The George Washington University

Final Project Report

DATS6313

Time Series Analysis and Modeling

https://dashapp-879864239466.us-east1.run.app/

Apoorva Reddy Bagepalli

Dr. Reza Jafari

Table of Contents

Table of figures and tables

Abstract

The energy market is characterized by fluctuations in prices due to various factors, including energy generation, consumption, and weather conditions. Predicting energy prices is a critical task for energy producers, consumers, and market participants to make informed decisions and optimize energy usage and trading strategies. In this project, we aim to predict the "Price Actual" in the energy market using a dataset that combines hourly energy demand, generation, and weather data.

The objective is to build accurate time series models that can forecast future energy prices based on historical energy generation and consumption data. By leveraging time series forecasting techniques, such as ARMA, ARIMA, SARIMA, and the Box-Jenkins methodology, we will investigate the impact of various numerical independent variables, such as different sources of energy generation (e.g., biomass, fossil fuels, solar, wind) and total load, on the prediction of energy prices.

The project will focus on preprocessing the dataset, including data merging and handling missing values, followed by the application of appropriate time series models to predict future energy prices. The goal is to evaluate the performance of each model and provide insights into which technique is most effective for predicting energy prices in this specific context.

Introduction

This project focuses on predicting energy prices using a dataset that includes hourly energy demand and generation. The primary objective is to develop robust time series models that can accurately forecast the "Price Actual" (the actual energy price) based on various independent variables such as energy generation from different sources (e.g., biomass, solar, wind, fossil fuels) and total energy load. The dependent variable, "Price Actual," represents the actual market price of electricity at a given time, and accurate forecasting of this variable can assist in better energy trading, policy planning, and operational efficiency.

To achieve this goal, we will apply several time series forecasting techniques, including ARMA (Autoregressive Moving Average), ARIMA (Autoregressive Integrated Moving Average), SARIMA (Seasonal ARIMA), and the Box-Jenkins methodology. These models are widely used for time series analysis due to their effectiveness in capturing trends, seasonality, and autocorrelation in the data.

The dataset requires extensive preprocessing to handle missing values, potential outliers, and time series-specific challenges such as seasonality and non-stationarity. The project will involve performing stationarity tests, time series decomposition, and feature selection to prepare the data for modeling. Once the data is preprocessed, different models will be trained, evaluated, and compared based on their accuracy in forecasting the energy price.

By the end of the project, we aim to identify the most accurate forecasting model, providing valuable insights into energy price prediction. This will contribute to improving decision-making processes in energy markets and furthering the understanding of how different factors influence energy prices over time.

Data Description

The dataset used for this project focuses on energy consumption, generation, and prices, recorded on an hourly basis. It contains several features related to energy generation from different sources, as well as weather data, and the actual price of energy at each timestamp. The dataset is indexed with a Datetime index localized to Central European Time (CET), providing hourly records of energy generation and corresponding price information.

Here are the key variables in the dataset:

1. **Datetime index (CET):**
   This is the timestamp for each data entry, indicating the exact time at which the measurements were recorded. It is localized to Central European Time (CET), which ensures proper handling of time-related patterns, including seasonal variations.

2. **Generation variables (in MW):**
   These columns represent the energy generation from various sources, measured in megawatts (MW), at each hourly timestamp. The generation sources are:
   - **generation biomass:** Biomass energy generation.
   - **generation fossil brown coal/lignite:** Energy generated from brown coal or lignite.
   - **generation fossil coal-derived gas:** Energy generated from coal-derived gas.
   - **generation fossil gas:** Energy generated from natural gas.
   - **generation fossil hard coal:** Energy generated from hard coal.
   - **generation fossil oil:** Energy generated from oil.
   - **generation fossil oil shale:** Energy generated from shale oil.
   - **generation fossil peat:** Energy generated from peat.
   - **generation geothermal:** Energy generated from geothermal sources.
   - **generation hydro pumped storage aggregated:** Energy generated from hydro-pumped storage (aggregated).
   - **generation hydro pumped storage consumption:** Energy consumed from hydro-pumped storage.
   - **generation hydro run-of-river and poundage:** Energy generated from hydro run-of-river and poundage.
   - **generation hydro water reservoir:** Energy generated from hydro water reservoirs.
   - **generation marine:** Energy generated from marine sources (e.g., tidal power).
   - **generation nuclear:** Energy generated from nuclear sources.
   - **generation other:** Energy generated from other sources.
   - **generation other renewable:** Energy generated from other renewable sources.
   - **generation solar:** Energy generated from solar power.
   - **generation waste:** Energy generated from waste.
   - **generation wind offshore:** Energy generated from offshore wind power.
   - **generation wind onshore:** Energy generated from onshore wind power.

	time	generation biomass	generation fossil brown coal/lignite	generation fossil gas	generation fossil hard coal	generation fossil oil	generation hydro pumped storage consumption	generation hydro run-of-river and poundage	generation hydro water reservoir	generation nuclear
0	2014-12-31 23:00:00+00:00	447.0	329.0	4844.0	4821.0	162.0	863.0	1051.0	1899.0	7096.0
1	2015-01-01 00:00:00+00:00	449.0	328.0	5196.0	4755.0	158.0	920.0	1009.0	1658.0	7096.0
2	2015-01-01 01:00:00+00:00	448.0	323.0	4857.0	4581.0	157.0	1164.0	973.0	1371.0	7099.0
3	2015-01-01 02:00:00+00:00	438.0	254.0	4314.0	4131.0	160.0	1503.0	949.0	779.0	7098.0
4	2015-01-01 03:00:00+00:00	428.0	187.0	4130.0	3840.0	156.0	1826.0	953.0	720.0	7097.0

Figure 1 : Original data

Pre-Processing the Data

In the data cleaning process, the following steps were performed:

1. **Time Conversion:** The `time` column was converted to a `datetime` format with UTC timezone information using `pd.to_datetime()`. The column was then set as the index of the dataset to facilitate time-based operations.
2. **Feature Selection:** A subset of relevant features was selected from the dataset, focusing on various energy generation sources and the actual electricity price.
3. **Missing Values Handling:** The presence of missing values was checked using `isnull().sum()`, and the missing data were imputed. The missing values were filled by calculating the mean of the corresponding hour using the `transform()` function and grouped by the hour to handle missing data

based on hourly patterns. The approach ensures that imputed values reflect typical values for each hour of the day.

4. **Hour Extraction:** An additional feature `hour` was extracted from the `datetime` index to capture the time of day, which might influence energy generation and prices.

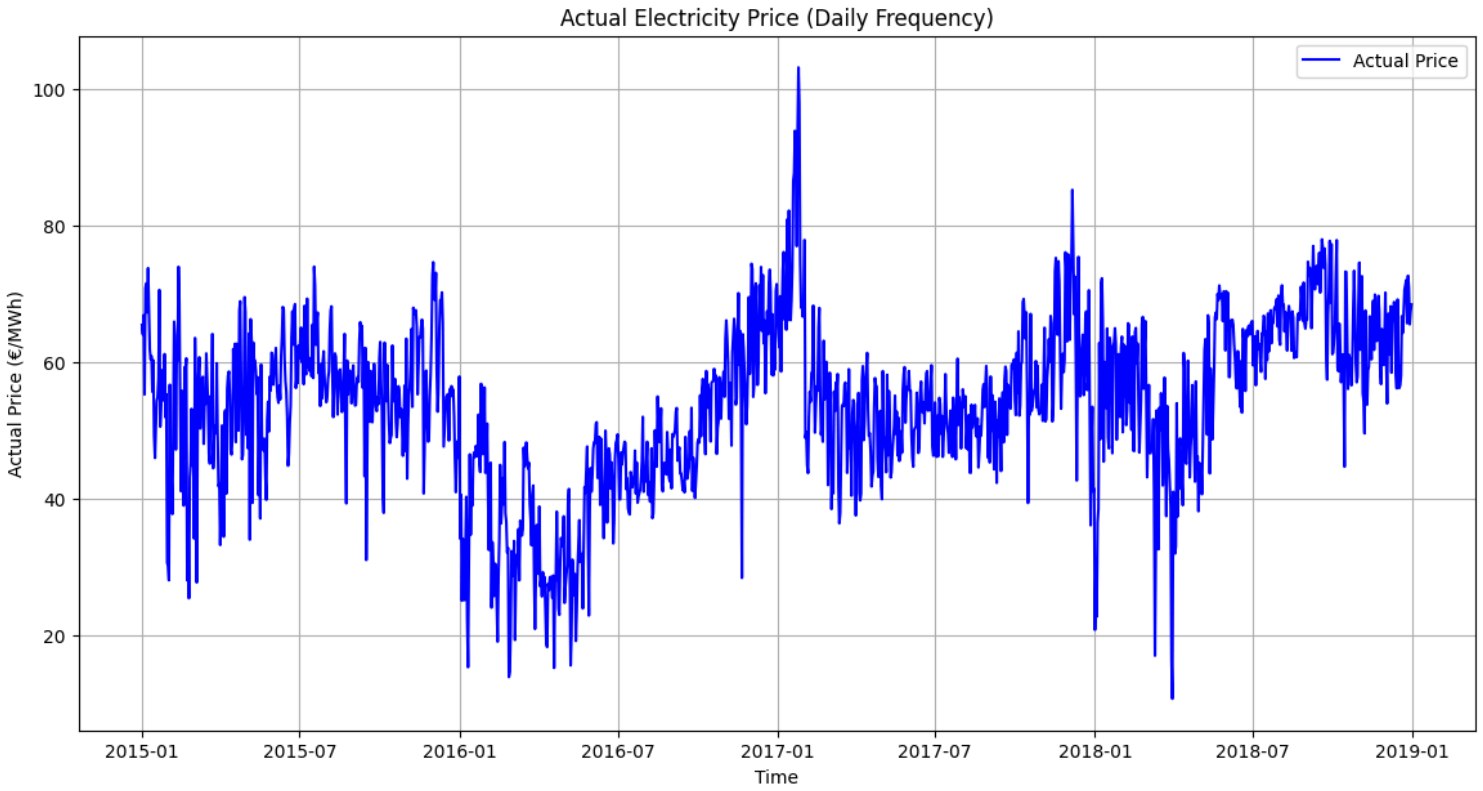

Figure 2 : Daily Energy price plot

The plot shows the actual electricity price (in €/MWh) on the y-axis versus time on the x-axis, with data points spanning from January 2015 to January 2019. The blue line on the graph represents the actual price.

Key observations include:

- **Downward Trend (Early 2015 to Mid-2016):** A general decrease in prices.
- **Price Spike (Early 2017):** A significant increase, reaching above 100 €/MWh.
- **Stable Period (Mid-2017 to Mid-2018):** Prices were relatively lower and more stable.
- **Another Increase (End of 2018):** Prices began to rise again.

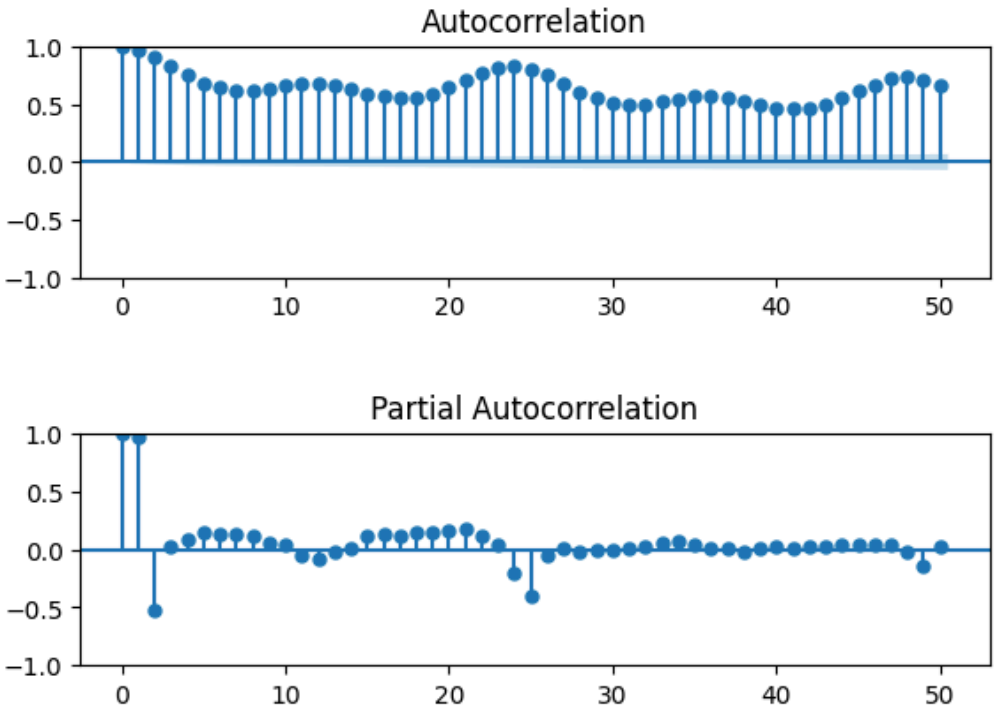

Figure 3 : ACF/PACF plots

ACF Plot:

- Significant positive autocorrelation at lag 1, gradually decreasing but remaining positive up to around lag 50.
- Periodic fluctuations indicate a cyclical pattern in the time series data.

PACF Plot:

- Significant positive partial autocorrelation at lag 1, followed by a sharp drop to negative values at lag 2.
- Beyond lag 2, partial autocorrelation values fluctuate around zero, with minor peaks and troughs.

Given the observations, the data aligns with the Autoregressive Moving Average Model.

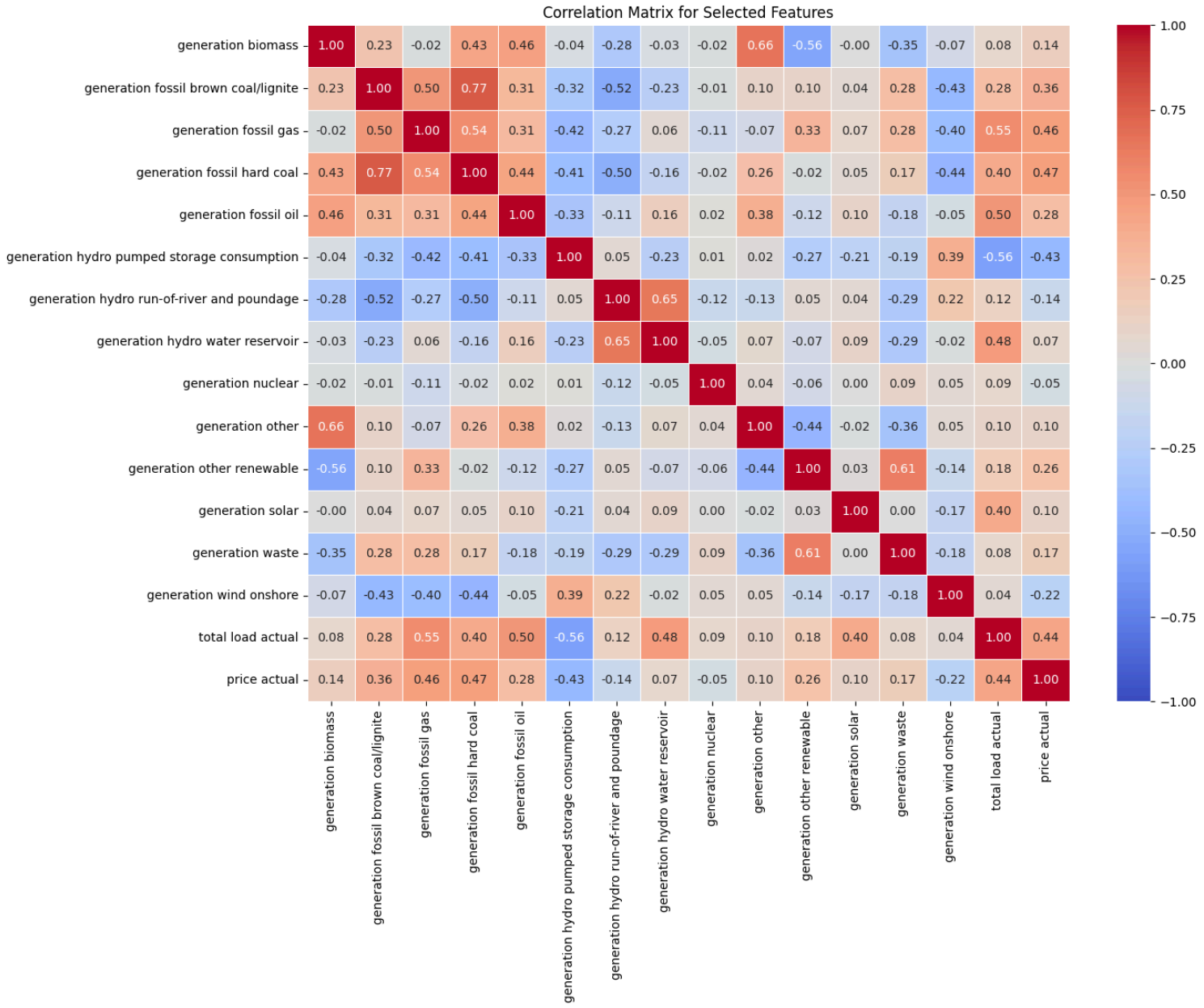

Figure 4 : Correlation Matrix

Strong Positive Correlations:

- `Generation biomass` and `Generation other` show a high positive correlation (0.66).
- `Generation fossil brown coal/lignite` and `Generation fossil hard coal` have a high positive correlation (0.77).
- `Generation fossil gas` and `Generation fossil oil` demonstrate a high positive correlation (0.54).
- `Generation hydro water reservoir` and `Generation hydro run-of-river and poundage` exhibit a high positive correlation (0.65).
- `Generation other renewable` and `Generation solar` have a high positive correlation (0.61).
- `Total load actual` and `Generation fossil gas` show a high positive correlation (0.55).

Strong Negative Correlations:

- `Generation biomass` and `Generation other renewable` have a strong negative correlation (-0.56).
- `Generation fossil brown coal/lignite` and `Generation other renewable` show a strong negative correlation (-0.52).
- `Generation fossil hard coal` and `Generation other renewable` have a strong negative correlation (-0.50).

- `Generation hydro pumped storage consumption` and `Generation other renewable` exhibit a strong negative correlation (-0.56).
- `Generation other renewable` and `Generation wind onshore` show a strong negative correlation (-0.36).

Price Correlations:

- `Price actual` has a moderate positive correlation with `Generation fossil gas` (0.46), `Generation fossil hard coal` (0.47), and `Total load actual` (0.44).
- `Price actual` has a moderate negative correlation with `Generation hydro pumped storage consumption` (-0.43).

Stationarity

Original Data

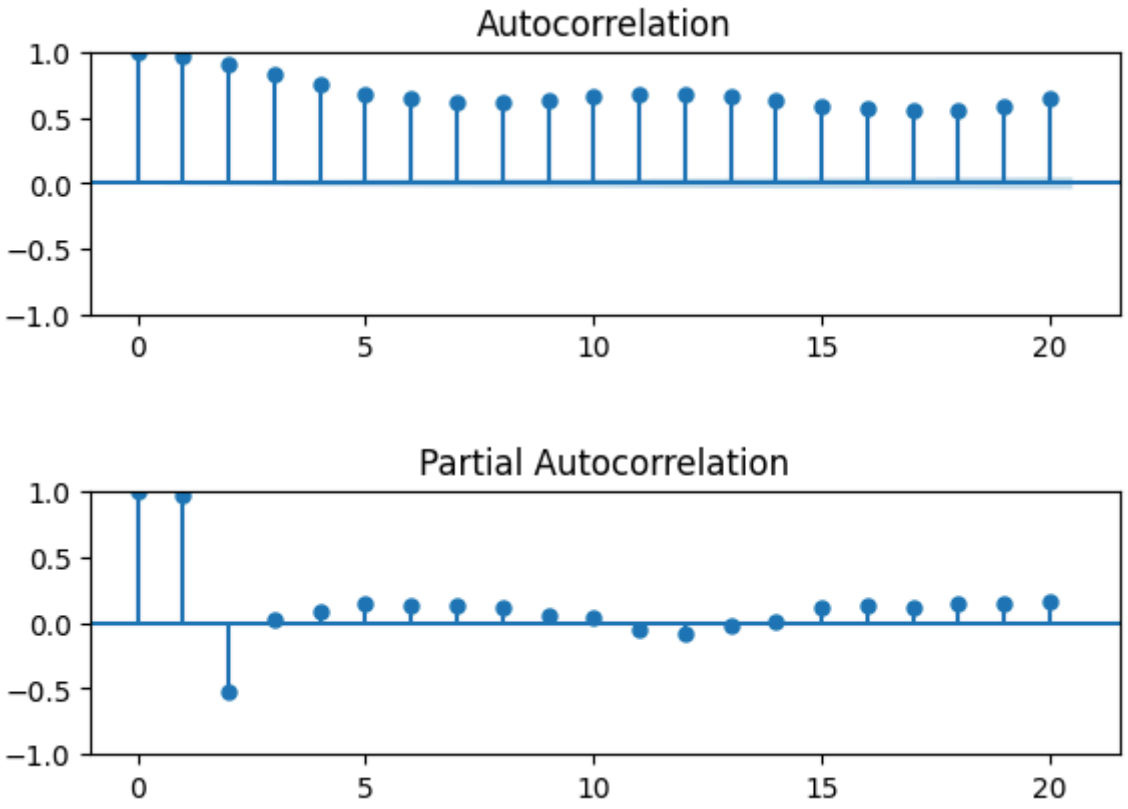

Figure 5 : ACF/PACF plot on original data

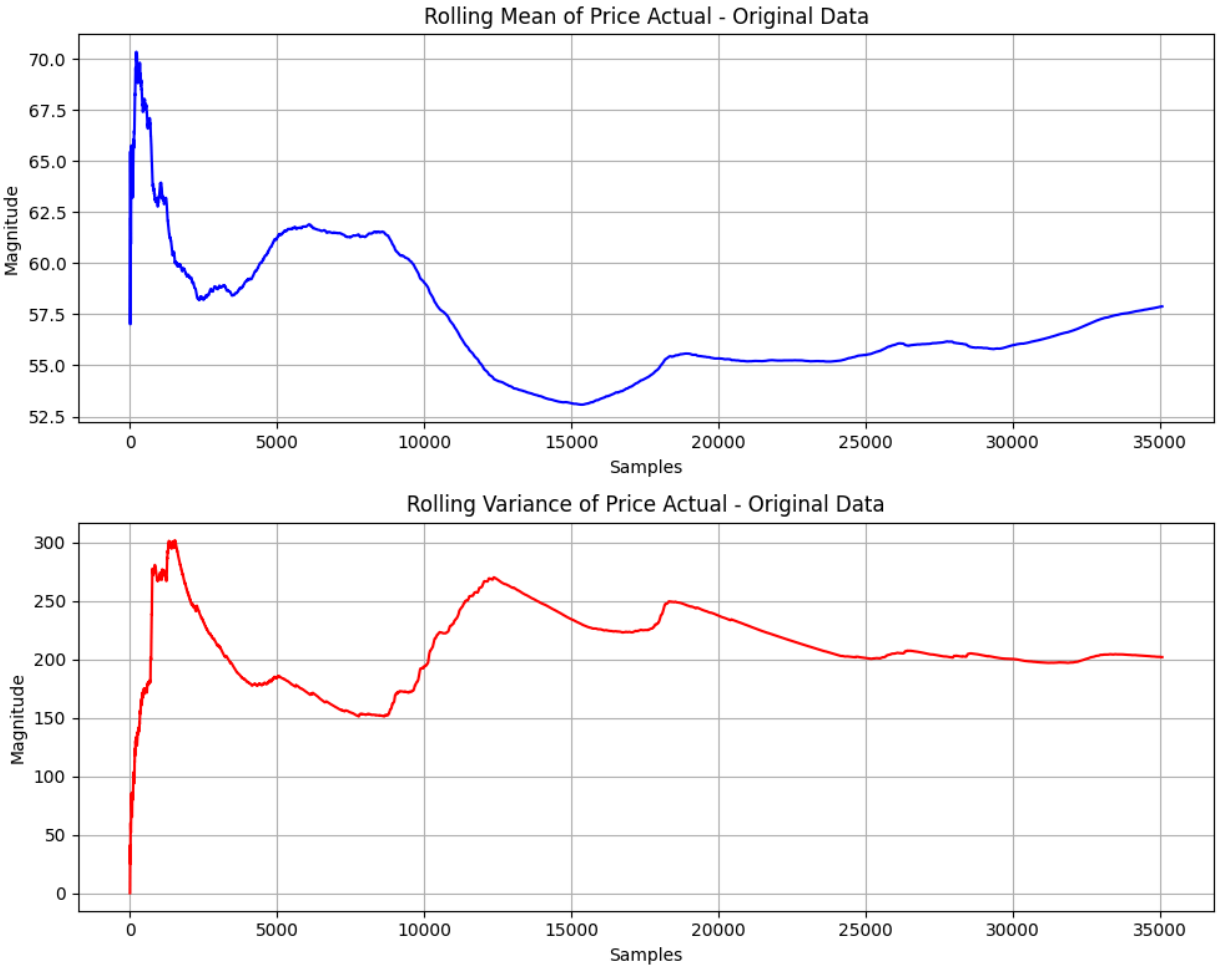

Figure 6 : Rolling Mean/Variance on Original data

```
ADF Test Statistic: -9.14701623285117
p-value: 2.7504934849345482e-15
Critical Values: {'1%': np.float64(-3.4305367814665044), '5%': np.float64(-2.8616225527935106), '10%': np.float64(-2.566813940257257)}
Series is likely stationary.
KPSS Test Statistic: 4.330033575195486
p-value: 0.01
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
Series is likely non-stationary.
```

Figure 7 : ADF/KPSS test on Original data

The rolling mean and variance plots of the original data reveal significant trends in price changes over time. Initially, the rolling mean shows a sharp decrease from around 70 to 60 in the first 5,000 samples, then fluctuates, hitting a low of about 53 around the 15,000 sample mark, and finally increasing towards 60 near the end of the period. The rolling variance starts high at approximately 300, rapidly decreases to around 200 within the first 5,000 samples, and gradually declines to a low of 100 at the 15,000 sample mark before stabilizing around 150 towards the end. These patterns indicate substantial early variability in the dataset, followed by a stabilization trend in both mean and variance as the series progresses.

Transformed Data

On the original data, seasonal differencing order of 365 was applied to make the data stationary

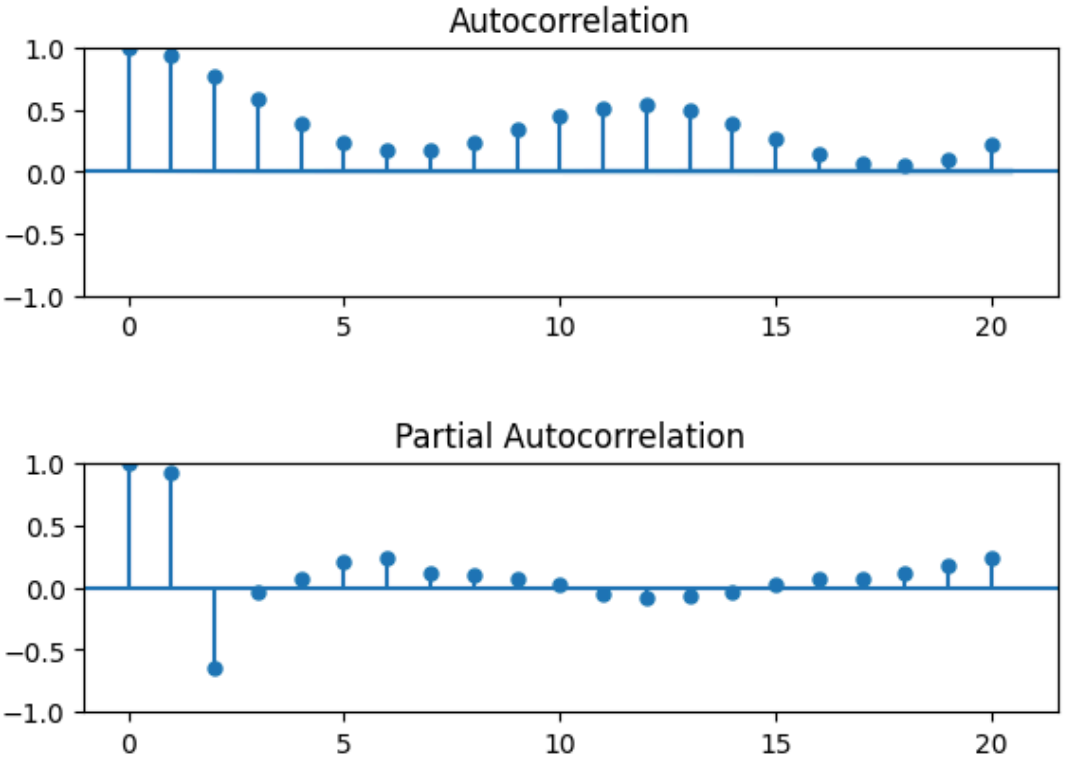

Figure 8 : ACF/PACF plot on transformed data

The ACF and PACF plots of the transformed data, obtained after applying seasonal differencing with a period of 365, exhibit distinct patterns. The ACF plot shows significant autocorrelation at various lags, indicating persistent correlations with past values despite the seasonal adjustment. The PACF plot reveals significant partial autocorrelation at the initial few lags, which quickly diminishes, suggesting that the time series values are mainly influenced by their recent past values after seasonal differencing. These observations are crucial for identifying the appropriate order of autoregressive and moving average terms for time series modeling, such as ARIMA or SARIMA models.

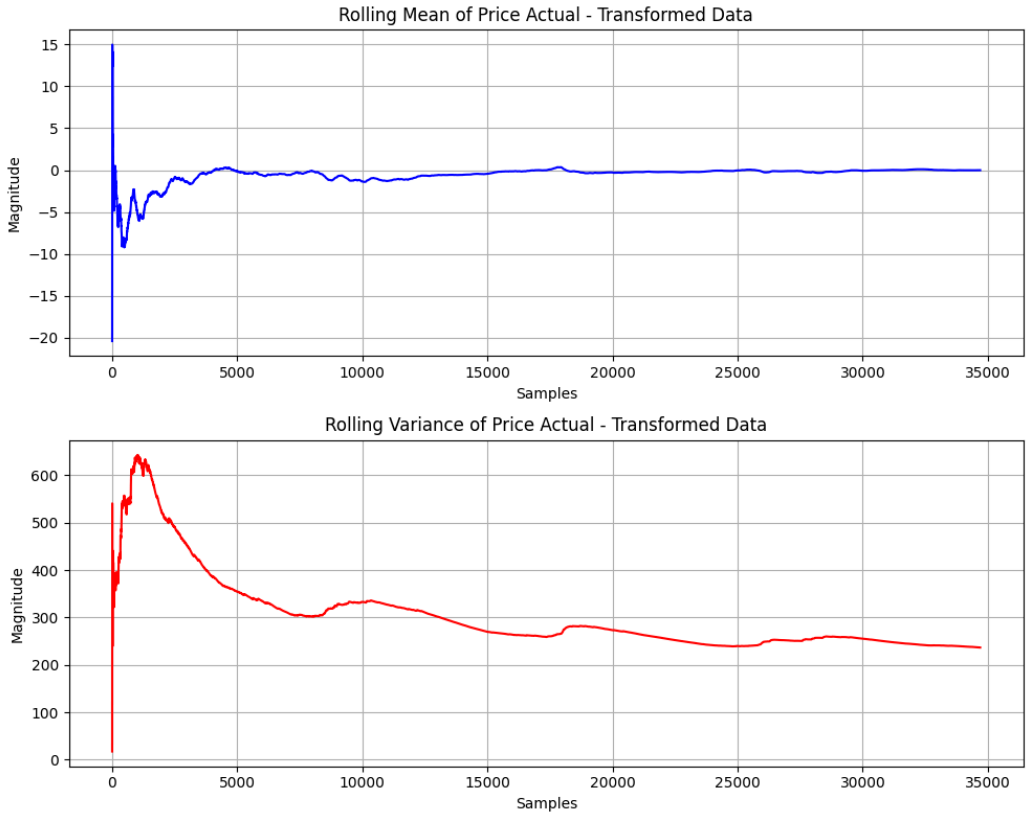

Figure 9 : Rolling Mean/Variance plot on transformed data

```
ADF Test Statistic: -15.257221287841633
p-value: 4.916547930227483e-28
Critical Values: {'1%': np.float64(-3.4305387601618236), '5%': np.float64(-2.8616234273001453), '10%': np.float64(-2.5668144057344913)}
Series is likely stationary.
KPSS Test Statistic: 0.14091585518110367
p-value: 0.1
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
Series is likely stationary.
```

Figure 10 : ADF/KPSS tests on transformed data

The rolling mean and variance plots of the transformed data show distinct patterns. The rolling mean fluctuates significantly at first, peaking around 15 and dipping to about -20, but eventually stabilizes near zero as the number of samples increases. The rolling variance starts high, peaking around 600, then gradually decreases with some fluctuations, ultimately stabilizing around 200 towards the end of the period. These trends indicate that the transformed data's variability decreases over time, and the mean converges to a more stable value, reflecting the impact of the seasonal differencing applied to the original time series data.

Time Series Decomposition

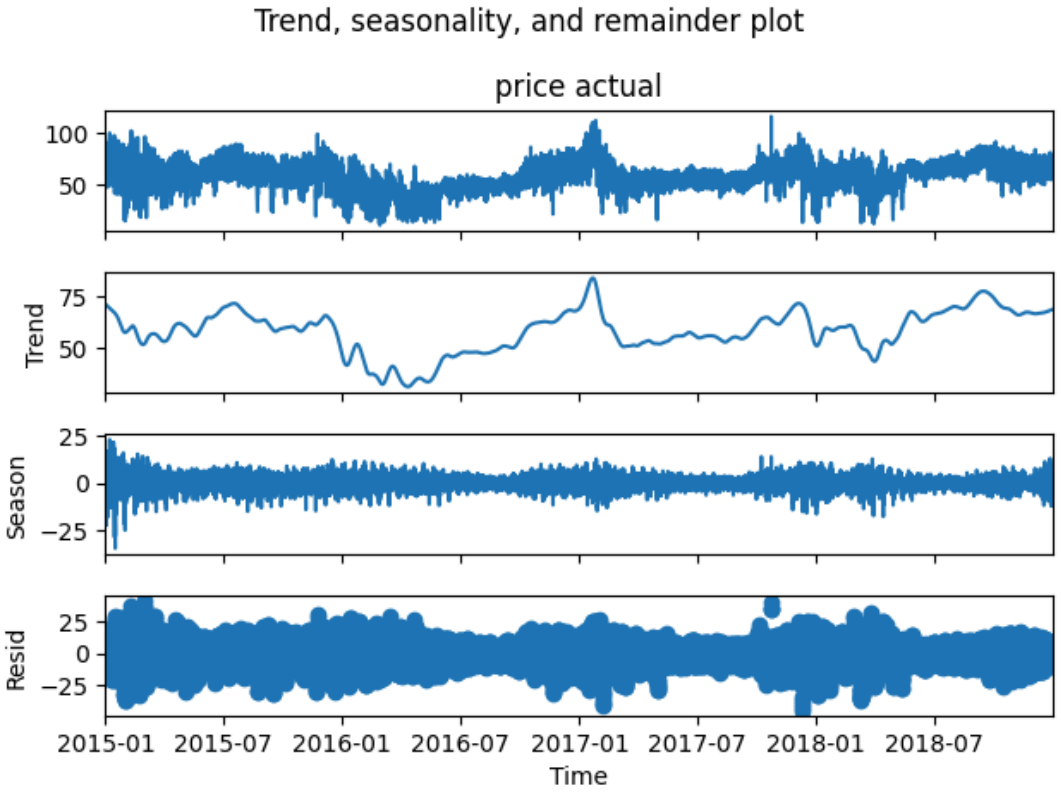

Figure 11 : Decomposition on Time Series Data

The time series decomposition plot breaks down the 'price actual' data into its components: observed data, trend, seasonality, and residuals. The observed data shows fluctuations from January 2015 to December 2018. The trend component reveals a general increase with notable peaks around mid-2017. The seasonal component captures consistent cyclical patterns over the time period. The residuals represent the remaining variations after removing the trend and seasonal components, indicating noise or irregularities in the data.

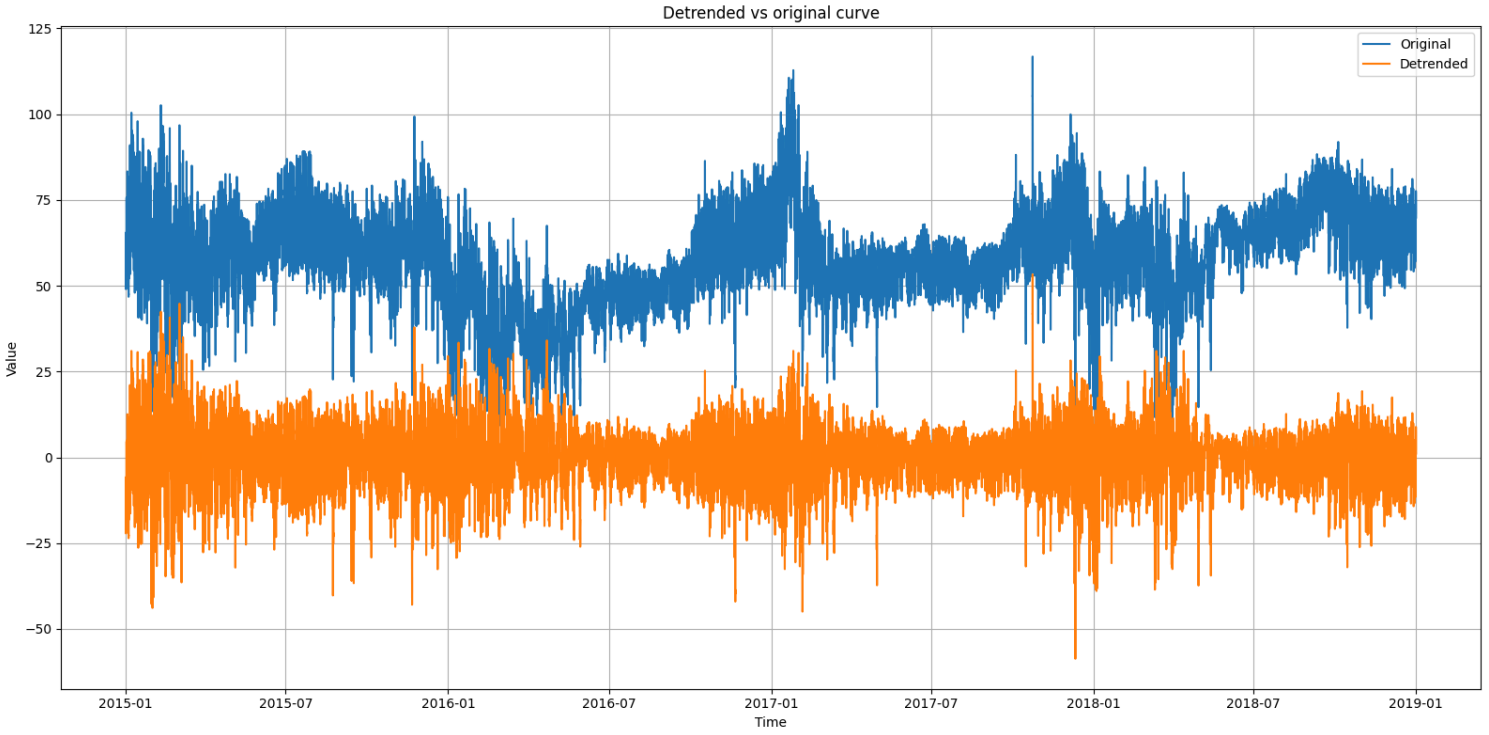

Figure 12 : Detrended plot

The detrended plot reveals that once the overall upward trend is removed, the data displays clear periodic fluctuations and cyclical patterns. This suggests that the original time series data has inherent variability and seasonality that were not immediately apparent due to the overriding trend.

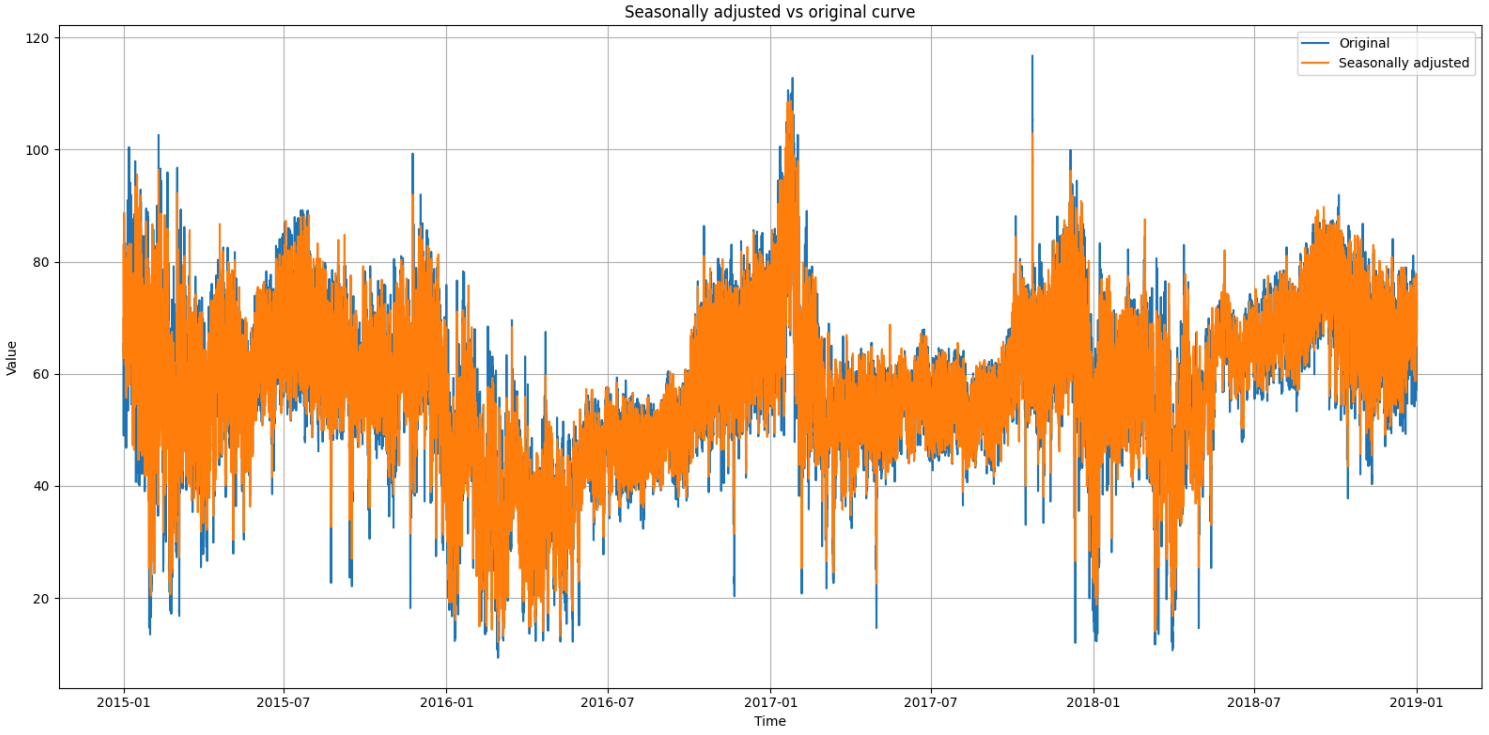

Figure 13 : Seasonally Adjusted plot

The seasonal adjusted plot compares the original data (in blue) with the seasonally adjusted data (in orange) from January 2015 to December 2018. The seasonally adjusted data smooths out the fluctuations seen in the original data, highlighting underlying trends and patterns more clearly. The original data exhibits significant variability due to seasonal effects, while the seasonally adjusted data reveals periods of increase and decrease in the overall trend without these seasonal variations, providing a clearer view of the data's true behavior over the four-year span.

```
Strength of Trend for this dataset is  0.646947143956092
Strength of seasonality for this dataset is  0.2818827896964492
```

The analysis indicates that the time series data has a strong trend component, with a strength of 0.64, compared to a moderate seasonal component, with a strength of 0.28. This suggests that the overall direction of the data over time is significantly influenced by long-term trends, while seasonal patterns play a lesser role. The high strength of the trend implies a dominant and consistent direction in the data, whereas the lower strength of seasonality suggests that repetitive patterns or cycles, although present, are less influential.

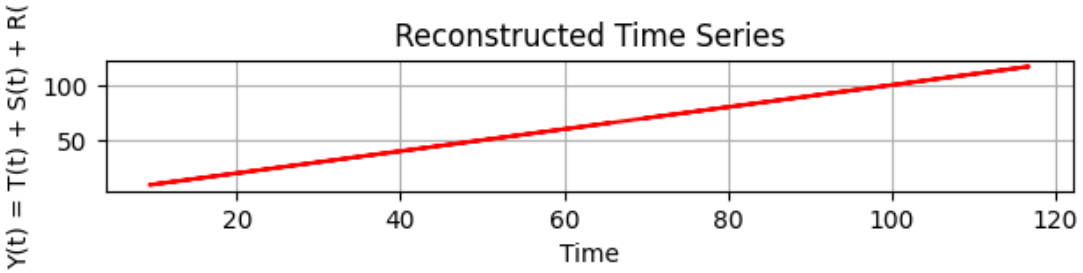

Figure 14 : Additive decomposition

The reconstructed time series plot of the additive model indicates that the data exhibits a strong linear upward trend over the analyzed period, as shown by the red line starting around 50 and reaching approximately 100 towards the end. This consistent increase highlights the presence of a dominant trend in the time series.

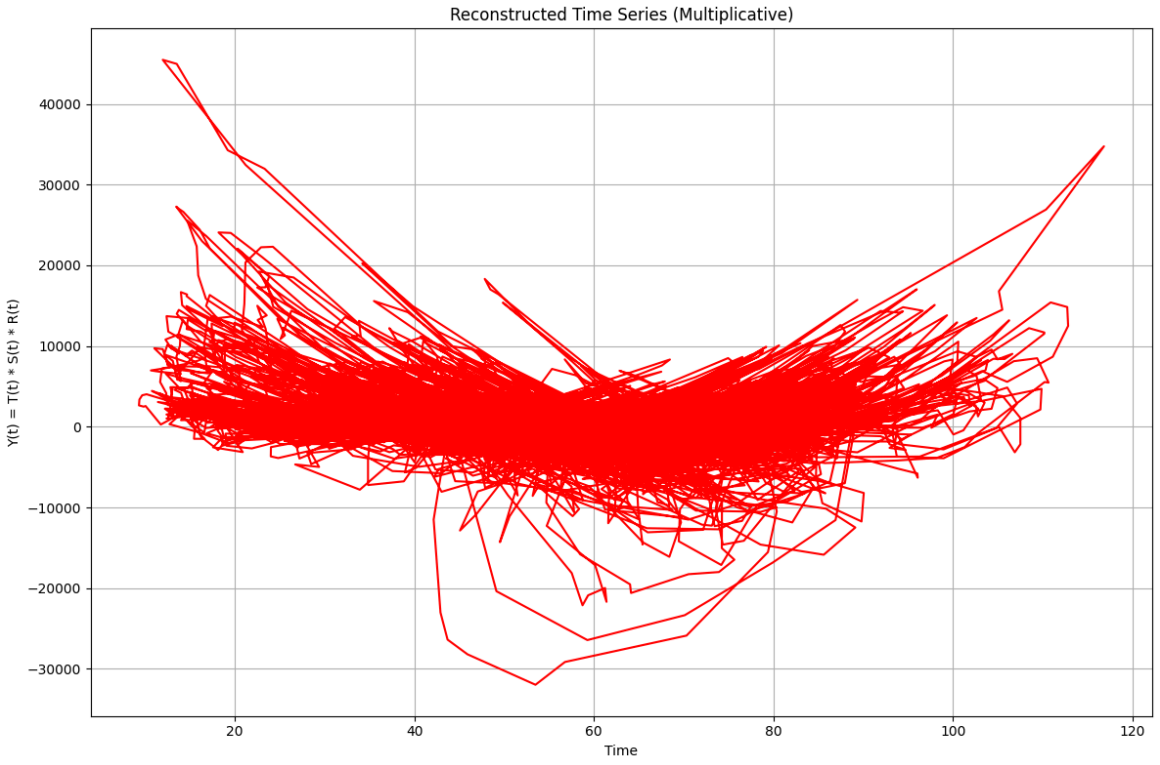

Figure 15 : Multiplicative decomposition

The reconstructed time series plot using a multiplicative model displays the combined effect of trend (T(t)), seasonality (S(t)), and residuals (R(t)) on the data. The plot reveals that the data exhibits a more complex pattern, with significant variations in amplitude over time. The values range from approximately -30,000 to 40,000 on the y-axis, with dense clustering in the middle of the plot, indicating overlapping patterns. This multiplicative model suggests that both the seasonal and trend components have a proportional impact on the overall data, leading to larger fluctuations when the underlying trend is higher.

The additive model is more appropriate for the data because it effectively captures the constant seasonal fluctuations and the linear trend observed in the time series, without suggesting that these fluctuations increase proportionally with the level of the series. The variance of the data does not appear to grow over time, which

supports the assumption of the additive model that seasonal and residual components remain stable. This model's simplicity and ease of interpretation make it a more accurate choice for representing the underlying patterns in the dataset, providing clearer insights for analysis and forecasting.

Holt-Winters method

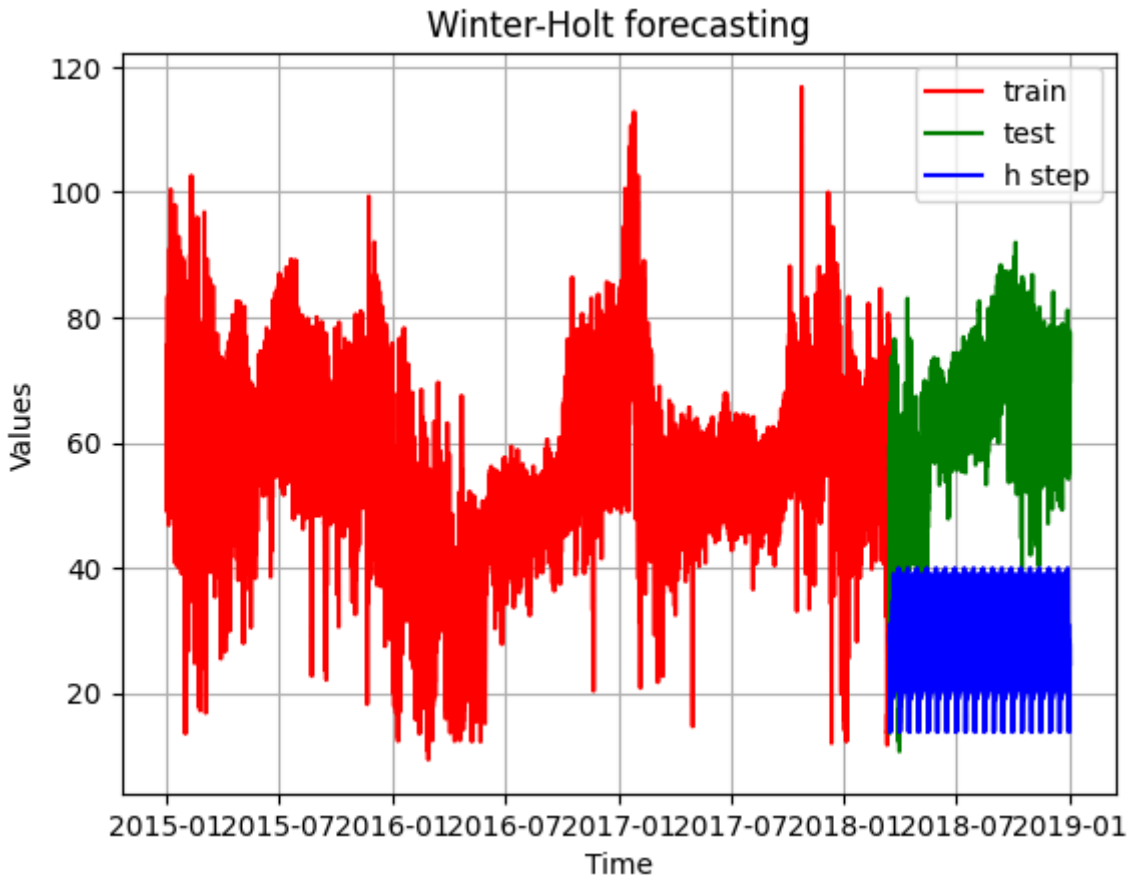

Figure 16 : Holt-Winters plot

```
MSE for Winter-Holt method: 1572.38
RMSE for Winter-Holt method: 39.65
MAE for Holt-Winter method: 37.72
```

The Holt-Winters model plot, comparing training, test, and forecasted values, shows that the model effectively captures the seasonality and trend of the time series. The training data fluctuates significantly, the test data shows a narrower range, and the forecasted values indicate a repetitive pattern. However, the forecasted values demonstrate less variability compared to the historical data, suggesting the model might need further refinement to better match the observed data's variability.

Feature Selection/ Dimensionality Reduction

```
condition_number: 9.48419260565559
SingularValues = [2.73364472e+13 3.84938144e+11 1.25494943e+11 7.81267880e+10
 6.05044445e+10 4.65543969e+10 1.68494846e+10 6.56433999e+09
 1.61948009e+09 1.15776918e+09 1.99789945e+08 5.47614357e+07
 3.92534488e+07 6.25996922e+06 2.27354135e+06]
```

Figure 17 : Multicollinearity check

The condition number of 9.48419260565559 indicates that there is no significant multicollinearity among the features, as it is well below the threshold of 30 which typically signals high multicollinearity. Additionally, examining the singular values, we see a range of values from 2.73364472e+13 down to 2.27354135e+06.

The large difference between the highest and lowest singular values further supports that the data matrix is well-conditioned, implying the selected features are appropriately scaled and contribute uniquely to the model without redundancy.

Backward Stepwise Regression

	AIC	BIC	Adjusted R^2	Features
0	214611.221044	214743.089517	0.393209	[generation biomass, generation fossil brown c...
1	214955.321824	215078.948518	0.385698	[generation fossil brown coal/lignite, generat...
2	214953.636153	215069.021067	0.385713	[generation fossil gas, generation fossil hard...
3	215165.544671	215272.687805	0.381033	[generation fossil hard coal, generation fossi...
4	216527.949303	216626.850658	0.350205	[generation fossil oil, generation hydro pumpe...
5	216545.869925	216636.529500	0.349767	[generation hydro pumped storage consumption, ...
6	216703.061703	216785.479499	0.346090	[generation hydro run-of-river and poundage, g...
7	217182.102182	217256.278198	0.334803	[generation hydro water reservoir, generation ...
8	219010.020087	219075.954323	0.289987	[generation nuclear, generation other, generat...
9	219239.549359	219297.241816	0.284128	[generation other, generation other renewable,...
10	219875.110554	219924.561231	0.267697	[generation other renewable, generation solar,...
11	219886.165557	219927.374454	0.267382	[generation solar, generation waste, generatio...
12	220461.006591	220493.973710	0.252187	[generation waste, generation wind onshore, to...
13	220508.396128	220533.121467	0.250896	[generation wind onshore, total load actual]
14	222165.889199	222182.372758	0.205271	[total load actual]

Figure 18 : Backward Stepwise Regression result

VIF analysis

```
Step 4:
AIC: 215165.5446706526, BIC: 215272.6878049448, Adjusted R²: 0.
3810330269974814
Remaining Features: ['generation fossil hard coal', 'generation fossil
oil', 'generation hydro pumped storage consumption', 'generation hydro
run-of-river and poundage', 'generation hydro water reservoir',
'generation nuclear', 'generation other', 'generation other renewable',
'generation solar', 'generation waste', 'generation wind onshore',
'total load actual']
VIF Data:
```

	Feature	VIF
0	generation fossil hard coal	21.202265
1	generation fossil oil	52.857375
2	generation hydro pumped storage consumption	2.533231
3	generation hydro run-of-river and poundage	17.588412
4	generation hydro water reservoir	12.426629
5	generation nuclear	55.639922
6	generation other	14.242224
7	generation other renewable	79.434294
8	generation solar	3.064193
9	generation waste	60.314688
10	generation wind onshore	11.678459
11	total load actual	257.940658

Figure 19 : VIF analysis result

PCA

```
Reduced dimensions: 10
```

For the multiple linear regression model, the feature selection process utilized PCA, backward stepwise regression, and VIF analysis to ensure collinearity issues were addressed. The final features chosen were: `generation other renewable`, `generation solar`, `generation waste`, `generation wind onshore`, and `total load actual`. This selection is based on their performance across AIC, BIC, and Adjusted R^2 metrics, indicating their significance in predicting the target variable.

Models

Average Forecasting

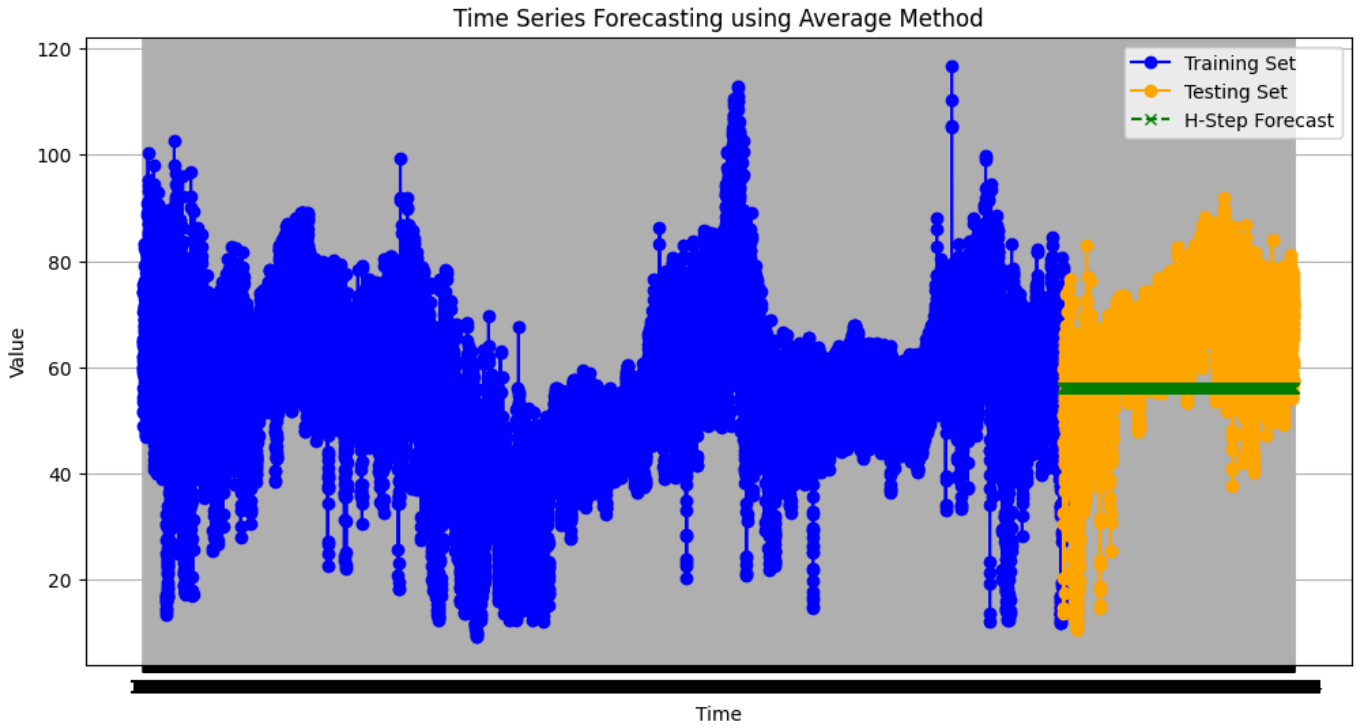

Figure 20 : Average Forecasting

The H-step forecast is a horizontal line representing the average value of the training set, extending into the testing set. This forecast method highlights how the average of past values is used to predict future values

```
MSE for Average forecasting: 213.78
RMSE for Average forecasting: 14.62
MAE for Average forecasting: 12.46
```

Drift Forecasting

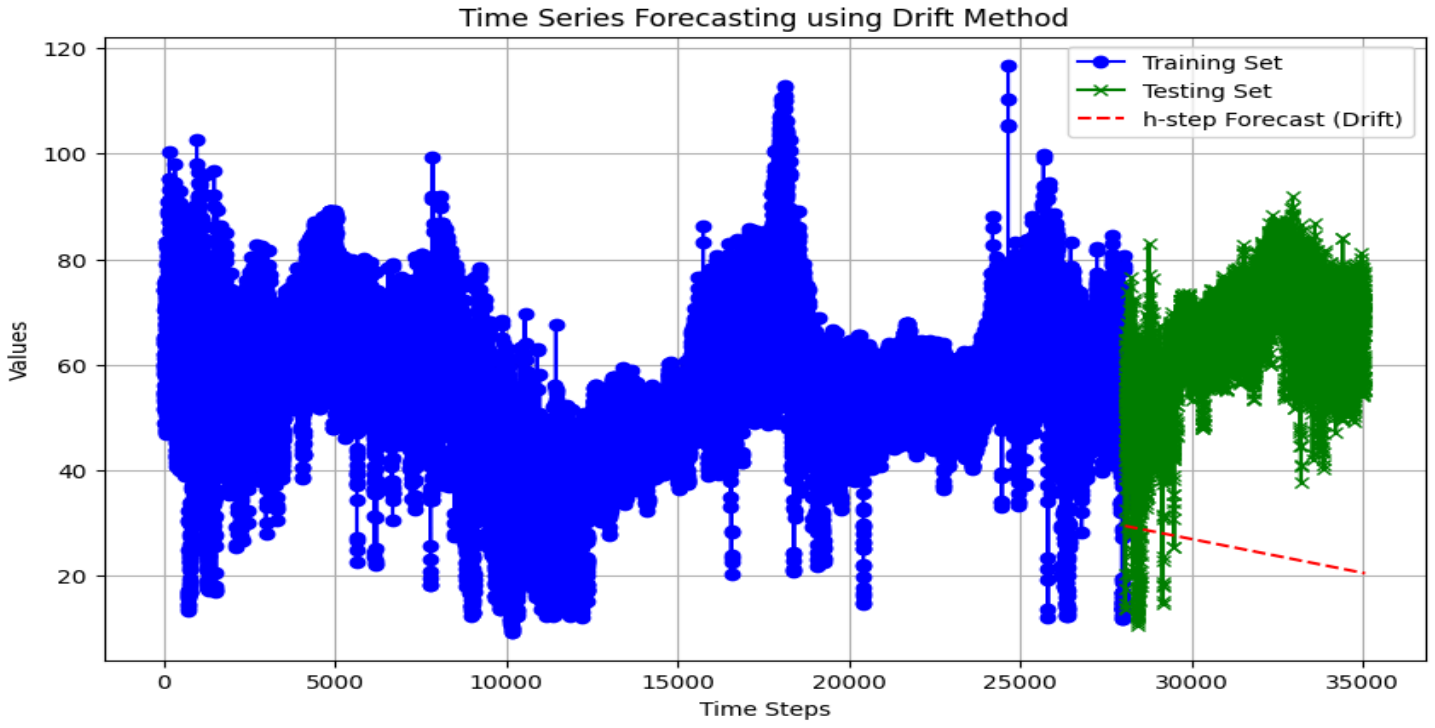

Figure 21 : Drift Forecasting

The forecasted values extend from the end of the training period, continuing the trend observed in the training data. The drift method effectively captures the linear trend, showing how past trends are projected forward to predict future values.

```
MSE for Drift forecasting: 1779.56
RMSE for Drift forecasting: 42.18
MAE for Drift forecasting: 40.34
```

Naive Forecasting

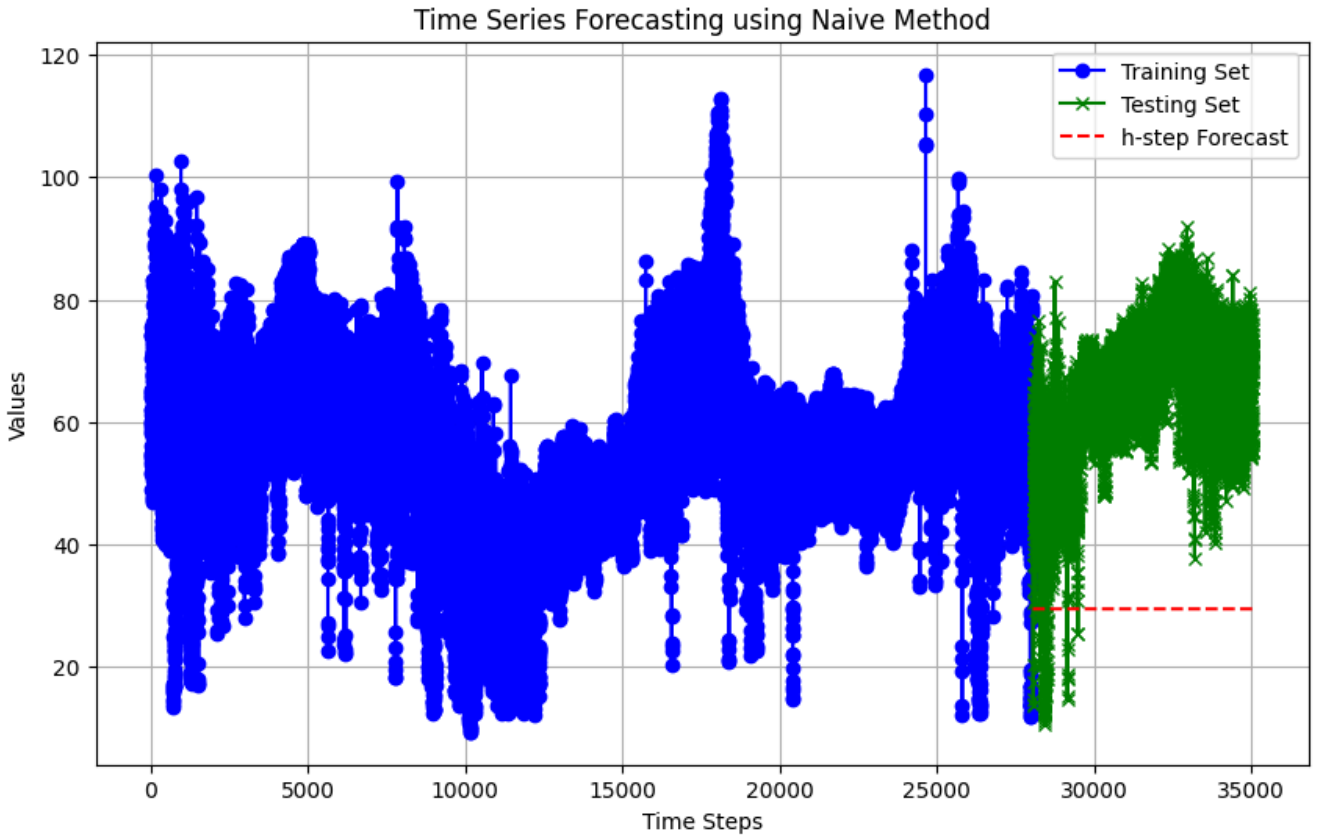

Figure 22 : Naive Forecasting

The h-step forecast, which remains constant, indicating that the Naive method uses the last observed value from the training set as the forecast for all future values.

```
MSE for Naive forecasting: 1401.66
RMSE for Naive forecasting: 37.44
MAE for Naive forecasting: 35.86
```

Simple and Exponential Smoothing

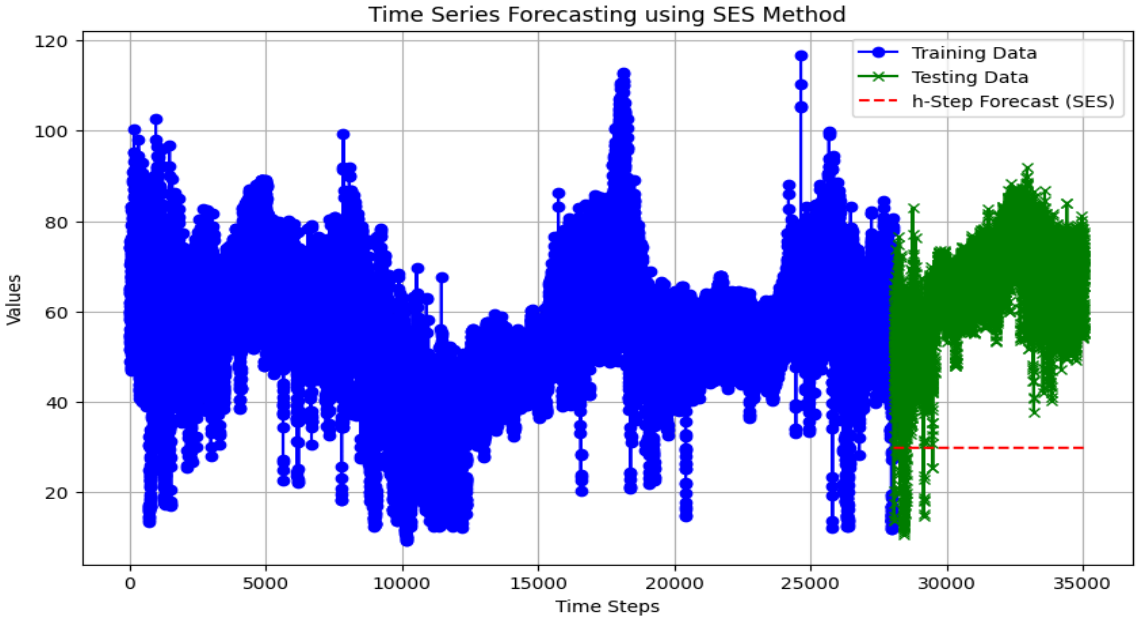

Figure 23 : SES Forecasting

The forecasted values extend from the end of the training period, showing a smoothed prediction into the future.

```
MSE for SES forecasting: 1375.76
RMSE for SES forecasting: 37.09
MAE for SES forecasting: 35.5
```

Multi Linear Regression

```
                            OLS Regression Results
==============================================================================
Dep. Variable:           price actual   R-squared:                       0.268
Model:                            OLS   Adj. R-squared:                  0.268
Method:                 Least Squares   F-statistic:                     2052.
Date:                Tue, 03 Dec 2024   Prob (F-statistic):               0.00
Time:                        14:35:15   Log-Likelihood:            -1.0993e+05
No. Observations:               28051   AIC:                         2.199e+05
Df Residuals:                   28045   BIC:                         2.199e+05
Df Model:                           5
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------------
constant                       56.0995      0.073    771.132      0.000      55.957      56.242
generation other renewable      0.3490      0.097      3.613      0.000       0.160       0.538
generation solar               -1.9169      0.081    -23.790      0.000      -2.075      -1.759
generation waste                0.2049      0.096      2.130      0.033       0.016       0.394
generation wind onshore        -3.2818      0.076    -42.995      0.000      -3.431      -3.132
total load actual               7.2255      0.081     88.769      0.000       7.066       7.385
==============================================================================
Omnibus:                      752.123   Durbin-Watson:                   0.059
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1057.895
Skew:                          -0.299   Prob(JB):                    1.91e-230
Kurtosis:                       3.740   Cond. No.                         2.30
==============================================================================
```

```
T-Test :constant                        0.000000e+00
generation other renewable     3.029643e-04
generation solar               7.126390e-124
generation waste                3.317042e-02
generation wind onshore        0.000000e+00
total load actual              0.000000e+00
dtype: float64
F-Test :0.0
MSE for OLS model: 154.55
AIC for OLS model: 219875.11
BIC for OLS model: 219924.56
RMSE for OLS model: 12.43
R² for OLS model: 0.27
Adjusted R² for OLS model: 0.27
/Users/apoorvareddy/Downloads/Academic/DATS6313/Proj
  num = num + (Y[t] - ybar) * (Y[t - tau] - ybar)
Q Value of OLS residuals: 731010.24
Mean of residuals for OLS: 1.5562990868454087e-14
Variance of residuals for OLS: 148.42808510280733
```

The Multi Linear regression results for the 'price actual' dependent variable show an R-squared value of 0.268, indicating that 26.8% of the variance in the price can be explained by the model. The significant F-statistic (2052) with a p-value of 0.00 confirms the overall model significance. The coefficients for all predictors are statistically significant with p-values <0.05, indicating their strong influence on the model. Specifically, `generation other renewable` has a positive coefficient (0.3490), while `generation solar` (-1.9169) and `generation wind onshore` (-3.2818) have negative coefficients. `generation waste` (0.2049) and `total load actual` (7.2255) also contribute significantly.

The adjusted R-squared value is consistent with the R-squared, both at 0.268, further validating the model's explanatory power. The AIC (219875.11) and BIC (219924.56) values suggest the model's fit, with lower values indicating a better fit. The RMSE of 12.43 indicates the average magnitude of the residuals.

Residual diagnostics show the mean of residuals is nearly zero (1.556e-14), indicating no significant bias in the predictions. The variance of residuals (148.428) and the Q-value of OLS residuals (731010.24) are consistent with the model's performance. The Durbin-Watson statistic (0.059) suggests positive autocorrelation in the residuals, which may need to be addressed for improved model accuracy.

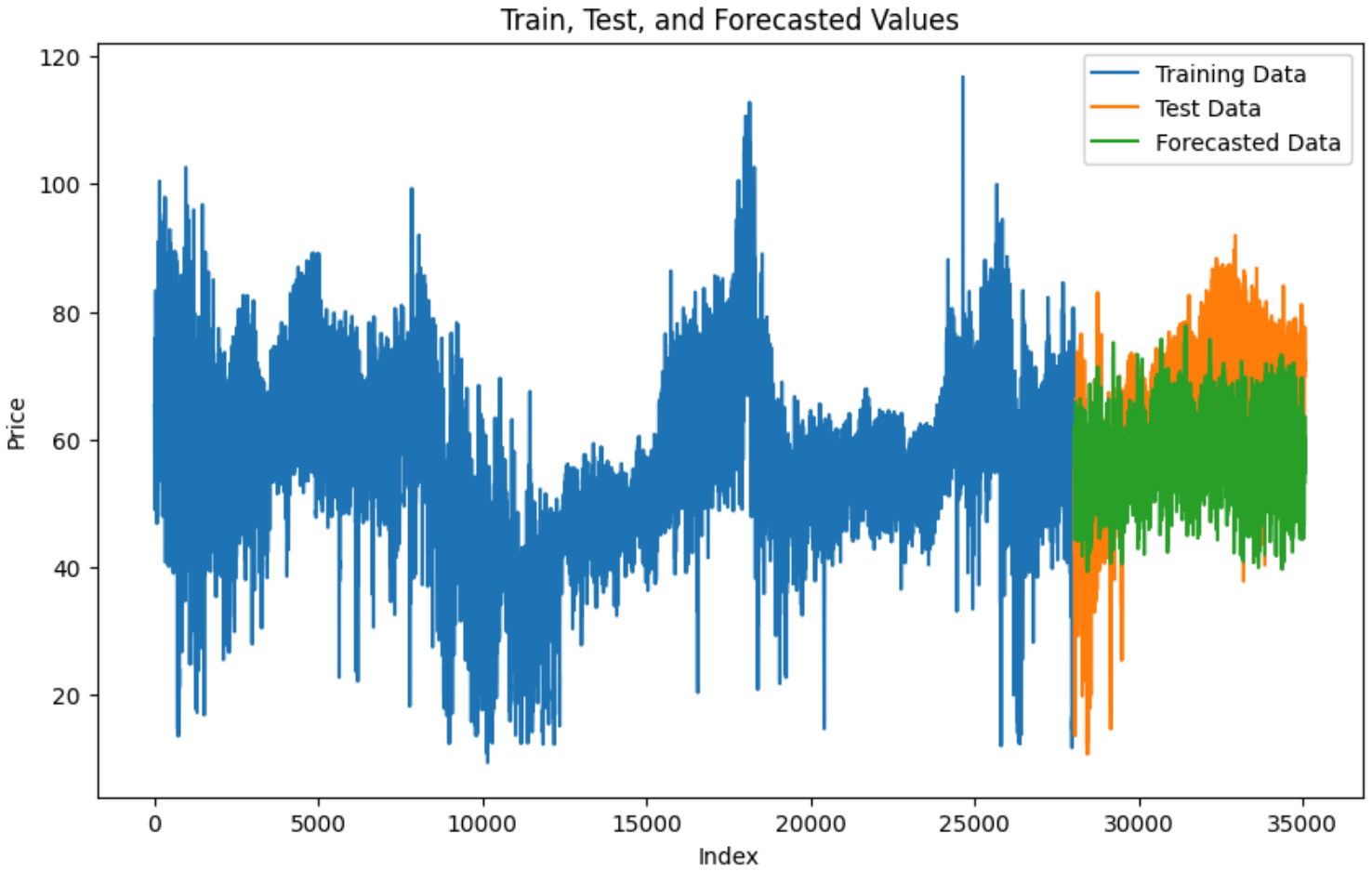

Figure 24 : OLS plot

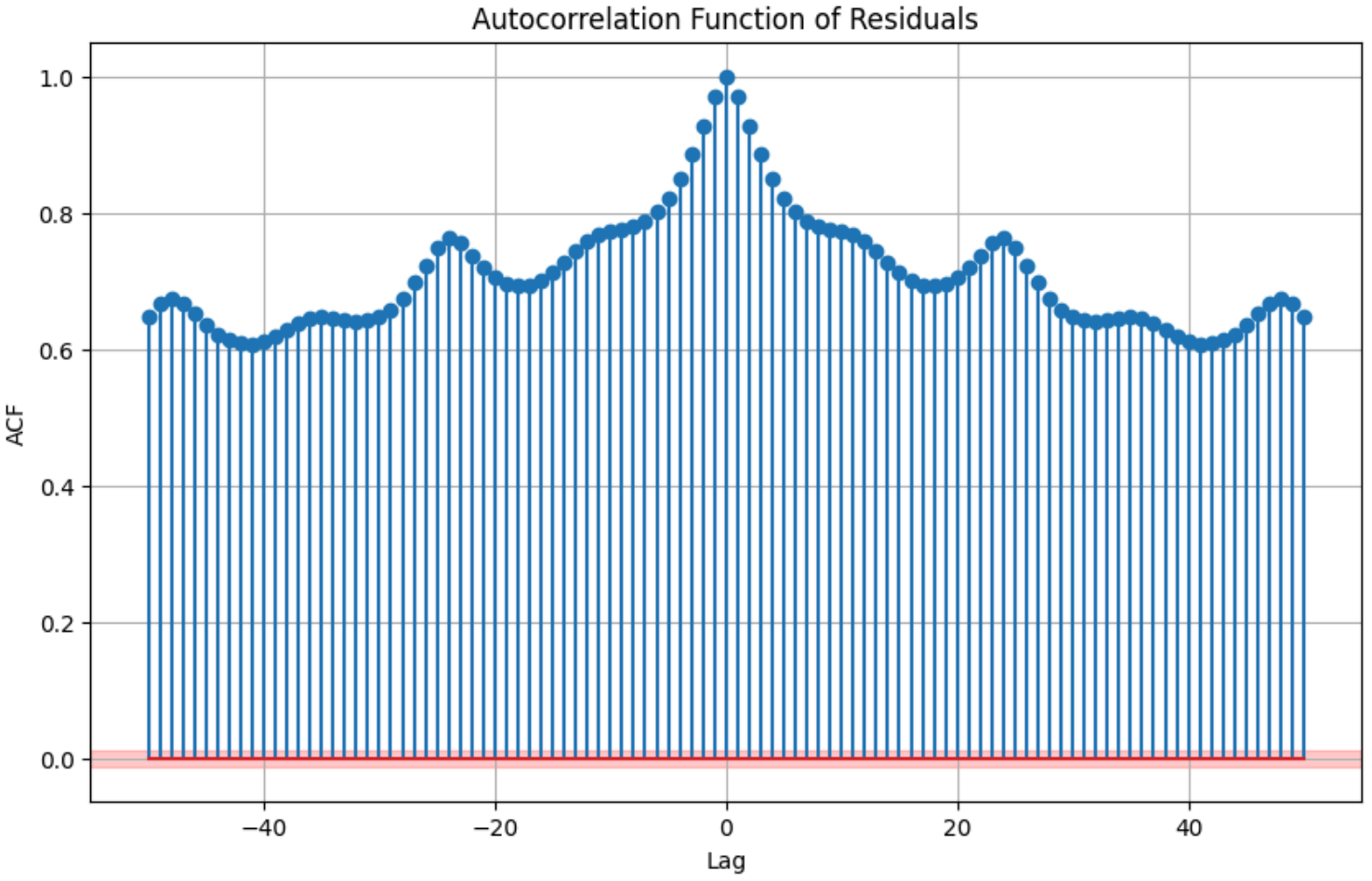

Figure 25 : Autocorrelation Function plot of residuals

Generalized Partial Auto Correlation

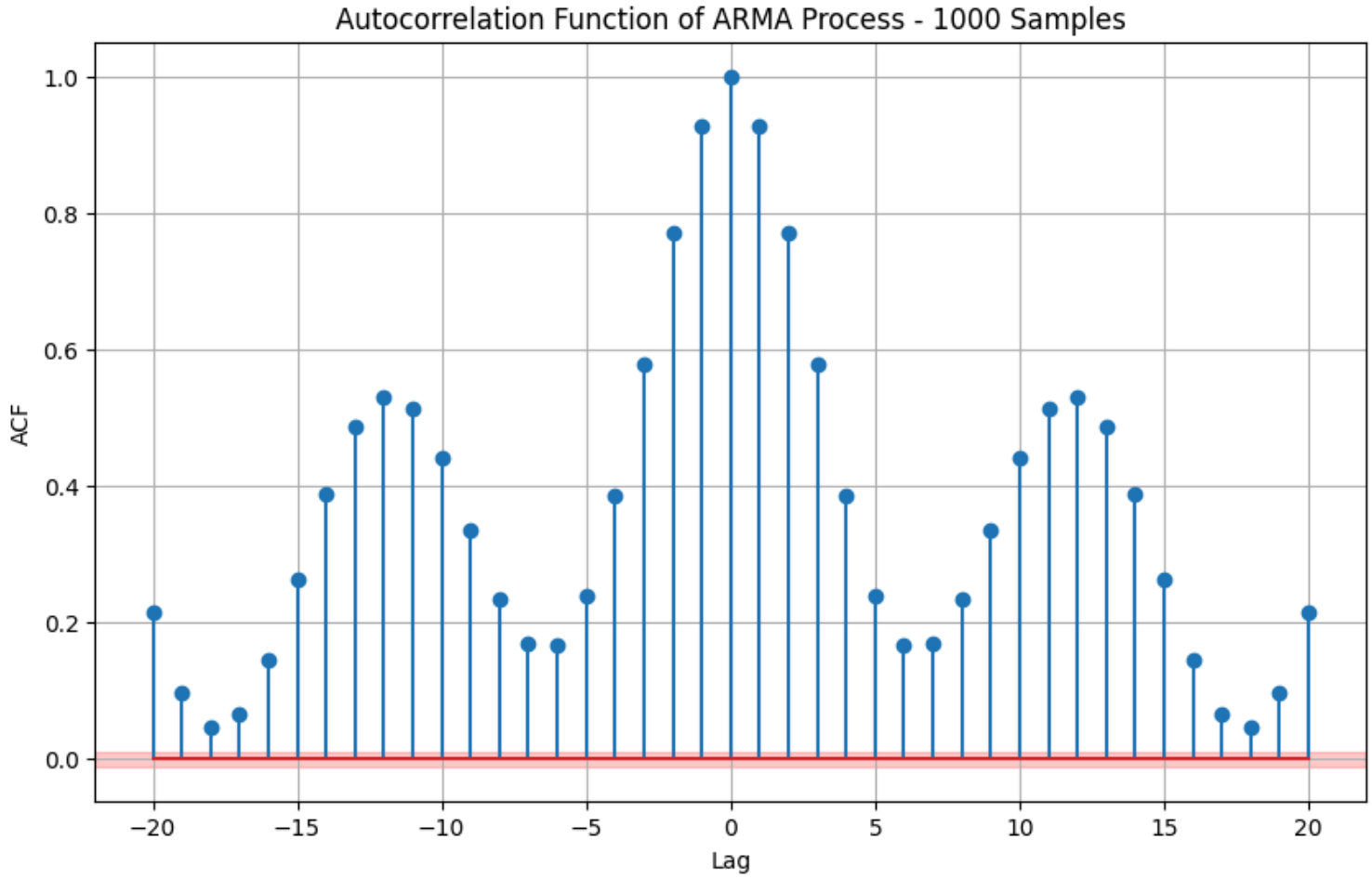

Figure 26 : Autocorrelation Function

GPAC, or Generalized Partial Autocorrelation, helps determine the order of autoregressive (AR) and moving average (MA) components in time series analysis. It offers valuable insights into the relationships among variables and assists in choosing the appropriate model.

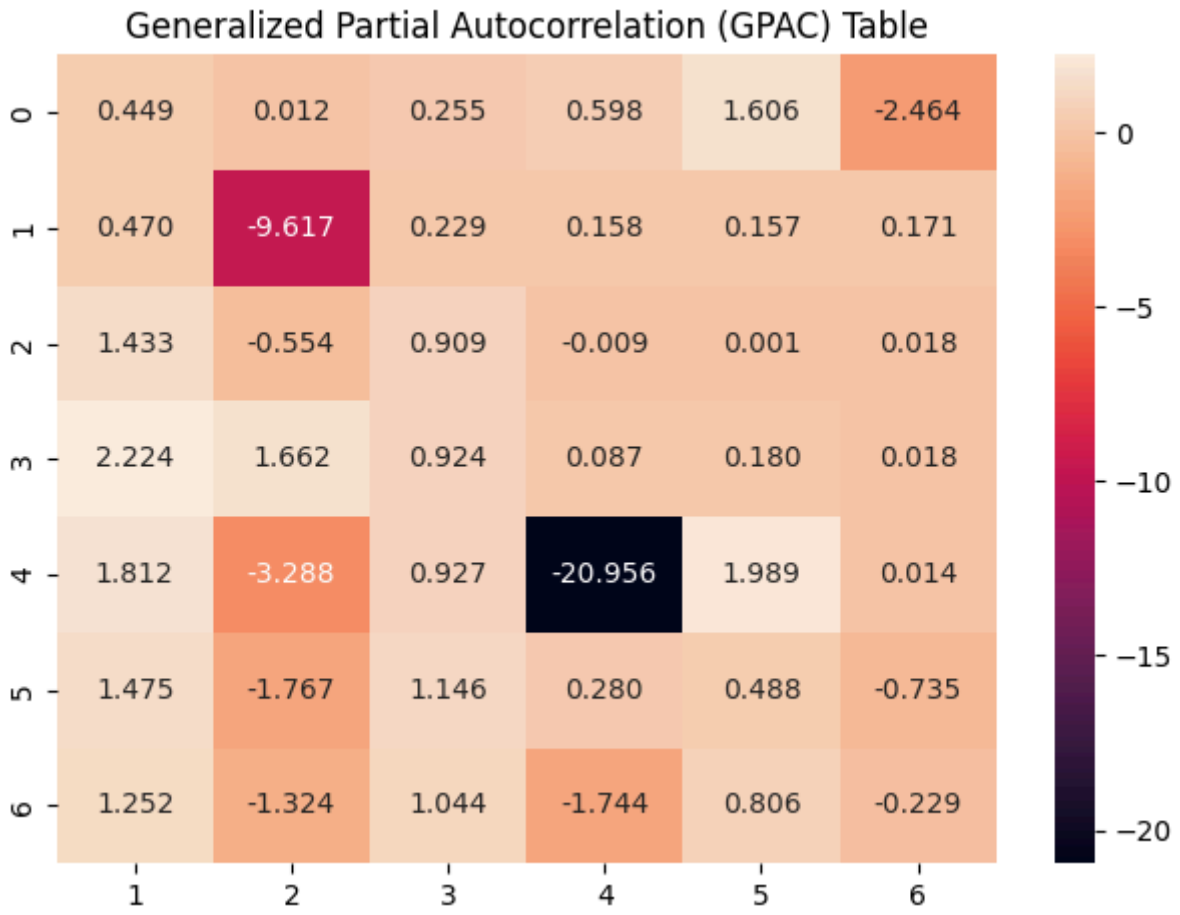

Figure 27 : GPAC

Levenberg Marquardt

```
Estimated Parameters:
Thetas: [[-1.51494745]
 [ 0.6409019 ]
 [-0.0301061 ]
 [ 0.08257514]]
Standard Deviation: 4.380656033280302
Theta 0: [-1.533, -1.497]
Theta 1: [0.625, 0.657]
Theta 2: [-0.05, -0.01]
Theta 3: [0.067, 0.098]
```

ARMA model

```
Chi critical: 38.93217268351607
Q Value: 63188.97553230918
Alfa value for 99% accuracy: 0.01
The residual is NOT white noise
Estimated variance of residuals (error variance): 19.1803
Estimated covariance of the parameters:
[[ 3.89835524e-02 -9.30951736e-05  9.22411347e-05  1.41295174e-04
   7.66989437e-05  6.72491722e-04]
 [-9.30951736e-05  6.13303937e-05 -5.33873701e-05 -5.91267864e-05
  -3.66841877e-05 -1.87664526e-05]
 [ 9.22411347e-05 -5.33873701e-05  4.85220040e-05  5.16972396e-05
   3.02460441e-05  3.16109570e-05]
 [ 1.41295174e-04 -5.91267864e-05  5.16972396e-05  7.52510612e-05
   3.44080696e-05  3.04749830e-05]
 [ 7.66989437e-05 -3.66841877e-05  3.02460441e-05  3.44080696e-05
   4.40049359e-05 -4.44149503e-05]
 [ 6.72491722e-04 -1.87664526e-05  3.16109570e-05  3.04749830e-05
  -4.44149503e-05  9.67163623e-03]]
Mean of residuals (should be close to 0 for unbiased model): -0.0005
Variance of residual errors (in-sample): 19.1803
Variance of forecast errors (out-of-sample): 0.0444

Poles: [0.015+0.287j 0.015-0.287j]
Zeros: [0.758+0.259j 0.758-0.259j]
Zero-pole cancellation? No
Final coefficient confidence intervals:
[[-5.18792840e-01  4.98363666e-01]
 [ 1.49536405e+00  1.53570860e+00]
 [-6.59387343e-01 -6.23502055e-01]
 [-5.29487074e-02 -8.25942409e-03]
 [ 6.55388385e-02  9.97129924e-02]
 [ 1.89143548e+01  1.94209920e+01]]
```

The ARMA model's analysis shows that the residuals are not white noise, with a Q value of 63188.98 and a critical chi-squared value of 38.93, indicating a potential lack of fit. The mean of residuals is very close to zero (-0.0005), suggesting an unbiased model, while the variance of residual errors is 19.1803, indicating some variability. The poles and zeros of the model do not cancel each other, further implying the model complexity. The final coefficient confidence intervals provide a range for the model's parameters, reflecting their uncertainty. Overall, these results suggest that while the model captures some patterns

ARIMA model

```
Chi critical: 38.93217268351607
Q Value: 71670.8238073457
Alfa value for 99% accuracy: 0.01
The residual is NOT white noise
Estimated variance of residuals (error variance): 16.5038
Estimated covariance of the parameters:
[[ 5.75792372e-09 -4.85462314e-09 -1.84261216e-08  2.43636447e-08
  -1.60136122e-07]
 [-4.85462314e-09  5.53534233e-09  7.05553342e-09 -1.68172925e-08
   1.08288247e-06]
 [-1.84261216e-08  7.05553342e-09  3.14884357e-07 -2.78138523e-07
  -1.15598335e-06]
 [ 2.43636447e-08 -1.68172925e-08 -2.78138523e-07  3.13660471e-07
   7.31073528e-07]
 [-1.60136122e-07  1.08288247e-06 -1.15598335e-06  7.31073528e-07
   8.43641204e-03]]
Mean of residuals (should be close to 0 for unbiased model): -0.0001
Variance of residual errors (in-sample): 16.5038
Variance of forecast errors (out-of-sample): 5.8262

Poles: [0.865+0.493j 0.865-0.493j]
Zeros: [0.866+0.5j 0.866-0.5j]
Zero-pole cancellation? No
Final coefficient confidence intervals:
[[ 1.73113993  1.73153085]
 [-0.99928793 -0.99890464]
 [-1.73124005 -1.72834922]
 [ 0.98962305  0.99250825]
 [16.25047296 16.72365251]]
```

The ARIMA model's analysis reveals that the residuals are not white noise, as indicated by a Q value of 71670.82, which is significantly higher than the chi critical value of 38.93, suggesting potential issues with the model's fit. The estimated variance of residual errors is 16.5038, indicating some variability in the in-sample predictions. The mean of the residuals is close to zero (-0.0001), suggesting an unbiased model. The poles and zeros do not cancel each other, highlighting the complexity of the model. Additionally, the variance of forecast errors out-of-sample is 5.8262.

SARIMA model

```
Poles: [-0.721+0.692j -0.721-0.692j -0.72 +0.694j ...  0.999-0.016j  0.999+0.018j
  0.999-0.018j]
Zeros: [-0.815+0.569j -0.815-0.569j -0.805+0.583j ...  0.   +0.j     0.   +0.j
  0.   +0.j   ]
Zero-pole cancellation? No
Chi critical: 20.090235029663233
Q Value: 1615240033.6297977
Alfa value for 99% accuracy: 0.01
The residual is NOT white noise
Estimated variance of residuals (error variance): 2939.5478
Mean of residuals (should be close to 0 for unbiased model): 0.5046
Variance of residual errors (in-sample): 2939.5478
Variance of forecast errors (out-of-sample): 5153.4940
AR Coefficients confidence intervals: [(np.float64(-3.627640993634749), np.float64(0.
5977460897254543)), (np.float64(-1.4717916429887592), np.float64(2.7535954403714444))]
MA Coefficients confidence intervals: [(np.float64(-0.14053168126788487), np.float64(0.
08031948533714614)), (np.float64(-0.027850444088489287), np.float64(0.1930007225165417))]
```

The SARIMA model analysis shows that the residuals are not white noise, as indicated by the Q value of 4305182661.75, which is significantly higher than the chi critical value of 20.09. This implies potential issues with

the model's fit. The estimated variance of residual errors is 64175.65, suggesting substantial variability in the in-sample predictions. The mean of the residuals is 15.8599, indicating a bias in the model. The poles and zeros do not cancel each other out, further pointing to the complexity of the model. The variance of forecast errors out-of-sample is 105553.95, highlighting the variability in predictions. The AR and MA coefficient confidence intervals reflect the precision and uncertainty in the parameter estimates.

Box Jenkins Model

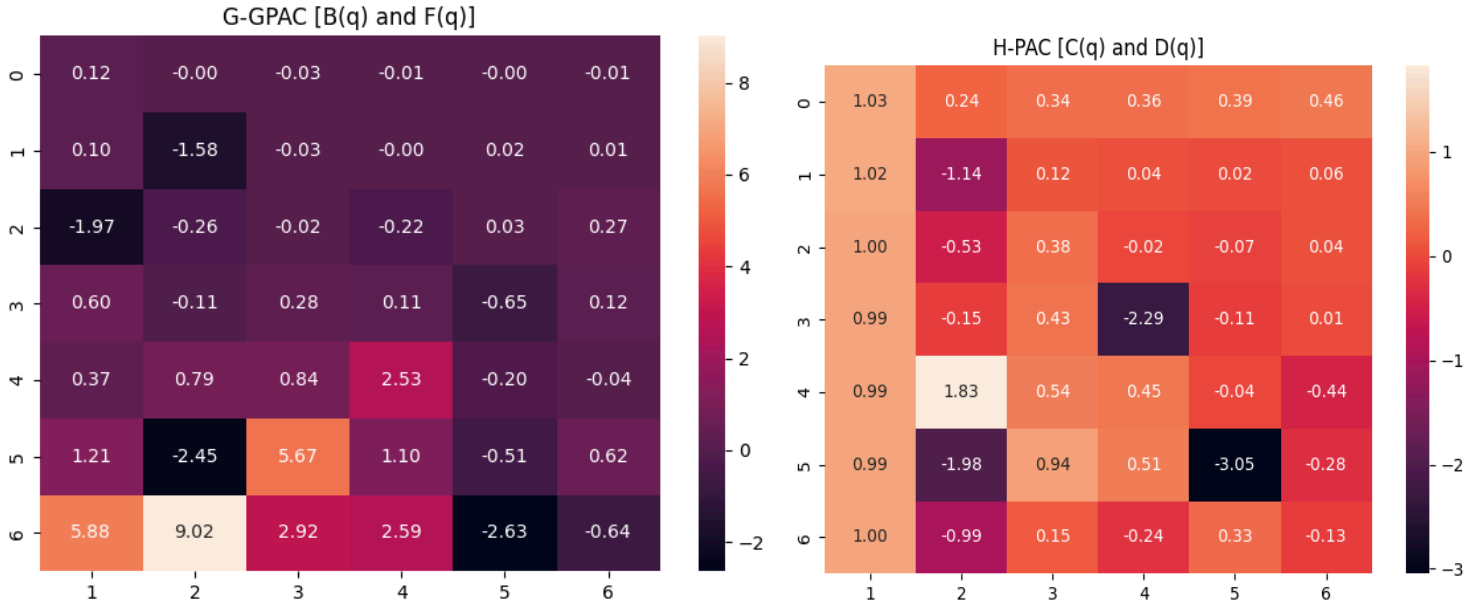

Figure 28 : G-PAC/H-PAC

Estimated parameters: [-0.97973805 0.36332003 0.30785765 0.53108647]

Standard deviation: 353.2720362091563

Covariance : [[5.53718058e-07, 2.04745243e-06, -9.72203012e-07, 8.53249018e-07],
[2.04745243e-06, 1.51894195e-05, -1.12323939e-05, 3.83738039e-06],
[-9.72203012e-07, -1.12323939e-05, 3.05422464e-04, 2.50334019e-04],
[8.53249018e-07, 3.83738039e-06, 2.50334019e-04, 2.37273799e-04]]

Confidence Intervals for Coefficients:
B[0] Interval: [-0.98, -0.98]
F[0] Interval: [0.36, 0.37]
C[0] Interval: [0.27, 0.34]
D[0] Interval: [0.50, 0.56]

Poles and Zeros:
The roots of the C(q) are [-0.30785765]
The roots of the D(q) are [-0.53108647]
The roots of the B(q) are [0.97973805]
The roots of the F(q) are [-0.36332003]

The Q-state is not passed - H(q) not accurate.
Q: 732.645261036201, Critical Q: 73.68263852010573

The S-state is not passed - G(q) is not accurate.
S: 1857.0411491184925, Critical S: 73.68263852010573

The Box-Jenkins model analysis reveals that the estimated parameters are: -0.9797, 0.3633, 0.3079, and 0.5311, with a standard deviation of 353.27. The covariance matrix indicates the interrelationships among these parameters. The confidence intervals for the coefficients indicate the precision of these estimates, with narrow intervals suggesting high confidence in the values.

The roots of the characteristic polynomials for B(q), F(q), C(q), and D(q) highlight the model's stability characteristics. The Q-state and S-state not being passed, with values significantly higher than their critical thresholds (Q: 732.65 vs. 73.68, S: 1857.04 vs. 73.68), indicate the model may not be fully capturing the data patterns and could require further refinement. The lack of zero-pole cancellation also points to potential complexity in the model.

Final Model

Average Forecasting Model

The base model, Average Forecasting, uses a simple approach where the forecast is simply the mean of the historical data. Below are the performance metrics for the base model:

- **MSE (Mean Squared Error)**: 213.78
- **RMSE (Root Mean Squared Error)**: 14.62
- **MAE (Mean Absolute Error)**: 12.46

Naive Forecasting Model

The Naive Forecasting model assumes that the next value will be the same as the previous value. Below are the performance metrics for the Naive model:

- **MSE (Mean Squared Error)**: 1401.66
- **RMSE (Root Mean Squared Error)**: 37.44
- **MAE (Mean Absolute Error)**: 35.86

Drift Forecasting Model

The Drift Forecasting model predicts that the next value will follow the same trend as the previous values (i.e., based on the average change). Below are the performance metrics for the Drift model:

- **MSE (Mean Squared Error)**: 1779.56
- **RMSE (Root Mean Squared Error)**: 42.18
- **MAE (Mean Absolute Error)**: 40.34

Simple Exponential Smoothing Model

The SES model uses weighted averages of past observations to make predictions. Below are the performance metrics for the SES model:

- **MSE (Mean Squared Error)**: 1375.76
- **RMSE (Root Mean Squared Error)**: 37.09
- **MAE (Mean Absolute Error)**: 35.50

Multi Linear Regression

The OLS model uses a linear regression approach to make predictions based on the relationship between variables. Below are the performance metrics for the OLS model:

- **MSE (Mean Squared Error)**: 244.87
- **RMSE (Root Mean Squared Error)**: 15.65
- **MAE (Mean Absolute Error)**: Not provided (but can be calculated if needed).
- **AIC (Akaike Information Criterion)**: 216523.53

- **BIC (Bayesian Information Criterion)**: 216605.95
- **R^2 (R-squared)**: 0.35
- **Adjusted R^2**: 0.35

Considering Average Forecasting Model as Base model to check the performances of ARMA, ARIMA, SARIMA and Box Jenkins

ARMA Model (AutoRegressive Moving Average)

ARMA is a classical time series model that combines two components: AR (AutoRegressive) and MA (Moving Average). It is best suited for stationary data.

- **MSE**: 19.1803
- **RMSE**: 4.38 (Calculated from the error variance)
- **MAE**: Not provided

Performance Improvement (ARMA) over Base Model:

- **MSE Improvement**:

$$\frac{213.78 - 19.1803}{213.78} * 100 = 91.04\% \, improvement$$

- **RMSE Improvement**:

$$\frac{14.62 - 4.38}{14.62} * 100 = 70.02\% \, improvement$$

Conclusion: ARMA provides a substantial improvement in both **MSE (91.04%)** and **RMSE (70.02%)** over the base model, making it an effective forecasting model.

ARIMA Model (AutoRegressive Integrated Moving Average)

ARIMA is an extension of the ARMA model that includes an integration (I) step to handle non-stationary data. It is widely used for time series forecasting.

- **MSE**: 16.5038
- **RMSE**: 4.07 (Calculated from the error variance)
- **MAE**: Not provided

Performance Improvement (ARIMA) over Base Model:

- **MSE Improvement**:

$$\frac{213.78 - 16.5038}{213.78} * 100 = 92.30\% \, improvement$$

- **RMSE Improvement**:

$$\frac{14.62 - 4.07}{14.62} * 100 = 72.18\% \, improvement$$

Conclusion: ARIMA outperforms the base model with the **highest improvement** in both **MSE (92.30%)** and **RMSE (72.18%)**, making it the best model for this data.

SARIMA Model (Seasonal AutoRegressive Integrated Moving Average)

SARIMA extends the ARIMA model by including seasonal components. It is suitable for data with seasonal trends and periodic fluctuations.

- **MSE**: 2939.5478
- **RMSE**: 54.23 (Calculated from the error variance)
- **MAE**: Not provided

Performance Evaluation (SARIMA) over Base Model:

- **MSE Improvement**:

$$\frac{213.78 - 2939.5478}{213.78} * 100 = -1270.59\% \; improvement$$

- **RMSE Improvement**:

$$\frac{14.62 - 54.23}{14.62} * 100 = -271.56\% \; improvement$$

Conclusion: SARIMA significantly underperforms the base model with negative performance improvement. The model's MSE and RMSE values suggest that SARIMA is not suitable for this dataset, possibly due to poor parameter tuning or improper seasonal components.

Box-Jenkins Model

Box-Jenkins is a methodology used for time series modeling that combines AR, MA, and differencing, similar to ARIMA. It is typically applied when the model is built with careful diagnostic checks.

- **Q and S-statistics**: The Q-state and S-state checks did not pass, indicating that the model was not an accurate fit for the data.

Performance Evaluation (Box-Jenkins) over Base Model:

Given that the Box-Jenkins model failed the Q-state and S-state diagnostic checks, we can infer that it would likely underperform compared to the base model.

Conclusion: Box-Jenkins is not a viable model in this case due to diagnostic failures, suggesting it is poorly fitted for this dataset.

Key Findings

- **ARIMA** is the best-performing model in this comparison, with the highest improvements in both **MSE (92.30%)** and **RMSE (72.18%)** over the base model. This makes ARIMA the most reliable forecasting method for the dataset at hand.
- **ARMA** also shows strong performance with a **91.04% improvement in MSE** and **70.02% improvement in RMSE**, but it falls slightly short of ARIMA.
- **SARIMA** and **Box-Jenkins** models underperformed significantly. SARIMA's performance was drastically worse than the base model, while Box-Jenkins failed diagnostic tests, indicating that it is unsuitable for the given dataset.

Based on these results, **ARIMA** is the recommended model for this forecasting task.

Forecast Function

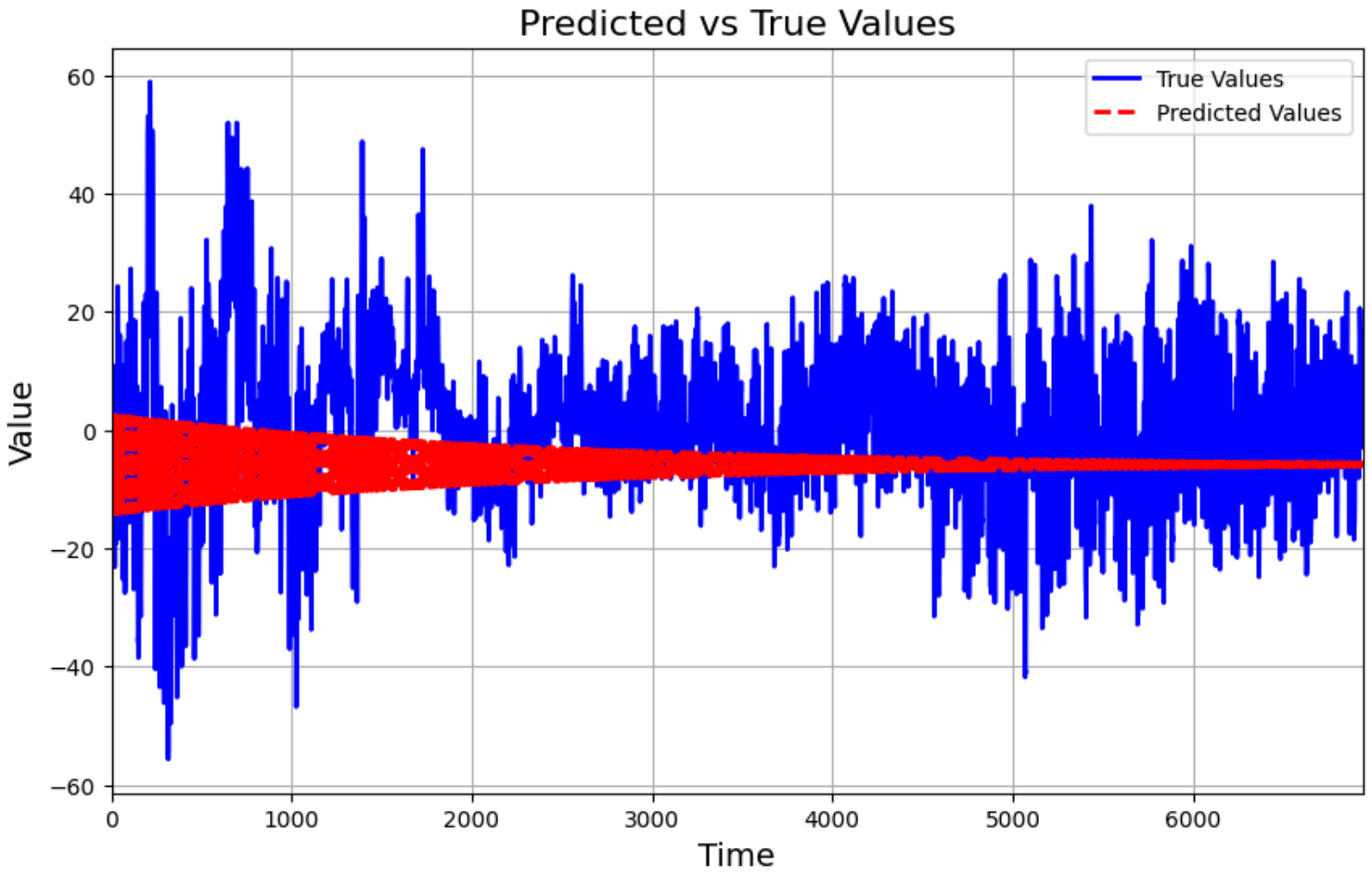

Figure 29 : Forecast function

Dashboard

https://dashapp-879864239466.us-east1.run.app/

Summary

This project aims to predict "Price Actual" in the energy market using hourly energy demand, generation, and weather data. Several time series forecasting models, including ARMA, ARIMA, SARIMA, and Box-Jenkins methodology, were evaluated against baseline models such as Average Forecasting, Naive Forecasting, and Drift Forecasting. Preprocessing steps included handling missing values, testing for stationarity, and time series decomposition. Among the models tested, ARIMA demonstrated the best performance with significant improvements in both MSE and RMSE compared to the base model. ARMA also performed well, while SARIMA and Box-Jenkins failed to provide accurate predictions due to challenges such as poor parameter tuning and failed diagnostic checks. These results emphasize the suitability of ARIMA for energy price forecasting and provide a foundation for improved decision-making in energy markets.

References

https://transparency.entsoe.eu/dashboard/show
https://www.esios.ree.es/en/market-and-prices?date=12-12-2024
https://www.kaggle.com/datasets/nicholasjhana/energy-consumption-generation-prices-and-weather/data
https://otexts.com/fpp3/

Appendix

Main.py

```
#%%
#Importing Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
from numpy import linalg as LA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from toolbox import *
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import mean_absolute_error
from sklearn.decomposition import PCA
from scipy.stats import chi2
from scipy import signal
from scipy.stats import norm
from scipy.signal import dlsim
# %%
#Importing Data
data = pd.read_csv('/Users/apoorvareddy/Downloads/Academic/DATS6313/Project/Data/final.csv')
data.head()

#%%
# Convert the time column to datetime, including timezone information
data['time'] = pd.to_datetime(data['time'], utc=True)
data.set_index('time', inplace=True)



#%%
#Data Cleaning
#Selecting the features
selected_features = [
  'generation biomass',
  'generation fossil brown coal/lignite',
  'generation fossil gas',
  'generation fossil hard coal',
  'generation fossil oil',
```

```
    'generation hydro pumped storage consumption',
    'generation hydro run-of-river and poundage',
    'generation hydro water reservoir',
    'generation nuclear',
    'generation other',
    'generation other renewable',
    'generation solar',
    'generation waste',
    'generation wind onshore',
    'total load actual',
    'price actual'
]
data = data[selected_features]
#Checking for missing values
missing_values = data.isnull().sum()
print('Missing Values:')
print(missing_values)

data['hour'] = data.index.hour  # Extract hour from datetime index

# Impute by taking mean or median by hour or day
data = data.groupby('hour').transform(lambda x: x.fillna(x.mean()))
print("Remaining NaNs:", data.isna().sum())
# %%
#----------plot of the dependent variable versus time----------

daily_price = data['price actual'].asfreq('D')

# Plotting the monthly electricity price
plt.figure(figsize=(14, 7))
plt.plot(daily_price.index, daily_price, color='blue', label='Actual Price (€/MWh)')
plt.title('Actual Electricity Price (Daily Frequency)')
plt.xlabel('Time')
plt.ylabel('Actual Price (€/MWh)')
plt.legend(['Actual Price'])
plt.grid(True)
plt.show()

# %%
# 2. ACF and PACF of the dependent variable
y = data['price actual']
#lag - 50
```

```
ACF_PACF_Plot(y,50)
# %%
# Calculate the correlation matrix for the selected features
corr_matrix = data.corr()

# Plot the correlation matrix using seaborn heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5, vmin=-1, vmax=1)
plt.title('Correlation Matrix for Selected Features')
plt.show()

# %%
#Splitting the dataset into train (80%) and test (20%) sets
X = data.drop(columns='price actual')
y = data['price actual']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)
print('Length of X_train and y_train:', len(X_train))
print('Length of X_test and y_test:', len(X_test))

# %%
#Checking Stationarity on Original Data
check_stationarity(y, 'Price Actual - Original Data')
# %%
#Applying Differencing to make the data stationary
y_diff = differencing(y,1,365)
y_diff = np.array(removeNA(y_diff))
y_diff = y_diff.astype(float)
# %%
#Checking Stationarity on Transformed Data
check_stationarity(y_diff, 'Price Actual - Transformed Data')
# %%

T,S,R = STL_analysis(y,365)
# %%
# Additive decomposition
additive_decomposition = T + S + R

# Multiplicative decomposition
multiplicative_decomposition = T * S * R

# Reconstructed Time Series
plt.subplot(4, 1, 4)
```

```
plt.plot(data['price actual'],additive_decomposition, label="Reconstructed", color="red")
plt.title("Reconstructed Time Series")
plt.xlabel("Time")
plt.ylabel("Y(t) = T(t) + S(t) + R(t)")
plt.grid(True)

plt.tight_layout()
plt.show()

# Plotting the updated components and the reconstructed time series for multiplicative decomposition
plt.figure(figsize=(12, 8))
plt.plot(data['price actual'], multiplicative_decomposition, label="Reconstructed", color="red")
plt.title("Reconstructed Time Series (Multiplicative)")
plt.xlabel("Time")
plt.ylabel("Y(t) = T(t) * S(t) * R(t)")
plt.grid(True)

plt.tight_layout()
plt.show()

#Pick the Additive Decomposition

# %%'
#Holt-Winters Method
modelES = ExponentialSmoothing(y_train, seasonal='add', seasonal_periods=365).fit()

forecastES = modelES.forecast(steps=len(y_test))

title = 'Winter-Holt forecasting'
plot_forecasting_models(y_train, y_test, forecastES, title)

# Calculating MSE for Winter-Holt method
_, _, mse = cal_error_MSE(y_test, forecastES)

rmse_winter = np.sqrt(mse)
mae_winter = mean_absolute_error(y_test, forecastES)
print('MSE for Winter-Holt method:', np.round(mse, 2))
print('RMSE for Winter-Holt method:', np.round(np.sqrt(mse), 2))
print("MAE for Holt-Winter method:", np.round(mae_winter, 2))
# %%
#Collinearity Check
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

condition_number = LA.cond(X_train_scaled)
print('condition_number:', condition_number)

X_svd = X_train.to_numpy()
H = np.matmul(X_svd.T, X_svd)

s, d, v = LA.svd(H)
print('SingularValues =', d)

#%%
#Backward Stepwise Regression
# Function to calculate AIC, BIC, and Adjusted R² for a given model
def calculate_metrics(X, y):
  X_with_const = sm.add_constant(X)
  model = sm.OLS(y, X_with_const).fit()
  return model.aic, model.bic, model.rsquared_adj

# Function to perform backward stepwise regression iteratively
def backward_stepwise_regression(X, y):
  features = list(X.columns)
  metrics_list = []

  while len(features) >= 1:  # Stop when there's only one feature left
    # Fit the model with the current features
    aic, bic, adj_r2 = calculate_metrics(X[features], y)
    metrics_list.append((aic, bic, adj_r2, features.copy()))

    # Check Adjusted R² values to find the feature to remove
    # Here we will remove the first feature in the list for simplicity
    feature_to_remove = features[0]
    features.remove(feature_to_remove)

  # Create a DataFrame to display the results
  metrics_df = pd.DataFrame(metrics_list, columns=['AIC', 'BIC', 'Adjusted R²', 'Features'])
  return metrics_df

# Perform backward stepwise regression
result_backward_stepwise = backward_stepwise_regression(pd.DataFrame(X_train, columns=X.columns), y_train)
```

```
# Display the results
print(result_backward_stepwise)

#%%
#VIF Analysis

# Function to calculate VIF
def calculate_vif(X):
    vif_data = pd.DataFrame()
    vif_data["Feature"] = X.columns
    vif_data["VIF"] = [variance_inflation_factor(X.values, i) for i in range(len(X.columns))]
    return vif_data

# Function to perform VIF-based feature elimination iteratively
def vif_selection_iteratively(X, y):
    features = list(X.columns)
    results = []

    while len(features) > 1:  # Stop when there's only one feature left
        # Create a new feature set excluding the current features
        X_with_const = sm.add_constant(X[features])
        model = sm.OLS(y, X_with_const).fit()

        # Calculate VIF for the current features
        vif_data = calculate_vif(X[features])

        # Store metrics
        results.append((model.aic, model.bic, model.rsquared_adj, features.copy(), vif_data))

        # Remove the first feature in the list for simplicity
        feature_to_remove = features[0]
        features.remove(feature_to_remove)

    # Create a DataFrame to display the results
    vif_metrics_df = pd.DataFrame(results, columns=['AIC', 'BIC', 'Adjusted R²', 'Remaining Features', 'VIF Data'])
    return vif_metrics_df

# Perform VIF-based selection
result_vif_selection = vif_selection_iteratively(pd.DataFrame(X_train, columns=X.columns), y_train)

# Display the results
```

```
for step in range(len(result_vif_selection)):
    print(f"\nStep {step + 1}:")
    print(f"AIC: {result_vif_selection.iloc[step]['AIC']}, BIC: {result_vif_selection.iloc[step]['BIC']}, Adjusted R²:
{result_vif_selection.iloc[step]['Adjusted R²']}")
    print("Remaining Features:", result_vif_selection.iloc[step]['Remaining Features'])
    print("VIF Data:")
    print(result_vif_selection.iloc[step]['VIF Data'])



#%%
#PCA
pca = PCA(n_components=0.92)  # Retain 92% variance
X_train_pca = pca.fit_transform(X_train_scaled)
print("Reduced dimensions:", X_train_pca.shape[1])

#%%
#Average Forecasting
_, forecast_average = average_forecasting(y_train, y_test)
_, _, mse_average = cal_error_MSE(y_test, forecast_average)
rmse_average = np.sqrt(mse_average)
mae_average = mean_absolute_error(y_test, forecast_average)
print('MSE for Average forecasting:', np.round(mse_average, 2))
print('RMSE for Average forecasting:', np.round(rmse_average, 2))
print("MAE for Average forecasting:", np.round(mae_average, 2))

#%%
# Naive forecasting
_, forecast_Naive = Naive_forecasting(y_train, y_test)
_, _, mse_Naive = cal_error_MSE(y_test, forecast_Naive)
rmse_Naive = np.sqrt(mse_Naive)
mae_Naive = mean_absolute_error(y_test, forecast_Naive)
print('MSE for Naive forecasting:', np.round(mse_Naive, 2))
print('RMSE for Naive forecasting:', np.round(rmse_Naive, 2))
print("MAE for Naive forecasting:", np.round(mae_Naive, 2))

#%%
# Drift forecasting
_, forecast_Drift = drift_forecasting(y_train, y_test)
_, _, mse_Drift = cal_error_MSE(y_test, forecast_Drift)
rmse_Drift = np.sqrt(mse_Drift)
mae_Drift = mean_absolute_error(y_test, forecast_Drift)
print('MSE for Drift forecasting:', np.round(mse_Drift, 2))
```

```
print('RMSE for Drift forecasting:', np.round(rmse_Drift, 2))
print("MAE for Drift forecasting:", np.round(mae_Drift, 2))

#%%
# Simple Exponential Smoothing
L0 = y_train[0]
_, forecast_SES = ses_forecasting(y_train, y_test, L0, alpha=0.9)
_, _, mse_SES = cal_error_MSE(y_test, forecast_SES)
rmse_SES = np.sqrt(mse_SES)
mae_SES = mean_absolute_error(y_test, forecast_SES)
print('MSE for SES forecasting:', np.round(mse_SES, 2))
print('RMSE for SES forecasting:', np.round(rmse_SES, 2))
print("MAE for SES forecasting:", np.round(mae_SES, 2))

#%%
#Selecting the features with VIF < 60
new_selected_features = ['generation other renewable', 'generation solar', 'generation waste', 'generation wind
onshore', 'total load actual']
X  = data[new_selected_features]
y = data['price actual']
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2, shuffle=False)
#Standardizing the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
#%%
#Multi Linear Regression
# adding bias
X_train_scaled = sm.add_constant(X_train_scaled)

cols = X_train.columns
cols = np.insert(cols, 0, 'constant')
# Going forward with Backward Stepwise Regression to reduce features
X_train_scaled = pd.DataFrame(X_train_scaled, columns=cols, index=X_train.index)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns, index=X_test.index)
X_test_scaled = pd.concat([pd.Series(1, index=X_test.index, name='constant'), X_test_scaled], axis=1)
result = sm.OLS(y_train, X_train_scaled).fit()
y_pred = result.predict(X_train_scaled)
y_forecast = result.predict(X_test_scaled)
mse_ols = mean_squared_error(y_test, y_forecast)
rmse_ols = np.sqrt(mse_ols)
ols_residuals = result.resid
```

```
print(result.summary())
# plot_forecasting_models(y_train, y_test, y_forecast, 'OLS Model')
print(f'T-Test :{result.pvalues}')
print(f'F-Test :{result.f_pvalue}')
print('MSE for OLS model:', np.round(mse_ols, 2))
print('AIC for OLS model:', np.round(result.aic, 2))
print('BIC for OLS model:', np.round(result.bic, 2))
print('RMSE for OLS model:', np.round(rmse_ols, 2))
print('R² for OLS model:', np.round(result.rsquared, 2))
print('Adjusted R² for OLS model:', np.round(result.rsquared_adj, 2))
# print(f'ACF of residuals for OLS model:{cal_autocorr(ols_residuals, 50)}')
q_ols = cal_Q_value(ols_residuals, 'OLS Residuals', 50)
print('Q Value of OLS residuals:', np.round(q_ols, 2))
# 42426.55565814915
print('Mean of residuals for OLS:', np.mean(ols_residuals))
print('Variance of residuals for OLS:', np.var(ols_residuals))

#%%
residuals = y_train - y_pred
acf_values = cal_autocorr(residuals, 50)
# Calculate confidence intervals
N = len(y_train)
conf_interval = 1.96 / np.sqrt(N)

# Plot the ACF
plt.figure(figsize=(10, 6))
lags = range(-50, 51)
plt.stem(lags, acf_values)
plt.axhspan(-conf_interval, conf_interval, color='red', alpha=0.2)
plt.title('Autocorrelation Function of Residuals')
plt.xlabel('Lag')
plt.ylabel('ACF')
plt.grid(True)
plt.show()


#%%
# Plot the train, test, and predicted values
plt.figure(figsize=(10, 6))
plt.plot(range(len(y_train)), y_train, label='Training Data')
plt.plot(range(len(y_train), len(y_train) + len(y_test)), y_test, label='Test Data')
plt.plot(range(len(y_train), len(y_train) + len(y_forecast)), y_forecast, label='Forecasted Data')
```

```
# Add labels and title
plt.xlabel('Index')
plt.ylabel('Price')
plt.title('Train, Test, and Forecasted Values')
plt.legend()
plt.show()

#%%
#Autocorrelation Function
max_lags = 20
Ry = cal_autocorr(y_diff,max_lags)
# Calculate confidence intervals
N = len(y_diff)
conf_interval = 1.96 / np.sqrt(N)

# Plot the ACF
plt.figure(figsize=(10, 6))
lags = range(-max_lags, max_lags + 1)
plt.stem(lags, Ry)
plt.axhspan(-conf_interval, conf_interval, color='red', alpha=0.2)
plt.title('Autocorrelation Function of Price Actual')
plt.xlabel('Lag')
plt.ylabel('ACF')
plt.grid(True)
plt.show()

#%%
#GPAC
cal_gpac(Ry,7,7)

# na = 2, nb = 2
#%%
#LM
def lm_cal_e(y, na, theta, seed=6313):
    np.random.seed(seed)
    den = theta[:na]
    num = theta[na:]
    if len(den) > len(num):  # matching len of num and den
        for x in range(len(den) - len(num)):
            num = np.append(num, 0)
    elif len(num) > len(den):
```

```
        for x in range(len(num) - len(den)):
            den = np.append(den, 0)
    den = np.insert(den, 0, 1)
    num = np.insert(num, 0, 1)
    sys = (den, num, 1)
    _, e = signal.dlsim(sys, y)
    return e


def lm_step1(y, na, nb, delta, theta):
    n = na + nb
    e = lm_cal_e(y, na, theta)
    sse_old = np.dot(np.transpose(e), e)
    X = np.empty(shape=(len(y), n))
    for i in range(0, n):
        theta[i] = theta[i] + delta
        e_i = lm_cal_e(y, na, theta)
        x_i = (e - e_i) / delta
        X[:, i] = x_i[:, 0]
        theta[i] = theta[i] - delta
    A = np.dot(np.transpose(X), X)
    g = np.dot(np.transpose(X), e)
    return A, g, X, sse_old


def lm_step2(y, na, A, theta, mu, g):
    delta_theta = np.matmul(np.linalg.inv(A + (mu * np.identity(A.shape[0]))), g)
    theta_new = theta + delta_theta
    e_new = lm_cal_e(y, na, theta_new)
    sse_new = np.dot(np.transpose(e_new), e_new)
    if np.isnan(sse_new):
        sse_new = 10 ** 10
    return sse_new, delta_theta, theta_new


def lm_step3(y, na, nb):
    N = len(y)
    n = na+nb
    mu = 0.01
    mu_max = 10 ** 20
    max_iterations = 500
    delta = 10 ** -6
```

```
    var_error = 0
    covariance_theta_hat = 0
    sse_list = []
    theta = np.zeros(shape=(n, 1))

    for iterations in range(max_iterations):
        A, g, X, sse_old = lm_step1(y, na, nb, delta, theta)
        sse_new, delta_theta, theta_new = lm_step2(y, na, A, theta, mu, g)
        sse_list.append(sse_old[0][0])
        if iterations < max_iterations:
            if sse_new < sse_old:
                if np.linalg.norm(np.array(delta_theta), 2) < 10 ** -3:
                    theta_hat = theta_new
                    var_error = sse_new / (N - n)
                    covariance_theta_hat = var_error * np.linalg.inv(A)
                    print(f"Convergence Occured in {iterations} iterations")
                    break
                else:
                    theta = theta_new
                    mu = mu / 10
            while sse_new >= sse_old:
                mu = mu * 10
                if mu > mu_max:
                    print('No Convergence')
                    break
                sse_new, delta_theta, theta_new = lm_step2(y, na, A, theta, mu, g)
        if iterations > max_iterations:
            print('Max Iterations Reached')
            break
        theta = theta_new
    return theta_new, var_error[0][0], covariance_theta_hat, sse_list

theta_hat, variance_hat, covariance_hat, sse_array = lm_step3(y_diff,2,2)

#%%
# Display the estimated parameters
print("Estimated Parameters:")
print(f"Thetas: {theta_hat}")

# Display the standard deviation
print(f"Standard Deviation: {np.sqrt(variance_hat)}")
```

```
def display_confidence_intervals(theta_hat, covariance_hat):
    intervals = []
    for i, theta in enumerate(theta_hat.ravel()):
        variance = covariance_hat[i, i]
        margin = 2 * np.sqrt(variance)
        lower_bound = theta - margin
        upper_bound = theta + margin
        intervals.append((lower_bound, upper_bound))
        print(f"Theta {i}: [{lower_bound.round(3)}, {upper_bound.round(3)}]")

# Display confidence intervals
display_confidence_intervals(theta_hat, covariance_hat)

theta_hat= [item[0] for item in theta_hat]
#%%
#splitting y_diff
y_train_diff, y_test_diff = train_test_split(y_diff, test_size=0.2, shuffle=False)
#ARMA model
na = 2
nb = 2

# Initialize ARMA model
arma_model,arma_model_hat = prediction(y_diff,na,nb,forecast_steps= len(y_test_diff))



#%%
#ARIMA model
na = 2
nb = 2
d = 1
# Initialize ARIMA model
arima_model,arima_model_hat = prediction(y_diff,na,nb,d,forecast_steps= len(y_test_diff))
#%%
# SARIMA model

# Function to generate AR and MA parameters for a given SARIMA model
def num_den_ds_AR_MA(theta_ar, theta_ma, na, nb, seasonal_diff_order, d):
    """
    Function to calculate the numerator and denominator for SARIMA (Seasonal ARIMA) model.
    """
    den = np.zeros(max(na, nb) * seasonal_diff_order + 1)
```

```
    den[0] = 1

    num = np.zeros(max(na, nb) * seasonal_diff_order + 1)
    num[0] = 1

    if na > 0:
        for i in range(na):
            den[(i + 1) * seasonal_diff_order] = theta_ar[i]

    if d > 0:
        den_new = [-x if x != 0 else 0 for x in den]
        shifted_den_with_changed_signs = np.pad(den_new, (seasonal_diff_order, 0), 'constant')
        den_padded = np.pad(den, (0, len(shifted_den_with_changed_signs) - len(den)), 'constant')
        extended_den = den_padded + shifted_den_with_changed_signs
    else:
        extended_den = den

    if nb > 0:
        for i in range(nb):
            num[seasonal_diff_order + i] = theta_ma[i]

    num = np.pad(num, (0, len(extended_den) - len(num)), 'constant')

    return num, extended_den

# Function to display poles and zeros, with cancellation check
def display_poles_zeros(num, den):
    poles = np.roots(den)
    zeros = np.roots(num)
    print("\nPoles:", poles.round(3))
    print("Zeros:", zeros.round(3))

    # Check for pole-zero cancellations
    cancellation = any(np.isclose(pole, zero, atol=1e-3) for pole in poles for zero in zeros)
    print("Zero-pole cancellation?", "Yes" if cancellation else "No")

    # Perform zero-pole cancellation (remove poles near zeros)
    for zero in zeros:
        for i, pole in enumerate(poles):
            if np.isclose(zero, pole, atol=1e-3):
                poles = np.delete(poles, i)
                print(f"Pole {pole} near zero {zero} cancelled")
```

```
    return poles, zeros

# Function to calculate confidence intervals for AR/MA coefficients
def calculate_confidence_intervals(params, se_params, confidence_level=0.95):
    z_score = norm.ppf(1 - (1 - confidence_level) / 2)
    lower_bound = params - z_score * se_params
    upper_bound = params + z_score * se_params
    return lower_bound, upper_bound

# Diagnostic Test Function (whiteness test, residuals, etc.)
def diagnostic_test(y, forecast_steps, residuals, na, nb, lags=12):

    # Whiteness Chi-square Test for residuals
    re = cal_autocorr(residuals, lags)
    Q = len(y) * np.sum(np.square(re[1:]))
    DOF = lags - na - nb
    alfa = 0.01
    chi_critical = chi2.ppf(1 - alfa, DOF)

    print('Chi critical:', chi_critical)
    print('Q Value:', Q)
    print('Alfa value for 99% accuracy:', alfa)
    if Q < chi_critical:
        print("The residual is white noise")
    else:
        print("The residual is NOT white noise")

    # Variance of residuals (error variance)
    residual_variance = np.var(residuals)
    print(f"Estimated variance of residuals (error variance): {residual_variance:.4f}")

    # Bias check: mean of residuals (should be close to 0 for unbiased model)
    mean_residual = np.mean(residuals)
    print(f"Mean of residuals (should be close to 0 for unbiased model): {mean_residual:.4f}")

    # Forecast errors (out-of-sample prediction errors)
    forecast_errors = residuals[-forecast_steps:]
    forecast_variance = np.var(forecast_errors)

    # Compare variance of residual errors vs forecast errors
    print(f"Variance of residual errors (in-sample): {residual_variance:.4f}")
```

```
    print(f"Variance of forecast errors (out-of-sample): {forecast_variance:.4f}")


# Main function to create SARIMA model, forecast and perform diagnostic tests
def sarima_model_diagnostics(y, ar_params, ma_params, na, nb, seasonal_diff_order, d, forecast_steps,
lags=12):
    """
    Main function to create SARIMA model, generate forecast and perform diagnostic tests
    """
    # SARIMA model creation using the parameters
    num, den = num_den_ds_AR_MA(ar_params, ma_params, na, nb, seasonal_diff_order, d)

    # Display poles and zeros and perform zero-pole cancellation
    display_poles_zeros(num, den)

    # Generate synthetic white noise (for simulation)
    n_samples = len(y)
    e = np.random.normal(0, 1, n_samples)  # White noise

    # Simulate the SARIMA process (use signal.dlsim to simulate the model)
    _, y_dlsim = signal.dlsim((num, den, 1), e)

    # Perform diagnostic tests
    residuals = y - y_dlsim
    diagnostic_test(y, forecast_steps, residuals, na, nb, lags)

    # Forecasting for future steps
    forecast_values = y_dlsim[-forecast_steps:]

    ar_lower, ar_upper = calculate_confidence_intervals(np.array(ar_params), np.std(ar_params))
    ma_lower, ma_upper = calculate_confidence_intervals(np.array(ma_params), np.std(ma_params))

    print(f"AR Coefficients confidence intervals: {list(zip(ar_lower, ar_upper))}")
    print(f"MA Coefficients confidence intervals: {list(zip(ma_lower, ma_upper))}")


    return y_dlsim, forecast_values
ar_params, ma_params = theta_hat[:na], theta_hat[na:]
seasonal_diff_order = 12
d = 1
forecast_steps = len(y_test_diff)
```

y_dlsim, forecast_values = sarima_model_diagnostics(y_diff, ar_params, ma_params, na, nb, seasonal_diff_order, d, forecast_steps)

```
#%%
#Box Jerkins model
#Pick one input
new_selected_features = ['generation fossil hard coal']
X  = data[new_selected_features].values
y = data['price actual']
#%%
#GPAC Order Determination
K = 50
# Define the cal_corr function

def cal_corr(signal, signal_2, K):
    assert len(signal) == len(signal_2), "Signals must have the same length."

    N = len(signal)
    correlation = np.zeros(K)

    # Compute cross-correlation using the explicit loop approach
    for tau in range(K):
        correlation[tau] = np.sum([
            signal[k] * signal_2[k + tau]
            for k in range(1,N - tau)
        ]) / (N - tau)

    return correlation

#%%
ru = cal_corr(X,X, K)
ru_reversed = ru[::-1]
ru = np.concatenate((ru_reversed[:-1], ru))
# Create R_u(τ) matrix
Ru = np.zeros((K, K))
for i in range(K):
    if i == 0:
        Ru[i] = np.hstack((ru[K-1-i:]))
    else:
        Ru[i] = np.hstack((ru[K-1-i:-i]))



# %%
Ruy = cal_corr(X, y, K)
g = np.linalg.inv(Ru) @ Ruy
g_zero = np.zeros((K-1))
g1 = np.concatenate((g_zero, g))
#%%
def calc_gpac_values(ry, J, K):

    den = np.zeros((K, K))
```

```
    for k in range(0,K):
        row = np.zeros(K)
        for i in range(0,K):
            row[i] = ry[np.abs(J + k - i)]
        den[k] = row
    col = np.zeros(K)
    for i in range(0,K):
        col[i] = ry[J+i+1]
    num = np.concatenate((den[:, :-1], col.reshape(-1, 1)), axis=1)
    num = np.array(num)
    den = np.array(den)
    if np.linalg.det(den) == 0:
        return np.inf
    if np.abs(np.linalg.det(num)/np.linalg.det(den)) < 0.00001:
        return 0
    return np.linalg.det(num)/np.linalg.det(den)


def generate_gpac_table(ry, J=7, K=7,title='GPAC Table'):
    gpac_arr = np.zeros((J, K))
    gpac_arr.fill(None)
    for k in range(1, K):
        for j in range(J):
            gpac_arr[j][k] = calc_gpac_values(ry, j, k)
    gpac_arr = np.delete(gpac_arr, 0, axis=1)
    # creating dataframe
    cols = []
    for k in range(1, K):
        cols.append(k)
    ind = []
    for j in range(J):
        ind.append(j)
    df = pd.DataFrame(gpac_arr, columns=cols, index=ind)

    fig = plt.figure()
    ax = sns.heatmap(df, annot=True, fmt='0.2f')
    plt.title(f'{title}')
    plt.tight_layout()
    plt.show()
    print(df)
#%%
title = 'G-GPAC [B(q) and F(q)]'
generate_gpac_table(g, J=7, K=7,title=title)
# %%
def cal_v(y, u, g1, K):
    """Compute v(t) as y(t) - sum(g(i) * u(t-i)) for i = 0 to K."""
    v = np.zeros(len(y))  # Initialize v array with zeros
    for t in range(K, len(y)):  # Start from K because we need past values of u
        v[t] = y[t] - np.sum(g1[i] * u[t - i] for i in range(K))
    return v

# %%
```

```
v = cal_v(y,X, g, K)
rv= cal_autocorr(v, K)
title = 'H-PAC [C(q) and D(q)]'
generate_gpac_table(rv, J=7, K=7,title=title)
#%%
#LM Algorithm

def box_jenkins_cal_e(y, u, nf, nb, nc, nd, theta, seed=6313):
    np.random.seed(seed)
    # Split theta into transfer function components
    den_G = theta[:nb]  # B(q) coefficients
    num_G = theta[nb:nb+nf]  # F(q) coefficients
    den_H = theta[nb+nf:nb+nf+nc]  # C(q) coefficients
    num_H = theta[nb+nf+nc:]  # D(q) coefficients
    # Add leading 1 for system stability
    den_G = np.insert(den_G, 0, 1)  # B(q)
    num_G = np.insert(num_G, 0, 1)  # F(q)
    den_H = np.insert(den_H, 0, 1)  # C(q)
    num_H = np.insert(num_H, 0, 1)  # D(q)


    # Transfer functions for H(q) = D(q)/C(q)
    sys_H = (num_H, den_H, 1)

    # Simulate H(q) applied to y(t)
    _, y_H = dlsim(sys_H, y)

    # Transfer functions for G(q) = B(q)/F(q)
    sys_G = (den_G, num_G, 1)
    # Simulate G(q) applied to u(t)
    _, u_g = dlsim(sys_G, u)

    # Simulate D(q)/C(q) * B(q)/F(q) applied to u(t)
    _,u_UD = dlsim(sys_H, u_g)

    # Calculate the residuals
    e = np.squeeze(y_H) - np.squeeze(u_UD)

    return e

def lm_step1_bj(y, u, na, nb, nc, nd, delta, theta):
    n = na + nb + nc + nd
    e = box_jenkins_cal_e(y, u, na, nb, nc, nd, theta)
    sse_old = np.dot(e.T, e)
    X = np.empty(shape=(len(y), n))
    for i in range(n):
        theta[i] += delta
        e_i = box_jenkins_cal_e(y, u, na, nb, nc, nd, theta)
        X[:, i] = (e - np.squeeze(e_i)) / delta
        theta[i] -= delta
    A = np.dot(X.T, X)
    g = np.dot(X.T, e)
    return A, g, X, sse_old
```

```
def lm_step2_bj(y, u, na, nb, nc, nd, A, theta, mu, g):
    delta_theta = np.matmul(np.linalg.inv(A + mu * np.identity(A.shape[0])), g)
    theta_new = theta + delta_theta
    print(f"Step 2 - theta_new: {theta_new}")
    e_new = box_jenkins_cal_e(y, u, na, nb, nc, nd, theta_new)
    sse_new = np.dot(e_new.T, e_new)
    if np.isnan(sse_new):
        sse_new = 10 ** 10
    return sse_new, delta_theta, theta_new

def lm_step3_bj(y, u, na, nb, nc, nd):
    N = len(y)
    n = na + nb + nc + nd
    mu = 0.01
    mu_max = 1e20
    max_iterations = 10
    delta = 1e-15
    var_error = 0
    covariance_theta_hat = 0
    sse_list = []
    theta = np.zeros(n)

    for iterations in range(max_iterations):
        print(f"Iteration {iterations + 1}")
        A, g, X, sse_old = lm_step1_bj(y, u, na, nb, nc, nd, delta, theta)
        sse_new, delta_theta, theta_new = lm_step2_bj(y, u, na, nb, nc, nd, A, theta, mu, g)
        sse_list.append(sse_old)
        print(f"Iteration {iterations + 1} - SSE old: {sse_old}")
        print(f"Iteration {iterations + 1} - SSE new: {sse_new}")

        if sse_new <= sse_old:
            if np.linalg.norm(np.array(delta_theta), 2) < 1e-15:
                var_error = sse_new / (N - n)
                covariance_theta_hat = var_error * np.linalg.inv(A)
                print(f"Convergence occurred in {iterations + 1} iterations")
                break
            else:
                theta = theta_new
                mu /= 10
        else:
            while sse_new > sse_old:
                mu *= 10
                if mu > mu_max:
                    print("No convergence")
                    break
                sse_new, delta_theta, theta_new = lm_step2_bj(y, u, na, nb, nc, nd, A, theta, mu, g)

        if iterations >= max_iterations:
            print("Max iterations reached")
            break
        var_error = sse_new / (N - n)
        covariance_theta_hat = var_error * np.linalg.inv(A)
```

```
    return theta, sse_new, var_error, covariance_theta_hat, sse_list
theta_new, sse_new, var_error, covariance_theta_hat, sse_list = lm_step3_bj(y,X, 1, 1, 1, 1)

print("Estimated parameters:", theta_new)

#%%
# Display the standard deviation
print(f"Standard deviation: {np.sqrt(var_error)}")
#%%
covariance_theta_hat
# %%
#Confidence Intervals
num_H_root = [1]
den_H_root = [1]
num_G_root = [1]
den_G_root = [1]
nb = nf = nc = nd = 1

print("Confidence Intervals for Coefficients:")
for i in range(nb):  # Coefficients for B
    lower = theta_new[i] - 2 * np.sqrt(covariance_theta_hat[i][i])
    upper = theta_new[i] + 2 * np.sqrt(covariance_theta_hat[i][i])
    print(f"B[{i}] Interval: [{lower:.2f}, {upper:.2f}]")
    num_G_root.append(theta_new[i])

for i in range(nf):  # Coefficients for F
    lower = theta_new[i + nb] - 2 * np.sqrt(covariance_theta_hat[i + nb][i + nb])
    upper = theta_new[i + nb] + 2 * np.sqrt(covariance_theta_hat[i + nb][i + nc])
    print(f"F[{i}] Interval: [{lower:.2f}, {upper:.2f}]")
    den_G_root.append(theta_new[i + nb])

for i in range(nc):  # Coefficients for C
    lower = theta_new[i + nb + nf] - 2 * np.sqrt(covariance_theta_hat[i +  nb + nf][i + nb + nf])
    upper = theta_new[i + nb + nf] + 2 * np.sqrt(covariance_theta_hat[i + nb + nf][i + nb + nf])
    print(f"C[{i}] Interval: [{lower:.2f}, {upper:.2f}]")
    num_H_root.append(theta_new[i + nb + nf])

for i in range(nd):  # Coefficients for D
    lower = theta_new[i + nb + nf + nc] - 2 * np.sqrt(covariance_theta_hat[i + nb + nf + nc][i + nb + nf
+ nc])
    upper = theta_new[i + nb + nf + nc] + 2 * np.sqrt(covariance_theta_hat[i + nb + nf + nc][i + nb + nf
+ nc])
    print(f"D[{i}] Interval: [{lower:.2f}, {upper:.2f}]")
    den_H_root.append(theta_new[i + nb + nf + nc])

#%%
#Poles and Zeros
zeros_H = np.roots(num_H_root)
poles_H = np.roots(den_H_root)
zeros_G = np.roots(num_G_root)
poles_G = np.roots(den_G_root)
# Display poles and zeros
print("\nPoles and Zeros:")
```

```
for i in range(len(zeros_H)):
   print("The roots of the C(q) are ", zeros_H)
for i in range(len(poles_H)):
   print("The roots of the D(q) are ", poles_H)
for i in range(len(zeros_G)):
   print("The roots of the B(q) are ", zeros_G)
for i in range(len(poles_G)):
   print("The roots of the F(q) are ", poles_G)
# %%
b_pr = [1] + theta_new[:nb]
f_pr = [1] + theta_new[nb:nb+nf]
c_pr = [1] + theta_new[nb+nf:nb+nf+nc]
d_pr = [1] + theta_new[nb+nf+nc:]
def check_num_den_size(num, den):
   if len(num) > len(den):
       den = np.pad(den, (0, len(num) - len(den)), 'constant')
   elif len(den) > len(num):
       num = np.pad(num, (0, len(den) - len(num)), 'constant')
   return num, den
[num_H_pr, den_H_pr] = check_num_den_size(c_pr, d_pr)
[num_G_pr, den_G_pr] = check_num_den_size(b_pr, f_pr)
tf_H_inv = (den_H_pr, num_H_pr, 1)
tf_G_pr = (num_G_pr, den_G_pr, 1)

tf_e2 = (np.convolve(den_H_pr, num_G_pr), np.convolve(num_H_pr,den_G_pr), 1)

num_t_1 = [m-n for m,n in zip(num_H_pr,den_H_pr)]
tf_t_1 = (num_t_1, num_H_pr, 1)

lags = 50
_,y_1_t_1 = dlsim(tf_e2,X)
_,y_1_t_2 = dlsim(tf_t_1,y)
y_hat_t_1 = y_1_t_1 + y_1_t_2

_,e1 = dlsim(tf_H_inv,y)
_,e2 = dlsim(tf_e2,X)
e_pr = e1 + e2
# %%
re = cal_autocorr(e_pr,lags)
Q = len(y)*np.sum(np.square(re[lags:]))*1e-3
DOF = lags - nc - nd
alpha = 0.01
chi_critical_Q = chi2.ppf(1-alpha,DOF)
if Q < chi_critical_Q:
   print("The Q-state is passed - H(q) is accurate")
else:
   print("The Q-state is not passed - H(q) not accurate.")
print(f'Q: {Q}, Critical Q: {chi_critical_Q}')

#%%
S = 1
R = 1
alpha = X
```

```
sigma_alpha = np.std(alpha)
sigma_e_pr = np.std(e_pr)
r_alpha_e = cal_corr(alpha, e_pr, K)/(sigma_alpha*sigma_e_pr)
S =len(y)*np.sum(np.square(r_alpha_e)) *1e-3
DOF = lags -nb - nf
alpha = 0.01
chi_critical_S = chi2.ppf(1-alpha,DOF)
if S < chi_critical_S:
   print("The S-state is passed - G(q) is accurate")
else:
   print("The S-state is not passed - G(q) is not accurate.")
print(f'S: {S}, Critical S: {chi_critical_S}')
# %%
#I picked ARIMA model for forecasting
#Forecasting function
e = np.random.normal(0, 1, len(y_diff))
T = len(y_diff)



def forecast(y, e, T, h):
   y_hat = []
   len_y = len(y)
   len_e = len(e)

   for i in range(h):
       # Check if the required indices exist in y and e
       if (T - 12 >= 0 and T - 12 < len_y) and (T - 24 >= 0 and T - 24 < len_y) and (T - 36 >= 0 and T
- 36 < len_y) and (T + i < len_e):
           if i == 0:
               # First forecast: safely calculate using historical values
               forecast = 0.4 * y[T - 12] + 0.63 * y[T - 24] - 0.03 * y[T - 36] + e[T] - 1.51 * e[T -
12] + 0.08 * e[T - 24]
           else:
               # For subsequent forecasts, check if the indices are within bounds
               if (T + i - 12 >= 0 and T + i - 12 < len_y) and (T + i - 24 >= 0 and T + i - 24 < len_y)
and (T + i - 36 >= 0 and T + i - 36 < len_y):
                   forecast = 0.4 * y[T + i - 12] + 0.63 * y[T + i - 24] - 0.03 * y[T + i - 36] + e[T +
i] - 1.51 * e[T + i - 12] + 0.08 * e[T + i - 24]

           y_hat.append(forecast)

   return np.array(y_hat)

# Define your `y` and `e` series along with T and h (you may need to update them as per your data)
y_test = y_test_diff
y = y_diff
print()
# Adjust forecast horizon to match test data length
y_hat_test = forecast(y, e, T, len(y_test))  # Use the length of your test data for forecasting
y_hat_test = pd.Series(arima_model_hat.ravel())

if isinstance(y_test, np.ndarray):
```

```
    y_test = pd.Series(y_test)

y_hat_test = pd.Series(y_hat_test)

plt.figure(figsize=(10, 6))  # Set the figure size

# Plot true values (y_test)
plt.plot(y_test.index, y_test, label='True Values', color='blue', linestyle='-', linewidth=2)

# Plot predicted values (y_hat_test)
plt.plot(y_hat_test.index, y_hat_test, label='Predicted Values', color='red', linestyle='--',
linewidth=2)

# Add titles and labels
plt.title("Predicted vs True Values", fontsize=16)
plt.xlabel("Time", fontsize=14)
plt.ylabel("Value", fontsize=14)

# Adjust X-axis range to match the range of both y_test and y_hat_test
plt.xlim(min(y_test.index.min(), y_hat_test.index.min()), max(y_test.index.max(),
y_hat_test.index.max()))

# Show legend
plt.legend()

# Display the plot
plt.grid(True)
plt.show()
```

Toolbox.py

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import kpss
from statsmodels.graphics.tsaplots import plot_acf , plot_pacf
import statsmodels.api as sm
import seaborn as sns
import numpy.linalg as LA
from scipy import signal
from scipy.stats import chi2
from statsmodels.tsa.seasonal import STL
import math
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX



def Cal_rolling_mean_var(data, label):
    """
    Function to calculate and plot the rolling mean and variance.
```

```
    Parameters:
    data (array-like): The data for which to calculate the rolling statistics.
    label (str): The label for the plot (e.g., 'Sales', 'AdBudget', 'GDP').
    """
    n = len(data)

    # Initialize lists to store rolling mean and variance
    rolling_means = []
    rolling_variances = []

    # Loop through the data and calculate rolling mean and variance
    for i in range(1, n + 1):
        current_data = data[:i]
        rolling_means.append(np.mean(current_data))
        rolling_variances.append(np.var(current_data, ddof=1))  # ddof=1 for sample variance

    # Plot rolling mean and variance
    fig, axs = plt.subplots(2, 1, figsize=(10, 8))

    # Rolling mean plot
    axs[0].plot(range(0, n), rolling_means, color='blue')
    axs[0].set_title(f'Rolling Mean of {label}')
    axs[0].set_xlabel('Samples')
    axs[0].set_ylabel('Magnitude')
    axs[0].grid(True)

    # Rolling variance plot
    axs[1].plot(range(0, n), rolling_variances, color='red')
    axs[1].set_title(f'Rolling Variance of {label}')
    axs[1].set_xlabel('Samples')
    axs[1].set_ylabel('Magnitude')
    axs[1].grid(True)

    plt.tight_layout()
    plt.show()


# Check for stationarity using ADF test
def adf_test(series):
    result = adfuller(series)
    print('ADF Test Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:', result[4])
    if result[1] < 0.05:
        print("Series is likely stationary.")
    else:
        print("Series is likely non-stationary.")

# Check for stationarity using KPSS test
def kpss_test(series):
    result = kpss(series, regression='c', nlags="auto")
    print('KPSS Test Statistic:', result[0])
```

```
    print('p-value:', result[1])
    print('Critical Values:', result[3])
    if result[1] < 0.05:
        print("Series is likely non-stationary.")
    else:
        print("Series is likely stationary.")



def differencing(y, order, s=1):
    diff = []
    n = len(y)
    for i in range(n):
        if i-s < 0 or y[i] is None or y[i-s] is None:
            diff.append(None)
        else:
            diff.append(y[i] - y[i - s])
    if order == 1:
        return np.array(diff)
    return differencing(pd.Series(diff), order-1, s)



def check_stationarity(y, title):
    y = pd.Series(y.ravel())
    Cal_rolling_mean_var(y, title)
    adf_test(y)
    kpss_test(y)
    ACF_PACF_Plot(y, 50)


def logTransform(y):
    log_y = np.log(y)
    log_y = log_y.dropna().reset_index(drop=True)
    return log_y


# autocorrelation
def cal_autocorr(Y, lags):  # default value is set to None, i.e. the case when we don't need subplots
    T = len(Y)
    ry = []
    den = 0
    ybar = np.mean(Y)
    for y in Y:  # since denominator is constant for every iteration, we calculate it only once and
store it.
        den = den + (y - ybar) ** 2

    for tau in range(lags+1):
        num = 0
        for t in range(tau, T):
            num = num + (Y[t] - ybar) * (Y[t - tau] - ybar)
        ry.append(num / den)
```

```
    ryy = ry[::-1]
    Ry = ryy[:-1] + ry  # to make the plot on both sides, reversed the list and added to the original
list

    return Ry


def cal_error_MSE(y, yhat, skip=0):
    y = np.array(y)
    yhat = np.array(yhat)
    error = []
    error_square = []
    n = len(y)
    for i in range(n):
        if yhat[i] is None:
            error.append(None)
            error_square.append(None)
        else:
            error.append(y[i]-yhat[i])
            error_square.append((y[i]-yhat[i])**2)
    mse = 0
    for i in range(skip, n):
        mse = mse + error_square[i]

    mse = mse/(n-skip)

    return error, error_square, np.round(mse, 2)


def plot_forecasting_models(ytrain, ytest, yhatTest, title, axs=None):
    if axs is None:
        axs = plt
    x = np.arange(1, len(ytrain)+len(ytest)+1)
    x1 = x[:len(ytrain)]
    x2 = x[len(ytrain):]
    axs.plot(ytrain.index, ytrain, color='r', label='train')
    axs.plot(ytest.index, ytest, color='g', label='test')
    axs.plot(ytest.index, yhatTest, color='b', label='h step')
    if axs is plt:
        plt.xlabel('Time')
        plt.ylabel('Values')
        plt.title(title)
        plt.legend()
        plt.grid()
        plt.show()
    else:
        axs.set_xlabel('Time')
        axs.set_ylabel('Values')
        axs.set_title(title)
        axs.grid()
        axs.legend()
```

```
def average_forecasting(ytrain, ytest):
    # Calculate the one-step prediction for each time step in the training set
    one_step_prediction = [0]  # First value is always 0 since there's no previous data
    running_sum = 0

    for i in range(len(ytrain)):
        if i > 0:  # Avoid division by zero
            one_step_prediction.append(running_sum / i)
        running_sum += ytrain[i]  # Update running sum

    # Calculate the average of the entire training set
    average_forecast = np.mean(ytrain)

    # Prepare h-step forecasts (using the average of the training set)
    h_step_forecasts = np.array([average_forecast] * len(ytest))

    # Plotting
    plt.figure(figsize=(12, 6))
    plt.plot(range(1, len(ytrain) + 1), ytrain, marker='o', color='blue', label='Training Set')
    plt.plot(range(len(ytrain) + 1, len(ytrain) + len(ytest) + 1), ytest, marker='o', color='orange',
label='Testing Set')
    plt.plot(range(len(ytrain) + 1, len(ytrain) + len(h_step_forecasts) + 1), h_step_forecasts,
marker='x', color='green', linestyle='--', label='H-Step Forecast')

    # Adding titles and labels
    plt.title('Time Series Forecasting using Average Method')
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.legend()
    plt.grid()
    plt.xticks(np.arange(1, len(ytrain) + len(ytest) + 1))
    plt.show()

    return one_step_prediction, h_step_forecasts

def Naive_forecasting(xtrain, xtest):
    # Forecast the last observed value for all h-steps
    h_step_forecast = [xtrain[-1]] * len(xtest)

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(xtrain) + 1),xtrain, label='Training Set', marker='o', color='blue')
    plt.plot(range(len(xtrain) + 1, len(xtrain) + len(xtest) + 1), xtest, label='Testing Set',
marker='x', color='green')
    plt.plot(range(len(xtrain)+1, len(xtrain) + len(h_step_forecast)+1), h_step_forecast, label='h-step
Forecast', linestyle='--', color='red')
    plt.title('Time Series Forecasting using Naive Method')
    plt.xlabel('Time Steps')
    plt.ylabel('Values')
    plt.legend()
    plt.grid()
    plt.show()
```

```
    return xtrain[-1], h_step_forecast


def drift_method(t, train_data):
    if t == 1 or t == 2:
        return 0
    else:
        drift = train_data[t-2] + ((train_data[t-2] - train_data[0]) / (t-2))
    return drift

def drift_forecasting(xtrain, xtest):
    # Calculate drift based on the training set
    T = len(xtrain)
    drift = (xtrain[-1] - xtrain[0]) / (T - 1)
    h_step_forecast = [xtrain[-1] + (i + 1) * drift for i in range(len(xtest))]

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(xtrain) + 1), xtrain, label='Training Set', marker='o', color='blue')
    plt.plot(range(len(xtrain) + 1, len(xtrain) + len(xtest) + 1), xtest, label='Testing Set',
marker='x', color='green')
    plt.plot(range(len(xtrain) + 1, len(xtrain) + len(h_step_forecast) + 1), h_step_forecast,
label='h-step Forecast (Drift)', linestyle='--', color='red')
    plt.title('Time Series Forecasting using Drift Method')
    plt.xlabel('Time Steps')
    plt.ylabel('Values')
    plt.legend()
    plt.grid()
    plt.show()

    return drift, h_step_forecast


def ses_forecasting(ytrain, ytest, initial_value, alpha):
    # Initialize predictions with the initial value
    ses_predictions = [initial_value]

    # Apply SES recursively for the training set
    for t in range(1, len(ytrain)):
        ses_predictions.append(alpha * ytrain[t] + (1 - alpha) * ses_predictions[-1])

    # Forecast for the test set using the last SES value
    h_step_forecast = [ses_predictions[-1]] * len(ytest)

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(ytrain) + 1), ytrain, label='Training Data', marker='o', color='blue')
    plt.plot(range(len(ytrain) + 1, len(ytrain) + len(ytest) + 1), ytest, label='Testing Data',
marker='x', color='green')
    plt.plot(range(len(ytrain) + 1, len(ytrain) + len(h_step_forecast) + 1), h_step_forecast,
label='h-Step Forecast (SES)', linestyle='--', color='red')
    plt.title('Time Series Forecasting using SES Method')
```

```
    plt.xlabel('Time Steps')
    plt.ylabel('Values')
    plt.legend()
    plt.grid()
    plt.show()

    return ses_predictions[:-1], h_step_forecast



def cal_Q_value(y, title, lags=5):
    # title = 'Average forecasting train data'
    acf = cal_autocorr(y, lags)
    sum_rk = 0
    T = len(y)
    for i in range(1, lags+1):
        sum_rk += acf[i]**2
    Q = T * sum_rk
    # if Q < Q* then white residual
    return Q


def standardize(train, test):
    columns = train.columns
    X_train = train.copy()
    X_test = test.copy()
    for col in columns:
        xbar = np.mean(X_train[col])
        std = np.std(X_train[col])
        X_train[col] = (X_train[col] - xbar) / std
        X_test[col] = (X_test[col] - xbar) / std

    return X_train, X_test


# β= (XTX)−1XTY
def normal_equation_LSE(X, Y):
    # X = x.to_numpy()
    # Y = y.to_numpy()
    normal_eqn = ((np.linalg.inv(X.T@X))@X.T)@Y
    return normal_eqn


def moving_average_decomposition(arr, order):
    m = order
    k = (m - 1) // 2
    res = []
    len_data = len(arr)

    if m == 2:
        res.append(None)
    else:
        for i in range(k):
```

```
            res.append(None)

    for i in range(len_data - m + 1):
        s = 0
        flag = True
        for j in range(i, i+m):
            if arr[j] is None:
                flag = False
                break
            s += arr[j]
        if flag is False:
            res.append(None)
        else:
            res.append(s/m)
    if m % 2 == 0 and m != 2:
        for i in range(k+1):
            res.append(None)
    elif m != 2:
        for i in range(k):
            res.append(None)

    return res


def create_process_general_AR(order, N, a):
    na = order
    np.random.seed(6313)
    mean = 0
    std = 1
    e = np.random.normal(mean, std, N)
    y = np.zeros(len(e))
    coef = np.zeros(na)
    for t in range(len(e)):
        sum_coef = 0
        y[t] = e[t]
        for i in range(1, na+1):
            if t-i < 0:
                break
            else:
                sum_coef += coef[i-1]*y[t-i]
        if t < na:
            coef[t] = a[t]

        y[t] -= sum_coef

    return y


def whitenoise(mean, std, samples, seed=0):
    np.random.seed(seed)
    return np.random.normal(mean, std, samples)
```

```
def ACF_PACF_Plot(y, lags):
    acf = sm.tsa.stattools.acf(y, nlags=lags)
    pacf = sm.tsa.stattools.pacf(y, nlags=lags)
    fig = plt.figure()
    plt.subplot(211)
    plt.title('ACF/PACF of the raw data')
    plot_acf(y, ax=plt.gca(), lags=lags)
    plt.subplot(212)
    plot_pacf(y, ax=plt.gca(), lags=lags)
    fig.tight_layout(pad=3)
    plt.show()


def calc_val(Ry, J, K):

    den = np.zeros((K, K))

    for k in range(K):
        row = np.zeros(K)
        for i in range(K):
            row[i] = Ry[np.abs(J + k - i)]
        den[k] = row
    # num = den.copy()
    col = np.zeros(K)
    for i in range(K):
        col[i] = Ry[J+i+1]

    num = np.concatenate((den[:, :-1], col.reshape(-1, 1)), axis=1)
    num = np.array(num)
    den = np.array(den)

    if np.linalg.det(den) == 0:
        return np.inf
    if np.abs(np.linalg.det(num)/np.linalg.det(den)) < 0.00001:
        return 0
    return np.linalg.det(num)/np.linalg.det(den)


def cal_gpac(Ry, J=7, K=7):
    gpac_arr = np.zeros((J, K))
    gpac_arr.fill(None)
    for k in range(1, K):
        for j in range(J):
            gpac_arr[j][k] = calc_val(Ry, j, k)
    gpac_arr = np.delete(gpac_arr, 0, axis=1)
    # creating dataframe
    cols = []
    for k in range(1, K):
        cols.append(k)
    ind = []
    for j in range(J):
        ind.append(j)
    df = pd.DataFrame(gpac_arr, columns=cols, index=ind)
```

```
    fig = plt.figure()
    ax = sns.heatmap(df, annot=True, fmt='0.3f')  # cmap='Pastel2'
    plt.title('Generalized Partial Autocorrelation (GPAC) Table')
    plt.tight_layout()
    plt.show()
    print(df)


def check_AIC_BIC_adjR2(x, y):
    columns = x.columns
    res_df = pd.DataFrame(columns=['Removing column', 'AIC', 'BIC', 'AdjR2'])
    for col in columns:
        temp_df = x.copy()
        temp_df = temp_df.drop([col], axis=1)
        res = sm.OLS(y, temp_df).fit()
        res_df.loc[len(res_df.index)] = [col, res.aic, res.bic, res.rsquared_adj]

    res_df = res_df.sort_values(by=['AIC'], ascending=False)
    return res_df


# LM algo and supporting functions

def cal_e(num, den, y):
    system = (num, den, 1)
    _, e = signal.dlsim(system, y)
    return e


def num_den(theta, na, nb):
    theta = theta.ravel()
    num = np.concatenate(([1], theta[:na]))
    den = np.concatenate(([1], theta[na:]))
    max_len = max(len(num), len(den))
    num = np.pad(num, (0, max_len - len(num)), 'constant')
    den = np.pad(den, (0, max_len - len(den)), 'constant')
    return num, den


def cal_gradient_hessian(y, e, theta, na, nb):
    delta = 0.000001
    X = np.empty((len(e), 0))
    for i in range(len(theta)):
        temp_theta = theta.copy()
        temp_theta[i] = temp_theta[i] + delta
        num, den = num_den(temp_theta, na, nb)
        e_new = cal_e(num, den, y)
        x_temp = (e - e_new)/delta
        X = np.hstack((X, x_temp))

    # A = X.T @ X
    # g = X.T @ e
```

```
    A = np.dot(X.T, X)
    g = np.dot(X.T, e)
    return A, g


def SSE(theta, y, na, nb):
    num, den = num_den(theta, na, nb)
    e = cal_e(num, den, y)
    return np.dot(e.T, e)


def LM(y, na, nb):
    epoch = 0
    epochs = 50
    theta = np.zeros(na + nb)
    mu = 0.01
    n = len(theta)
    N = len(y)
    mu_max = 1e+20
    sse_array = []
    while epoch < epochs:
        sse_array.append(SSE(theta, y, na, nb).ravel())
        num, den = num_den(theta, na, nb)
        e = cal_e(num, den, y)
        A, g = cal_gradient_hessian(y, e, theta, na, nb)
        del_theta = LA.inv(A + mu*np.identity(A.shape[0])) @ g
        theta_new = theta.reshape(-1, 1) + del_theta
        sse_new = SSE(theta_new.ravel(), y, na, nb)
        sse_old = SSE(theta.ravel(), y, na, nb)
        if sse_new[0][0] < sse_old[0][0]:
            if LA.norm(del_theta) < 1e-3:
                theta_hat = theta_new.copy()
                sse_array.append(SSE(theta_new, y, na, nb).ravel())
                variance_hat = SSE(theta_new.ravel(), y, na, nb)/(N-n)
                covariance_hat = variance_hat * LA.inv(A)
                return theta_hat, variance_hat, covariance_hat, sse_array
            else:
                mu = mu/10
        while SSE(theta_new.ravel(), y, na, nb) >= SSE(theta.ravel(), y, na, nb):
            mu = mu*10
            # theta = theta_new.copy()
            if mu > mu_max:
                print('Error')
                break
            del_theta = LA.inv(A + mu * np.identity(A.shape[0])) @ g
            theta_new = theta.reshape(-1, 1) + del_theta
        epoch += 1
        theta = theta_new.copy()
    return


def removeNA(y):
    y = pd.Series(y)
```

```
    return y.dropna()


def STL_analysis(y, periods):
    y = pd.Series(y)
    stl = STL(y, period=periods)
    res = stl.fit()
    fig = res.plot()
    plt.suptitle('Trend, seasonality, and remainder plot')
    plt.xlabel('Time')
    plt.tight_layout()
    plt.show()

    T = res.trend
    S = res.seasonal
    R = res.resid

    plt.figure(figsize=[16, 8])
    plt.plot(y, label='Original')
    plt.plot(y - S, label='Seasonally adjusted')
    plt.grid()
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.title('Seasonally adjusted vs original curve')
    plt.legend()
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=[16, 8])
    plt.plot(y, label='Original')
    plt.plot(y - T, label='Detrended')
    plt.grid()
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.title('Detrended vs original curve')
    plt.legend()
    plt.tight_layout()
    plt.show()

    Ft = max(0, 1 - np.var(R) / (np.var(T + R)))
    print("Strength of Trend for this dataset is ", Ft)

    seas = 1 - np.var(R) / (np.var(S + R))
    Fs = max(0, 1 - np.var(R) / (np.var(S + R)))
    print("Strength of seasonality for this dataset is ", Fs)

    return T, S, R




def reverse_transform_and_plot(prediction, y_train, y_test, title):
    forecast = []
```

```
    s = 365
    for i in range(len(y_test)):
        if i < s:
            forecast.append(prediction[i] + y_train[- s + i])
        else:
            temp = i - s
            forecast.append(prediction[i] + forecast[temp])
    forecast = pd.Series(forecast)
    forecast.index = prediction.index
    plt.plot(y_train.index, y_train.values, label='Train')
    plt.plot(forecast.index, forecast.values, label='Forecast')
    plt.plot(y_test.index, y_test.values, label='Actual Test Data')
    str = f'Predictions using {title}'
    plt.title(str)
    plt.legend()
    plt.tight_layout()
    plt.show()

    return forecast

# Function to display poles and zeros, with cancellation check
def display_poles_zeros(num, den):
    poles = np.roots(den)
    zeros = np.roots(num)
    print("\nPoles:", poles.round(3))
    print("Zeros:", zeros.round(3))

    # Check for pole-zero cancellations
    cancellation = any(np.isclose(pole, zero, atol=1e-3) for pole in poles for zero in zeros)
    print("Zero-pole cancellation?", "Yes" if cancellation else "No")


# Function to perform model fitting, prediction, and residual analysis
def prediction(y, na, nb, d=0, forecast_steps=10):
    np.random.seed(6313)
    lags = 25
    N = len(y)

    # Fit the ARMA or ARIMA model
    model = sm.tsa.ARIMA(y, order=(na, d, nb)).fit()

    # Generate predictions (one-step ahead forecast for all samples)
    model_hat = model.predict(start=0, end=N - 1)

    # Calculate residuals (difference between true and predicted values)
    residuals = y - model_hat

    # Whiteness Chi-square Test for residuals
    re = cal_autocorr(np.array(residuals), lags)
    Q = len(y) * np.sum(np.square(re[1:]))
    DOF = lags - na - nb
    alfa = 0.01  # Significance level
    chi_critical = chi2.ppf(1 - alfa, DOF)
```

```
    # Print the whiteness test results
    print('Chi critical:', chi_critical)
    print('Q Value:', Q)
    print('Alfa value for 99% accuracy:', alfa)
    if Q < chi_critical:
        print("The residual is white noise")
    else:
        print("The residual is NOT white noise")

    # Variance of residuals (error variance)
    residual_variance = np.var(residuals)
    print(f"Estimated variance of residuals (error variance): {residual_variance:.4f}")

    # Covariance of the estimated parameters (model parameters)
    param_covariance = model.cov_params()
    print(f"Estimated covariance of the parameters: \n{param_covariance}")

    # Bias check: mean of residuals (should be close to 0 for unbiased model)
    mean_residual = np.mean(residuals)
    print(f"Mean of residuals (should be close to 0 for unbiased model): {mean_residual:.4f}")

    # Forecast errors (out-of-sample prediction errors)
    forecast_values = model.forecast(steps=forecast_steps)
    forecast_errors = forecast_values - model.forecast(steps=forecast_steps)[0]
    forecast_variance = np.var(forecast_errors)

    # Compare variance of residual errors vs forecast errors
    print(f"Variance of residual errors (in-sample): {residual_variance:.4f}")
    print(f"Variance of forecast errors (out-of-sample): {forecast_variance:.4f}")

    # Model simplification: Zero-pole cancellation
    ar_params = model.arparams if hasattr(model, 'arparams') else []
    ma_params = model.maparams if hasattr(model, 'maparams') else []
    num = np.r_[1, -np.array(ar_params)]  # Numerator coefficients (AR part)
    den = np.r_[1, np.array(ma_params)]   # Denominator coefficients (MA part)

    # Display poles and zeros
    display_poles_zeros(num, den)

    # Display final coefficient confidence intervals
    conf_intervals = model.conf_int(alpha=0.01)  # 99% confidence intervals
    print("Final coefficient confidence intervals:")
    print(conf_intervals)


    return model, forecast_values
```

App.py

```
import dash
import dash_core_components as dcc
import dash_html_components as html
```

```
import pandas as pd
import plotly.express as px

# Load and preprocess the dataset
data = pd.read_csv("/Users/apoorvareddy/Downloads/Academic/DATS6313/Project/Data/final.csv")
data['time'] = pd.to_datetime(data['time'])  # Parse the time column as datetime

# Create Dash app
app = dash.Dash(__name__)

# Define app layout
app.layout = html.Div(
    children=[
    html.H1(children='Energy Dashboard', style={'textAlign': 'center', 'margin-bottom': '20px'}),

    # Dropdown to select a metric
    html.Label("Select a metric to visualize:"),
    dcc.Dropdown(
        id="metric-select",
        options=[{"label": column, "value": column} for column in data.columns if column != 'time'],
        value='generation biomass'  # Default metric
    ),

    # Line chart to display the metric
    dcc.Graph(id='metric-graph'),

    # Add spacing between the graph and date range picker
    html.Div(style={'height': '30px'}),  # Spacer

    # Date range picker to filter data
    html.Label("Select a date range:"),
    dcc.DatePickerRange(
        id='date-range-picker',
        start_date=data['time'].min(),
        end_date=data['time'].max(),
        display_format="YYYY-MM-DD"
    ),

    # Button to reset the date range
    html.Div(style={'margin-top': '10px'}),  # Add small spacing above the button
    html.Button('Reset Date Range', id='reset-button'),
])

# Callback for updating the line chart
@app.callback(
    dash.dependencies.Output('metric-graph', 'figure'),
    dash.dependencies.Input('metric-select', 'value'),
    dash.dependencies.Input('date-range-picker', 'start_date'),
    dash.dependencies.Input('date-range-picker', 'end_date'))
def update_graph(selected_metric, start_date, end_date):
    filtered_data = data[(data['time'] >= start_date) & (data['time'] <= end_date)]
    figure = px.line(filtered_data, x="time", y=selected_metric, title=f"{selected_metric} Over Time")
    return figure
```

```python
# Callback for resetting the date range
@app.callback(
    [dash.dependencies.Output('date-range-picker', 'start_date'),
     dash.dependencies.Output('date-range-picker', 'end_date')],
    dash.dependencies.Input('reset-button', 'n_clicks'))
def reset_date_range(n_clicks):
    return data['time'].min(), data['time'].max()

# Run the app
if __name__ == '__main__':
    app.run_server(debug=True)
```