4790.987	4806.913	22.076	108.206	17.147	91.563
(1,1,2)	4789.982	4809.889	22.176	108.025	16.839	90.383
(2,1,1)	4790.225	4810.132	22.157	108.043	16.850	90.430
(2,1,2)	4791.924	4815.813	22.174	107.988	16.890	90.617
(1,1,3)	4791.933	4815.822	22.185	108.001	16.879	90.562

```
lowestGasBIC_order = scoreMetrics_Gas_df["BIC"].idxmin()
lowestGasBIC_value = scoreMetrics_Gas_df["BIC"].min()
print(f"Order with Lowest BIC value: {lowestGasBIC_order}")
print(f"Lowest BIC value: {lowestGasBIC_value}")

lowestGasBIC_cols = scoreMetrics_Gas_df.loc[lowestGasBIC_order]
display(lowestGasBIC_cols)
```

(1,1,1)

```
Order with Lowest BIC value: (1,1,1)
Lowest BIC value: 2131.965

AIC              2116.039
BIC              2131.965
Train MAPE         11.345
Train RMSE          3.716
Validate MAPE       9.344
Validate RMSE       2.961
Name: (1,1,1), dtype: float64
```

Observations:

- ARIMA order of (1,1,1) has the lowest BIC value so it is the best order to be used for Gas Consumption.
- Validate MAPE is 2% lower than the train MAPE so it means there is a lower different in errors between the test and predicted data than the train and predicted data.
- Validate RMSE is also lower than the train RMSE.

```
lowestWaterBIC_order = scoreMetrics_Water_df["BIC"].idxmin()
lowestWaterBIC_value = scoreMetrics_Water_df["BIC"].min()
print(f"Order with Lowest BIC value: {lowestWaterBIC_order}")
print(f"Lowest BIC value: {lowestWaterBIC_value}")

lowestWaterBIC_cols = scoreMetrics_Water_df.loc[lowestWaterBIC_order]
display(lowestWaterBIC_cols)
```

(1,1,1)

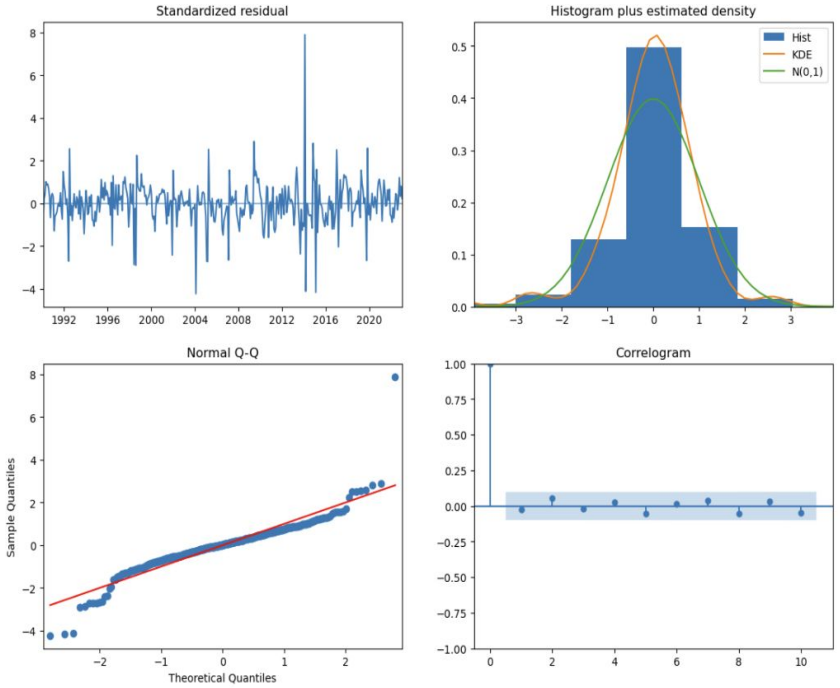
```
Order with Lowest BIC value: (1,1,1)
Lowest BIC value: 4806.913

AIC              4790.987
BIC              4806.913
Train MAPE         22.076
Train RMSE        108.206
Validate MAPE      17.147
Validate RMSE      91.563
Name: (1,1,1), dtype: float64
```

Observations:

- ARIMA order of (1,1,1) has the lowest BIC value so it is the best order to be used for Water Consumption.
- Validate MAPE is 4% lower than the train MAPE so it means there is a lower different in errors between the test and predicted data than the train and predicted data.
- Validate RMSE is also lower than the train RMSE.

Step 5: Model Improvement (ARIMA Gas)

```
Gas_ARIMA_model = ARIMA(order=(1,1,1)).fit(energyConsumption_df["Gas Consumption (tons)"])
Gas_ARIMA_model.plot_diagnostics(figsize=(14,10))
plt.show()
```



Observations:

- There is no obvious pattern in the residuals in the standardized residual plot, suggests that this is a good model.
- The histogram is slightly symmetrical and the QQ plot shows all points lying on the line which means that the data follows a normal distribution.
- The correlogram shows that 95% of the correlations for lag do lie within the threshold, which shows that this is a good model.

```
Gas_ARIMA_model.summary()
```

6]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	397
Model:	SARIMAX(1, 1, 1)	**Log Likelihood**	-1054.019
Date:	Thu, 10 Aug 2023	**AIC**	2116.039
Time:	08:13:24	**BIC**	2131.965
Sample:	01-01-1990	**HQIC**	2122.348
	- 01-01-2023		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	0.0133	0.018	0.728	0.467	-0.023	0.049
ar.L1	0.4336	0.037	11.732	0.000	0.361	0.506
ma.L1	-0.9013	0.032	-28.510	0.000	-0.963	-0.839
sigma2	11.9797	0.429	27.933	0.000	11.139	12.820

Ljung-Box (L1) (Q):	0.22	**Jarque-Bera (JB):**	2255.67
Prob(Q):	0.64	**Prob(JB):**	0.00
Heteroskedasticity (H):	2.21	**Skew:**	0.57
Prob(H) (two-sided):	0.00	**Kurtosis:**	14.64

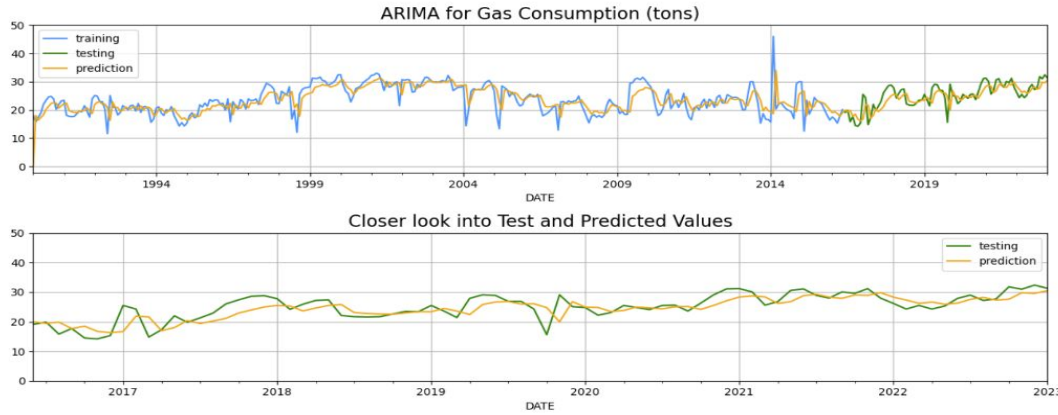
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Observations:

- All coefficients are significant as p-value are less than 0.5.
- Ljung Box test shows that the residuals are independently distributed.
- Heteroscedasticity test shows that the residuals do not show variance.
- Jarque-Bera Test shows the data is not normally distributed against an alternative of another distribution.

Step 5: Model Improvement (ARIMA Gas)

```
#Visualise train and test time series in a time series plot
fig, ax = plt.subplots(2,1,figsize=(13,6))

#Plot training, testing & predicted values on the graph
train_set["Gas Consumption (tons)"].plot(ax=ax[0], label="training", color='dodgerblue')
test_set["Gas Consumption (tons)"].plot(ax=ax[0], label="testing", color='green')
pred_full_Gas.plot(ax=ax[0], label='prediction', color="orange")
ax[0].legend()
ax[0].set_title(f"ARIMA for Gas Consumption (tons)", fontsize=16)
ax[0].set_yticks(np.arange(0,51,10))
ax[0].grid(True)

#Focus on the predicted values and the test values
test_set["Gas Consumption (tons)"].plot(ax=ax[1], label="testing", color='green')
gas.plot(ax=ax[1], label='prediction', color="orange")
ax[1].legend()
ax[1].set_title(f"Closer look into Test and Predicted Values", fontsize=16)
ax[1].set_yticks(np.arange(0,51,10))
ax[1].grid(True)
plt.tight_layout()
plt.show()
```

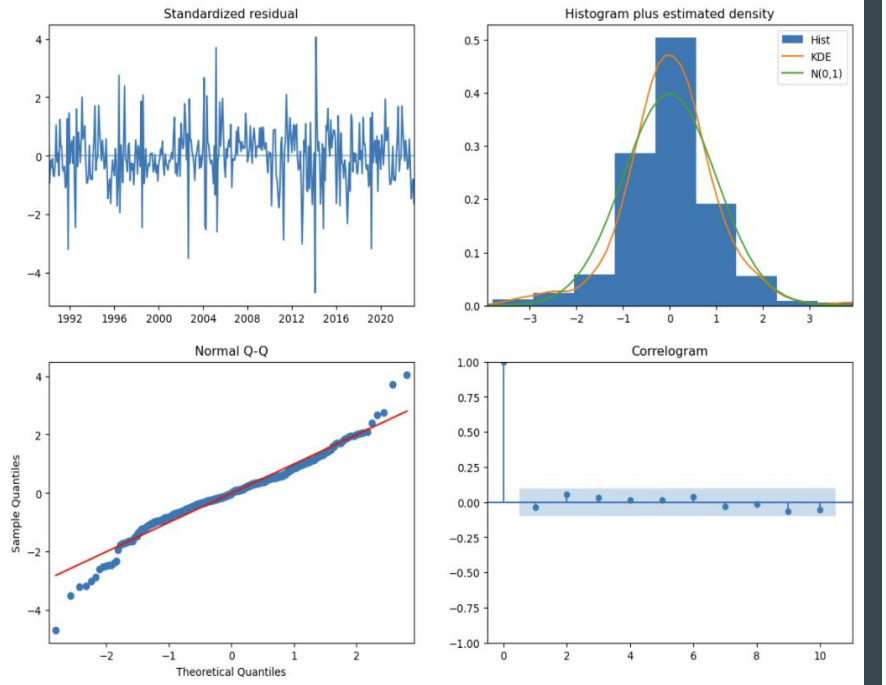


Observations:

- For Gas Consumption, the predicted data seems to match the training and testing data trend even though they are not exactly similar.

Step 5: Model Improvement (ARIMA Water)

```
#Diagnostic plot
Water_ARIMA_model = ARIMA(order=(1,1,1)).fit(energyConsumption_df["Water Consumption (tons)"])
Water_ARIMA_model.plot_diagnostics(figsize=(14,10))
plt.show()
```



Observations:

- There is no obvious pattern in the residuals in the standardized residual plot, suggests that this is a good model.
- The histogram is slightly symmetrical and the QQ plot shows all points lying on the line which means that the data follows a normal distribution.
- The correlogram shows that 95% of the correlations for lag do lie within the threshold, which shows that this is a good model.

```
Water_ARIMA_model.summary()
```

[2]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	397
Model:	SARIMAX(1, 1, 1)	**Log Likelihood**	-2391.494
Date:	Thu, 10 Aug 2023	**AIC**	4790.987
Time:	08:23:35	**BIC**	4806.913
Sample:	01-01-1990	**HQIC**	4797.296
	- 01-01-2023		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-0.0600	0.395	-0.152	0.879	-0.834	0.714
ar.L1	0.4397	0.045	9.737	0.000	0.351	0.528
ma.L1	-0.9294	0.024	-39.224	0.000	-0.976	-0.883
sigma2	1.028e+04	523.320	19.635	0.000	9249.795	1.13e+04

Ljung-Box (L1) (Q):	0.51	**Jarque-Bera (JB):**	128.13
Prob(Q):	0.48	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.50	**Skew:**	-0.30
Prob(H) (two-sided):	0.02	**Kurtosis:**	5.72

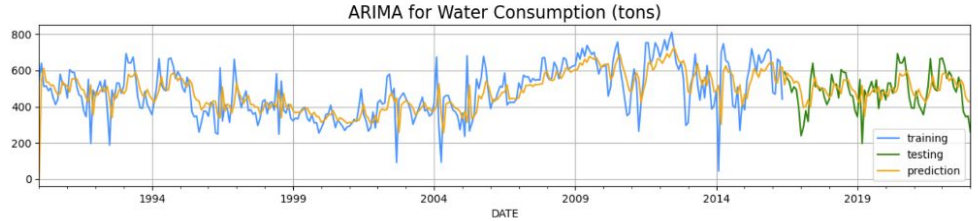
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Observations:

- All coefficients are significant as p-value are less than 0.5.
- Ljung Box test shows that the residuals are not independently distributed.
- Heteroscedasticity test shows that the residuals do not show variance.
- Jarque-Bera Test shows the data is not normally distributed against an alternative of another distribution.

Step 5: Model Improvement (ARIMA Water)

```
#Visualise train and test time series in a time series plot
fig, ax = plt.subplots(2,1,figsize=(13,6))

#Plot training, testing & predicted values on the graph
train_set["Water Consumption (tons)"].plot(ax=ax[0], label="training", color='dodgerblue')
test_set["Water Consumption (tons)"].plot(ax=ax[0], label="testing", color='green')
pred_full_Water.plot(ax=ax[0], label='prediction', color="orange")
ax[0].legend()
ax[0].set_title(f"ARIMA for Water Consumption (tons)", fontsize=16)
ax[0].grid(True)

test_set["Water Consumption (tons)"].plot(ax=ax[1], label="testing", color='green')
water.plot(ax=ax[1], label='prediction', color="orange")
ax[1].legend()
ax[1].set_title(f"Closer look into Test and Predicted Values", fontsize=16)
ax[1].grid(True)
plt.tight_layout()
plt.show()
```



Observations:

- For Water Consumption, the predicted data seems to match the training and testing data trend even though they are not exactly similar.

Step 5: Model Improvement (SARIMA Electricity)

Electricity Consumption pmdarima

```
scoreMetrics_values={}

#Values of p, q, P & Q
p_values = np.arange(0, 3)
q_values = np.arange(0, 4)
P_values = np.arange(0, 3)
Q_values = np.arange(0, 3)

for i in list(product(p_values, q_values, P_values, Q_values)):
    order = (i[0], 1, i[1])
    seasonal_order = (i[2], 1, i[3], 12)
    model = ARIMA(order=order, seasonal_order=seasonal_order)
    (electricity_trainPredicted, electricity, pred_full_Electricity, arima_modelElectricity, elec_aic, elec_bic,
    elec_train_MAPE, elec_train_RMSE, elec_validate_MAPE, elec_validate_RMSE) = walk_forward_validate(
        "Electricity Consumption (MWh)", train_set, test_set, model, energyConsumption_df
    )

    #concatenate order as a string
    order_string=f'({i[0]},1,{i[1]}), ({i[2]},1,{i[3]},12)'

    #store the score metrics in the dictionaries
    scoreMetrics_values[order_string] = {'AIC': round(elec_aic,3), 'BIC': round(elec_bic,3),
                                         'Train MAPE': elec_train_MAPE, 'Train RMSE': elec_train_RMSE,
                                         'Validate MAPE': elec_validate_MAPE, 'Validate RMSE': elec_validate_RMSE}

#Convert the dictionary to dataframe
scoreMetrics_Electricity_df = pd.DataFrame.from_dict(scoreMetrics_values, orient='index').sort_values(
    by=["BIC","Validate MAPE", "Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
display(scoreMetrics_Electricity_df.head(15))
```

	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(1,1,1), (2,1,1,12)	3540.218	3567.872	2.486	50.117	2.303	30.474
(1,1,1), (0,1,1,12)	3550.888	3570.641	2.503	50.214	2.394	31.265
(2,1,1), (2,1,1,12)	3540.350	3571.955	2.482	50.101	2.280	30.339
(1,1,2), (2,1,1,12)	3541.214	3572.819	2.485	50.092	2.290	30.391
(0,1,3), (2,1,1,12)	3543.050	3574.655	2.507	50.128	2.281	30.494

Best BIC

Best MAPE

```
lowestElectricityBIC_order = scoreMetrics_Electricity_df["BIC"].idxmin()
lowestElectricityBIC_value = scoreMetrics_Electricity_df["BIC"].min()
print(f"Order with Lowest BIC value: {lowestElectricityBIC_order}")
print(f"Lowest BIC value: {lowestElectricityBIC_value}")

lowestElectricityBIC_cols = scoreMetrics_Electricity_df.loc[lowestElectricityBIC_order]
display(lowestElectricityBIC_cols)
```

```
Order with Lowest BIC value: (1,1,1), (2,1,1,12)
Lowest BIC value: 3567.872

AIC              3540.218
BIC              3567.872
Train MAPE          2.486
Train RMSE         50.117
Validate MAPE       2.303
Validate RMSE      30.474
Name: (1,1,1), (2,1,1,12),
```

Best BIC

```
scoreMetrics_Electricity_df.sort_values(
    by=["Validate MAPE", "BIC" ,"Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
```

	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(2,1,1), (2,1,1,12)	3540.350	3571.955	2.482	50.101	2.280	30.339
(0,1,3), (2,1,1,12)	3543.050	3574.655	2.507	50.128	2.281	30.494
(2,1,2), (2,1,2,12)	3541.906	3581.413	2.473	50.075	2.281	30.264
(1,1,2), (2,1,1,12)	3541.214	3572.819	2.485	50.092	2.290	30.391
(2,1,2), (2,1,1,12)	3542.989	3578.545	2.476	50.074	2.294	30.325
...	...	...	...	...	...	...
(0,1,1), (0,1,0,12)	3742.753	3754.605	3.042	53.297	3.001	40.805
(1,1,0), (0,1,0,12)	3764.756	3776.608	3.124	53.772	3.016	41.643
(0,1,0), (0,1,0,12)	3780.555	3788.457	3.157	54.117	3.021	42.591
(2,1,0), (1,1,0,12)	3678.675	3698.429	2.827	51.804	3.033	38.875
(0,1,1), (1,1,0,12)	3693.682	3709.485	2.832	52.044	3.104	39.976

108 rows × 6 columns

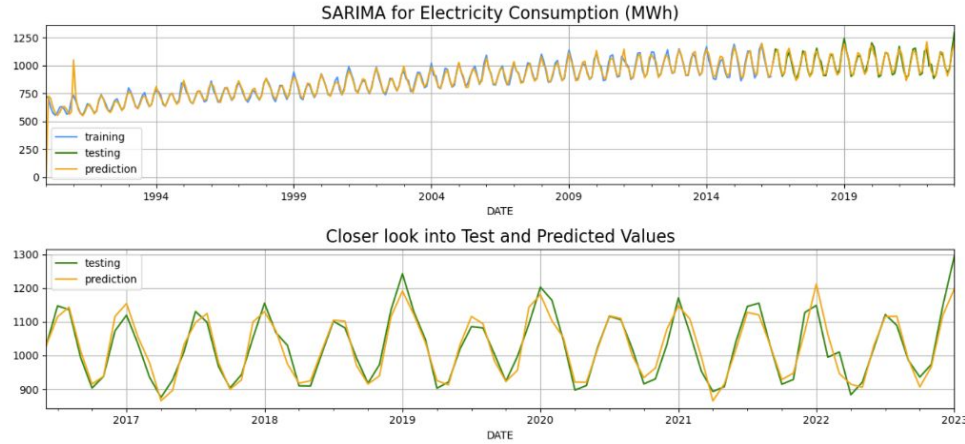
```
lowestElectricityMAPE_order = scoreMetrics_Electricity_df["Validate MAPE"].idxmin()
lowestElectricityMAPE_value = scoreMetrics_Electricity_df["Validate MAPE"].min()
print(f"Order with Lowest Validate MAPE value: {lowestElectricityMAPE_order}")
print(f"Lowest Validate MAPE value: {lowestElectricityMAPE_value}")

lowestElectricityMAPE_cols = scoreMetrics_Electricity_df.loc[lowestElectricityMAPE_order]
display(lowestElectricityMAPE_cols)
```

```
Order with Lowest Validate MAPE value: (2,1,1), (2,1,1,12)
Lowest Validate MAPE value: 2.28

AIC              3540.350
BIC              3571.955
Train MAPE          2.482
Train RMSE         50.101
Validate MAPE       2.280
Validate RMSE      30.339
Name: (2,1,1), (2,1,1,12), dtype: float64
```

Best MAPE

Step 5: Model Improvement (SARIMA Electricity)

```python
#Visualise train and test time series in a time series plot
fig, ax = plt.subplots(2,1,figsize=(13,6))

#Plot training, testing & predicted values on the graph
train_set["Electricity Consumption (MWh)"].plot(ax=ax[0], label="training", color='dodgerblue')
test_set["Electricity Consumption (MWh)"].plot(ax=ax[0], label="testing", color='green')
pred_full_Elec.plot(ax=ax[0], label='prediction', color="orange")
ax[0].legend()
ax[0].set_title(f"SARIMA for Electricity Consumption (MWh)", fontsize=16)
ax[0].grid(True)

test_set["Electricity Consumption (MWh)"].plot(ax=ax[1], label="testing", color='green')
electricity.plot(ax=ax[1], label='prediction', color="orange")
ax[1].legend()
ax[1].set_title(f"Closer look into Test and Predicted Values", fontsize=16)
ax[1].grid(True)
plt.tight_layout()
plt.show()
```



Observations:

- For Electricity Consumption, the predicted data seems to match the actual data trend very accurately.

Step 5: Model Improvement (SARIMA Electricity)

```
#print summary for Electricity Consumption
arima_modelElec.summary()
```

5]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	397
Model:	SARIMAX(2, 1, 1)x(2, 1, 1, 12)	**Log Likelihood**	-1762.175
Date:	Fri, 11 Aug 2023	**AIC**	3540.350
Time:	05:46:57	**BIC**	3571.955
Sample:	0	**HQIC**	3552.886
	- 397		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-0.0413	0.015	-2.840	0.005	-0.070	-0.013
ar.L1	0.5426	0.047	11.472	0.000	0.450	0.635
ar.L2	-0.0619	0.050	-1.244	0.213	-0.160	0.036
ma.L1	-0.9712	0.016	-62.000	0.000	-1.002	-0.941
ar.S.L12	-0.0511	0.061	-0.843	0.399	-0.170	0.068
ar.S.L24	-0.2547	0.064	-4.003	0.000	-0.379	-0.130
ma.S.L12	-0.6733	0.059	-11.358	0.000	-0.789	-0.557
sigma2	544.1684	32.582	16.701	0.000	480.308	608.029

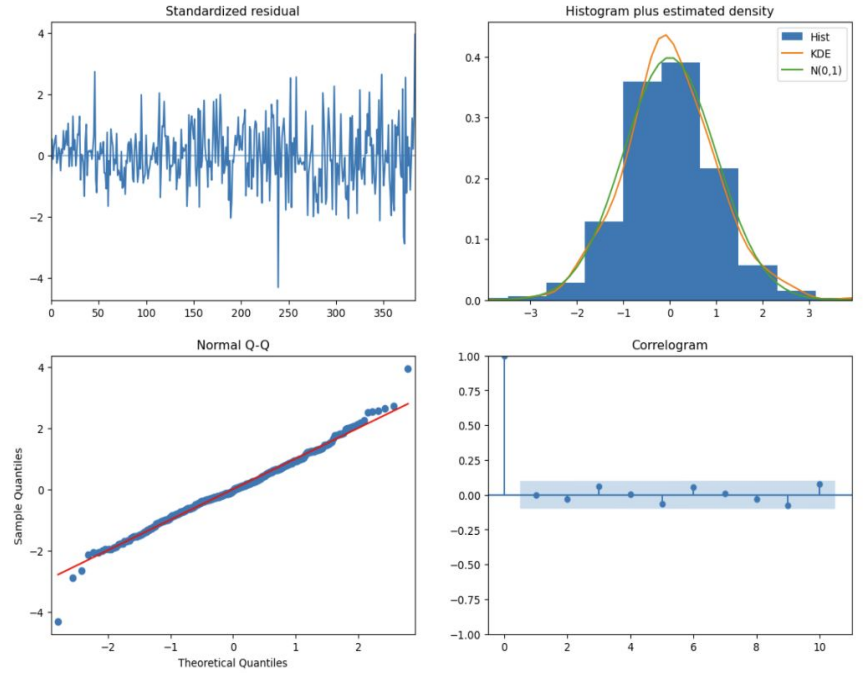
Ljung-Box (L1) (Q):	0.00	**Jarque-Bera (JB):**	23.47
Prob(Q):	0.98	**Prob(JB):**	0.00
Heteroskedasticity (H):	2.90	**Skew:**	0.05
Prob(H) (two-sided):	0.00	**Kurtosis:**	4.21

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

Observations:

- All coefficients are significant as all p-value are less than 0.5.
- Ljung Box test shows that the residuals are independently distributed, especially with a p-value so close to 1, suggesting it is a good model.
- Heteroscedasticity test shows that the residuals do not show variance.
- Jarque-Bera Test shows the data is not normally distributed against an alternative of another distribution.

```
#Plot diagnostic plot
arima_modelElec.plot_diagnostics(figsize=(14,10))
plt.show()
```



Observations:

- There is no obvious trend in the residuals in the standardized residual plot, showing that this can be a good fit.
- The histogram is symmetrical and the QQ plot shows all points lying on the line which means that the data follows a normal distribution.
- The correlogram shows that 95% of the correlations for lag do lie within the threshold, which shows that this is a good model.

Step 5: Model Improvement (VARMA - All Cols)

All Columns

- As mentioned, I will fit all columns into VARIMA to find the best suitable order as Johansen Test suggests that all variables have a cointergated relationship with one another.

Jupyter Report will show how this was evaluated: Requires more iterations for a better order to be found

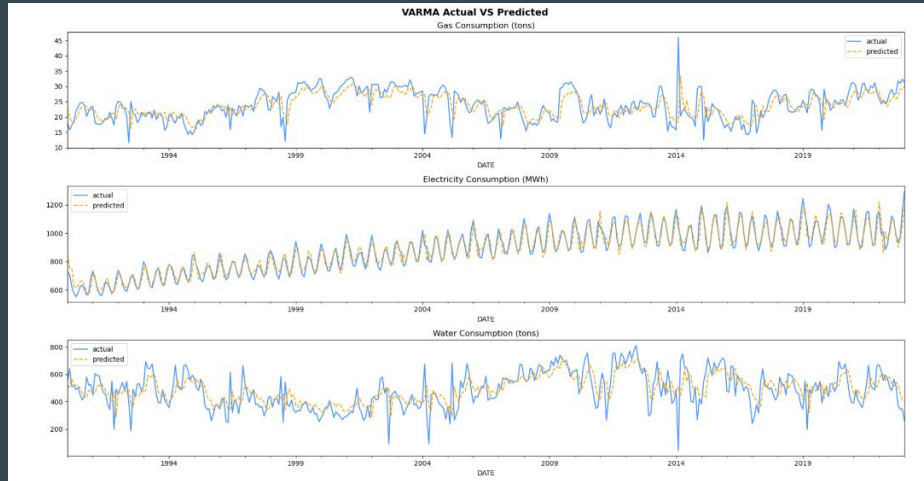
```
scoreMetrics_values={}

#Values of p & q
p_values = np.arange(0, 5)
q_values = np.arange(0, 5)
columns = ['Gas Consumption (tons)', 'Electricity Consumption (MWh)', 'Water Consumption (tons)']

for i in list(product(p_values, q_values)):
    order = (i[0], i[1])
    if order!=(0, 0):
        varma_model = VARMAX(endog=train_set[columns], order=order).fit(disp=False)
        (trainPredicted, testPredicted, pred_full, varma_model, aic, bic,
        v_Train_MAPE, v_Train_RMSE, v_Validate_MAPE, v_validate_RMSE)  = walk_forward_validate_varma(
            columns, train_set, test_set, varma_model, order
        )

        #concatenate order as a string
        order_string=f'({i[0]},{i[1]})'

        #store the score metrics in the dictionaries
        scoreMetrics_values[order_string] = {'AIC': round(aic,3), 'BIC': round(bic,3),
                                             'Train MAPE': v_Train_MAPE, 'Train RMSE': v_Train_RMSE,
                                             'Validate MAPE': v_Validate_MAPE, 'Validate RMSE': v_validate_RMSE}

#Convert the dictionary to dataframe
scoreMetrics_varma_AllCols=pd.DataFrame.from_dict(scoreMetrics_values, orient='index').sort_values(
    by=["BIC","Validate MAPE", "Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
display(scoreMetrics_varma_AllCols)
```

	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(4,1)	10965.274	11180.270	12.270	48.256	10.413	48.440
(4,0)	11003.509	11182.673	12.294	49.108	10.400	49.626
(3,1)	11049.429	11228.593	12.476	49.603	10.356	49.713
(3,2)	11016.253	11231.249	12.348	49.110	10.242	48.097

```
Order with Lowest BIC value: (4,1)
Lowest BIC value: 11180.27

AIC              10965.274
BIC              11180.270
Train MAPE          12.270
Train RMSE          48.256
Validate MAPE       10.413
Validate RMSE       48.440
Name: (4,1), dtype: float64
```

Observations:

- VARMA order of (4,1) has the lowest BIC value so it is the best order to be used for all these columns.
- Validate MAPE is 2% lower than the train MAPE so it means there is a lower different in errors between the test and predicted data than the train and predicted data.
- Validate RMSE is also quite close to the value of the train RMSE.
- Therefore, there should be a better order for the model that would give a better result if more iterations of order are being placed.



Observations:

- For all 3 columns, it seems like the predicted data does not seem to match the actual data but it looks much better than the graphs fitted by the previous order.

Step 5: Model Improvement (VARMA - Water_Gas)

Gas Consumption & Water Consumption

- As shown in Granger's Causality Test & Engle Granger Cointegration Test, I will fit Gas Consumption & Water Consumption into VARIMA to find the best suitable order as there is a causlity and cointegration relationship between these two variables.

```
scoreMetrics_values={}

#Values of p & q
p_values = np.arange(0, 5)
q_values = np.arange(0, 5)
columns = ['Gas Consumption (tons)', 'Water Consumption (tons)']

for i in list(product(p_values, q_values)):
    order = (i[0], i[1])
    if order!=(0, 0):
        varma_model = VARMAX(endog=train_set[columns], order=order).fit(disp=False)
        (trainPredicted, testPredicted, pred_full, varma_model, aic, bic,
        va_Train_MAPE, va_Train_RMSE, va_Validate_MAPE, va_validate_RMSE)  = walk_forward_validate_varma(
            columns, train_set, test_set, varma_model, order
        )

        #concatenate order as a string
        order_string=f'({i[0]},{i[1]})'

        #store the score metrics in the dictionaries
        scoreMetrics_values[order_string] = {'AIC': round(aic,3), 'BIC': round(bic,3),
                                             'Train MAPE': va_Train_MAPE, 'Train RMSE': va_Train_RMSE,
                                             'Validate MAPE': va_Validate_MAPE, 'Validate RMSE': va_validate_RMSE}

#Convert the dictionary to dataframe
scoreMetrics_varma_GasWater=pd.DataFrame.from_dict(scoreMetrics_values, orient='index').sort_values(
    by=["BIC","Validate MAPE", "Train MAPE","Validate RMSE","Train RMSE"],
    ascending=[True, True, True,True,True])
display(scoreMetrics_varma_GasWater)
```

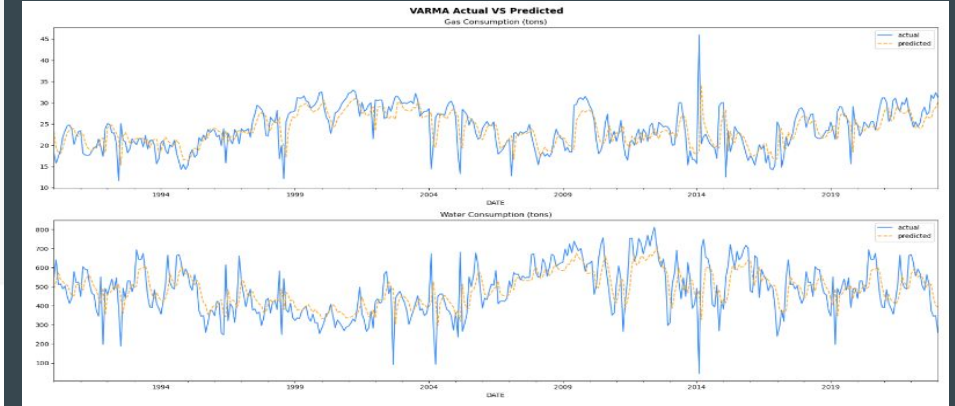
	AIC	BIC	Train MAPE	Train RMSE	Validate MAPE	Validate RMSE
(1,1)	6855.624	6907.383	16.783	53.829	12.845	45.666
(2,0)	6857.742	6909.500	16.817	54.298	12.710	45.282
(1,0)	6874.542	6910.374	17.203	55.770	12.596	45.357
(2,1)	6859.265	6926.949	16.828	53.734	12.828	46.545
(3,0)	6860.386	6928.070	16.725	53.806	12.880	46.076

```
lowestGasWaterBIC_order = scoreMetrics_varma_GasWater["BIC"].idxmin()
lowestGasWaterBIC_value = scoreMetrics_varma_GasWater["BIC"].min()
print(f"Order with Lowest BIC value: {lowestGasWaterBIC_order}")
print(f"Lowest BIC value: {lowestGasWaterBIC_value}")

lowestGasWaterBIC_cols = scoreMetrics_varma_GasWater.loc[lowestGasWaterBIC_order]
display(lowestGasWaterBIC_cols)
```

```
Order with Lowest BIC value: (1,1)
Lowest BIC value: 6907.383

AIC              6855.624
BIC              6907.383
Train MAPE         16.783
Train RMSE         53.829
Validate MAPE      12.845
Validate RMSE      45.666
Name: (1,1), dtype: float64
```

Observations:

- VARMA order of (1,1) has the lowest BIC value so it is the best order to be used for Gas Consumption & Water Consumption columns.
- Validate MAPE is 4% lower than the train MAPE so it means there is a lower different in errors between the test and predicted data than the train and predicted data.
- Validate RMSE is also much lower than the value of the train RMSE.
- Therefore, this is quite a good model for these features.



Observations:

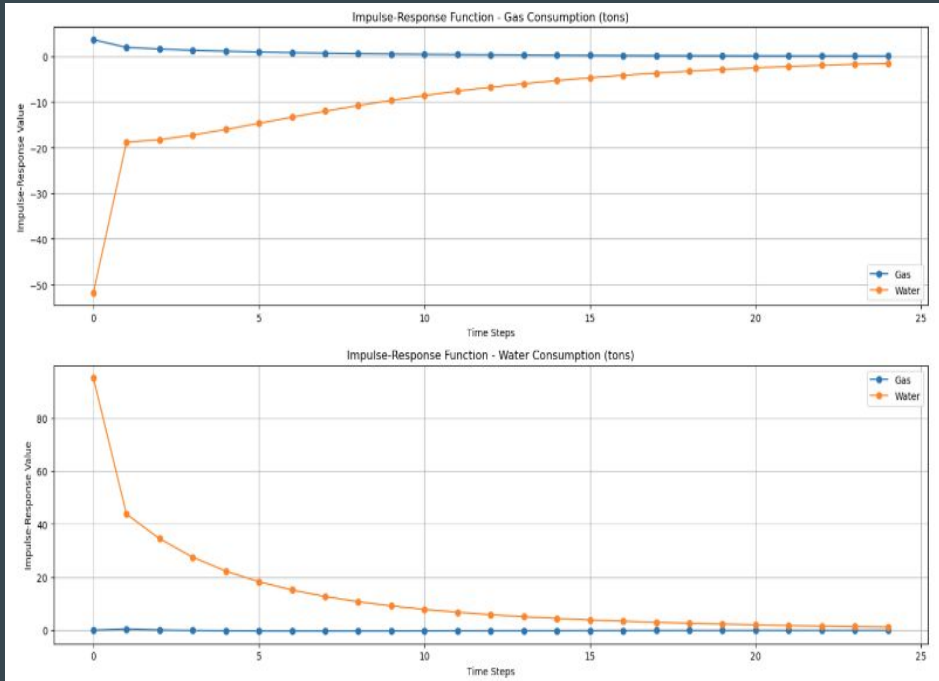
- For these 2 columns, it seems like the predicted data seems to be a good estimate of match the actual data.

Step 5: Model Improvement (VARMA - IRF)

VARMA (Impulse-Response Function)

- Provides insights into how an impulse to one variable affects both that variable and other variables in the system over multiple time periods by showing how the variables in the system respond to this impulse over time.
- It captures the lagged effect of the impulse on each variable.
- For each variable in the system, the IRF plots how the variable's value deviates from the normal path due to impulse.
- The direction and magnitude of the impact can be observed from the shape of the IRF curve. If the curve rises, it suggests a positive response, and if it falls, it suggests a negative response.

```
#Impulse-Response Function
varma_model = VARMAX(endog=train_set[columns], order=(1,1)).fit(disp=False)

gas_irfs = varma_model.impulse_responses(24, impulse=0, orthogonalized=True)
water_irfs = varma_model.impulse_responses(24, impulse=1, orthogonalized=True)

#Plot impulse reaction graph for better visualisation
fig, ax = plt.subplots(2, 1, figsize=(15,9))

index=0
irfs=[gas_irfs, water_irfs]
for col in columns:
    ax[index].plot(range(0,25), irfs[index], marker='o', label=["Gas", "Water"])
    ax[index].set_xlabel("Time Steps")
    ax[index].set_ylabel("Impulse-Response Value")
    ax[index].set_title(f"Impulse-Response Function - {col}")
    ax[index].legend()
    ax[index].grid(True)
    index+=1

plt.tight_layout()
plt.show()
```



Observations:

- In response to Gas Consumption, Water Consumption has a negative response which shows that Gas Consumption do indeed pose a significant causality to Water Consumption, as supported by the Granger's Causility Test.
- Gradually, the IRF converges to zero gradually as the impulse effects became to die out.

Step 5: Model Improvement (Durbin Watson Test)

VARMA (Durbin Watson Test)

Determines whether there is evidence of autocorrelation (serial correlation) in the residuals of a regression analysis. Checks for the presence of first-order autocorrelation. Test statistic take the values between 0 and 4. A value close to 2 suggests no autocorrelation (null hypothesis cannot be rejected). A value significantly less than 2 indicates positive autocorrelation (rejection of null hypothesis in favor of alternative). A value significantly greater than 2 indicates negative autocorrelation (rejection of null hypothesis in favor of alternative)

- H_0: There is no first-order autocorrelation in the residuals.
- H_1: There is first-order autocorrelation in the residuals.

```
#Conduct durbin watson test
def durbin_watson_test(df, resid=None):
    cols, stat = [], []
    out = durbin_watson(resid)
    for col, val in zip(df.columns, out):
        cols.append(col)
        stat.append(round(val, 2))
    dw_test = pd.DataFrame(stat, index=cols, columns=['Durbin Watson Test Statistic'])
    return dw_test
```

```
#Perform durbin watson test
dw_results = durbin_watson_test(df=energyConsumption_df[columns], resid=varma_model.resid)
display(dw_results)
```

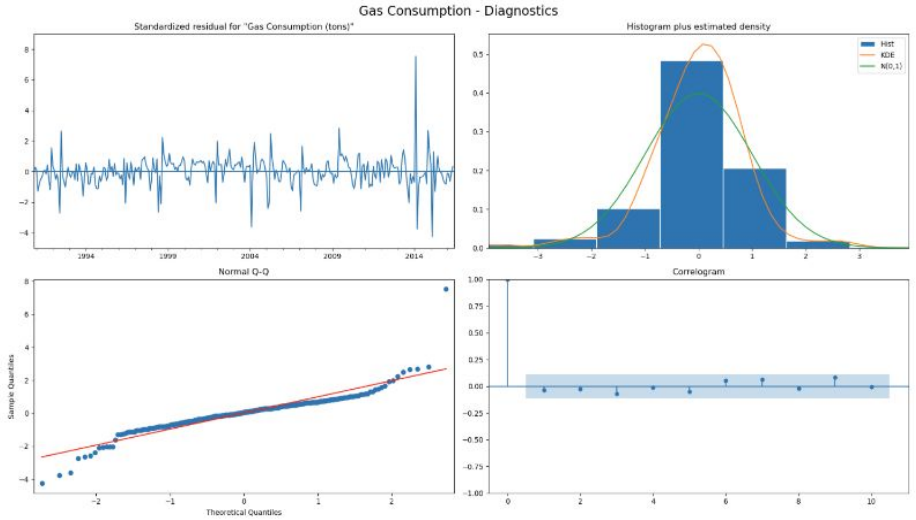
	Durbin Watson Test Statistic
Gas Consumption (tons)	2.06
Water Consumption (tons)	1.92

Observations:

- For Gas Consumption, Durbin Watson Test Statistic shows that there is a negative autocorrelation as test statistic is larger than 2 while there is a positive autocorrelation for Water Consumption as test statistic is lesser than 2.

Step 5: Model Improvement (Diagnostics Plot)

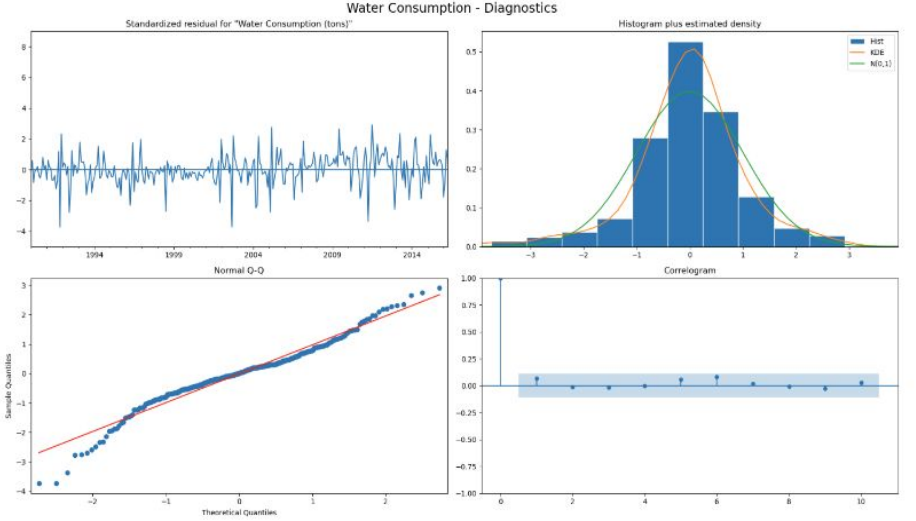
```
#Diagnostics plot for Gas Consumption
fig=varma_model.plot_diagnostics(variable=0)
plt.gcf().suptitle('Gas Consumption - Diagnostics', fontsize=18)
ax = fig.axes[0]
ax.set_xlim(250, 557)
ax.set_ylim(-5,9)
ax.axhline(0)
plt.tight_layout()
plt.show()
```



Observations for Gas Consumption:

- There is no obvious trend in the residuals in the standardized residual plot, suggests that this is a good model.
- The histogram is slightly symmetrical and the QQ plot shows all points lying on the line which means that the data follows a normal distribution.
- The correlogram shows that 95% of the correlations for lag do lie within the threshold, which shows that this is a good model.

```
#Diagnostics plot for Water Consumption
fig=varma_model.plot_diagnostics(variable=1)
plt.gcf().suptitle('Water Consumption - Diagnostics', fontsize=18)
ax = fig.axes[0]
ax.set_xlim(240, 555)
ax.set_ylim(-5,9)
ax.axhline(0)
plt.tight_layout()
plt.show()
```



Observations for Water Consumption:

- There is no obvious pattern or trend in the residuals in the standardized residual plot, showing that this model can be a good fit.
- The histogram is slightly symmetrical and the QQ plot shows all points lying on the line which means that the data follows a normal distribution.
- The correlogram shows that 95% of the correlations for lag do lie within the threshold, which shows that this is a good model.

Step 5: Model Improvement (Summary Results)

```
#Try out p=1, q=1
columns = ['Gas Consumption (tons)', 'Water Consumption (tons)']
varma_model = varma_model = VARMAX(endog=train_set[columns], order=(1,1)).fit(disp=False)
#Summary
varma_model.summary()
```

Statespace Model Results

Dep. Variable:	['Gas Consumption (tons)', 'Water Consumption (tons)']	**No. Observations:**	317
Model:	VARMA(1,1)	**Log Likelihood**	-2744.862
	+ intercept	**AIC**	5515.725
Date:	Fri, 11 Aug 2023	**BIC**	5564.591
Time:	16:19:32	**HQIC**	5535.244
Sample:	01-01-1990		
	- 05-01-2016		
Covariance Type:	opg		

Ljung-Box (L1) (Q):	0.43, 1.54	**Jarque-Bera (JB):**	2157.75, 59.97
Prob(Q):	0.51, 0.21	**Prob(JB):**	0.00, 0.00
Heteroskedasticity (H):	2.12, 1.39	**Skew:**	0.79, -0.41
Prob(H) (two-sided):	0.00, 0.09	**Kurtosis:**	15.68, 4.97

Look at Prob for each test

Results for equation Gas Consumption (tons)

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	7.9184	2.261	3.503	0.000	3.487	12.349
L1.Gas Consumption (tons)	0.7724	0.067	11.583	0.000	0.642	0.903
L1.Water Consumption (tons)	-0.0054	0.002	-2.267	0.023	-0.010	-0.001
L1.e(Gas Consumption (tons))	-0.1743	0.069	-2.545	0.011	-0.309	-0.040
L1.e(Water Consumption (tons))	0.0097	0.004	2.746	0.006	0.003	0.017

Results for equation Water Consumption (tons)

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	133.9090	58.055	2.307	0.021	20.123	247.695
L1.Gas Consumption (tons)	-1.5998	1.601	-0.999	0.318	-4.738	1.539
L1.Water Consumption (tons)	0.8022	0.065	12.250	0.000	0.674	0.931
L1.e(Gas Consumption (tons))	2.9725	1.916	1.552	0.121	-0.782	6.727
L1.e(Water Consumption (tons))	-0.3416	0.092	-3.708	0.000	-0.522	-0.161

Error covariance matrix

	coef	std err	z	P>|z|	[0.025	0.975]
sqrt.var.Gas Consumption (tons)	3.6825	0.096	38.265	0.000	3.494	3.871
sqrt.cov.Gas Consumption (tons).Water Consumption (tons)	-51.7898	5.275	-9.817	0.000	-62.129	-41.450
sqrt.var.Water Consumption (tons)	95.0667	3.322	28.615	0.000	88.555	101.578

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

Observations:

- Ljung Box test shows that the residuals are independently distributed for all 2 variables.
- Heteroscedasticity test shows that the residuals do not show variance for gas consumption but residuals show variances for water consumption.
- Jarque-Bera Test shows the data is not normally distributed against an alternative of another distribution for all 3 variables.

Summary

- SARIMA works better for **Electricity Consumption** which clearly show **seasonality** in the data.
- ARIMA tends to perform better for Gas Consumption & Water Consumption which do not show seasonality in the data.
- VARMA work well when there is causality and cointegration between multiple variables but it will require additional data transformation as shown.
- Model improvement to find the most suitable order for each data will **require more time and higher processing speed** as **larger number of iterations** will be required to find the **best order**.
- Vector Error Correction Models (VECM) could be run if there was more time for variables that show **long term relationship and no short term relationship** between one another based on the **Engle-Granger Cointegration Test results** & **Granger Causality Test**.