KNU
Department of Electronics, Information and Communication Engineering

BIG DATA ANALYTICS OF ENERGY TIME SERIES

2024

Professor Kihyeon Kwon

Salaki Reynaldo Joshua (Ph.D Student)

TABLE OF CONTENTS

01 Time series analysis of solar power generation data

02 DC Group

03 AC Group

04 SOL Group

05 TMP Group

1 TIME SERIES ANALYSIS OF SOLAR POWER GENERATION DATA

The attached file was prepared using the pv_2years_eng.csv data file (in the case of annual forecasts, if there is insufficient data, it is okay to use augmentation).Implementation by determining possible time prediction units among minutes, hours, days, months, quarters, and years. Additional techniques may also be used. Submit as Jupyter Notebook and PDF file

1. Using solar power generation data, pv_2years_eng, decompose the time series into trend, seasonal, and residual components and display them with plot(). At this time, decompose using the additive vs. multiplicative model and present the results.

2. Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the autocorrelation lag (lag) using the Statsmodels acf() function and plot_acf(). Find .

3. Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the partial autocorrelation lag (lag) using the Statsmodels pacf() function and plot_acf().

4. Using solar power generation data, pv_2years_eng, implement it as a moving average (MA), and display the moving average prediction results as plot().

5. Using solar power generation data, pv_2years_eng, implement triple exponential smoothing and display the triple exponential smoothing results with plot().

6. Using solar power generation data, pv_2years_eng, implement autoregressive (AR) and display the autoregressive (AR) results in plot().

Our CSV data comprises 17 columns representing various parameters related to a photovoltaic system. These columns are labeled as follows: 'DC_V' for DC voltage, 'DC_A' for DC current, 'DC_W' for DC output, 'AC_V_RS' for AC voltage in phase R to S, 'AC_V_ST' for AC voltage in phase S to T, 'AC_V_TR' for AC voltage in phase T to R, 'AC_A_R' for AC current in phase R, 'AC_A_S' for AC current in phase S, 'AC_A_T' for AC current in phase T, 'AC_W' for AC output, 'AC_TOT' for AC cumulative power generation, 'AC_FREQ' for AC frequency, 'AC_POW' for AC power factor, 'SOL_RAD_SLOPE' for solar radiation slope in W/㎡, 'SOL_RAD_LEVEL' for horizontal solar radiation in W/㎡, 'TMP_MODU' for module temperature in ℃, and 'TMP_CLI' for outdoor temperature in ℃. These columns collectively provide comprehensive data on the performance and environmental conditions of the photovoltaic system.

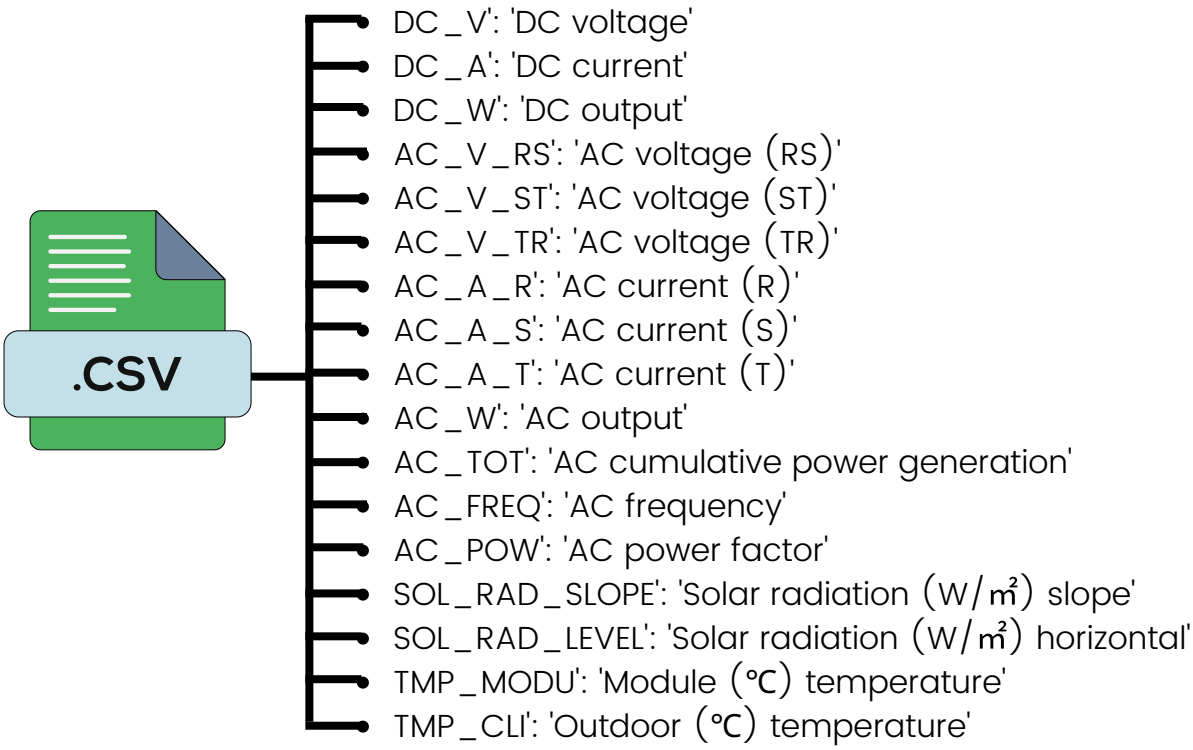


Figure 1. Column Information

FOUR ANALYSIS GROUPS

01

DC Group

- DC_V': 'DC voltage'
- DC_A': 'DC current'
- DC_W': 'DC output'

02

AC Group

- AC_V_RS': 'AC voltage (RS)'
- AC_V_ST': 'AC voltage (ST)'
- AC_V_TR': 'AC voltage (TR)'
- AC_A_R': 'AC current (R)'
- AC_A_S': 'AC current (S)'
- AC_A_T': 'AC current (T)'
- AC_W': 'AC output'
- AC_TOT': 'AC cumulative power generation'
- AC_FREQ': 'AC frequency'
- AC_POW': 'AC power factor'

03

SOL Group

- SOL_RAD_SLOPE': 'Solar radiation (W/㎡) slope'
- SOL_RAD_LEVEL': 'Solar radiation (W/㎡) horizontal'

04

TMP Group

- TMP_MODU': 'Module (℃) temperature'
- TMP_CLI': 'Outdoor (℃) temperature'

1 DC GROUP

The columns labelled 'DC_V', 'DC_A', and 'DC_W' in the solar panel data indicate important characteristics concerning the direct current (DC) output of the photovoltaic system. The DC voltage, which expresses the electrical potential difference between two places in the system, is indicated in the 'DC_V' column. It offers information on how strong the electrical potential produced by the solar panels is. Concurrently, the 'DC_A' column represents DC current, signifying the movement of electrical charge in the system. This metric is crucial for determining how much electricity the solar panels are producing. Lastly, the DC output—which is the real electrical power produced by the solar panels—is indicated in the 'DC_W' column. It is a quantitative indicator of the energy produced by the system and is the result of the DC voltage and current. When combined, these columns provide useful data regarding the solar panel array's effectiveness and performance, making it easier to monitor and optimize its operation.

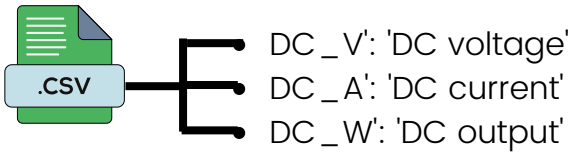


Figure 2. Analysis 1

Question 1

Using solar power generation data, pv_2years_eng, decompose the time series into trend, seasonal, and residual components and display them with plot(). At this time, decompose using the additive vs. multiplicative model and present the results.

This preprocessing steps prepare the solar panel data for further analysis, ensuring that it is structured appropriately, free from missing values, and ready for time series analysis or modeling.

```python
import pandas as pd
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Convert the date column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Set the date column as the index
data.set_index('date', inplace=True)

# Handle missing values by filling with mean
data.fillna(data.mean(), inplace=True)

# Add a small value to avoid zero or negative values for multiplicative decomposition
data = data + 1e-8

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Specify frequency
freq = 'H'
```

Import Libraries: The code imports necessary libraries such as pandas for data manipulation, statsmodels for time series analysis, and matplotlib for plotting.

Load Data: It loads the solar panel data from a CSV file into a pandas DataFrame named '**data**'.

Convert Date Column to Datetime Format: It converts the '**date**' column in the DataFrame data to datetime format using **pd.to_datetime().** This ensures that the dates are interpreted as datetime objects, enabling time-based operations.

Set Date Column as Index: The '**date**' column, now in datetime format, is set as the index of the DataFrame '**data**' using '**set_index()**'. This restructures the DataFrame so that it is indexed by dates, which is common for time series data.

Handle Missing Values: Missing values in the DataFrame are filled with the mean of each column using '**fillna()**'. This approach replaces any NaN values with the mean of the respective column, ensuring that the dataset remains complete and suitable for analysis.

Add a Small Value: A small value (1e-8) is added to the entire DataFrame '**data**'. This is done to avoid potential issues with zero or negative values, particularly when using multiplicative decomposition methods. The addition of a small positive value ensures that the data remains valid for such methods.

Define Columns of Interest: The variable **columns_of_interest** is defined as a list containing the names of columns that are of interest for further analysis. In this case, the columns 'DC_V', 'DC_A', and 'DC_W' are specified, likely representing important metrics related to the solar panel data.

Specify Frequency: The variable '**freq**' is assigned the value 'H', indicating that the time series data is at an hourly frequency. This information can be crucial for certain time series analysis techniques that rely on the frequency of the data, such as seasonal decomposition or forecasting.

This code efficiently decomposes the time series data for each specified column into its trend, seasonal, and residual components using both additive and multiplicative models and visualizes the results for further analysis. Additionally, it sets a larger figure size (18x14) for the plots to ensure clarity and readability.

```python
# Decompose the time series using both additive and multiplicative models
for column in columns_of_interest:
    # Additive decomposition
    decomposition_add = seasonal_decompose(data[column], model='additive', period=24, extrapolate_trend='freq')

    # Multiplicative decomposition
    decomposition_mul = seasonal_decompose(data[column], model='multiplicative', period=24, extrapolate_trend='freq')

    # Plot the decomposed components
    plt.figure(figsize=(18, 14))
    plt.subplot(3, 2, 1)
    plt.plot(decomposition_add.trend, label='Additive Trend')
    plt.title('Additive Trend')
    plt.legend()

    plt.subplot(3, 2, 2)
    plt.plot(decomposition_add.seasonal, label='Additive Seasonal')
    plt.title('Additive Seasonal')
    plt.legend()

    plt.subplot(3, 2, 3)
    plt.plot(decomposition_add.resid, label='Additive Residual')
    plt.title('Additive Residual')
    plt.legend()

    plt.subplot(3, 2, 4)
    plt.plot(decomposition_mul.trend, label='Multiplicative Trend')
    plt.title('Multiplicative Trend')
    plt.legend()

    plt.subplot(3, 2, 5)
    plt.plot(decomposition_mul.seasonal, label='Multiplicative Seasonal')
    plt.title('Multiplicative Seasonal')
    plt.legend()

    plt.subplot(3, 2, 6)
    plt.plot(decomposition_mul.resid, label='Multiplicative Residual')
    plt.title('Multiplicative Residual')
    plt.legend()

    plt.suptitle(f'Decomposition of {data[column].name}', fontsize=12)
    plt.show()
```

For Loop Over Columns of Interest: The code iterates over each column specified in the **'columns_of_interest'** list.

Additive Decomposition:

- For each column, it performs additive decomposition using the **'seasonal_decompose'** function from statsmodels.
- The **'model'** parameter is set to 'additive' to specify additive decomposition.
- The **'period'** parameter is set to 24, indicating that the data has a daily seasonality.
- **'extrapolate_trend'** is set to 'freq' to ensure that the trend is extrapolated to cover the entire time range.

Multiplicative Decomposition:

- Similarly, multiplicative decomposition is performed for each column using the same function.
- The **'model'** parameter is set to 'multiplicative' to specify multiplicative decomposition.

Plotting Decomposed Components:

- For each decomposition model (additive and multiplicative), the code plots the trend, seasonal, and residual components.
- It creates a 3x2 subplot grid for each column, resulting in a total of six subplots.
- Each subplot shows one component of the decomposition (trend, seasonal, or residual) for either the additive or multiplicative model.
- The title of each subplot indicates the specific component being plotted (e.g., 'Additive Trend', 'Multiplicative Seasonal').
- Legends are added to the plots to indicate the corresponding decomposition component.
- The suptitle describes the decomposition being performed for the specific column.

Display Plots: After plotting all components for a column, the code displays the plot using **'plt.show()'.**

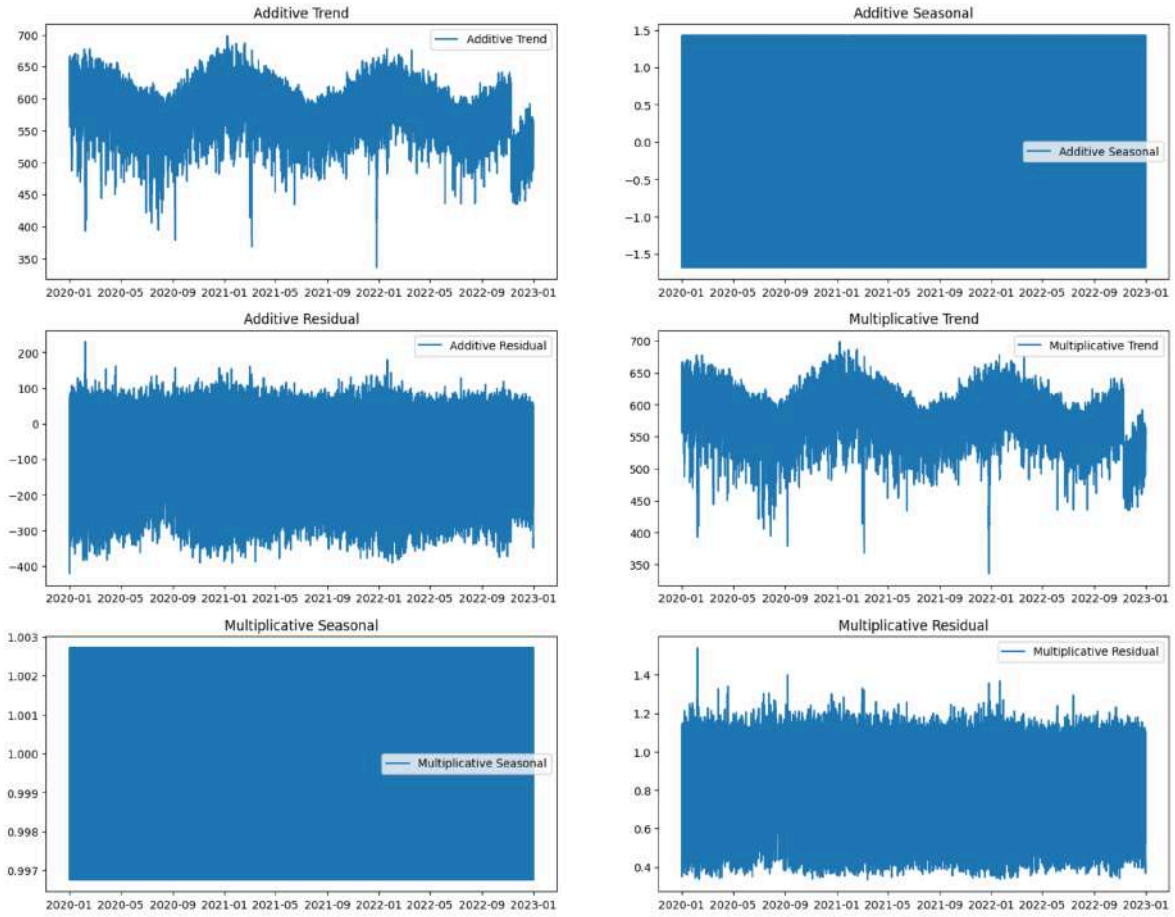


Figure 3. Decomposition of DC _ V

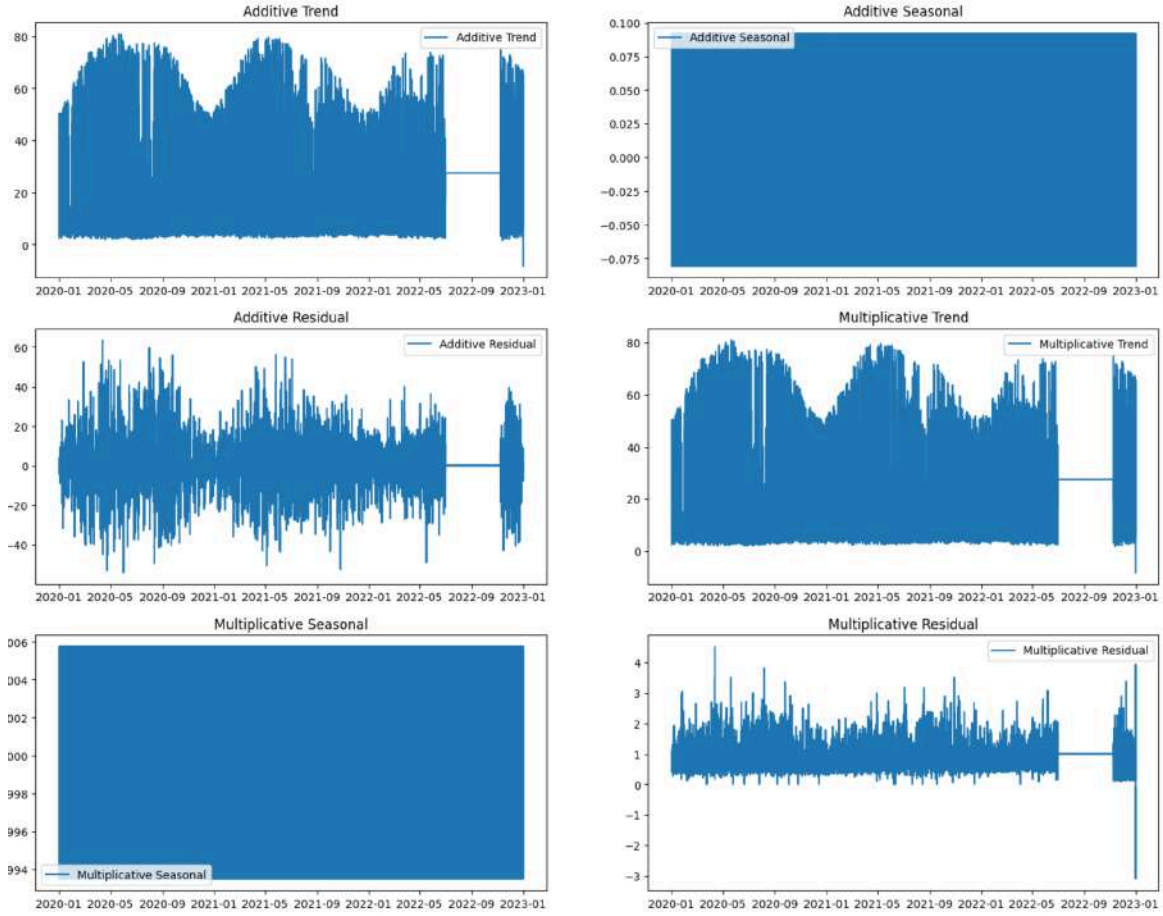


Figure 3. Decomposition of DC _ A

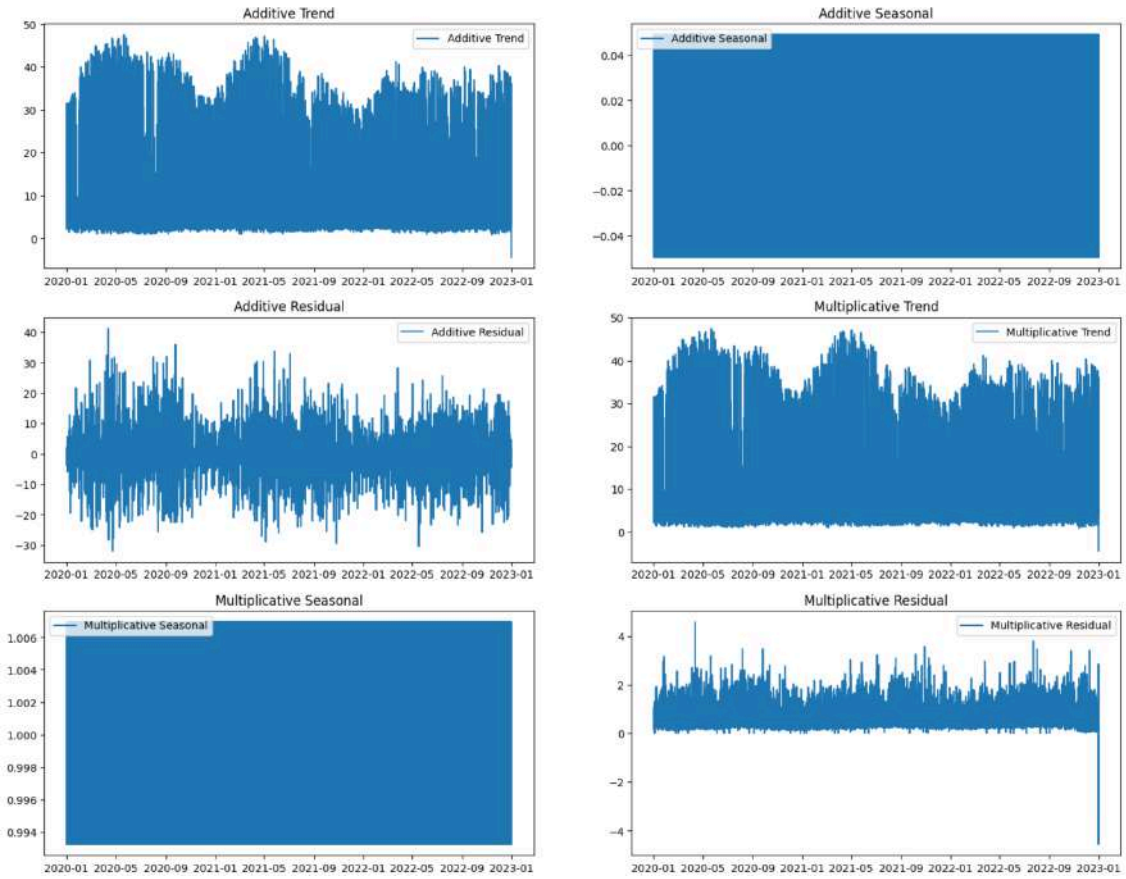


Figure 4. Decomposition of DC_W

Question 2

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the autocorrelation lag (lag) using the Statsmodels acf() function and plot_acf(). Find .

This structure prepares the data for further analysis by ensuring stationarity of the time series through the calculation of first-order differences for the specified columns of interest.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Import Libraries: The code imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **acf** from **statsmodels.tsa.stattools** for calculating autocorrelation.
- **plot_acf** from **statsmodels.graphics.tsaplots** for plotting autocorrelation.

Load Data: It loads the solar power generation data from a CSV file named 'pv_2years_eng.csv' using **pd.read_csv()** function and stores it in a DataFrame named **data.**

Defining Columns of Interest: **The code specifies a list named columns_of_interest** containing the names of columns that are of interest for further analysis. These columns represent different aspects of solar power generation: 'DC_V' (DC voltage), 'DC_A' (DC current), and 'DC_W' (DC output).

Ensuring Stationarity of the Time Series: For each column of interest, the code calculates the first-order difference by subtracting each value from its previous value using the **diff()** method. This is done to ensure stationarity of the time series data, which is a common requirement for many time series analysis techniques.

This structure enables visual inspection of the first-order difference and autocorrelation of each column, which are essential steps in time series analysis to understand the data's temporal patterns and dependencies.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate autocorrelation lag using acf() function
for column in columns_of_interest:
    acf_result = acf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Autocorrelation for {column}: {acf_result}')

# Plot autocorrelation using plot_acf() function
for column in columns_of_interest:
    plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference:

- It creates a figure with a size of 12x6 inches using **plt.figure(figsize=(12, 6)).**
- It iterates over each column of interest using a for loop with **enumerate(columns_of_interest, 1)** to track the index and column name.
- For each column, it creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)** to arrange the subplots vertically.
- It plots the first-order difference of the column against the index using **plt.plot(data.index, data[column + '_diff']).**
- Sets the title, xlabel, and ylabel for each subplot.
- Finally, it calls **plt.tight_layout()** to adjust the subplot layout and displays the plot using **plt.show().**

Calculating and Plotting Autocorrelation :For each column of interest:

- It calculates the autocorrelation up to lag 20 using **acf()** function from **statsmodels.tsa.stattools**. The **dropna()** method is used to remove any NaN values.
- Prints the autocorrelation results for each column.
- It then plots the autocorrelation function using **plot_acf()** function from **statsmodels.graphics.tsaplots**, specifying a maximum lag of 20.
- The title of each plot indicates the column for which autocorrelation is being calculated.

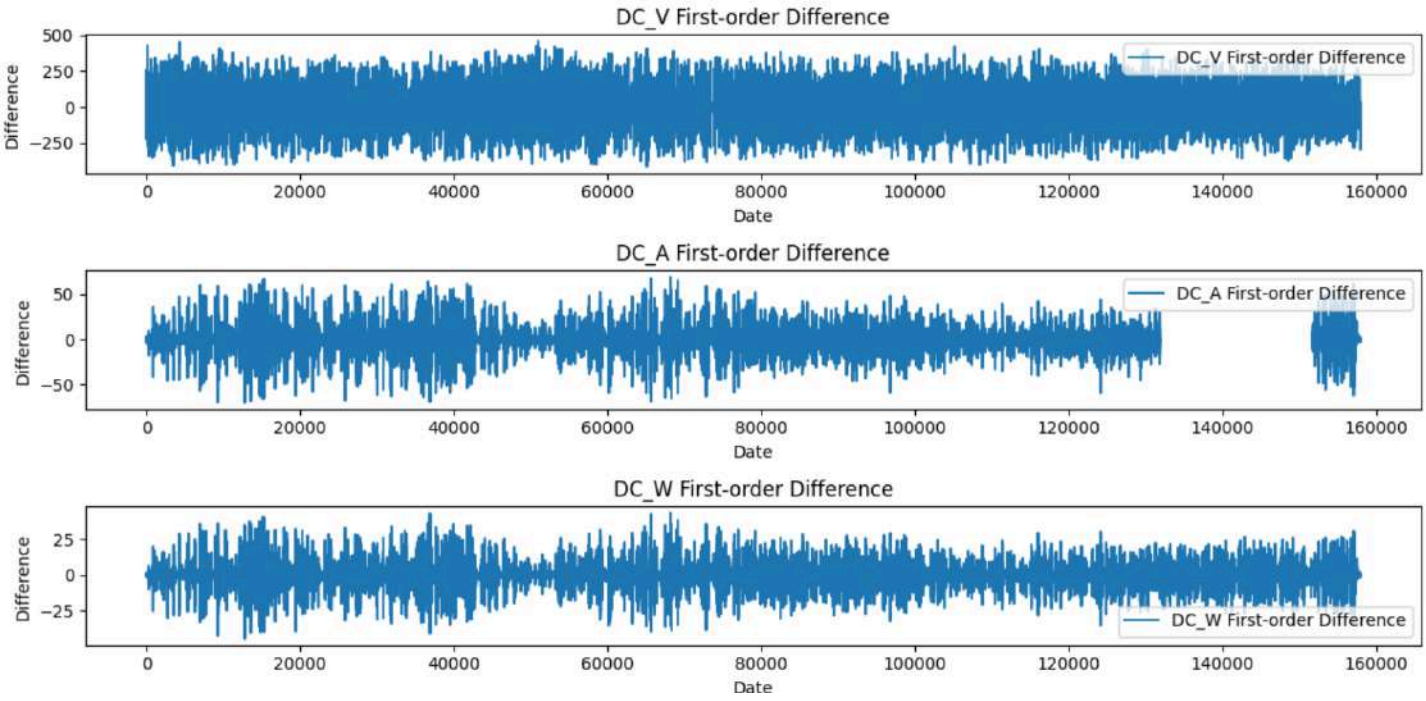


Figure 5. First-order difference of DC_V, DC_A, DC_W

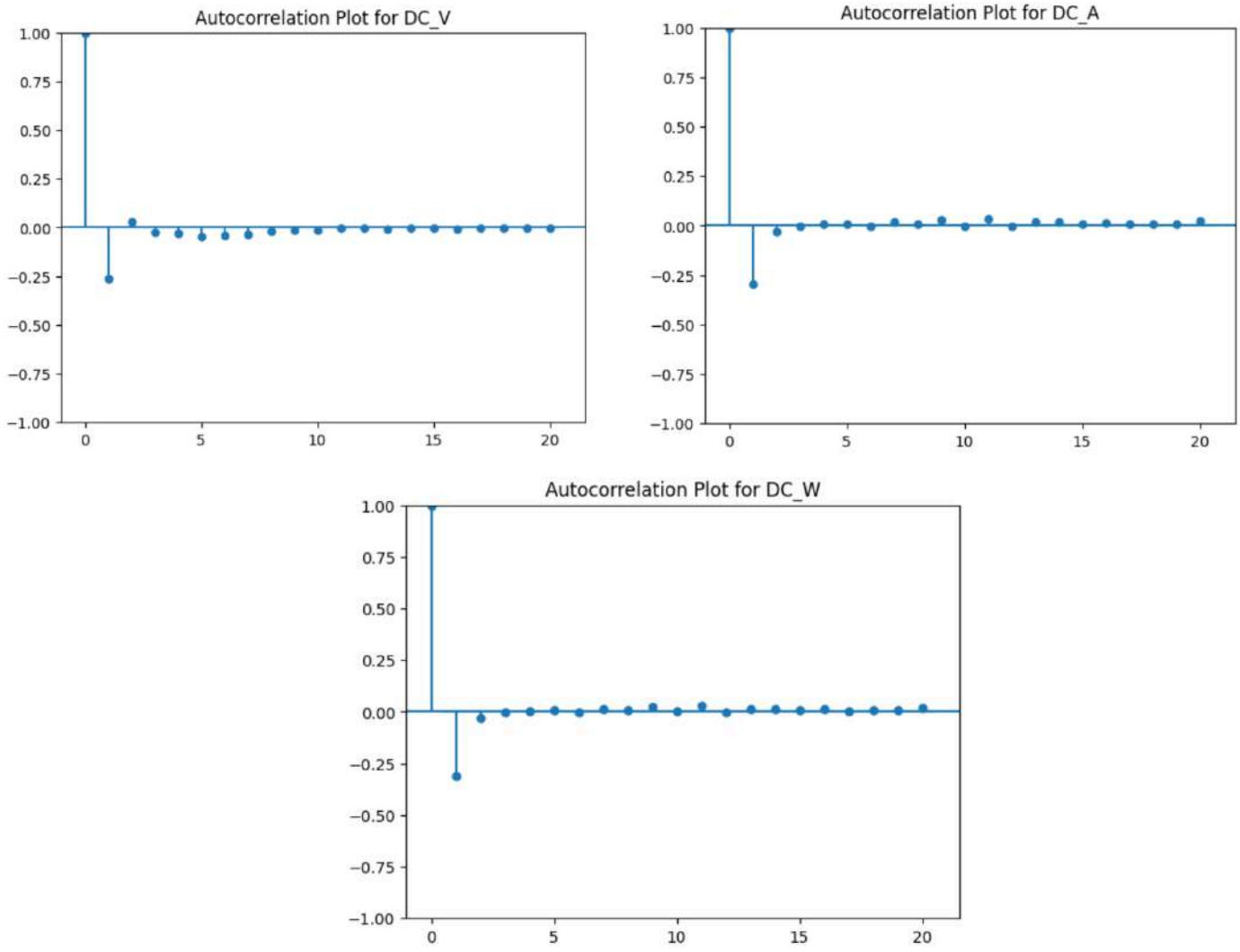


Figure 6. Autocorrelation of DC_V, DC_A, DC_W

Question 3

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the partial autocorrelation lag (lag) using the Statsmodels pacf() function and plot_acf().).

This code prepares the data by calculating the first-order difference for each column of interest, which is a common preprocessing step in time series analysis to stabilize the mean of the data and make it more amenable to modeling and analysis.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import pacf
from statsmodels.graphics.tsaplots import plot_pacf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Import: It imports necessary libraries such as Pandas for data manipulation, Matplotlib for plotting, and specific functions for calculating partial autocorrelation **(pacf)** and plotting partial autocorrelation plots **(plot_pacf)** from statsmodels.

Load Data: It loads the solar power generation data from the provided CSV file located at '/kaggle/input/solar-power-generation-data/pv_2years_eng.csv' into a Pandas DataFrame named **data.**

Defining Columns of Interest:
It defines a list **columns_of_interest** containing the names of the columns of interest, which are 'DC_V', 'DC_A', and 'DC_W'. These columns likely represent different aspects of solar power generation data.

Ensuring Stationarity of the Time Series:
It ensures stationarity of the time series data for each column of interest by calculating the first-order difference. This is done by applying the diff() function to each column of interest and storing the result in new columns with the suffix '_diff'. For example, for the 'DC_V' column, it creates a new column named 'DC_V_diff' containing the first-order differences of the 'DC_V' column. This process is performed in a loop over each column of interest.

This structure enables visual inspection of the first-order difference and autocorrelation of each column, which are essential steps in time series analysis to understand the data's temporal patterns and dependencies.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate partial autocorrelation lag using pacf() function
for column in columns_of_interest:
    pacf_result = pacf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Partial Autocorrelation for {column}: {pacf_result}')

# Plot partial autocorrelation using plot_pacf() function
for column in columns_of_interest:
    plot_pacf(data[column + '_diff'].dropna(), lags=20, title=f'Partial Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference: It plots the first-order difference of each column of interest. It first creates a figure with a specified size using **plt.figure(figsize=(12, 6*len(columns_of_interest)))**. Then, it iterates over each column of interest using a for loop. Within each iteration:

- It creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)**, where i is the current index of the subplot.
- It plots the first-order difference of the current column against the index (presumably representing time) using **plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')**.
- Sets the title of the subplot to indicate the column name and the fact that it's the first-order difference using **plt.title(f'{column} First-order Difference')**.
- Labels the x-axis as 'Date' and y-axis as 'Difference' using **plt.xlabel('Date')** and **plt.ylabel('Difference')**.
- Adds a legend to the subplot using **plt.legend()** to indicate which curve corresponds to which column.

Calculating Partial Autocorrelation:
It calculates the partial autocorrelation of the first-order difference for each column of interest using the **pacf()** function from statsmodels. It computes the partial autocorrelation up to a specified number of lags (in this case, 20 lags).

Plotting Partial Autocorrelation:
It plots the partial autocorrelation of the first-order difference for each column of interest using the **plot_pacf()** function from statsmodels. It generates a separate plot for each column with the title indicating the column name.

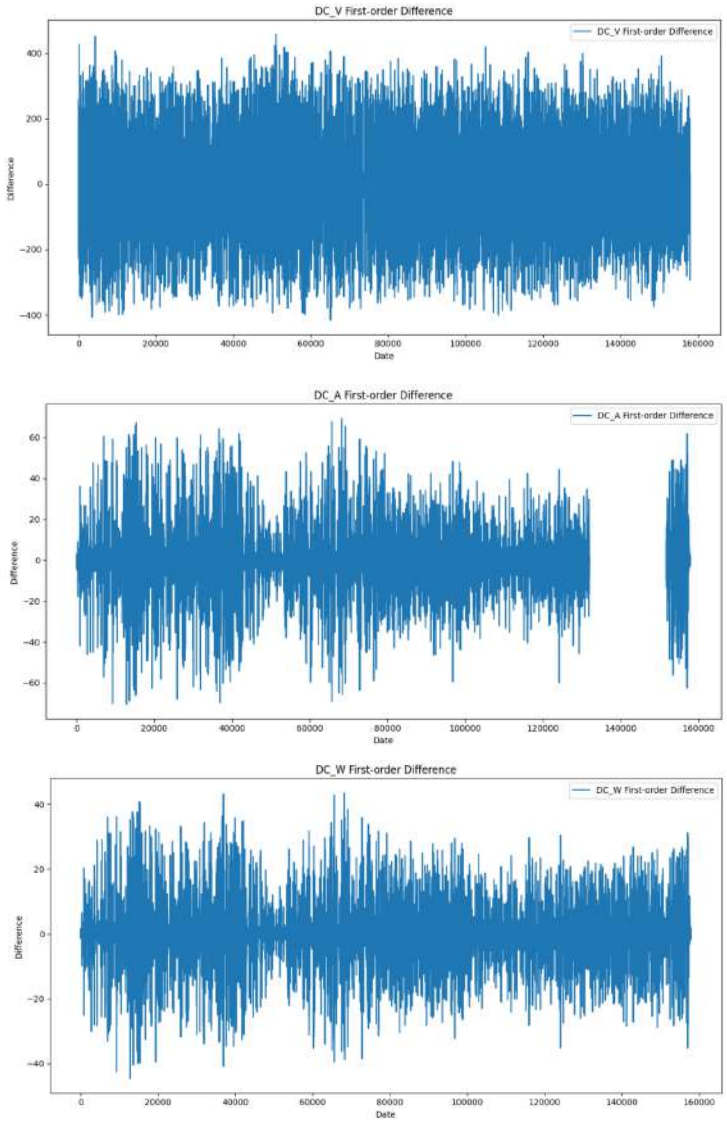


Figure 7. First-order difference of DC_V, DC_A, DC_W

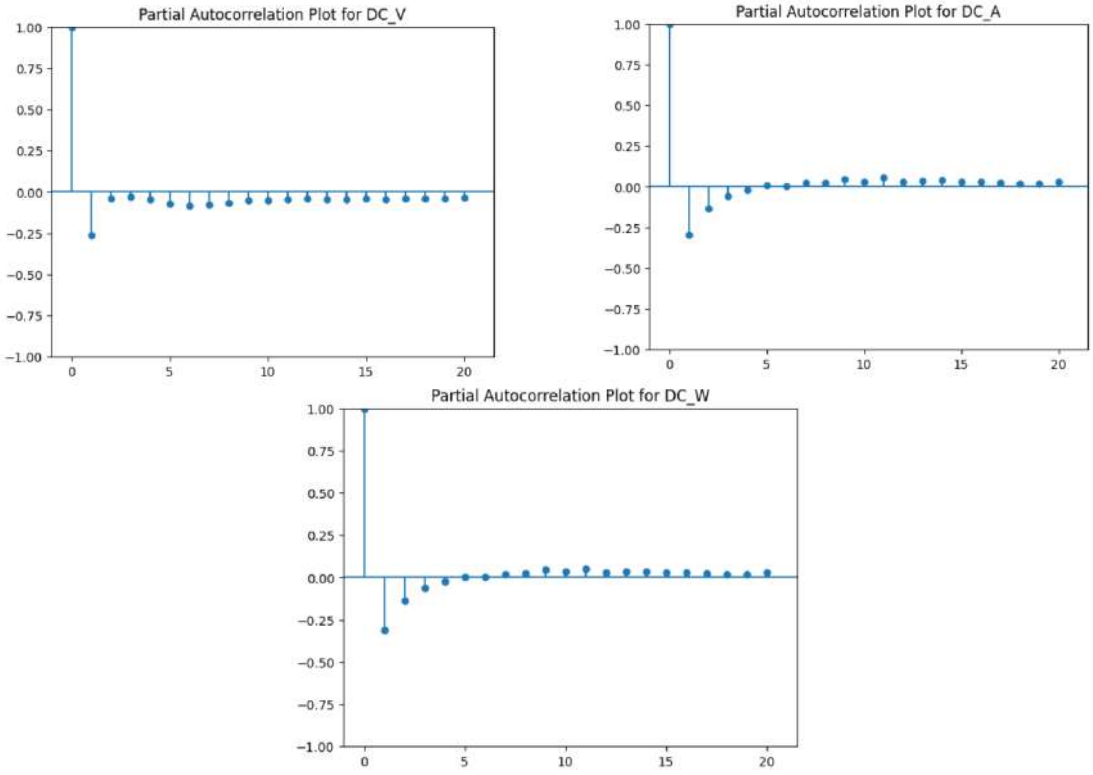


Figure 8. Partial Autocorrelation of DC_V, DC_A, DC_W

Question 4

Using solar power generation data, pv_2years_eng, implement it as a moving average (MA), and display the moving average prediction results as plot().

This code essentially visualizes the original data and the moving average prediction results for each specified column separately.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Plot original data and moving average for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))

    # Plot original data
    plt.plot(data.index, data[column], label=f'Original {column}')

    # Implement moving average (MA)
    window_size = 10  # Specify the window size for the moving average
    ma = data[column].rolling(window=window_size).mean()
    plt.plot(data.index, ma, label=f'{column} MA ({window_size} periods)')

    plt.title(f'Moving Average Prediction Results for {column}')
    plt.xlabel('Date')
    plt.ylabel('Value')
    plt.legend()
    plt.show()
```

Import: It imports necessary libraries such as Pandas for data manipulation, Matplotlib for plotting, and specific functions.

Load Data: It loads the solar power generation data from the specified CSV file using pandas and assigns it to the variable **data**.

Plot Original Data and Moving Average: It iterates over each column of interest. For each column:

- It creates a new figure with a specified size using **plt.figure(figsize=(12, 6)).**
- It plots the original data against the index of the DataFrame using **plt.plot(data.index, data[column], label=f'Original {column}').**
- It calculates the moving average (MA) for the current column using a rolling window with a specified size **(window_size).** The moving average is plotted against the index of the DataFrame.
- It sets the title, xlabel, ylabel, and legend for the plot.
- It displays the plot using **plt.show().**

Defining Columns of Interest:
It defines a list **columns_of_interest** containing the names of the columns for which we want to analyze the moving average. In this case, the columns of interest are 'DC_V', 'DC_A', and 'DC_W', representing DC voltage, DC current, and DC output, respectively.

This code essentially visualizes the original data and the combined moving average prediction results for all specified columns together.

```python
# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Plot original data and combined moving average for all columns
plt.figure(figsize=(12, 6))

# Plot original data for each column
for column in columns_of_interest:
    plt.plot(data.index, data[column], label=f'Original {column}')

# Calculate and plot combined moving average
window_size = 10  # Specify the window size for the moving average
combined_ma = data[columns_of_interest].rolling(window=window_size).mean().mean(axis=1)
plt.plot(data.index, combined_ma, label=f'Combined MA ({window_size} periods)', color='black', linestyle='--')

plt.title('Combined Moving Average Prediction Results for DC Voltage, DC Current, and DC Output')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

Define Columns of Interest: It defines a list **columns_of_interes**t containing the names of the columns for which we want to analyze the moving average. In this case, the columns of interest are 'DC_V', 'DC_A', and 'DC_W', representing DC voltage, DC current, and DC output, respectively.

Plot Original Data and Combined Moving Average:

- It creates a new figure with a specified size using **plt.figure(figsize=(12, 6)).**
- It iterates over each column of interest. For each column:
  - It plots the original data against the index of the DataFrame using **plt.plot(data.index, data[column], label=f'Original {column}').**
- It calculates the combined moving average for all columns by first calculating the moving average for each column using a rolling window with a specified size **(window_size).** Then, it calculates the mean of the moving averages across all columns. The combined moving average is plotted against the index of the DataFrame.
- It sets the title, xlabel, ylabel, and legend for the plot.
- It displays the plot using **plt.show().**

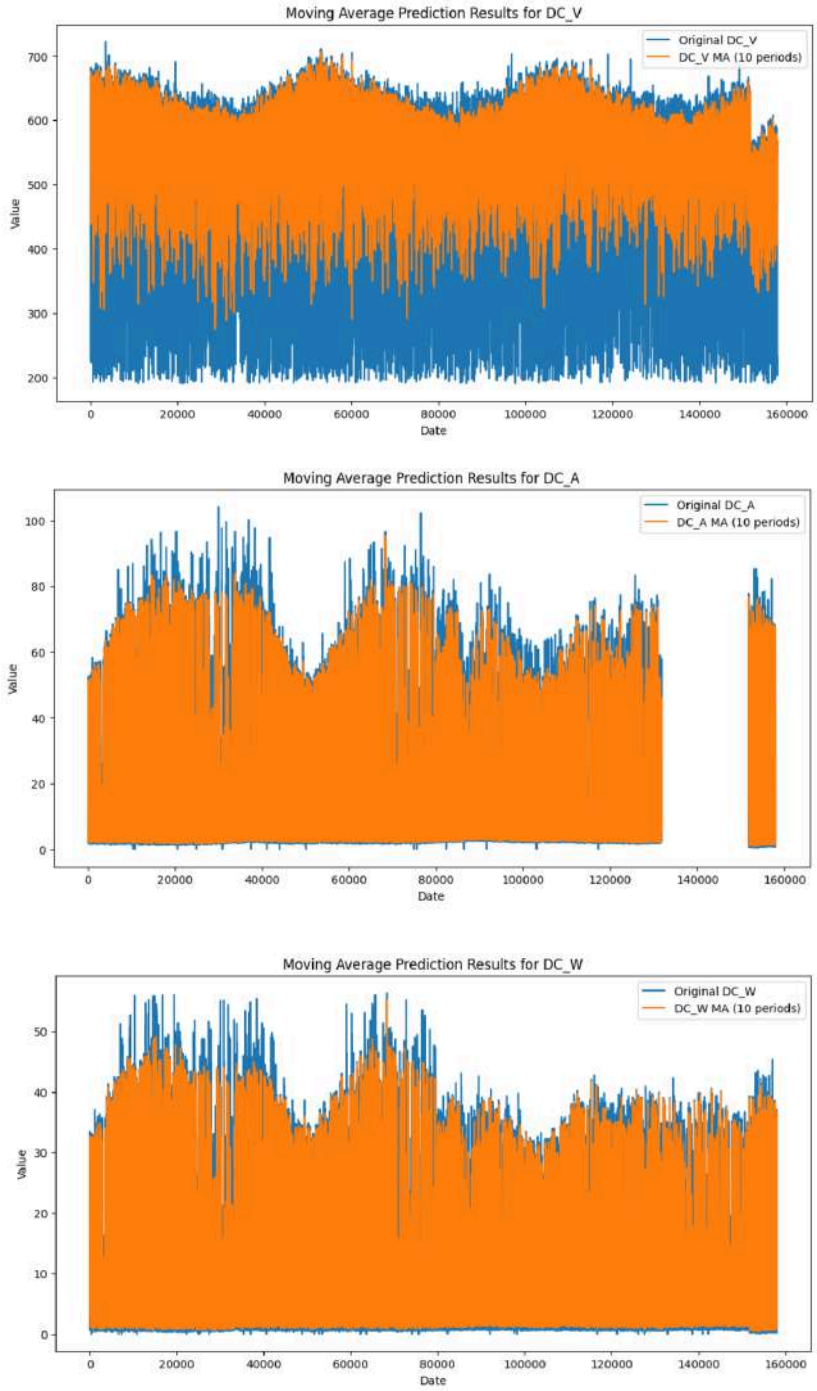


Figure 9. Moving Average of DC_V, DC_A and DC_W

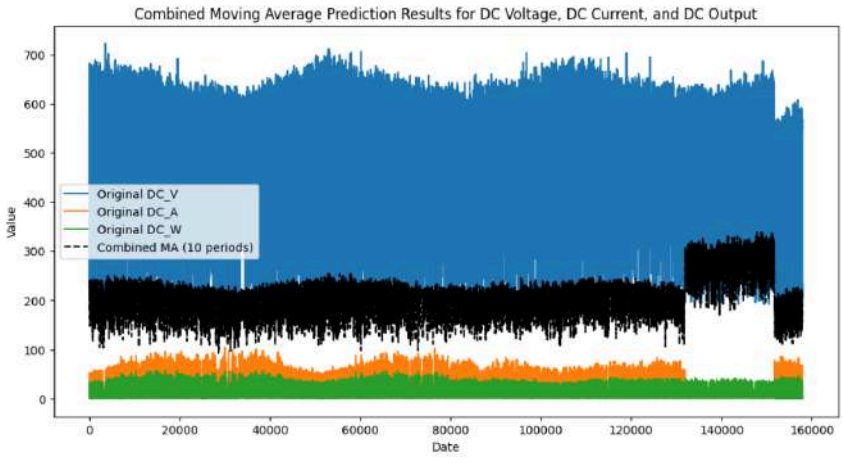


Figure 10. Combined Moving Average of DC_V, DC_A and DC_W

Question 5

Using solar power generation data, pv_2years_eng, implement triple exponential smoothing and display the triple exponential smoothing results with plot().

This code efficiently applies triple exponential smoothing to multiple time series columns in the dataset and stores the predictions for each column separately for further analysis or visualization.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv', parse_dates=['date'])

# Prepare the data
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']

# Implement triple exponential smoothing (Holt-Winters method) for each column
predictions = {}

for column in columns_of_interest:
    model = ExponentialSmoothing(data[column], trend='add', seasonal='add', seasonal_periods=12)
    result = model.fit()
    predictions[column] = result.predict(start=data.index[0], end=data.index[-1])
```

Import Libraries: The code imports necessary libraries:

- **pandas** for handling data frames
- **matplotlib.pyplot** for plotting
- **ExponentialSmoothing** from **statsmodels.tsa.holtwinters** module for implementing triple exponential smoothing.

Load and Prepare Data:

- The data is loaded from a CSV file.
- The '**date**' column is parsed as dates during the loading process.
- The index of the DataFrame is set to the '**date**' column.

Define Columns of Interest:
A list named columns_of_interest is defined, containing the column names for which triple exponential smoothing will be applied.

Triple Exponential Smoothing Implementation:

- A loop iterates over each column specified in **columns_of_interest.**
- For each column, an **ExponentialSmoothing** model is instantiated with the following parameters:
  - **data[column]:** The time series data for the current column.
  - **trend='add'**: Adding a linear trend component to the model.
  - **seasonal='add'**: Adding a seasonal component to the model.
  - **seasonal_periods**=12: Specifying the seasonal period (e.g., for monthly data, the period is 12).
- The **fit()** method is called on the model to fit it to the data and obtain the predictions.
- The predictions are stored in a dictionary named **predictions**, where the keys are column names and the values are the corresponding predicted time series data.

This code essentially visualizes the original data and the combined moving average prediction results for all specified columns together.

```python
# Plot the predictions for each column separately
plt.figure(figsize=(12, 6))

# Plot DC Voltage predictions
plt.plot(data.index, data['DC_V'], label='Original DC Voltage')
plt.plot(predictions['DC_V'].index, predictions['DC_V'], label='DC Voltage Predictions')

plt.title('Triple Exponential Smoothing Predictions for DC Voltage')
plt.xlabel('Date')
plt.ylabel('DC Voltage')
plt.legend()
plt.grid(True)
plt.show()

# Plot DC Current predictions
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['DC_A'], label='Original DC Current')
plt.plot(predictions['DC_A'].index, predictions['DC_A'], label='DC Current Predictions')

plt.title('Triple Exponential Smoothing Predictions for DC Current')
plt.xlabel('Date')
plt.ylabel('DC Current')
plt.legend()
plt.grid(True)
plt.show()

# Plot DC Output predictions
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['DC_W'], label='Original DC Output')
plt.plot(predictions['DC_W'].index, predictions['DC_W'], label='DC Output Predictions')

plt.title('Triple Exponential Smoothing Predictions for DC Output')
plt.xlabel('Date')
plt.ylabel('DC Output')
plt.legend()
plt.grid(True)
plt.show()
```

Plotting Predictions:

- The code creates three separate plots, one for each column of interest: DC Voltage, DC Current, and DC Output.
- Each plot is set to a figure size of 12x6 inches.

Plotting Original Data:

- For each column, the original time series data is plotted using **plt.plot()** function.
- The x-axis represents the dates, while the y-axis represents the values of the respective columns.
- The original data is plotted with a specific label for each column.

Plot Customization:

- Each plot has a title indicating the type of data (e.g., DC Voltage, DC Current, DC Output).
- The x-axis label is set to 'Date', and the y-axis label varies depending on the column being plotted.
- Legends are added to indicate the original data and the predictions.
- Grid lines are enabled for better visualization.

Displaying the Plots:

- Each plot is displayed using **plt.show().**

Plotting Predicted Data:

- For each column, the predictions obtained from triple exponential smoothing are plotted using **plt.plot()** function.
- The x-axis represents the dates, while the y-axis represents the predicted values of the respective columns.
- The predicted data is plotted with a specific label for each column.

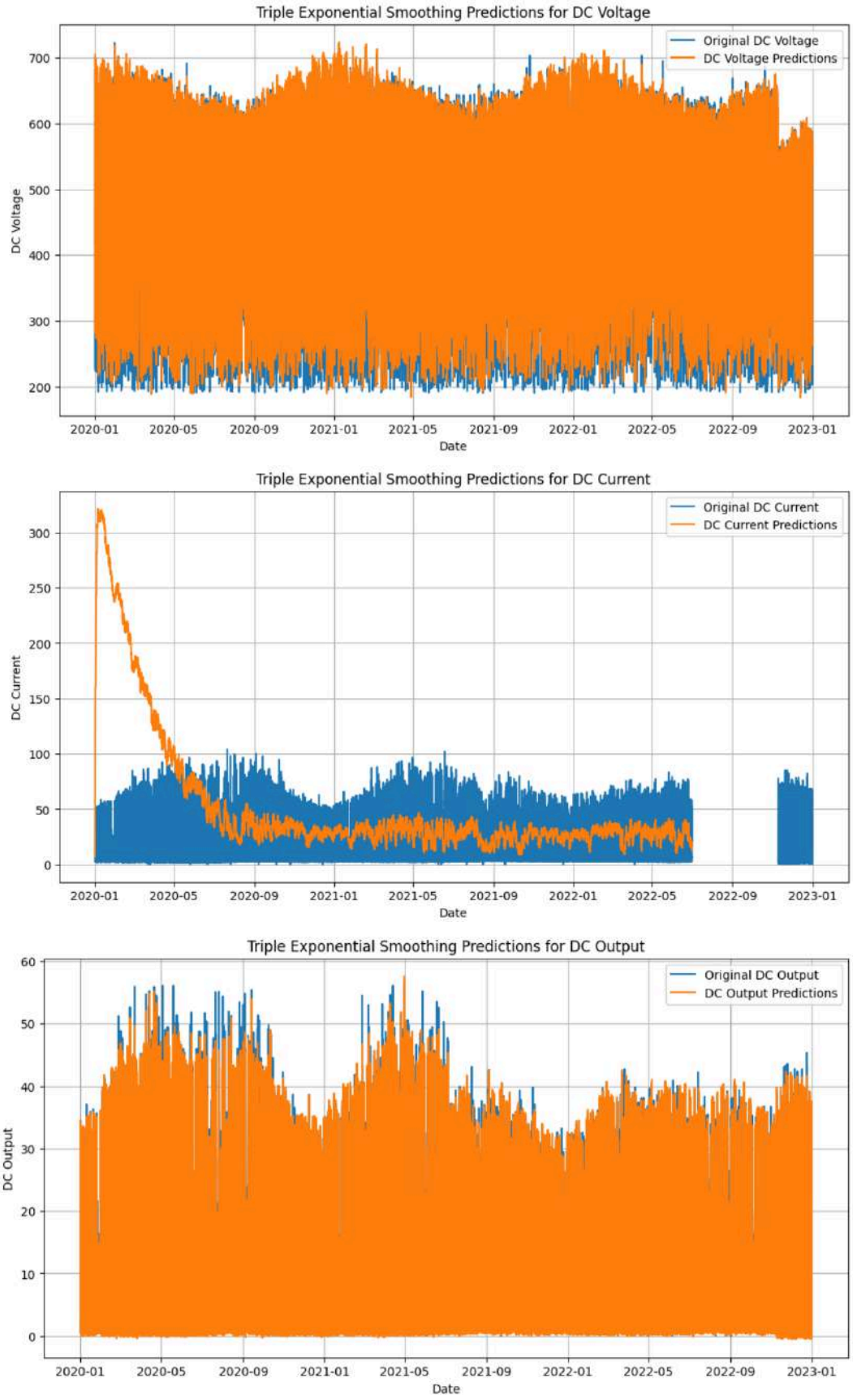


Figure 11. Triple Exponential Smooting Predictions of DC_V, DC_A and DC_W

Question 6

Using solar power generation data, pv_2years_eng, implement autoregressive (AR) and display the autoregressive (AR) results in plot().

This code segment sets the stage for autoregressive modeling by loading the data and defining the columns of interest for analysis. The actual autoregressive modeling and plotting of results are not included in this snippet.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AutoReg

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['DC_V', 'DC_A', 'DC_W']
```

Importing Libraries: It imports the necessary libraries:

- **pandas** as pd for data manipulation.
- **matplotlib.pyplot** as plt for plotting.
- **AutoReg** from **statsmodels.tsa.ar_model** for autoregressive modeling.

Loading Data:

- It loads the solar power generation data from the specified CSV file using **pd.read_csv().**
- The loaded data is stored in the DataFrame named **data.**

Defining Columns of Interest:

- It specifies the columns of interest for which autoregressive modeling will be performed.
- The columns of interest are defined in the list **columns_of_interest**, which includes 'DC_V', 'DC_A', and 'DC_W'.

This code allows for the visualization of autoregressive predictions for each column of interest in the dataset.

```python
# Plot autoregressive (AR) results for each column
for column in columns_of_interest:
    # Fit autoregressive (AR) model
    model = AutoReg(data[column].dropna(), lags=1)  # Using lag 1 for simplicity
    model_fit = model.fit()

    # Make predictions
    predictions = model_fit.predict(start=1, end=len(data[column]))

    # Plot original data and autoregressive (AR) predictions
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label='Original Data')
    plt.plot(data.index, predictions, label='Autoregressive (AR) Predictions', color='orange')

    plt.title(f'Autoregressive (AR) Model Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Looping Over Columns of Interest:

- It iterates over each column specified in the **columns_of_interest** list.

Fitting Autoregressive (AR) Model:

- For each column, it fits an autoregressive (AR) model using the AutoReg function from **statsmodels.tsa.ar_model.**
- The **lags** parameter is set to 1 for simplicity, meaning it's a first-order autoregressive model.
- The model is fitted to the non-null values of the column using the **fit()** method.

Plotting Predictions:

- The code creates three separate plots, one for each column of interest: DC Voltage, DC Current, and DC Output.
- Each plot is set to a figure size of 12x6 inches.

Making Predictions:

- After fitting the model, it makes predictions using the **predict()** method.
- The predictions are made for the entire length of the data, starting from the second data point **(start=1)** to the end.

Plotting Original Data and Predictions:

- For each column, it creates a new figure with a size of 12x6 inches.
- It plots the original data and the autoregressive predictions on the same plot.
- The original data is plotted using **plt.plot(data.index, data[column], label='Original Data').**
- The autoregressive predictions are plotted using **plt.plot(data.index, predictions, label='Autoregressive (AR) Predictions', color='orange').**
- It adds a title, xlabel, ylabel, legend, and grid to the plot for better visualization.
- Finally, it displays the plot using **plt.show().**

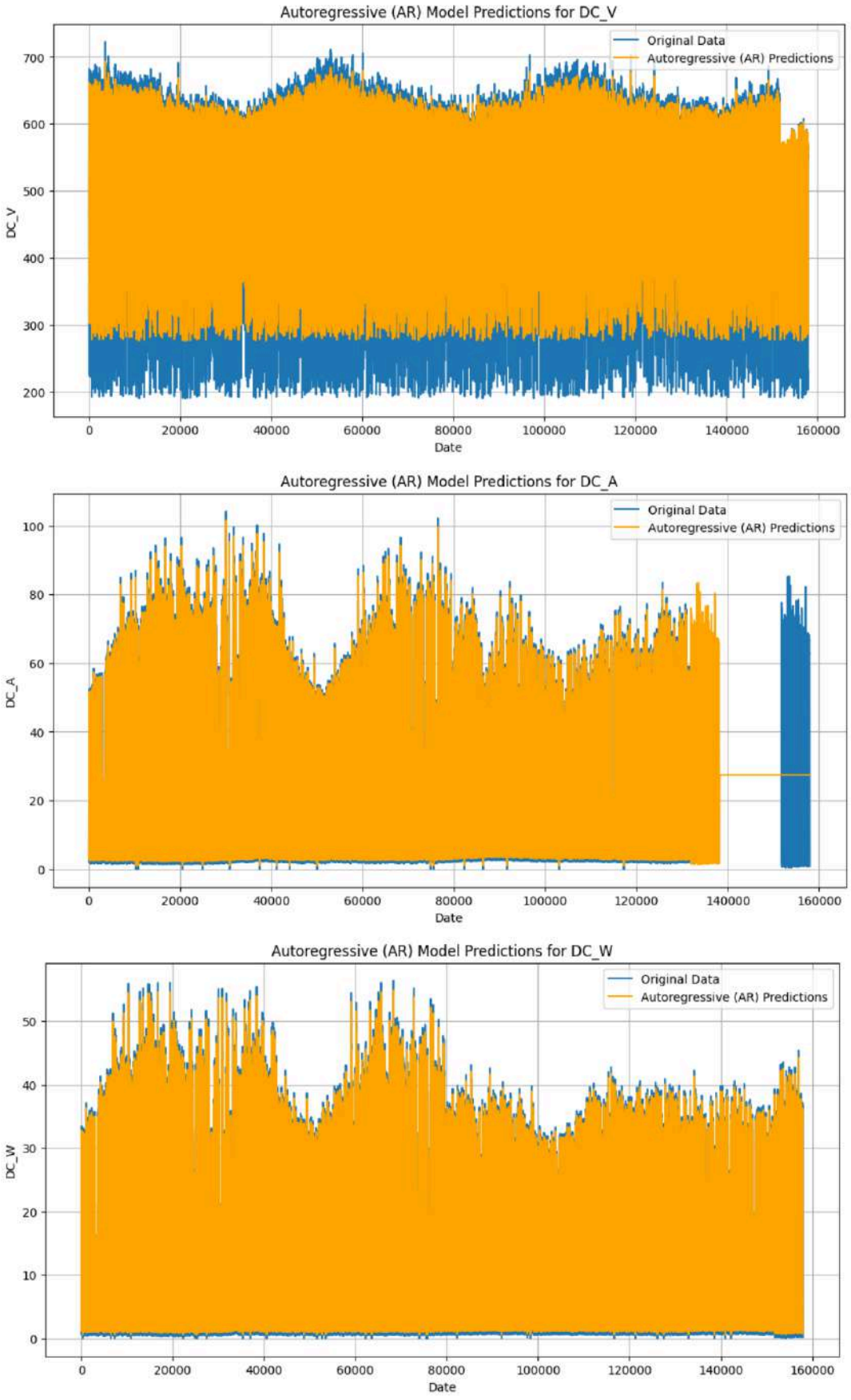


Figure 12. Autoregressive (AR) Model Predictions of DC_V, DC_A and DC_W

2 AC GROUP

Several columns in the solar panel data offer crucial details about the alternating current (AC) produced by the solar power system. The columns 'AC_V_RS', 'AC_V_ST', and 'AC_V_TR' denote the AC voltage in phases R to S, S to T, and T to R, correspondingly. By capturing the electrical potential changes between the various AC output phases, these voltage measurements provide information about the stability and quality of the electrical output produced by the solar panels. Furthermore, the AC currents in phases R, S, and T are shown by the columns "AC_A_R," "AC_A_S," and "AC_A_T," respectively. The electric charge flow through each phase is quantified by these current values, which help evaluate the efficiency of power generation and delivery. Moreover, the AC output—which represents the actual electrical power produced by the solar panels—is shown in the 'AC_W' column. The variable 'AC_TOT' represents the total power generated over a given period of time, offering a full measurement of the energy generated. Furthermore, 'AC_FREQ' indicates the AC frequency, which is essential to guaranteeing that electrical devices are in sync with the power source. Last but not least, the value 'AC_POW' denotes the AC power factor, which shows how well power is used and transmitted throughout the system. When combined, these columns provide a thorough picture of the solar panel system's AC output characteristics and performance, facilitating efficient monitoring, analysis, and operation optimization.

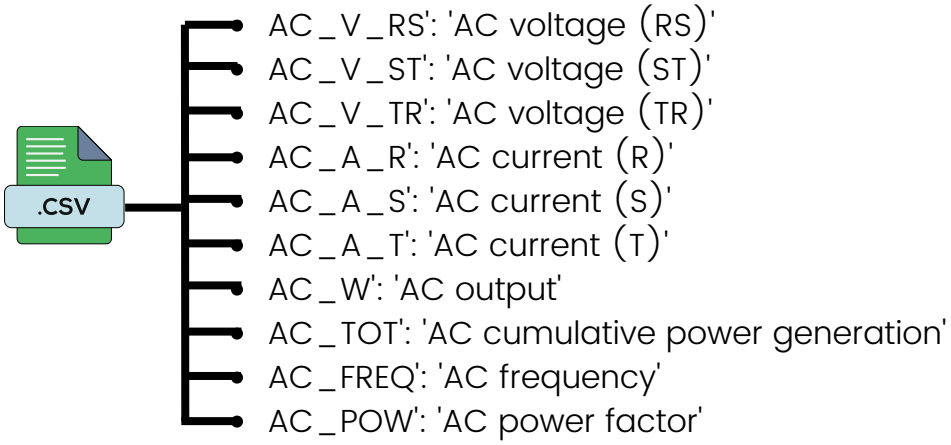
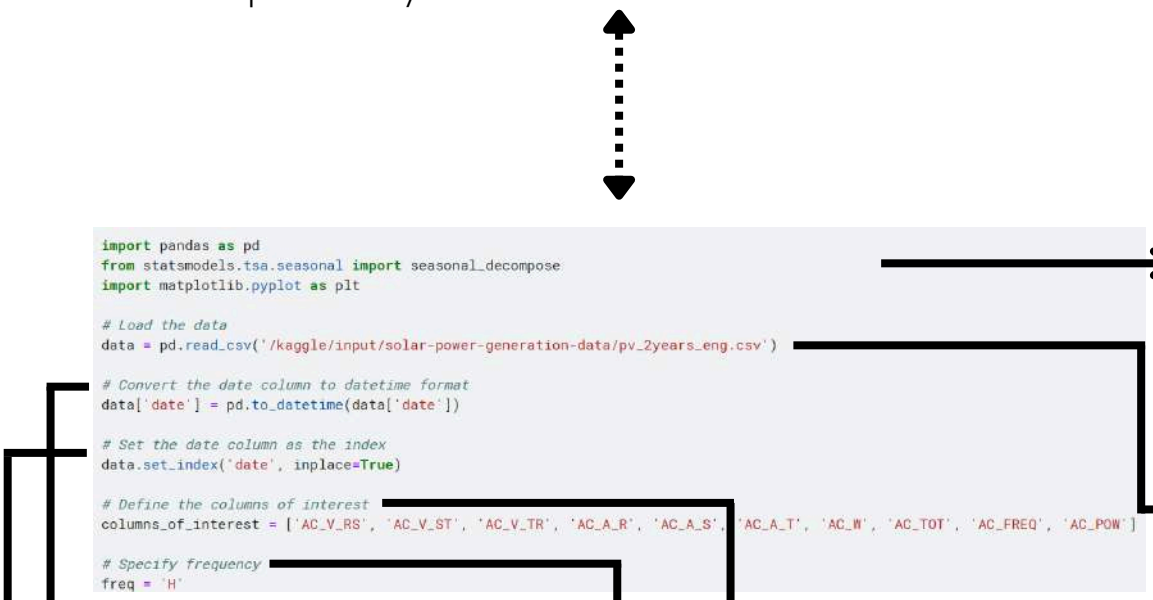


Figure 13. Analysis 2

Question 1

Using solar power generation data, pv_2years_eng, decompose the time series into trend, seasonal, and residual components and display them with plot(). At this time, decompose using the additive vs. multiplicative model and present the results.

This code segment prepares the solar power generation data by converting the date column to datetime format, setting it as the index, and defining the columns of interest for subsequent analysis.



```
import pandas as pd
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Convert the date column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Set the date column as the index
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S', 'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']

# Specify frequency
freq = 'H'
```

Import Libraries: The code imports the necessary libraries, including pandas for data manipulation, statsmodels for time series analysis (specifically seasonal decomposition), and matplotlib for visualization.

Load Data: The code loads the solar power generation data from a CSV file named 'pv_2years_eng.csv' located at the specified path using the pandas **'read_csv()'** function. The loaded data is stored in a DataFrame named **'data'**.

Convert Date Column to Datetime: It converts the 'date' column in the DataFrame **'data'** to datetime format using the **'pd.to_datetime()'** function. This conversion ensures that dates are interpreted correctly for time-based operations.

Set Date Column as Index: The 'date' column, now in datetime format, is set as the index of the DataFrame **'data'** using the **'set_index()'** method. Setting the index to the date column facilitates time series analysis and manipulation.

Define Columns of Interest: The code specifies a list named **'columns_of_interest'** containing the names of columns relevant for further analysis. These columns likely represent key metrics related to solar power generation, such as AC voltage, AC current, power output, etc.

Specify Frequency: The variable **'freq'** is assigned the value 'H', indicating that the time series data is at an hourly frequency. This information is essential for certain time series analysis techniques that rely on the frequency of the data, such as seasonal decomposition.

This code efficiently decomposes the time series data for each specified column into its trend, seasonal, and residual components using an additive model and visualizes the results for further analysis. It also handles exceptions gracefully, providing informative error messages if decomposition fails for any column.

```python
# Decompose the time series using both additive and multiplicative models
for column in columns_of_interest:
    try:
        # Additive decomposition
        decomposition_add = seasonal_decompose(data[column], model='additive', period=24, extrapolate_trend='freq')

        # Plot the decomposed components
        plt.figure(figsize=(12, 12))
        plt.subplot(3, 1, 1)
        plt.plot(decomposition_add.trend, label='Additive Trend')
        plt.title('Additive Trend')
        plt.legend()

        plt.subplot(3, 1, 2)
        plt.plot(decomposition_add.seasonal, label='Additive Seasonal')
        plt.title('Additive Seasonal')
        plt.legend()

        plt.subplot(3, 1, 3)
        plt.plot(decomposition_add.resid, label='Additive Residual')
        plt.title('Additive Residual')
        plt.legend()

        plt.subplots_adjust(hspace=0.5)  # Adjust the spacing between subplots

        plt.suptitle(f'Decomposition of {data[column].name} (Additive Model)', fontsize=12)
        plt.show()

    except ValueError as e:
        print(f"Additive decomposition couldn't be performed for column {column}: {str(e)}")
```

For Loop Over Columns of Interest: The code iterates over each column specified in the **'columns_of_interest'** list.

Additive Decomposition:

- Inside the loop, it attempts to perform additive decomposition for each column using the **'seasonal_decompose'** function from statsmodels.
- The **'model'** parameter is set to 'additive' to specify additive decomposition.
- The **'period'** parameter is set to 24, indicating that the data has a daily seasonality.
- **'extrapolate_trend'** is set to 'freq' to ensure that the trend is extrapolated to cover the entire time range.

Exception Handling:

- The code uses a try-except block to handle any ValueError exceptions that may occur during the decomposition process.
- If an exception occurs, it prints an error message indicating that additive decomposition couldn't be performed for the specific column.

Plotting Decomposed Components:

- For each column, it plots the trend, seasonal, and residual components of the additive decomposition.
- It creates a vertical layout with three subplots stacked vertically (3 rows, 1 column) using **'plt.subplot()'**.
- Each subplot shows one component of the decomposition (trend, seasonal, or residual).
- The title of each subplot indicates the specific component being plotted (e.g., 'Additive Trend', 'Additive Seasonal').
- Legends are added to the plots to indicate the corresponding decomposition component.
- The suptitle describes the decomposition being performed for the specific column using an additive model.

This code segment efficiently performs multiplicative decomposition on the time series data for each specified column, plots the decomposed components, and handles any exceptions gracefully, providing informative error messages if decomposition fails for any column.

```python
try:
    # Multiplicative decomposition
    decomposition_mul = seasonal_decompose(data[column], model='multiplicative', period=24, extrapolate_trend='freq')

    # Plot the decomposed components
    plt.figure(figsize=(12, 12))
    plt.subplot(3, 1, 1)
    plt.plot(decomposition_mul.trend, label='Multiplicative Trend')
    plt.title('Multiplicative Trend')
    plt.legend()

    plt.subplot(3, 1, 2)
    plt.plot(decomposition_mul.seasonal, label='Multiplicative Seasonal')
    plt.title('Multiplicative Seasonal')
    plt.legend()

    plt.subplot(3, 1, 3)
    plt.plot(decomposition_mul.resid, label='Multiplicative Residual')
    plt.title('Multiplicative Residual')
    plt.legend()

    plt.subplots_adjust(hspace=0.5)  # Adjust the spacing between subplots

    plt.suptitle(f'Decomposition of {data[column].name} (Multiplicative Model)', fontsize=16)
    plt.show()

except ValueError as e:
    print(f"Multiplicative decomposition couldn't be performed for column {column}: {str(e)}")
```

Multiplicative Decomposition:

- Inside a try block, it attempts to perform multiplicative decomposition for each column using the seasonal_decompose function from statsmodels.
- The model parameter is set to 'multiplicative' to specify multiplicative decomposition.
- The period parameter is set to 24, indicating that the data has a daily seasonality.
- extrapolate_trend is set to 'freq' to ensure that the trend is extrapolated to cover the entire time range.

Exception Handling:

- The code uses a except block to handle any ValueError exceptions that may occur during the decomposition process.
- If an exception occurs, it prints an error message indicating that multiplicative decomposition couldn't be performed for the specific column.

Plotting Decomposed Components:

- If the multiplicative decomposition is successful, it plots the trend, seasonal, and residual components of the multiplicative decomposition.
- It creates a vertical layout with three subplots stacked vertically (3 rows, 1 column) using '**plt.subplot()**'.
- Each subplot shows one component of the decomposition (trend, seasonal, or residual).
- The title of each subplot indicates the specific component being plotted (e.g., 'Multiplicative Trend', 'Multiplicative Seasonal').
- Legends are added to the plots to indicate the corresponding decomposition component.
- The suptitle describes the decomposition being performed for the specific column using a multiplicative model.

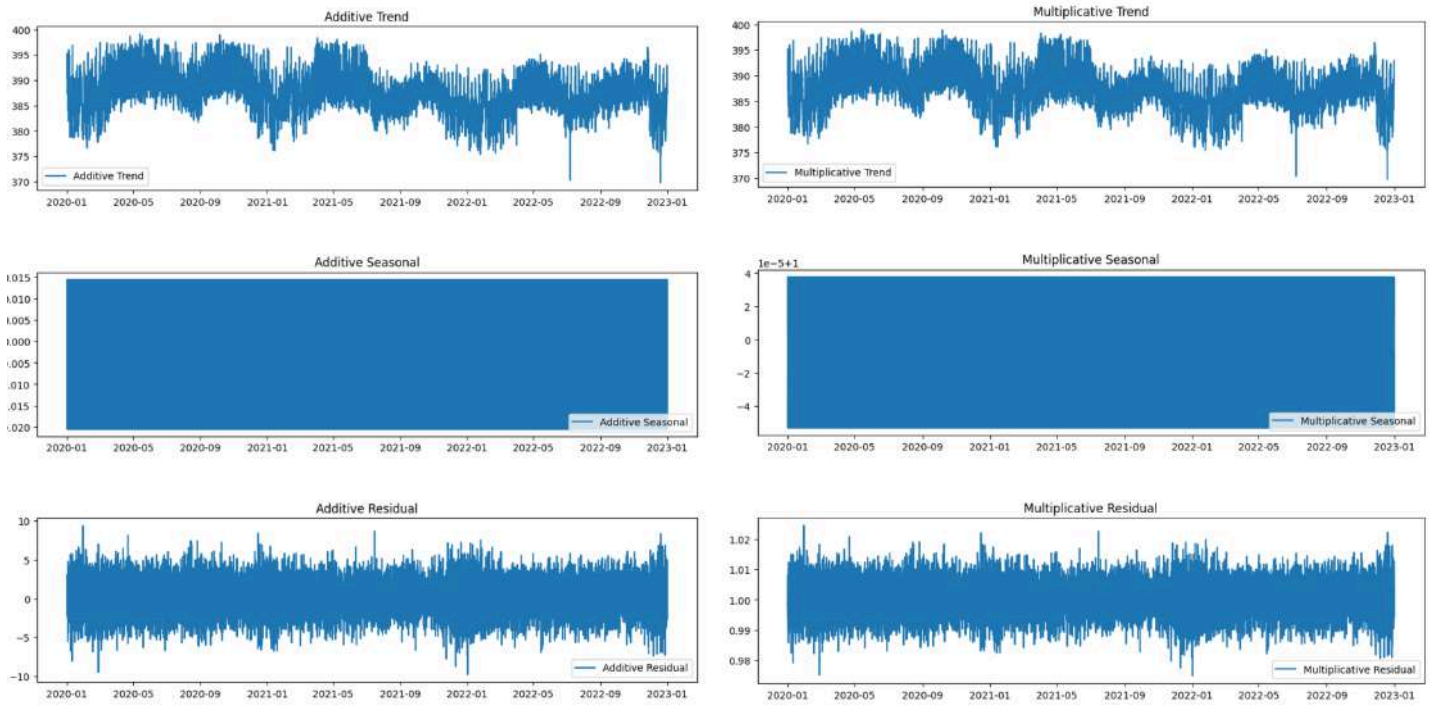


Figure 14. Decomposition of AC_V_RS (Additive and Multiplicative Model)

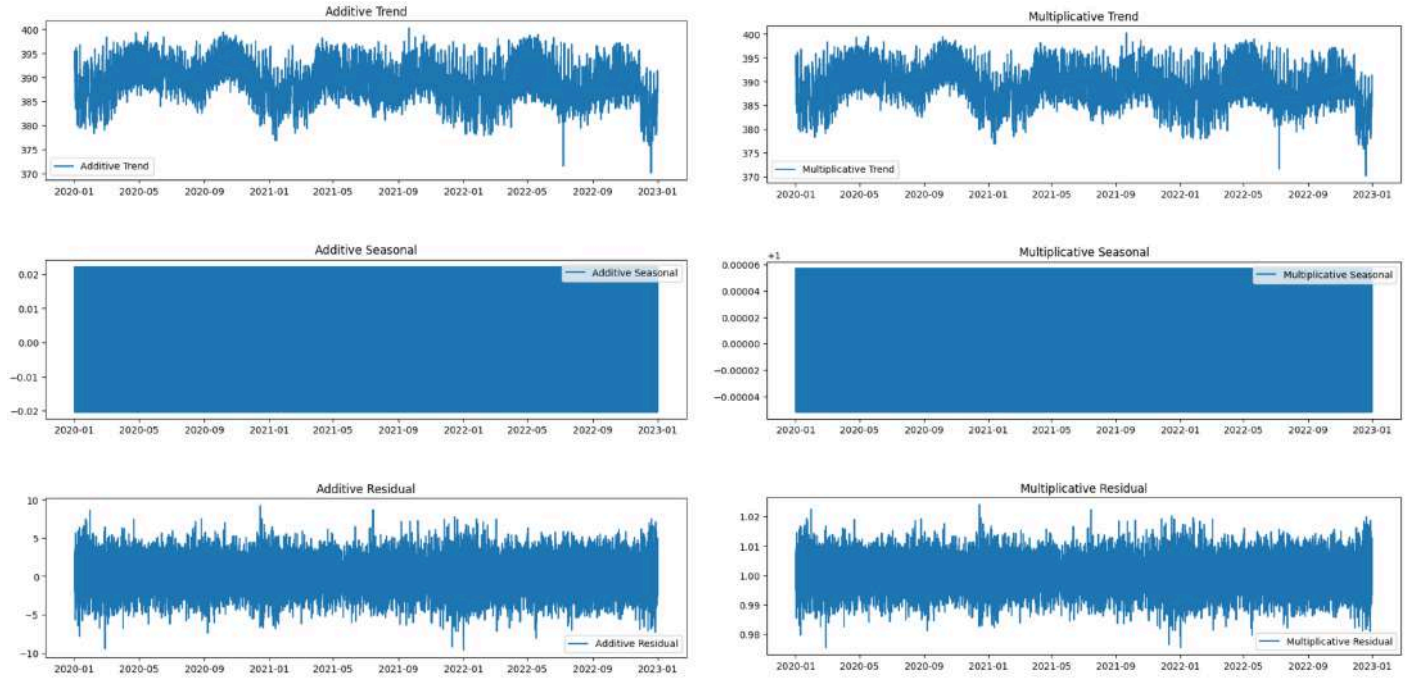


Figure 15. Decomposition of AC_V_ST (Additive and Multiplicative Model)

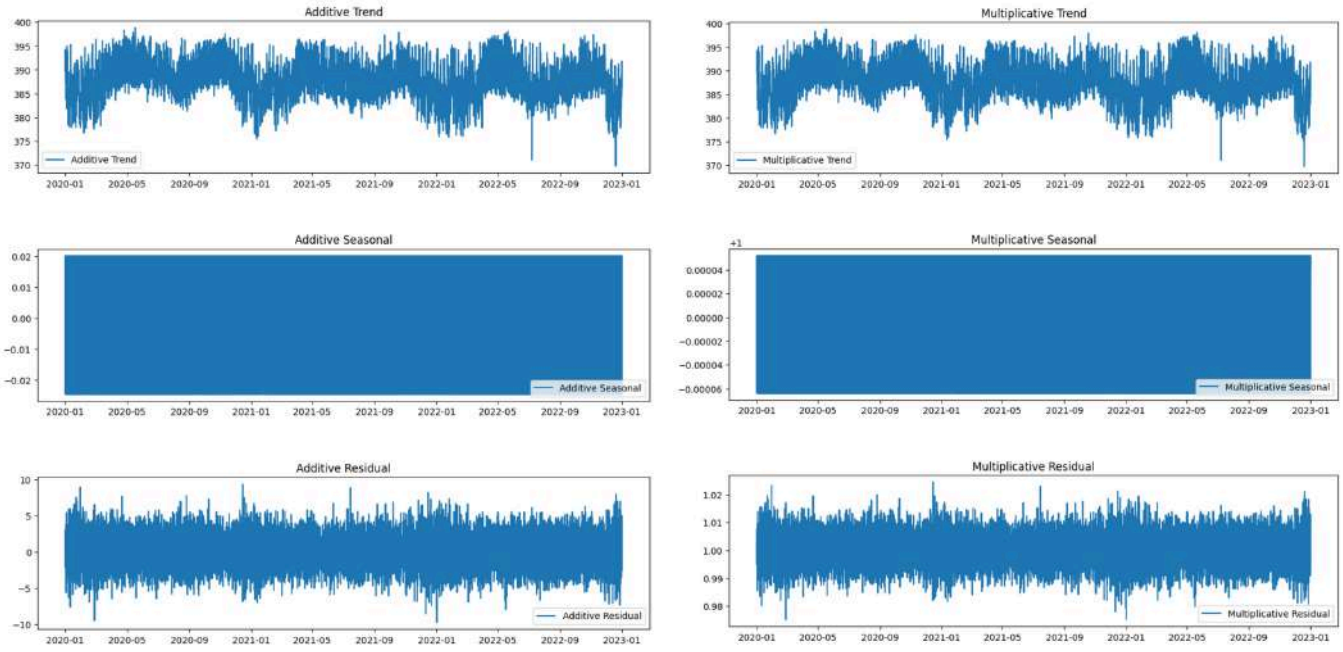
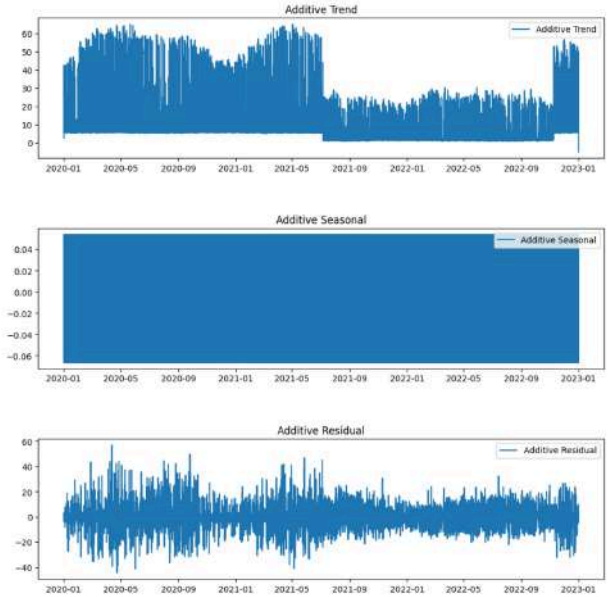
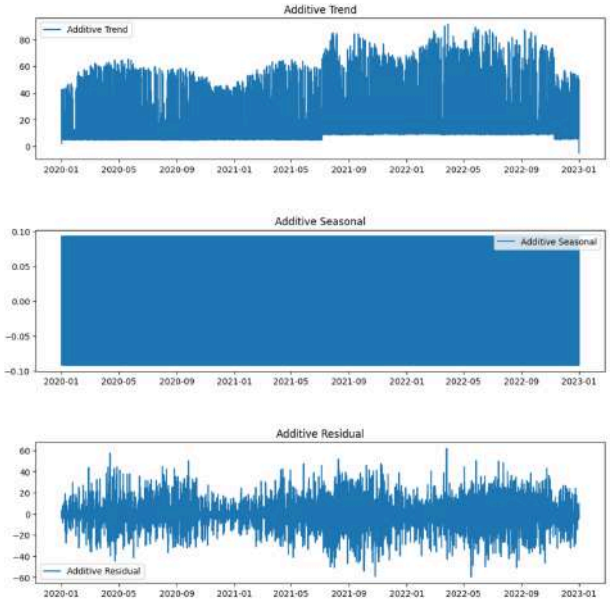


Figure 16. Decomposition of AC_V_TR (Additive and Multiplicative Model)



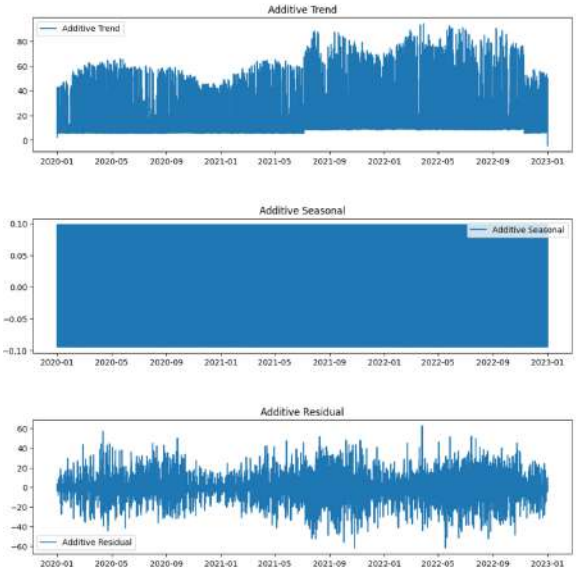
Multiplicative decomposition couldn't be performed for column AC_A_T: Multiplicative seasonality is not appropriate for zero and negative values

Figure 17. Decomposition of AC_A_R (Additive Model)



Multiplicative decomposition couldn't be performed for column AC_A_S: Multiplicative seasonality is not appropriate for zero and negative values

Figure 18. Decomposition of AC_A_S (Additive Model)



Multiplicative decomposition couldn't be performed for column AC_A_T: Multiplicative seasonality is not appropriate for zero and negative values

Figure 19. Decomposition of AC_A_T (Additive Model)

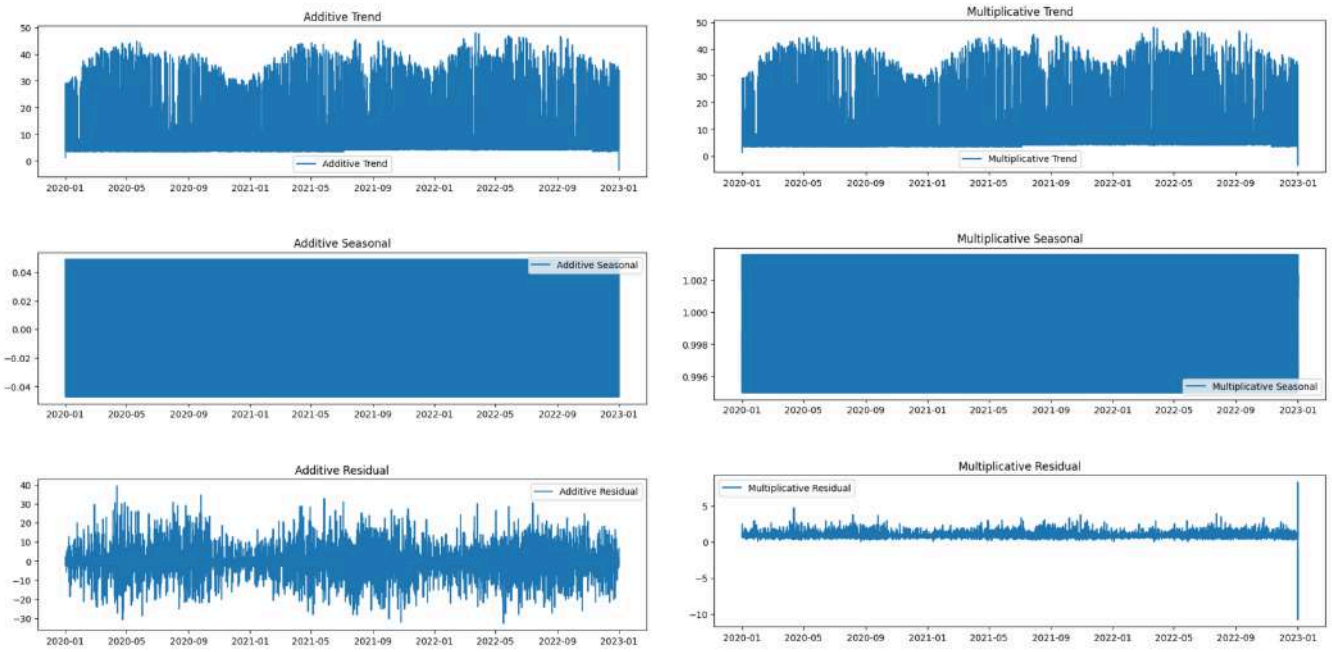
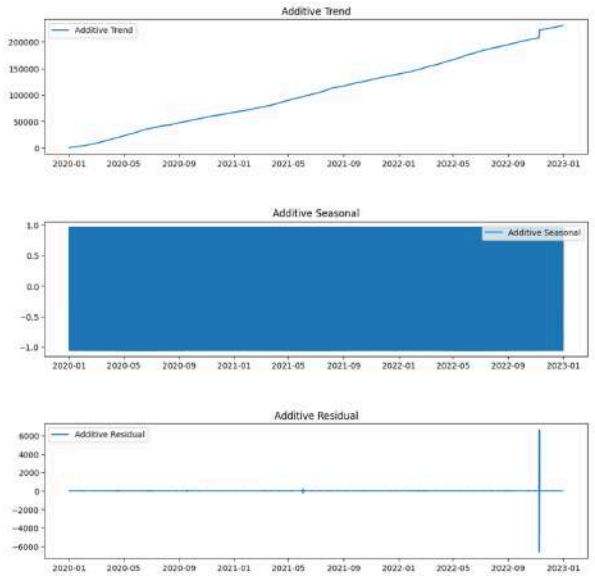


Figure 20. Decomposition of AC_W (Additive and Multiplicative Model)



Multiplicative decomposition couldn't be performed for column AC_TOT: Multiplicative seasonality is not appropriate for zero and negative values

Figure 21. Decomposition of AC_TOT (Additive Model)

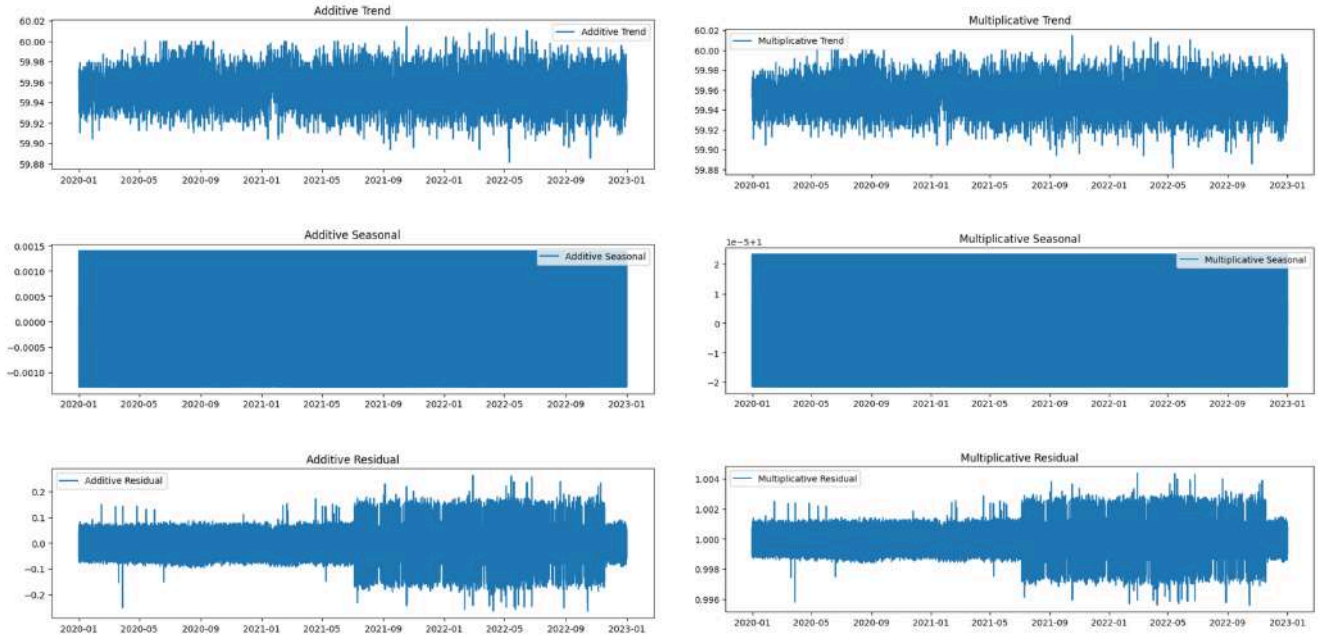
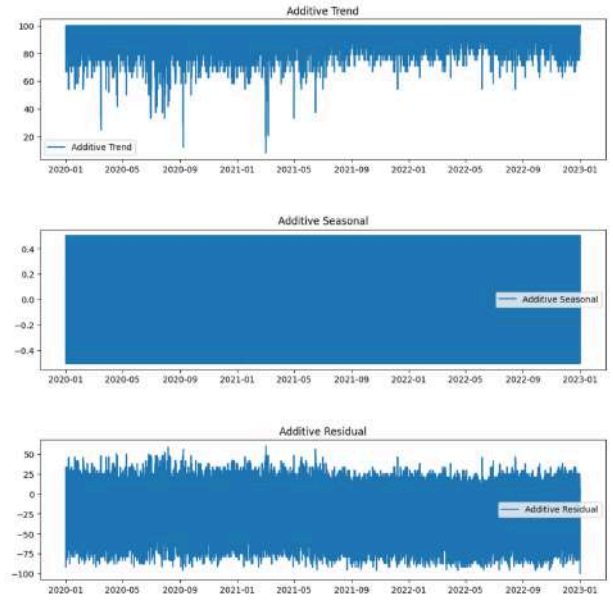


Figure 22. Decomposition of AC_FREQ (Additive and Multiplicative Model)



Multiplicative decomposition couldn't be performed for column AC_POW: Multiplicative seasonality is not appropriate for zero and negative values

Figure 23. Decomposition of AC_TOT (Additive and Multiplicative Model)

Question 2

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the autocorrelation lag (lag) using the Statsmodels acf() function and plot_acf(). Find .

This structure prepares the data for further analysis by ensuring stationarity of the time series through the calculation of first-order differences for the specified columns of interest.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S',
                       'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Importing Libraries: It imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **acf** from **statsmodels.tsa.stattools** for calculating autocorrelation.
- **plot_acf** from **statsmodels.graphics.tsaplots** for plotting autocorrelation.

Load Data: It loads the solar power generation data from the specified CSV file using **pd.read_csv()** function and stores it in a DataFrame named **data.**

Defining Columns of Interest:
It specifies a list named **columns_of_interest** containing the names of columns that are of interest for further analysis. These columns represent various aspects of AC power generation.

Ensuring Stationarity of the Time Series:

- For each column of interest, it calculates the first-order difference by subtracting each value from its previous value using the **diff()** method.
- It creates new columns in the DataFrame with names suffixed by '_diff', representing the first-order differences of the original columns.

This structure enables visual inspection of the first-order difference and autocorrelation of each column, which are essential steps in time series analysis to understand the data's temporal patterns and dependencies.

```python
# Plot the first-order difference with larger figure size and increased spacing
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate autocorrelation lag using acf() function
for column in columns_of_interest:
    acf_result = acf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Autocorrelation for {column}: {acf_result}')

# Plot autocorrelation using plot_acf() function
for column in columns_of_interest:
    plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference:

- It creates a figure with a larger size based on the number of columns of interest using **plt.figure(figsize=(12, 6*len(columns_of_interest)))**.
- It iterates over each column of interest using a for loop with **enumerate(columns_of_interest, 1)** to track the index and column name.
- For each column, it creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)** to arrange the subplots vertically.
- It plots the first-order difference of the column against the index using **plt.plot(data.index, data[column + '_diff'])**.
- Sets the title, xlabel, and ylabel for each subplot.
- Finally, it calls **plt.tight_layout()** to adjust the subplot layout and displays the plot using **plt.show()**.

Calculating and Plotting Autocorrelation :For each column of interest:

- It calculates the autocorrelation up to lag 20 using the acf() function from **statsmodels.tsa.stattools**. The **dropna()** method is used to remove any NaN values.
- Prints the autocorrelation results for each column.
- It then plots the autocorrelation function using the **plot_acf()** function from **statsmodels.graphics.tsaplots**, specifying a maximum lag of 20.
- The title of each plot indicates the column for which autocorrelation is being calculated.

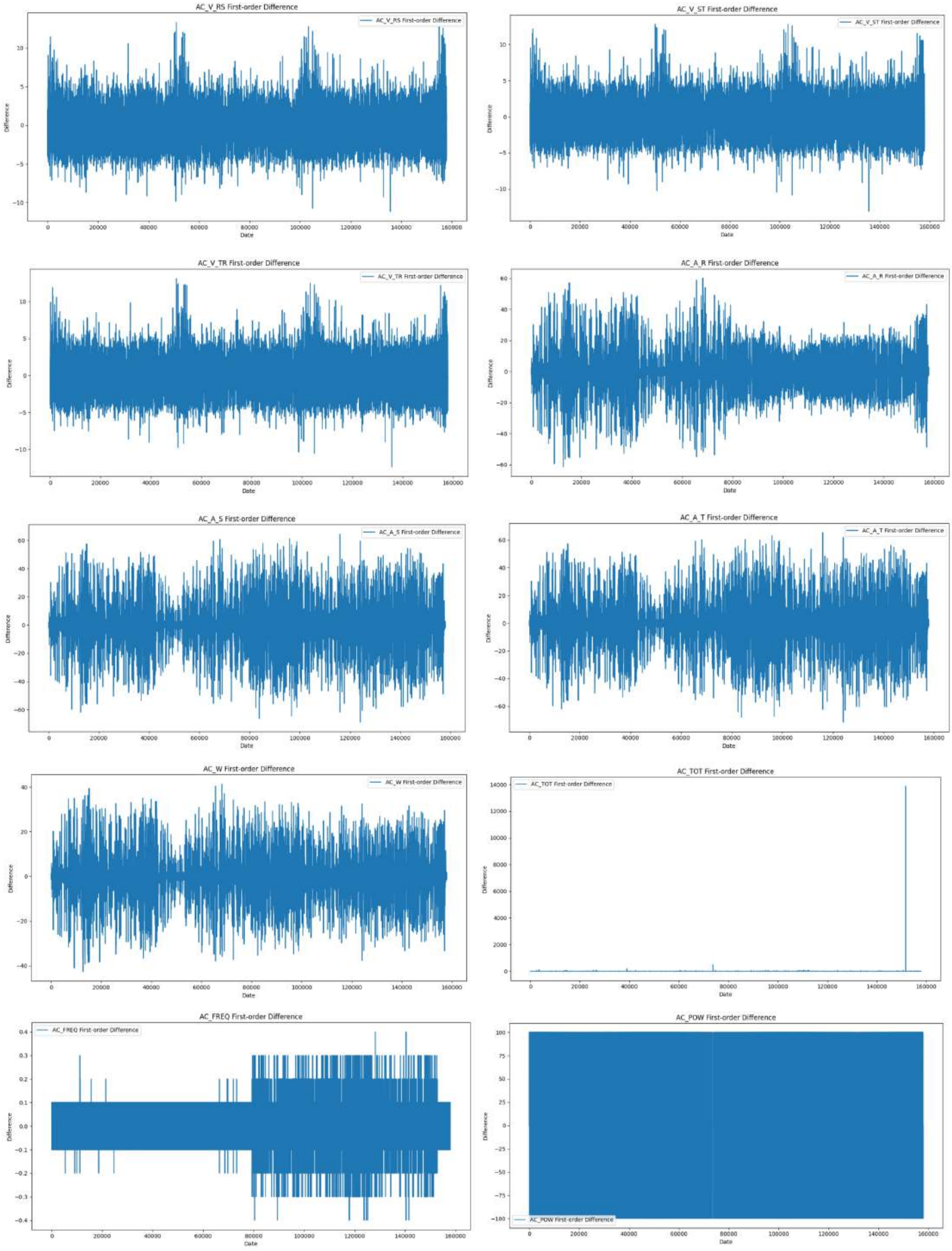


Figure 24. First-order difference of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

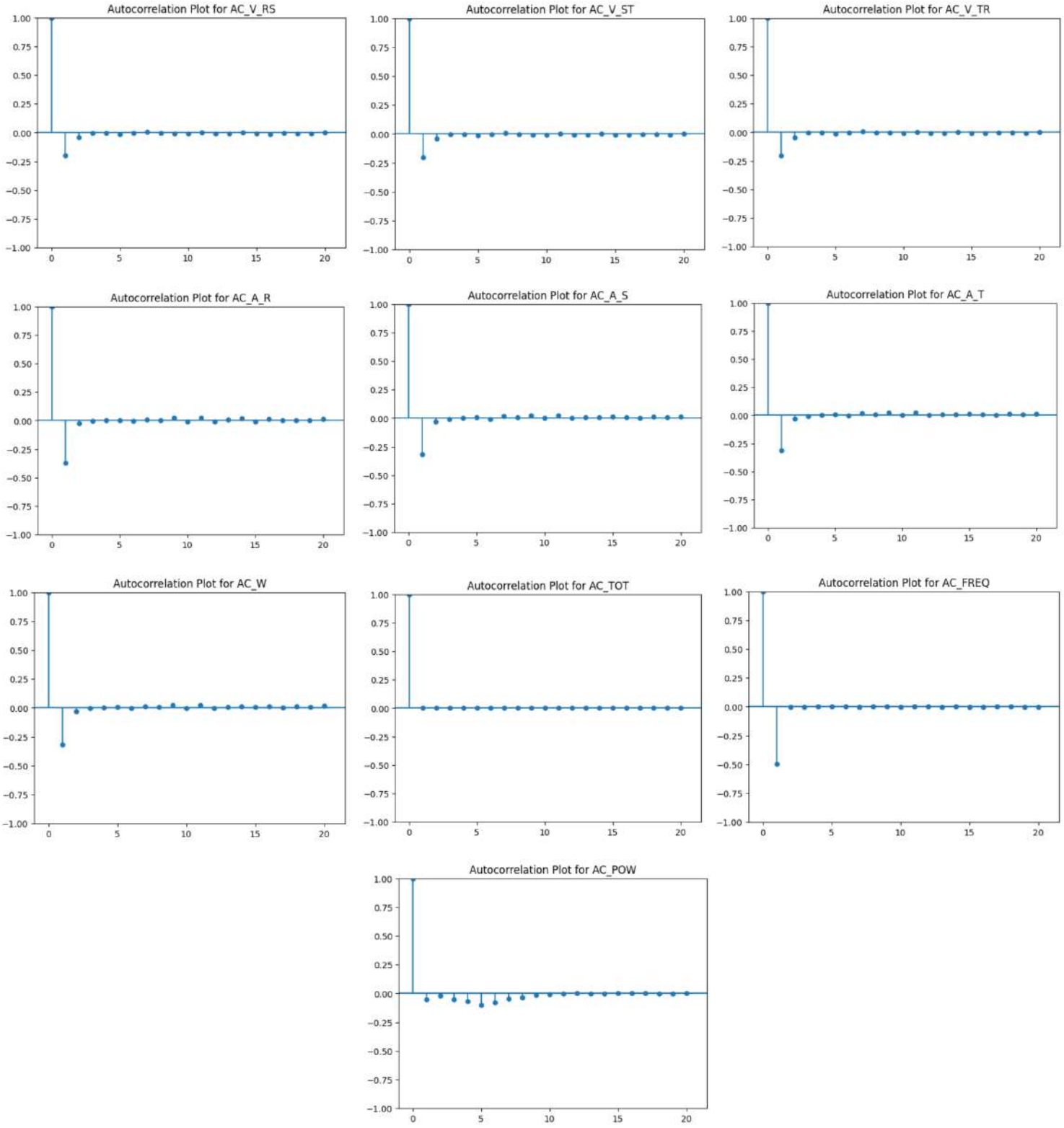


Figure 25. Autocorrelation of of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

Question 3

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the partial autocorrelation lag (lag) using the Statsmodels pacf() function and plot_acf().).

This code prepares the solar power generation data for analysis by ensuring stationarity through the calculation of first-order differences for the specified columns of interest.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import pacf
from statsmodels.graphics.tsaplots import plot_pacf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S',
                       'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Import: It imports the necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for data visualization.
- **pacf** from **statsmodels.tsa.stattools** to compute the partial autocorrelation function.
- **plot_pacf** from **statsmodels.graphics.tsaplots** to plot the partial autocorrelation.

Load Data: It reads the solar power generation data from the specified CSV file using **pd.read_csv()** and stores it in the DataFrame **data.**

Defining Columns of Interest:
It lists the columns of interest that need to be analyzed for stationarity. These columns represent different aspects of solar power generation, such as voltage, current, output, frequency, and power factor.

Ensuring Stationarity of the Time Series:
For each column of interest, it calculates the first-order difference by subtracting each value from its preceding value using the **diff()** function. The first-order difference is a common technique used to transform a non-stationary time series into a stationary one by removing trends or seasonality. The resulting differenced data is stored in new columns with names suffixed by **_diff.**

This code visually examines the first-order difference of the columns of interest and computes and plots the partial autocorrelation for each column to further analyze the time series data's stationarity and autocorrelation structure.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate partial autocorrelation lag using pacf() function
for column in columns_of_interest:
    pacf_result = pacf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Partial Autocorrelation for {column}: {pacf_result}')

# Plot partial autocorrelation using plot_pacf() function
for column in columns_of_interest:
    plot_pacf(data[column + '_diff'].dropna(), lags=20, title=f'Partial Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference:

- It creates a figure with a specified size using **plt.figure(figsize=(12, 6*len(columns_of_interest)))**.
- It iterates over each column of interest in **columns_of_interest**.
- For each column, it creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)**, where i is the current iteration index.
- It plots the first-order difference of the column against the index of the DataFrame using **plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')**.
- Sets the title of the subplot to indicate the column and the operation performed using **plt.title(f'{column} First-order Difference')**.
- Labels the x-axis as 'Date' and the y-axis as 'Difference' using **plt.xlabel('Date')** and **plt.ylabel('Difference')**, respectively.
- Finally, it displays the legend using **plt.legend()** and ensures proper layout using **plt.tight_layout()**. It shows the plot using **plt.show()**.

Calculating and Plotting Partial Autocorrelation:

- For each column in **columns_of_interest**, it calculates the partial autocorrelation using the **pacf()** function and stores the result in **pacf_result**.
- It prints the computed partial autocorrelation for each column.
- It then plots the partial autocorrelation using the **plot_pacf()** function, specifying the number of lags to consider and providing a title that includes the column name.
- Finally, it displays each plot using **plt.show()**.

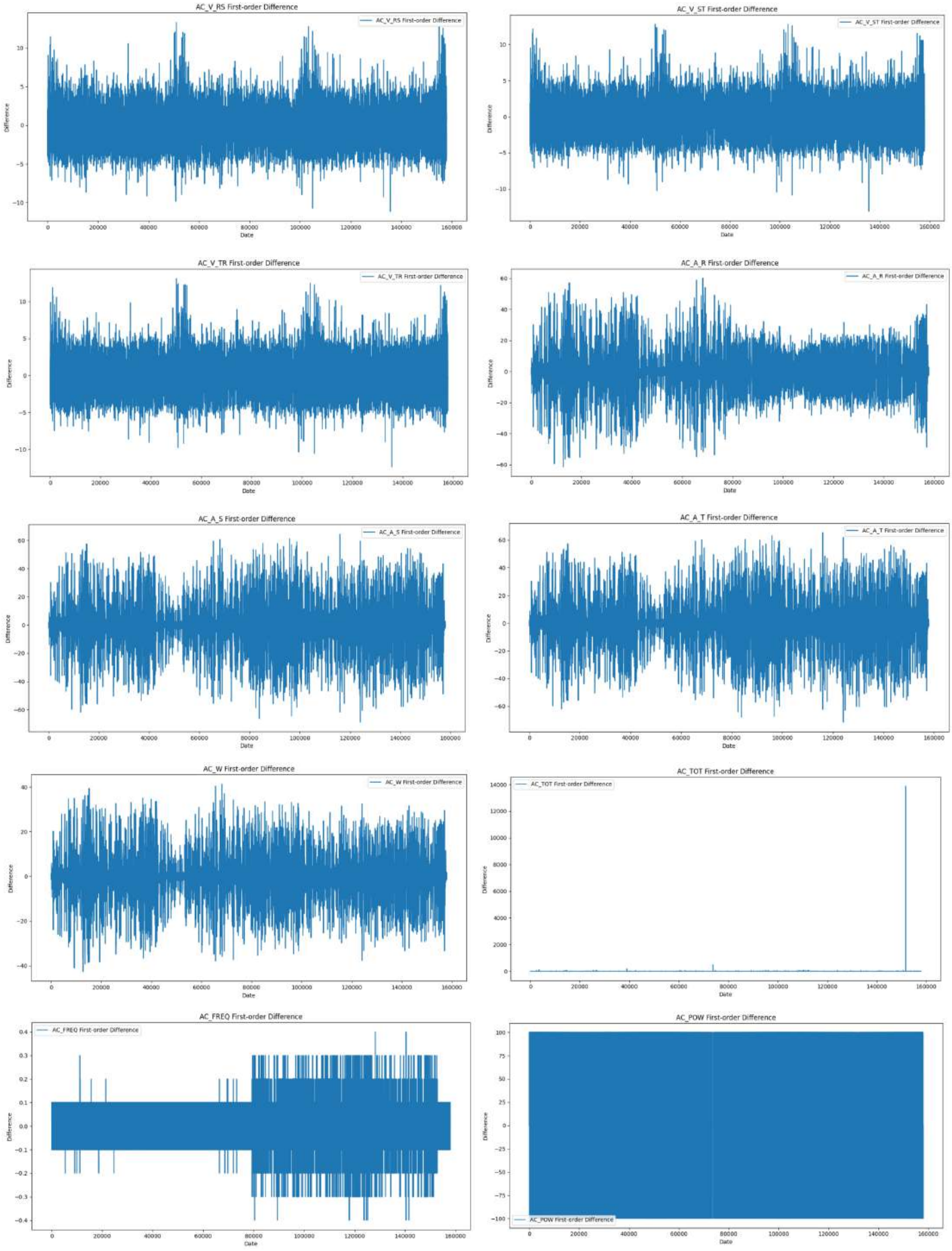


Figure 26. First-order difference of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

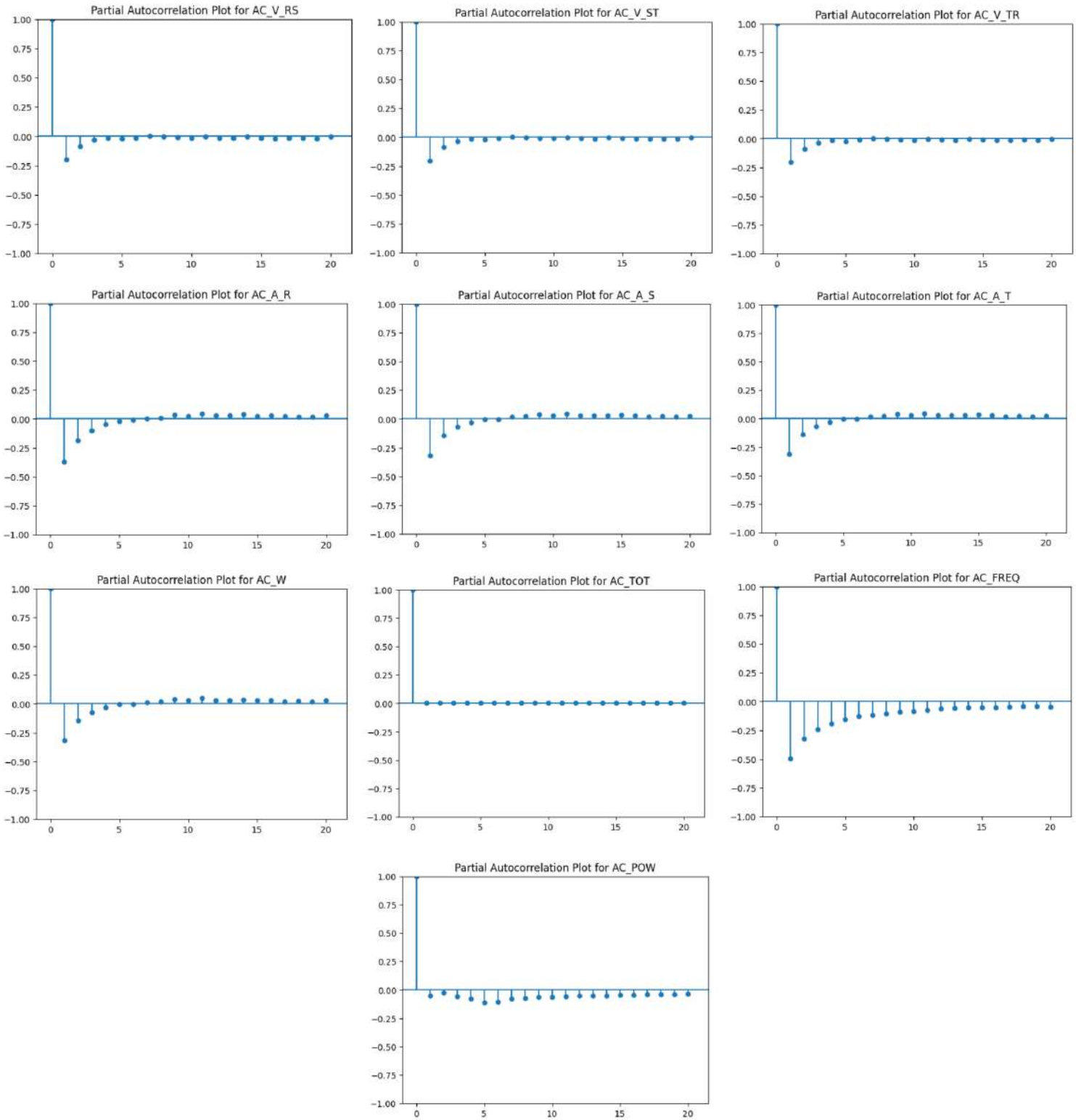


Figure 27. Partial Autocorrelation of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

Question 4

Using solar power generation data, pv_2years_eng, implement it as a moving average (MA), and display the moving average prediction results as plot().

This code essentially provides a visual representation of the original data along with its moving average for each selected column of interest.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S',
                       'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']

# Plot original data and moving average for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))

    # Plot original data
    plt.plot(data.index, data[column], label=f'Original {column}')

    # Implement moving average (MA)
    window_size = 10  # Specify the window size for the moving average
    ma = data[column].rolling(window=window_size).mean()
    plt.plot(data.index, ma, label=f'{column} MA ({window_size} periods)')

    plt.title(f'Moving Average Prediction Results for {column}')
    plt.xlabel('Date')
    plt.ylabel('Value')
    plt.legend()
    plt.show()
```

Import: It imports necessary libraries such as Pandas for data manipulation, Matplotlib for plotting, and specific functions.

Load Data: It imports the Pandas library and reads the solar power generation data from the provided CSV file **(pv_2years_eng.csv)** into a Pandas DataFrame.

Plot Original Data and Moving Average: It iterates through each column of interest and creates a separate plot for each. For each plot:

- It sets up a figure with a size of 12x6 inches.
- It plots the original data against the index (date) on the x-axis and the values of the selected column on the y-axis, labeling the line as "Original {column_name}".
- It calculates the moving average (MA) using a window size of 10 periods and plots the moving average line on the same plot, labeling it as " {column_name} MA (10 periods)".
- It adds a title to the plot indicating the column name for which the moving average is calculated.
- It labels the x-axis as "Date" and the y-axis as "Value".
- It adds a legend to differentiate between the original data and the moving average line.
- It displays the plot using **plt.show().**

Defining Columns of Interest:
It defines a list of columns of interest for which moving averages will be calculated. These columns include various measurements related to AC voltage, AC current, AC power, etc.

This code calculates and plots the combined moving average prediction results for multiple columns of interest in a solar power generation dataset.

```python
# Calculate and plot combined moving average
plt.figure(figsize=(12, 6))

# Plot original data for each column
for column in columns_of_interest:
    plt.plot(data.index, data[column], label=f'Original {column}')

# Calculate combined moving average
window_size = 10  # Specify the window size for the moving average
combined_ma = data[columns_of_interest].rolling(window=window_size).mean().mean(axis=1)

# Plot combined moving average
plt.plot(data.index, combined_ma, label=f'Combined MA ({window_size} periods)', color='black', linestyle='--')

plt.title('Combined Moving Average Prediction Results')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

Setup Figure: It creates a new figure with a size of 12x6 inches using **plt.figure(figsize=(12, 6)).** This figure will contain the plot of the combined moving average.

Plot Original Data: It iterates through each column of interest **(columns_of_interest)** and plots the original data for each column against the index (date) on the x-axis and the corresponding values on the y-axis. Each line is labeled as "Original {column_name}".

Calculate Combined Moving Average: It calculates the combined moving average by first applying the rolling mean function **(rolling(window=window_size).mean())** to each column in the **data** DataFrame with a window size of 10 periods. Then, it takes the mean across all columns using **.mean(axis=1).** This results in a single series representing the combined moving average for all columns.

Plot Combined Moving Average: It plots the combined moving average on the same plot as the original data. The combined moving average line is plotted against the index (date) on the x-axis and the calculated values on the y-axis. The line is labeled as "Combined MA (10 periods)". It is displayed in black color and dashed linestyle **(color='black', linestyle='--').**

- Set Title and Labels: It sets the title of the plot as "Combined Moving Average Prediction Results". It labels the x-axis as "Date" and the y-axis as "Value".
- Add Legend: It adds a legend to the plot to differentiate between the original data for each column and the combined moving average.
- Display Plot: It displays the plot using **plt.show().**

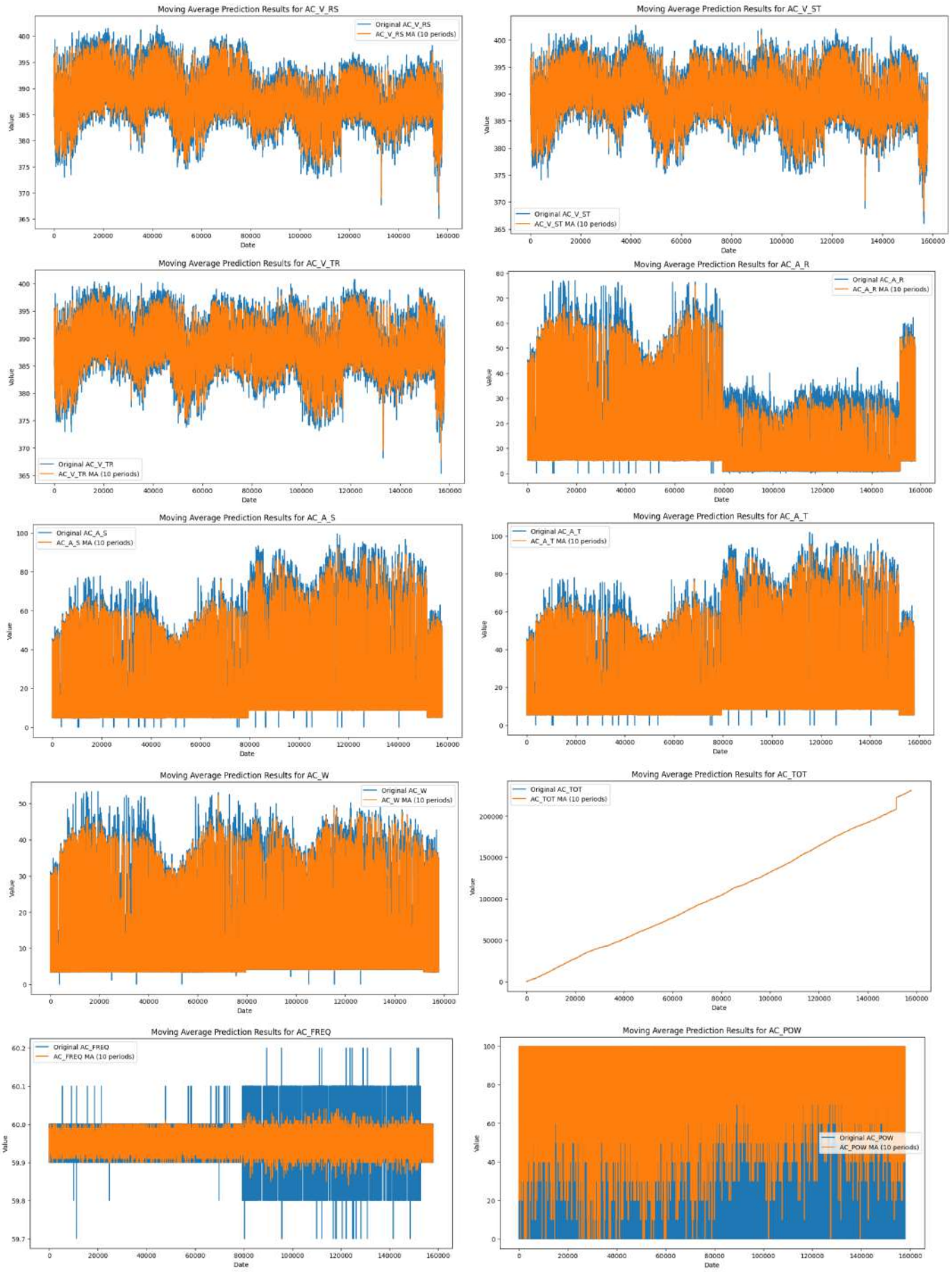


Figure 28. Moving Average of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

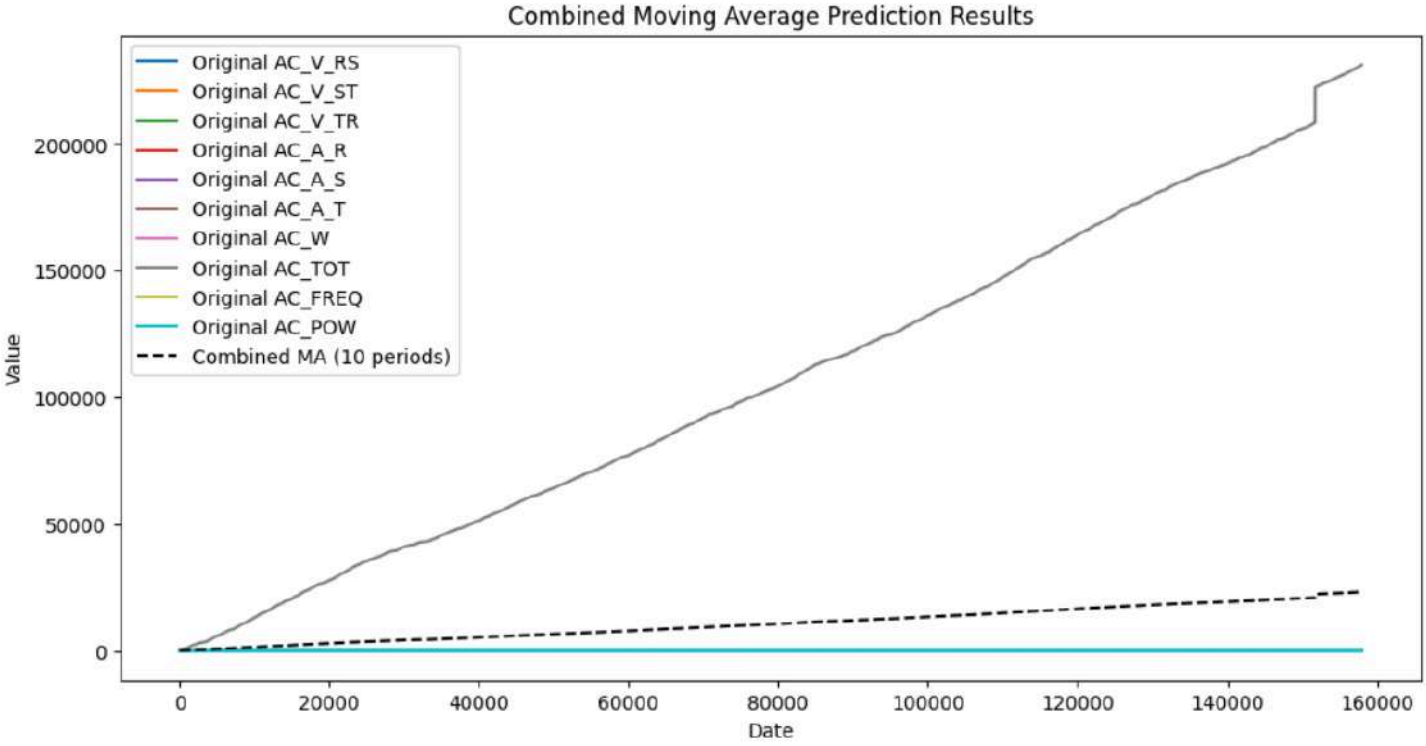


Figure 29. Combined Moving Average of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

Question 5

Using solar power generation data, pv_2years_eng, implement triple exponential smoothing and display the triple exponential smoothing results with plot().

This code segment applies triple exponential smoothing to the specified columns of the solar power generation dataset and stores the predictions for each column in a dictionary for further analysis or visualization.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv', parse_dates=['date'])

# Prepare the data
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S', 'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']

# Implement triple exponential smoothing (Holt-Winters method) for each column
predictions = {}

for column in columns_of_interest:
    model = ExponentialSmoothing(data[column], trend='add', seasonal='add', seasonal_periods=12)
    result = model.fit()
    predictions[column] = result.predict(start=data.index[0], end=data.index[-1])
```

Importing Libraries: It imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **ExponentialSmoothing** from **statsmodels.tsa.holtwinters** for implementing triple exponential smoothing.

Loading Data:

- It loads the solar power generation data from the specified CSV file path into a DataFrame **data.**
- The **parse_dates** parameter is used to ensure that the 'date' column is parsed as dates.

Data Preparation:

- It sets the 'date' column as the index of the DataFrame data using **set_index()** method.

Defining Columns of Interest:

- It defines the columns of interest for which triple exponential smoothing will be applied.
- These columns include AC voltage, AC current, AC power, and other related features.

Triple Exponential Smoothing:

- For each column of interest, it iterates through the list of columns **(columns_of_interest).**
- It initializes an **ExponentialSmoothing** model with additive trend and additive seasonal components.
- The seasonal period is set to 12, indicating monthly seasonal patterns.
- The model is then fitted to the data using the **fit()** method.
- Predictions are generated using the **predict()** method from the start date to the end date of the data.
- Predictions for each column are stored in the **predictions** dictionary with the column name as the key.

This code segment facilitates the visualization of the original data and the predictions obtained from the triple exponential smoothing model for each column of interest in the solar power generation dataset.

```
# Plot the predictions for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label=f'Original {column}')
    plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions')

    plt.title(f'Triple Exponential Smoothing Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Looping Through Columns of Interest:

- It iterates through each column in the **columns_of_interest** list.

Plotting:

- For each column, it creates a new figure with a size of 12x6 inches **(plt.figure(figsize=(12, 6))).**
- It plots the original data against the index (dates) for the current column using **plt.plot(data.index, data[column], label=f'Original {column}').**
- It plots the predictions obtained from the triple exponential smoothing model against the index for the current column using **plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions').**
  - Here, **predictions[column]** retrieves the predictions for the current column from the **predictions** dictionary.
- It adds a title to the plot indicating the type of prediction being displayed, e.g., "Triple Exponential Smoothing Predictions for DC Voltage".
- It sets the x-axis label to "Date" and the y-axis label to the name of the current column.
- It adds a legend to the plot to distinguish between the original data and the predictions.
- It enables grid lines on the plot for better readability using **plt.grid(True).**
- Finally, it displays the plot using **plt.show().**

Visualization:

- The loop generates separate plots for each column, allowing for easy comparison between the original data and the predictions.

Figure 30. Triple Exponential Smooting Predictions of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

Question 6

Using solar power generation data, pv_2years_eng, implement autoregressive (AR) and display the autoregressive (AR) results in plot().

This code prepares the data and sets up the autoregressive modeling process for the specified columns related to AC power generation.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AutoReg

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['AC_V_RS', 'AC_V_ST', 'AC_V_TR', 'AC_A_R', 'AC_A_S', 'AC_A_T', 'AC_W', 'AC_TOT', 'AC_FREQ', 'AC_POW']
```

Importing Libraries: It imports the necessary libraries:

- **pandas** as pd for data manipulation.
- **matplotlib.pyplot** as plt for plotting.
- **AutoReg** from **statsmodels.tsa.ar_model** for autoregressive modeling.

Loading Data:

- It loads the solar power generation data from the specified CSV file using **pd.read_csv().**
- The loaded data is stored in the DataFrame named **data.**

Defining Columns of Interest:

- It specifies the columns of interest related to AC power generation. These columns are:
  - 'AC_V_RS': AC voltage (RS)
  - 'AC_V_ST': AC voltage (ST)
  - 'AC_V_TR': AC voltage (TR)
  - 'AC_A_R': AC current (R)
  - 'AC_A_S': AC current (S)
  - 'AC_A_T': AC current (T)
  - 'AC_W': AC output
  - 'AC_TOT': AC cumulative power generation
  - 'AC_FREQ': AC frequency
  - 'AC_POW': AC power factor
- These columns represent various aspects of AC power generation.

This code effectively visualizes the autoregressive (AR) model predictions for each column of interest in the dataset.

```python
# Plot autoregressive (AR) results for each column
for column in columns_of_interest:
    # Fit autoregressive (AR) model
    model = AutoReg(data[column].dropna(), lags=1)  # Using lag 1 for simplicity
    model_fit = model.fit()

    # Make predictions
    predictions = model_fit.predict(start=1, end=len(data[column]))

    # Plot original data and autoregressive (AR) predictions
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label='Original Data')
    plt.plot(data.index, predictions, label='Autoregressive (AR) Predictions', color='orange')

    plt.title(f'Autoregressive (AR) Model Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Iterating Over Columns of Interest:

- It iterates over each column specified in the columns_of_interest variable.

Fitting Autoregressive (AR) Model:

- For each column, it creates an autoregressive (AR) model using the AutoReg function from statsmodels.tsa.ar_model.
- The model is fitted to the data using the fit() method.
- The parameter lags=1 specifies that the model is fitted with lag 1 for simplicity. This means it uses only the immediately preceding value for prediction.

Making Predictions:

- After fitting the model, it generates predictions using the predict() method.
- The start parameter is set to 1 to indicate the starting point of predictions, and the end parameter is set to the length of the data.

Plotting:

- It creates a new figure using plt.figure(figsize=(12, 6)).
- It plots the original data and the autoregressive (AR) predictions on the same plot using plt.plot().
- The original data is plotted using data.index as the x-axis and data[column] as the y-axis.
- The autoregressive (AR) predictions are plotted using the same x-axis values (data.index) and the predicted values (predictions).
- It sets the title of the plot to indicate that it shows the autoregressive (AR) model predictions for the specific column.
- The x-axis label is set to 'Date', and the y-axis label is set to the name of the column.
- A legend is added to the plot to distinguish between the original data and the predictions.
- Finally, it displays the plot using plt.show().

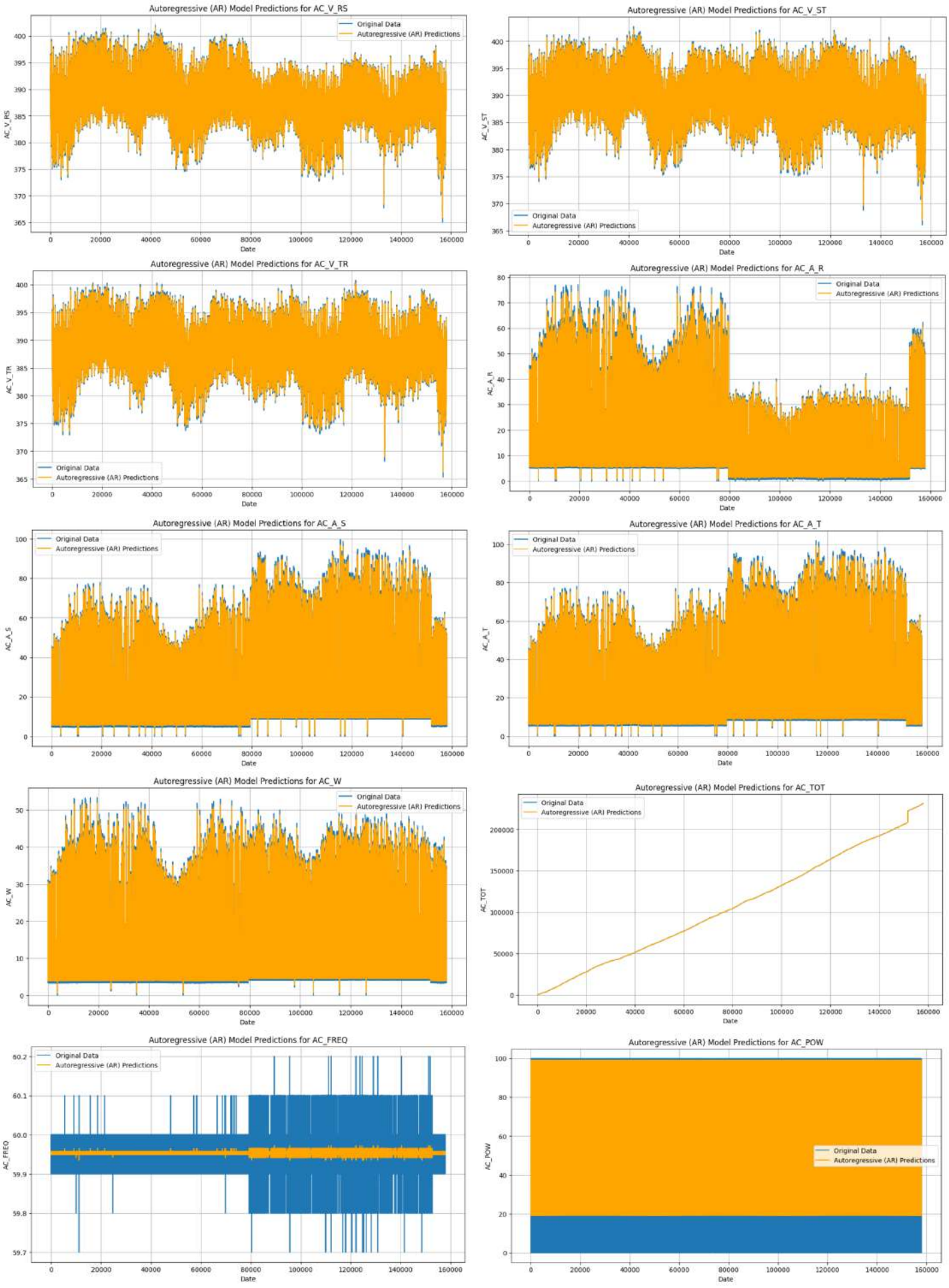
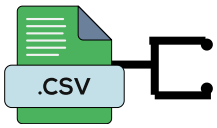


Figure 31. Autoregressive (AR) Model Predictions of AC_V_RS, AC_V_ST, AC_V_TR, AC_A_R, AC_A_S, AC_A_T, AC_W, AC_TOT, AC_FREQ, and AC_POW

3 SOL GROUP

For an understanding of the effectiveness and performance of photovoltaic systems, it is necessary to have access to the solar radiation levels provided by the columns 'SOL_RAD_SLOPE' and 'SOL_RAD_LEVEL' in solar panel data. The solar radiation slope, expressed in watts per square meter (W/㎡), is represented by the variable "SOL_RAD_SLOPE." This measure provides information on the slope or inclination of the solar panels and their orientation with respect to the sun by showing the rate of change in solar radiation intensity over a specified area. Conversely, 'SOL_RAD_LEVEL' denotes the horizontal solar radiation, which is similarly expressed in W/m^2. This measurement, which is independent of the panel's tilt or orientation, expresses the total amount of solar energy received on a horizontal surface. Solar panel operators can evaluate how well their systems capture solar energy and optimize panel location and orientation to maximize energy production by examining both measures. These columns are also essential for predicting energy production and assessing the overall effectiveness of solar power systems.



SOL_RAD_SLOPE': 'Solar radiation (W/㎡) slope'
SOL_RAD_LEVEL': 'Solar radiation (W/㎡) horizontal'

Figure 32. Analysis 3

Question 1

Using solar power generation data, pv_2years_eng, decompose the time series into trend, seasonal, and residual components and display them with plot(). At this time, decompose using the additive vs. multiplicative model and present the results.

This code prepares the solar power generation data by converting the date column to datetime format, setting it as the index, and defining the columns of interest for subsequent analysis.

```python
import pandas as pd
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Convert the date column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Set the date column as the index
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']

# Specify frequency
freq = 'H'
```

Import Libraries: The code imports the necessary libraries for data manipulation, time series analysis, and visualization. These include pandas for data handling, statsmodels for seasonal decomposition, and matplotlib for plotting.

Load Data: The code loads the solar power generation data from a CSV file named 'pv_2years_eng.csv'. It uses the **'read_csv()'** function from pandas to read the data into a DataFrame named **'data'.**

Convert Date Column to Datetime: It converts the 'date' column in the DataFrame **'data'** to datetime format using the **'pd.to_datetime()'** function. This conversion ensures that dates are interpreted correctly for time-based operations.

Set Date Column as Index: The 'date' column, now in datetime format, is set as the index of the DataFrame **'data'** using the **'set_index()'** method. Setting the index to the date column facilitates time series analysis and manipulation.

Define Columns of Interest: The code specifies a list named **'columns_of_interest'** containing the names of columns that are of interest for further analysis. In this case, the columns 'SOL_RAD_SLOPE' and 'SOL_RAD_LEVEL' are chosen, likely representing key metrics related to solar radiation.

Specify Frequency: The variable **'freq'** is assigned the value 'H', indicating that the time series data is at an hourly frequency. This information is essential for certain time series analysis techniques that rely on the frequency of the data, such as seasonal decomposition.

This code effectively decomposes the time series data for each specified column into its trend, seasonal, and residual components using both additive and multiplicative models, and it ensures that potential issues with zero values are addressed before decomposition.

```python
# Decompose the time series using both additive and multiplicative models
for column in columns_of_interest:
    # Additive decomposition
    decomposition_add = seasonal_decompose(data[column], model='additive', period=24, extrapolate_trend='freq')

    # Add a small positive constant to avoid zero values
    data[column] = data[column] + 1e-8

    # Multiplicative decomposition
    decomposition_mul = seasonal_decompose(data[column], model='multiplicative', period=24, extrapolate_trend='freq')

    # Plot the decomposed components
    plt.figure(figsize=(18, 14))
```

For Loop Over Columns of Interest: The code iterates over each column specified in the **'columns_of_interest'** list.

Additive Decomposition:

- For each column, it performs additive decomposition using the **'seasonal_decompose'** function from statsmodels.
- The **'model'** parameter is set to 'additive' to specify additive decomposition.
- The **'period'** parameter is set to 24, indicating that the data has a daily seasonality.
- **'extrapolate_trend'** is set to 'freq' to ensure that the trend is extrapolated to cover the entire time range.

Add a Small Positive Constant : A small positive constant (1e-8) is added to the column data. This is likely done to avoid potential issues with zero values during decomposition, especially when using multiplicative models.

Multiplicative Decomposition:

- Similarly, multiplicative decomposition is performed for each column using the same function.
- The model parameter is set to 'multiplicative' to specify multiplicative decomposition.

Plot the Decomposed Components: After both additive and multiplicative decomposition, the code creates a new figure for plotting the decomposed components using **plt.figure(figsize=(18, 14)).** The large figure size ensures that the plots are clear and easy to interpret.

This structure ensures that the decomposed components of both additive and multiplicative models are visualized clearly and labeled appropriately for analysis. The use of subplots helps organize and present the information in a structured manner.

```
# Additive model plots
plt.subplot(3, 2, 1)
plt.plot(decomposition_add.trend, label='Additive Trend')
plt.title('Additive Trend')
plt.legend()

plt.subplot(3, 2, 2)
plt.plot(decomposition_add.seasonal, label='Additive Seasonal')
plt.title('Additive Seasonal')
plt.legend()

plt.subplot(3, 2, 3)
plt.plot(decomposition_add.resid, label='Additive Residual')
plt.title('Additive Residual')
plt.legend()

# Multiplicative model plots
plt.subplot(3, 2, 4)
plt.plot(decomposition_mul.trend, label='Multiplicative Trend')
plt.title('Multiplicative Trend')
plt.legend()

plt.subplot(3, 2, 5)
plt.plot(decomposition_mul.seasonal, label='Multiplicative Seasonal')
plt.title('Multiplicative Seasonal')
plt.legend()

plt.subplot(3, 2, 6)
plt.plot(decomposition_mul.resid, label='Multiplicative Residual')
plt.title('Multiplicative Residual')
plt.legend()


plt.suptitle(f'Decomposition of {data[column].name}', fontsize=14)
plt.show()
```

Additive Model Plots:

- Three subplots are dedicated to plotting the trend, seasonal, and residual components of the additive decomposition.
- For each subplot:
  - It uses '**plt.subplot(3, 2, n)**' to create a subplot grid with 3 rows and 2 columns. n specifies the position of the subplot within this grid.
  - '**plt.plot()**' is used to plot the corresponding component (trend, seasonal, or residual) obtained from the additive decomposition.
  - '**plt.title()**' sets the title of the subplot to indicate the type of component being plotted (e.g., 'Additive Trend').
  - '**plt.legend()**' adds a legend to the plot to label the plotted component.

Multiplicative Model Plots:

- Similarly, three subplots are dedicated to plotting the trend, seasonal, and residual components of the multiplicative decomposition.
- For each subplot:
  - It uses '**plt.subplot(3, 2, n)**' to create a new subplot in the same grid.
  - '**plt.plot()**' is used to plot the corresponding component (trend, seasonal, or residual) obtained from the multiplicative decomposition.
  - '**plt.title()**' sets the title of the subplot to indicate the type of component being plotted (e.g., 'Multiplicative Trend').
  - '**plt.legend()**' adds a legend to the plot to label the plotted component.

Suptitle: '**plt.suptitle()**' sets the main title of the entire plot, indicating the decomposition of the specific column. It uses string formatting '**(f'Decomposition of {data[column].name}')**' to include the name of the

Show Plot: '**plt.show()**' is called to display the plot containing all the subplots.

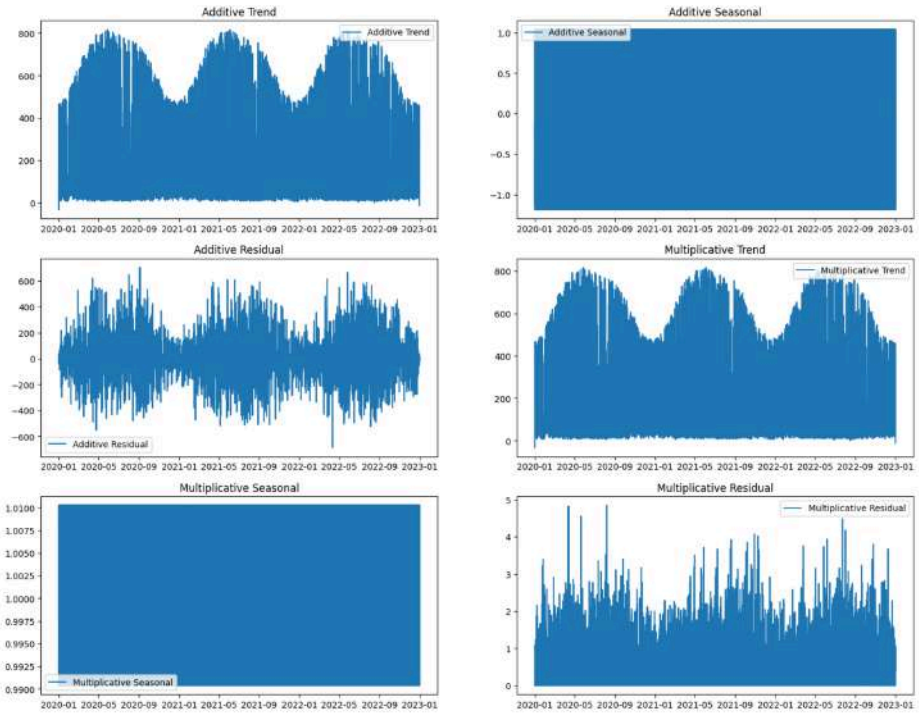


Figure 33. Decomposition of SOL_RAD_SLOPE (Additive and Multiplicative Model)

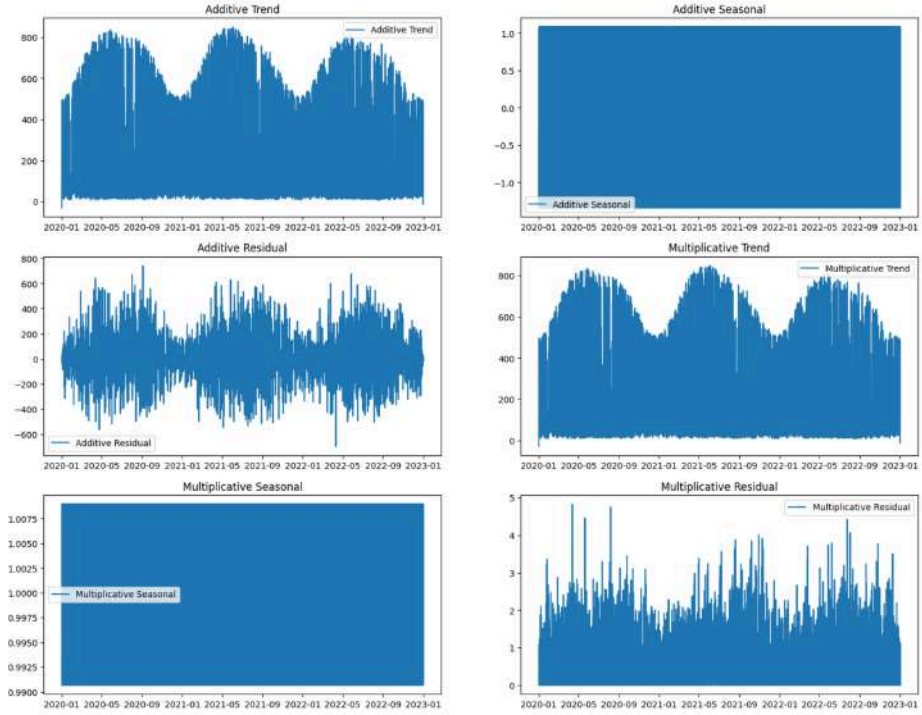


Figure 34. Decomposition of SOL_RAD_LEVEL' (Additive and Multiplicative Model)

Question 2

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the autocorrelation lag (lag) using the Statsmodels acf() function and plot_acf(). Find .

This structure prepares the data for further analysis by ensuring stationarity of the time series through the calculation of first-order differences for the specified columns of interest related to solar radiation.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Importing Libraries: It imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **acf** from **statsmodels.tsa.stattools** for calculating autocorrelation.
- **plot_acf** from **statsmodels.graphics.tsaplots** for plotting autocorrelation.

Load Data: It loads the solar power generation data from the specified CSV file using **pd.read_csv()** function and stores it in a DataFrame named **data.**

Defining Columns of Interest:
It specifies a list named **columns_of_interest** containing the names of columns that are of interest for further analysis. These columns represent various aspects of AC power generation.

Ensuring Stationarity of the Time Series:

- For each column of interest, it calculates the first-order difference by subtracting each value from its previous value using the **diff()** method.
- It creates new columns in the DataFrame with names suffixed by '_diff', representing the first-order differences of the original columns.

This structure enables visual inspection of the first-order difference and autocorrelation of each column, which are essential steps in time series analysis to understand the data's temporal patterns and dependencies.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate autocorrelation lag using acf() function
for column in columns_of_interest:
    acf_result = acf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Autocorrelation for {column}: {acf_result}')

# Plot autocorrelation using plot_acf() function
for column in columns_of_interest:
    plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference:

- It creates a figure for plotting with a size determined by the number of columns of interest and the specified aspect ratio using **plt.figure(figsize=(12, 6*len(columns_of_interest)))**.
- It iterates over each column of interest using a for loop with **enumerate(columns_of_interest, 1)** to track the index and column name.
- For each column, it creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)** to arrange the subplots vertically.
- It plots the first-order difference of the column against the index using plt.plot(data.index, data[column + '_diff']).
- Sets the title, xlabel, and ylabel for each subplot.
- Finally, it calls **plt.tight_layout()** to adjust the subplot layout and displays the plot using **plt.show()**.

Calculating and Printing Autocorrelation: For each column of interest:

- It calculates the autocorrelation up to lag 20 using the **acf()** function from **statsmodels.tsa.stattools.** The dropna() method is used to remove any NaN values.
- Prints the autocorrelation results for each column.

Plotting Autocorrelation: For each column of interest:

- It plots the autocorrelation function using the **plot_acf()** function from **statsmodels.graphics.tsaplots,** specifying a maximum lag of 20.
- The title of each plot indicates the column for which autocorrelation is being calculated.

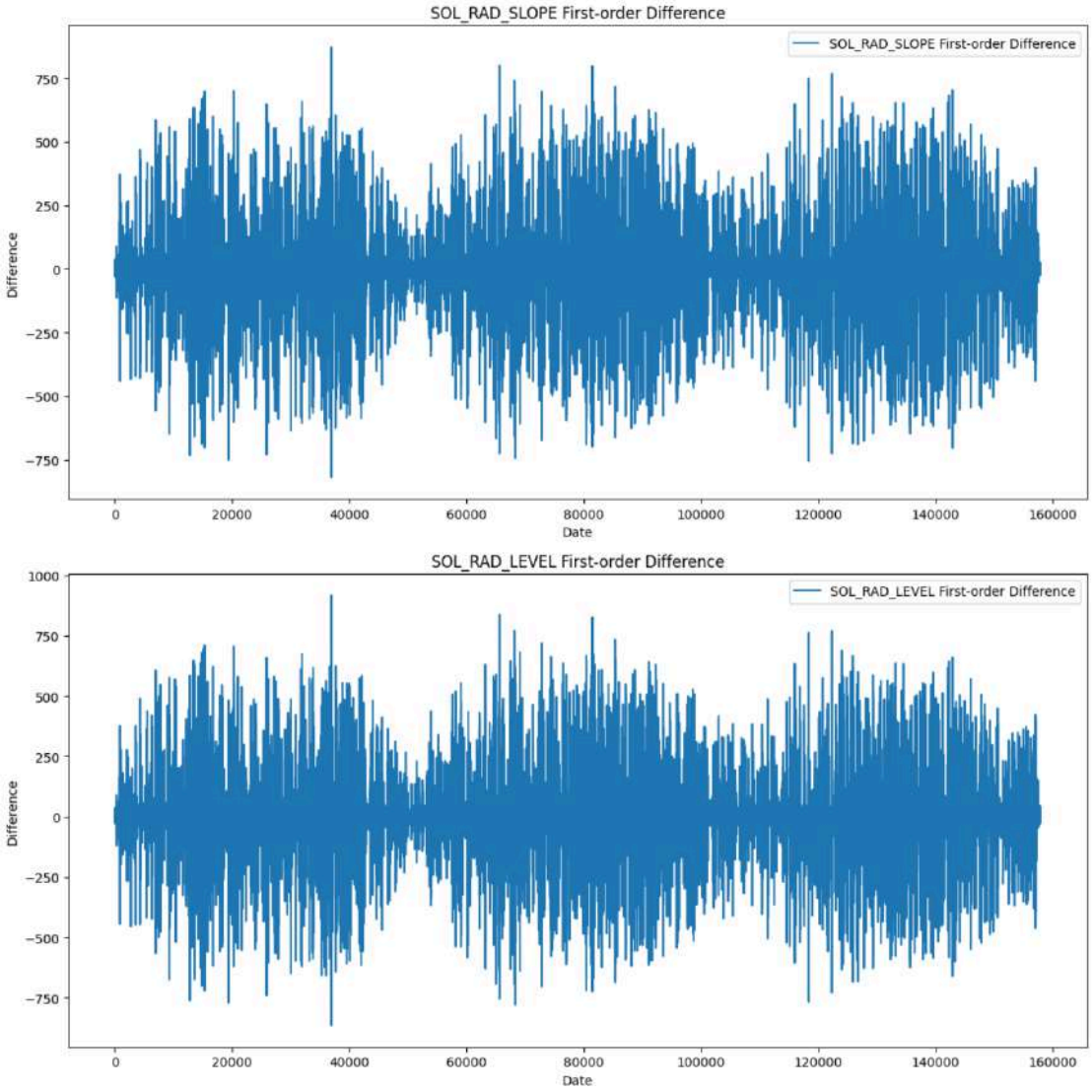


Figure 35. First-order difference of 'SOL_RAD_SLOPE', 'SOL_RAD_LEVEL'

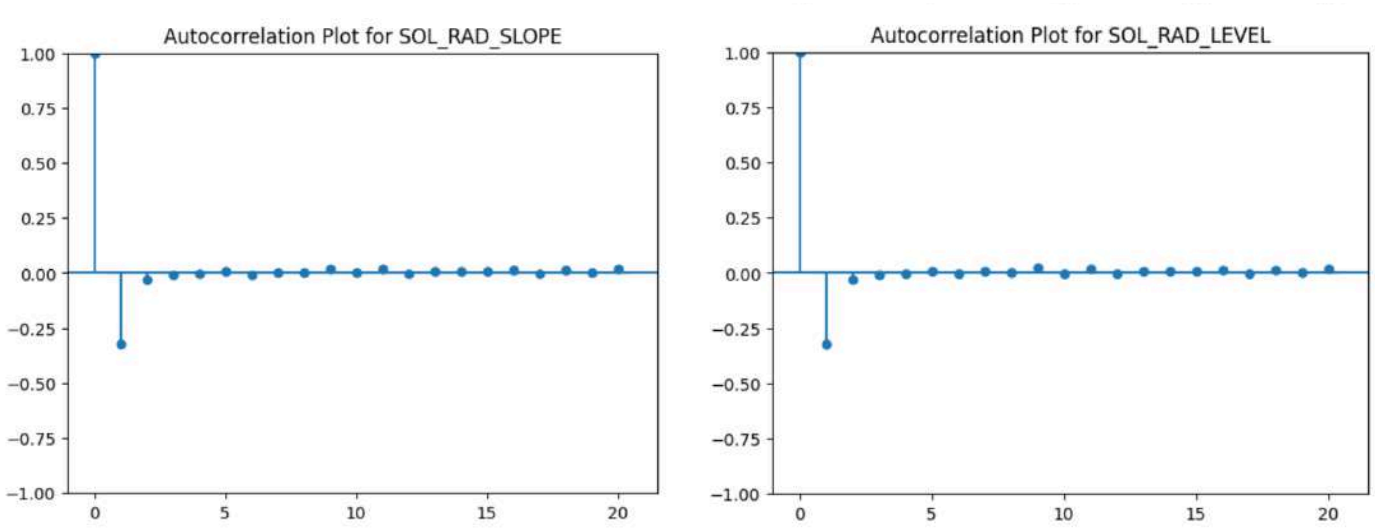


Figure 36. Autocorrelation of 'SOL_RAD_SLOPE', 'SOL_RAD_LEVEL'

Question 3

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the partial autocorrelation lag (lag) using the Statsmodels pacf() function and plot_acf().).

This code prepares the data by calculating the first-order difference for the specified columns of interest, which is a common preprocessing step in time series analysis to stabilize the mean of the series.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import pacf
from statsmodels.graphics.tsaplots import plot_pacf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Import: It imports necessary libraries such as pandas for data manipulation, matplotlib.pyplot for plotting, and functions related to time series analysis from statsmodels library.

Load Data: It reads the solar power generation data from the provided CSV file into a pandas DataFrame called **data.**

Defining Columns of Interest:
It defines the columns of interest as 'SOL_RAD_SLOPE' and 'SOL_RAD_LEVEL'. These are the columns from which the first-order difference will be calculated.

Ensuring Stationarity of the Time Series:
It ensures stationarity of the time series data by calculating the first-order difference for each column of interest. The first-order difference is calculated using the **diff()** function in pandas, which computes the difference between consecutive values in each column.

This code visually examines the first-order difference of the time series data and calculates and plots the partial autocorrelation for each column of interest. These analyses help in understanding the temporal dependencies and identifying the appropriate lag values for time series modeling and forecasting.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate partial autocorrelation lag using pacf() function
for column in columns_of_interest:
    pacf_result = pacf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Partial Autocorrelation for {column}: {pacf_result}')

# Plot partial autocorrelation using plot_pacf() function
for column in columns_of_interest:
    plot_pacf(data[column + '_diff'].dropna(), lags=20, title=f'Partial Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference: It plots the first-order difference for each column of interest. For each column, it creates a subplot in a single figure using matplotlib's **plt.subplot()** function. The first-order difference data is plotted against the index of the DataFrame **(data.index).** Each subplot is labeled with the column name and the title indicates that it represents the first-order difference.

Calculating Partial Autocorrelation:
It calculates the partial autocorrelation for each column of interest using the **pacf()** function from statsmodels. The partial autocorrelation function (PACF) measures the correlation between a variable and its lagged values while controlling for the values of the intermediate lags. It calculates the PACF up to a specified number of lags (in this case, 20).

Plotting Partial Autocorrelation: It plots the partial autocorrelation for each column of interest using the **plot_pacf()** function from statsmodels. Each plot represents the partial autocorrelation for a specific column up to the specified number of lags. The title of each plot indicates the column name for which the PACF is plotted.

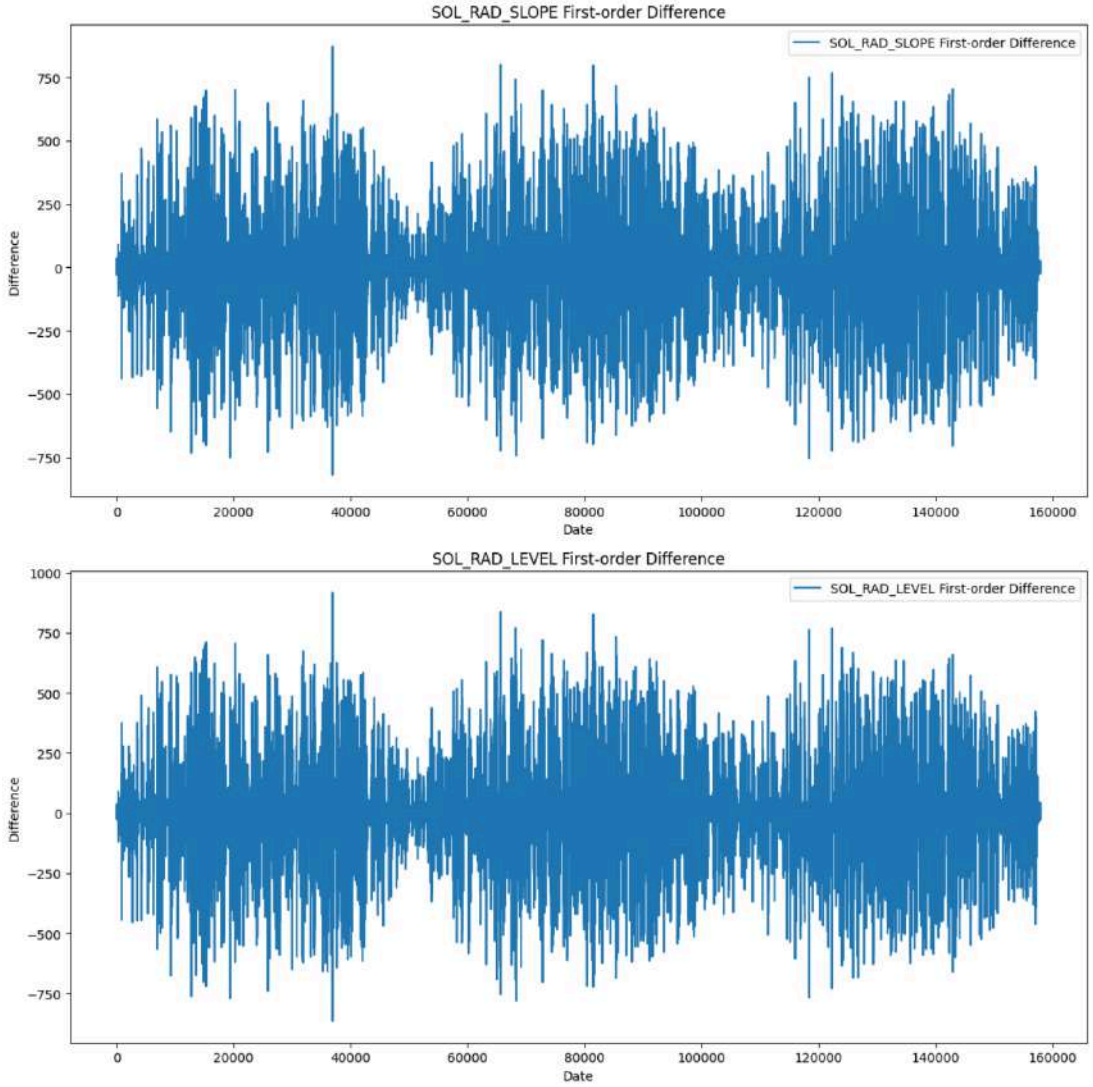


Figure 37. First-order difference of 'SOL_RAD_SLOPE', 'SOL_RAD_LEVEL'

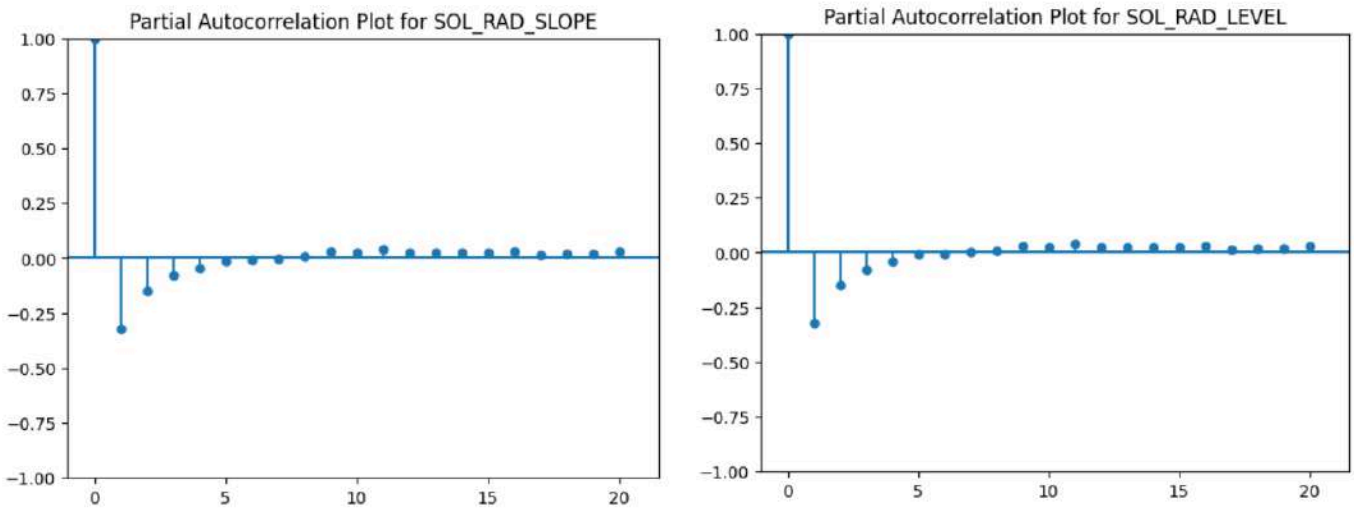


Figure 38. Partial Autocorrelation of 'SOL_RAD_SLOPE', 'SOL_RAD_LEVEL'

Question 4

Using solar power generation data, pv_2years_eng, implement it as a moving average (MA), and display the moving average prediction results as plot().

This code visualizes the original data along with the moving average prediction results for each column of interest, providing insights into the trend and smoothing effects of the moving average.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']

# Plot original data and moving average for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))

    # Plot original data
    plt.plot(data.index, data[column], label=f'Original {column}')

    # Implement moving average (MA)
    window_size = 10  # Specify the window size for the moving average
    ma = data[column].rolling(window=window_size).mean()
    plt.plot(data.index, ma, label=f'{column} MA ({window_size} periods)')

    plt.title(f'Moving Average Prediction Results for {column}')
    plt.xlabel('Date')
    plt.ylabel('Value')
    plt.legend()
    plt.show()
```

Import: It imports necessary libraries such as Pandas for data manipulation, Matplotlib for plotting, and specific functions.

Load Data: It imports the pandas library to work with data and loads a CSV file containing solar power generation data into a DataFrame named **data**. The data is assumed to be located at '/kaggle/input/solar-power-generation-data/pv_2years_eng.csv'.

Defining Columns of Interest:
It defines a list of columns of interest for which moving averages will be calculated. These columns include various measurements related to AC voltage, AC current, AC power, etc.

Plot Original Data and Moving Average: It iterates over each column of interest and creates a separate plot for each. For each column:

- It creates a new figure with a size of 12x6 inches.
- It plots the original data against the index (presumably date) and labels it as 'Original [column_name]'.
- It calculates the moving average (MA) using a rolling window of size 10 and plots it on the same plot, labeling it as '[column_name] MA (10 periods)'.
- It adds a title to the plot indicating the column name and the type of prediction ('Moving Average Prediction Results').
- It labels the x-axis as 'Date' and the y-axis as 'Value'.
- It adds a legend to differentiate between the original data and the moving average.
- It displays the plot.

This code calculates and plots the combined moving average prediction results for multiple columns of interest in a solar power generation dataset.

```python
# Calculate combined moving average
window_size = 10  # Specify the window size for the moving average
combined_ma = data[columns_of_interest].rolling(window=window_size).mean().mean(axis=1)

# Plot combined moving average
plt.figure(figsize=(12, 6))

# Plot original data for each column
for column in columns_of_interest:
    plt.plot(data.index, data[column], label=f'Original {column}')

# Plot combined moving average
plt.plot(data.index, combined_ma, label=f'Combined MA ({window_size} periods)', color='black', linestyle='--')

plt.title('Combined Moving Average Prediction Results')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

Calculate Combined Moving Average: It calculates the combined moving average across all columns of interest. It uses a window size of 10 for the moving average. The **.rolling(window=window_size)**. mean() method computes the moving average for each column separately, and **.mean(axis=1)** calculates the mean across all columns at each time point, resulting in a single combined moving average series.

Plotting Original Data and Combined Moving Average: It then plots the original data and the combined moving average on the same plot. For each column of interest:

- It iterates over each column of interest and plots the original data against the index (presumably date). Each column's data is labeled as 'Original [column_name]'.
- It plots the combined moving average calculated earlier against the index. The combined moving average is labeled as 'Combined MA (10 periods)' and plotted in black with a dashed line.

Setup Figure: It creates a new figure with a size of 12x6 inches using **plt.figure(figsize=(12, 6)).** This figure will contain the plot of the combined moving average.

Plot Customization: It sets the title of the plot as 'Combined Moving Average Prediction Results' and labels the x-axis as 'Date' and the y-axis as 'Value'. It adds a legend to differentiate between the original data and the combined moving

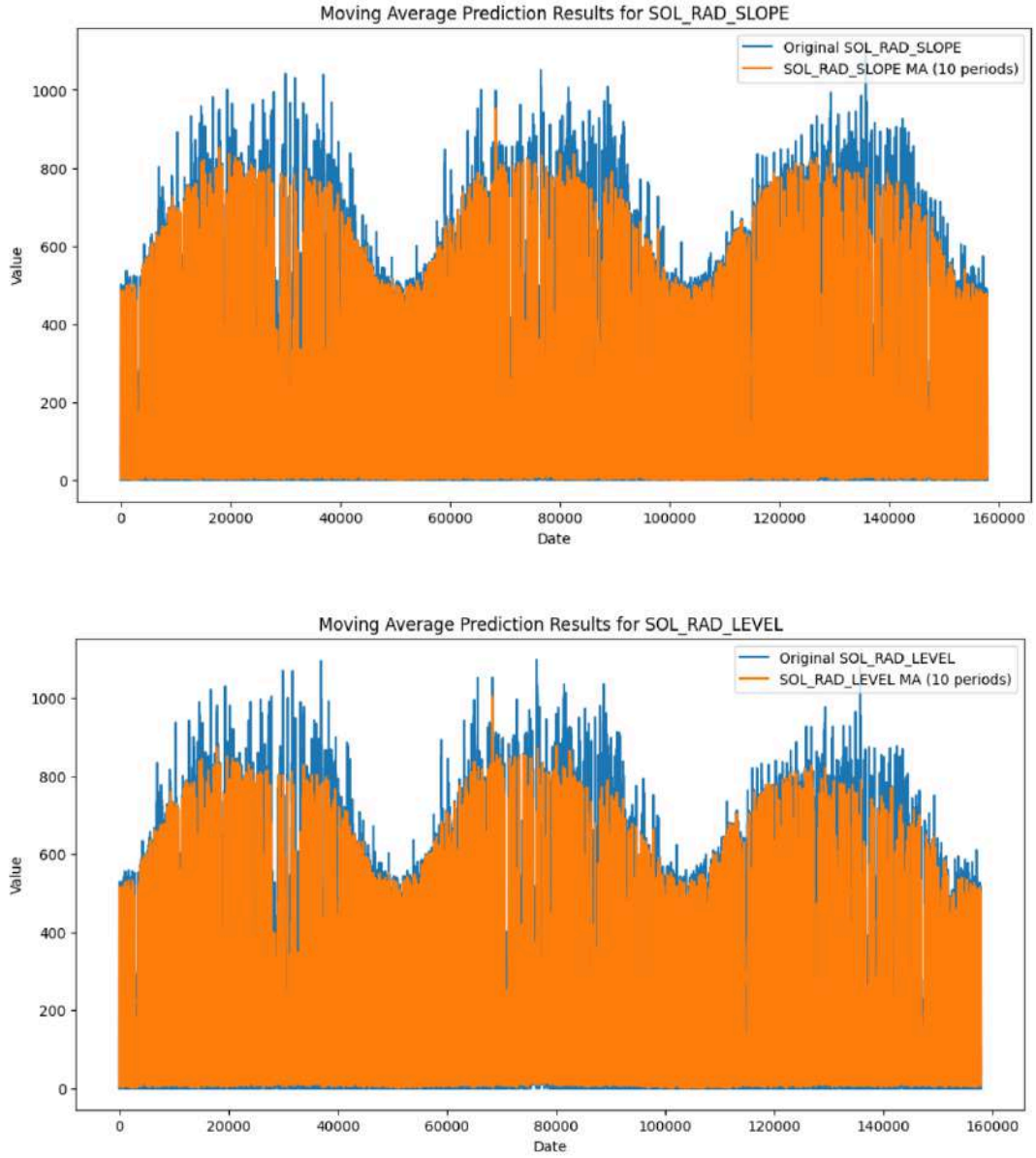


Figure 39. Moving Average of SOL_RAD_SLOPE and SOL_RAD_LEVEL

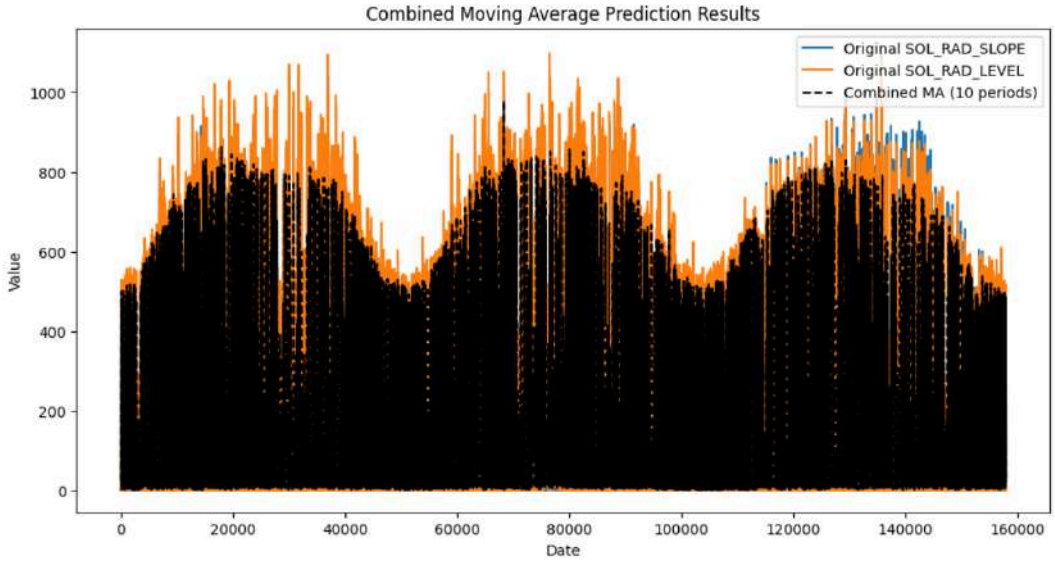


Figure 40. Combined Moving Average of SOL_RAD_SLOPE and SOL_RAD_LEVEL

Question 5

Using solar power generation data, pv_2years_eng, implement triple exponential smoothing and display the triple exponential smoothing results with plot().

This code segment prepares the data, defines columns of interest, and applies triple exponential smoothing to each column, storing the predictions for visualization or further analysis.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv', parse_dates=['date'])

# Prepare the data
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']

# Implement triple exponential smoothing (Holt-Winters method) for each column
predictions = {}

for column in columns_of_interest:
    model = ExponentialSmoothing(data[column], trend='add', seasonal='add', seasonal_periods=12)
    result = model.fit()
    predictions[column] = result.predict(start=data.index[0], end=data.index[-1])
```

Importing Libraries: It imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **ExponentialSmoothing** from **statsmodels.tsa.holtwinters** for implementing triple exponential smoothing.

Define Columns of Interest: The columns of interest for which triple exponential smoothing will be applied are specified in the **columns_of_interest** list, containing 'SOL_RAD_SLOPE' and 'SOL_RAD_LEVEL'.

Data Loading and Preparation:

- It imports necessary libraries, including pandas for data manipulation, matplotlib.pyplot for plotting, and ExponentialSmoothing from statsmodels.tsa.holtwinters for triple exponential smoothing.
- The solar power generation data is loaded from the specified CSV file ('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv') using pd.read_csv().
- The 'date' column in the dataset is parsed as datetime using the parse_dates parameter in pd.read_csv().
- The 'date' column is set as the index of the DataFrame using data.set_index('date', inplace=True).

Triple Exponential Smoothing:

- For each column of interest, a triple exponential smoothing model is fitted.
- Within the loop:
  - **ExponentialSmoothing()** is used to initialize the triple exponential smoothing model for the current column with parameters such as trend ('add'), seasonal ('add'), and seasonal_periods (12, indicating yearly seasonality).
  - **fit()** is called on the model to fit the data.
  - **predict()** is then used to obtain predictions for the entire time range of the dataset, from the first date to the last date.

Storage of Predictions:

- Predictions for each column are stored in a dictionary called **predictions**, where the keys are column names ('SOL_RAD_SLOPE' and 'SOL_RAD_LEVEL') and the values are the corresponding prediction time series.

This code segment iterates over each column of interest, plots the original data, and overlays the predictions obtained from triple exponential smoothing for comparison. It provides a clear visualization of how well the triple exponential smoothing model fits the original data for each column.

```python
# Plot the predictions for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label=f'Original {column}')
    plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions')

    plt.title(f'Triple Exponential Smoothing Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Loop Through Columns of Interest:

- The loop iterates over each column in the **columns_of_interest** list.

Plotting:

- For each column, a new figure is created with a figsize of (12, 6) using **plt.figure(figsize=(12, 6)).**
- The original data for the current column is plotted against the index (date) using **plt.plot(data.index, data[column], label=f'Original {column}').**
- The predictions for the current column obtained from triple exponential smoothing are plotted against the index (date) using **plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions').**

Setting Plot Titles and Labels:

- The title of the plot is set to indicate that it shows the predictions obtained from triple exponential smoothing for the current column, using **plt.title(f'Triple Exponential Smoothing Predictions for {column}').**
- The x-axis label is set to 'Date' using **plt.xlabel('Date').**
- The y-axis label is set to the current column name using **plt.ylabel(column).**

Legend and Grid:

- A legend is displayed to distinguish between the original data and the predictions using **plt.legend().**
- Grid lines are added to the plot using **plt.grid(True).**

Displaying the Plot:

- Finally, the plot for the current column is displayed using **plt.show().**

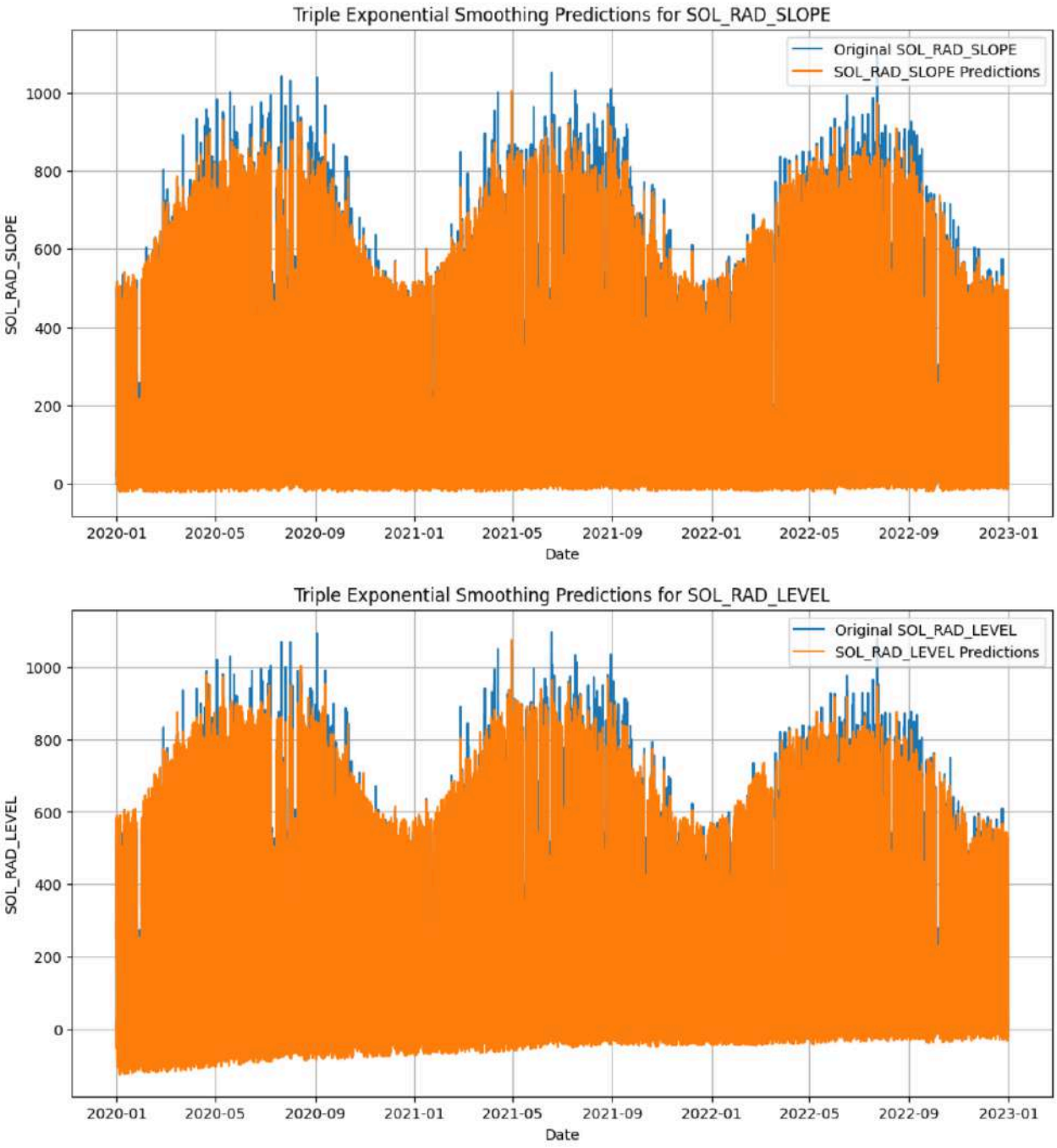


Figure 41. Triple Exponential Smooting Predictions of SOL_RAD_SLOPE and SOL_RAD_LEVEL

Question 6

Using solar power generation data, pv_2years_eng, implement autoregressive (AR) and display the autoregressive (AR) results in plot().

This code structure allows for the efficient fitting of autoregressive (AR) models to multiple columns of interest and visualizing the predictions for each column.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AutoReg

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['SOL_RAD_SLOPE', 'SOL_RAD_LEVEL']
```

Importing Libraries: It imports the necessary libraries:

- **pandas** as pd for data manipulation.
- **matplotlib.pyplot** as plt for plotting.
- **AutoReg** from **statsmodels.tsa.ar_model** for autoregressive modeling.

Loading Data:

- It loads the solar power generation data from the specified CSV file using **pd.read_csv().**
- The loaded data is stored in the DataFrame named **data.**

Defining Columns of Interest: It defines a list columns_of_interest containing the names of the columns for which autoregressive (AR) models will be fitted and predictions visualized.

This structure allows for the efficient visualization of autoregressive (AR) model predictions for multiple columns of interest.

```python
# Plot autoregressive (AR) results for each column
for column in columns_of_interest:
    # Fit autoregressive (AR) model
    model = AutoReg(data[column].dropna(), lags=1)  # Using lag 1 for simplicity
    model_fit = model.fit()

    # Make predictions
    predictions = model_fit.predict(start=1, end=len(data[column]))

    # Plot original data and autoregressive (AR) predictions
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label='Original Data')
    plt.plot(data.index, predictions, label='Autoregressive (AR) Predictions', color='orange')

    plt.title(f'Autoregressive (AR) Model Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Making Predictions:

- After fitting the model, it generates predictions using the **predict()** method of the fitted model.
- The predictions are made for the entire length of the data, starting from index 1 to the end of the column.

Loop Over Columns of Interest:

- It iterates over each column specified in the columns_of_interest list.

Fitting Autoregressive (AR) Model:

- For each column, it initializes an autoregressive (AR) model using the AutoReg function.
- The model is fitted to the data of the respective column, with missing values dropped using .dropna().
- In this case, a lag of 1 (lags=1) is used for simplicity.

Plotting Original Data and Predictions:

- For each column, it creates a new figure with a size of 12x6 using plt.figure(figsize=(12, 6)).
- It plots the original data (data[column]) and the predictions generated by the autoregressive (AR) model on the same plot using plt.plot().
- The original data is plotted against the index of the data (assuming it represents time).
- The title of each plot indicates that it shows the autoregressive (AR) model predictions for the specific column.
- It sets the x-axis label to 'Date' and the y-axis label to the name of the respective column.
- A legend is added to the plot to distinguish between the original data and the predictions.
- Grid lines are enabled on the plot for better visualization of data trends.

Displaying Plots:

- After plotting the data and predictions for each column, it displays the plot using plt.show().

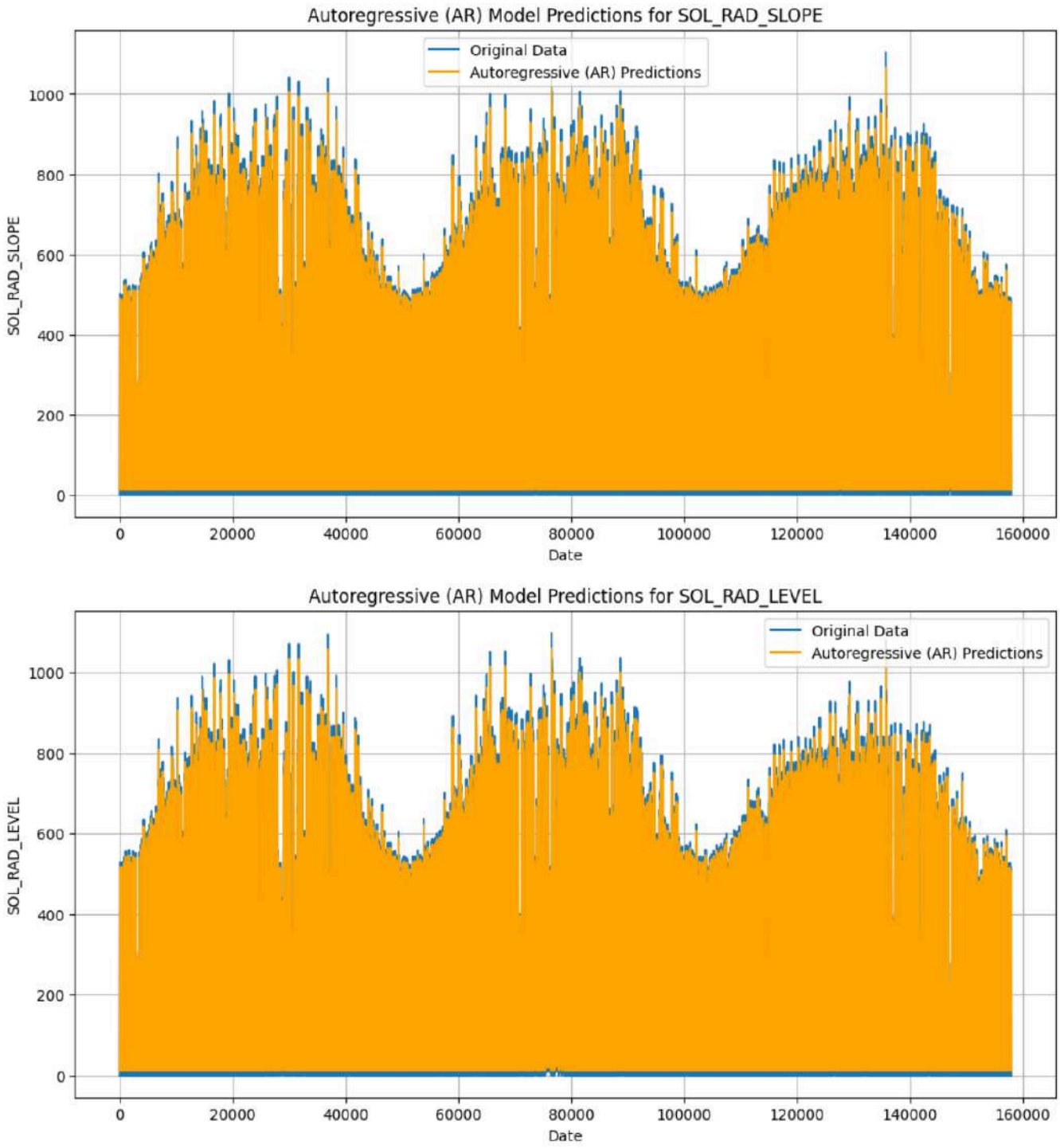
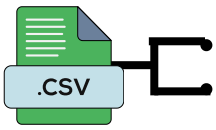


Figure 42. Autoregressive (AR) Model Predictions of SOL_RAD_SLOPE and SOL_RAD_LEVEL

4 TMP GROUP

For an understanding of the effectiveness and performance of photovoltaic systems, temperature measurements from the solar panel data columns 'TMP_MODU' and 'TMP_CLI' are required. The temperature of the solar panels itself is shown by the variable 'TMP_MODU,' which stands for 'Module (℃) temperature'. This measure is important since temperature has an impact on solar panel performance, with higher temperatures frequently resulting in lower efficiency because of greater resistance in the semiconductor materials. By keeping an eye on the temperature of the module, one may determine how temperature variations affect power output and make necessary modifications to the cooling or operation systems to maximize efficiency. However, "TMP_CLI," which stands for "Outdoor (℃) temperature," denotes the surrounding air temperature of the solar panels. The temperature outside has an impact on the panels' temperature as well as the system's overall efficiency. The performance of the panels may be impacted by increased module temperatures brought on by greater outside temperatures. Monitoring both the module and external temperatures enables a thorough understanding of the environmental factors influencing the production of solar power. When combined, these temperature readings offer insightful information for solar energy system performance monitoring, maintenance scheduling, and system optimization.



TMP_MODU': 'Module (℃) temperature'
TMP_CLI': 'Outdoor (℃) temperature'

Figure 43. Analysis 4

Question 1

Using solar power generation data, pv_2years_eng, decompose the time series into trend, seasonal, and residual components and display them with plot(). At this time, decompose using the additive vs. multiplicative model and present the results.

This code segment loads the solar power generation data, converts the date column to datetime format, sets it as the index, and defines the columns of interest for subsequent analysis.

```
import pandas as pd
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Convert the date column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Set the date column as the index
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']
```

Import Libraries: The code imports the necessary libraries for data manipulation, time series analysis, and visualization. These include pandas for data handling, statsmodels for seasonal decomposition, and matplotlib for plotting.

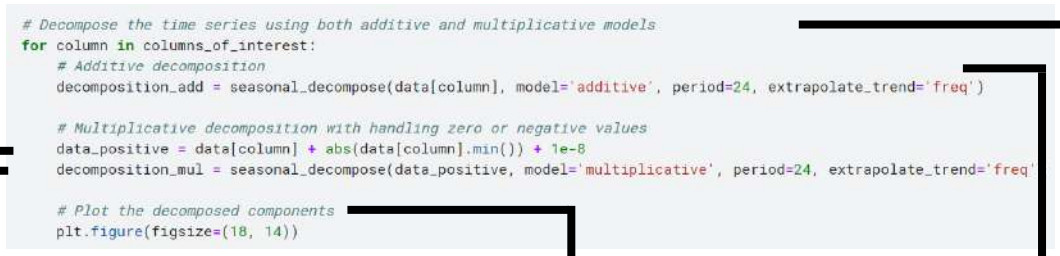
- **Load Data:** It loads the solar power generation data from a CSV file named 'pv_2years_eng.csv'. This is achieved using the **read_csv()** function from pandas, which reads the data into a DataFrame named **data**.

Convert Date Column to Datetime: The 'date' column in the DataFrame data is converted to datetime format using the **pd.to_datetime()** function. This conversion ensures that dates are interpreted correctly for time-based operations.

Set Date Column as Index: The 'date' column, now in datetime format, is set as the index of the DataFrame **data** using the **set_index()** method. Setting the index to the date column facilitates time series analysis and manipulation.

Define Columns of Interest: The code specifies a list named **columns_of_interest** containing the names of columns that are of interest for further analysis. In this case, the columns 'TMP_MODU' and 'TMP_CLI' are chosen, likely representing temperature readings related to solar panel operation.

This structure ensures that the time series data for each specified column is decomposed into its trend, seasonal, and residual components using both additive and multiplicative models, preparing the data for further analysis or modeling. Additionally, it addresses potential issues with zero or negative values in the data to ensure accurate decomposition results.



```
# Decompose the time series using both additive and multiplicative models
for column in columns_of_interest:
    # Additive decomposition
    decomposition_add = seasonal_decompose(data[column], model='additive', period=24, extrapolate_trend='freq')

    # Multiplicative decomposition with handling zero or negative values
    data_positive = data[column] + abs(data[column].min()) + 1e-8
    decomposition_mul = seasonal_decompose(data_positive, model='multiplicative', period=24, extrapolate_trend='freq'

    # Plot the decomposed components
    plt.figure(figsize=(18, 14))
```

For Loop Over Columns of Interest: The code iterates over each column specified in the columns_of_interest list, likely containing the names of columns representing various metrics related to solar power generation.

Additive Decomposition:

- For each column, it performs additive decomposition using the **seasonal_decompose** function from the statsmodels library.
- The **model** parameter is set to 'additive' to specify additive decomposition.
- The **period** parameter is set to 24, indicating that the data exhibits daily seasonality.
- **extrapolate_trend** is set to 'freq' to ensure that the trend component is extrapolated to cover the entire time range.

Handling Zero or Negative Values for Multiplicative Decomposition:

- Before performing multiplicative decomposition, the code handles potential zero or negative values in the data.
- It creates a new DataFrame **data_positive** by adding the absolute value of the minimum value in the column to each value and then adding a small positive constant (1e-8). This is likely done to prevent issues such as division by zero or negative values during multiplicative decomposition, ensuring that all values are positive.

Plot the Decomposed Components: After both additive and multiplicative decompositions, the code creates a new figure for plotting the decomposed components using **plt.figure(figsize=(18, 14)).** The large figure size ensures that the plots are clear and easy to interpret.

Multiplicative Decomposition:

- After handling zero or negative values, multiplicative decomposition is performed for each column using the same **seasonal_decompose** function.
- The model parameter is set to 'multiplicative' to specify multiplicative decomposition, which assumes that the seasonal component varies proportionally with the trend component.

This structure ensures that the decomposed components of both additive and multiplicative models are visualized clearly and labeled appropriately for analysis. The use of subplots helps organize and present the information in a structured manner.

```
# Additive model plots
plt.subplot(3, 2, 1)
plt.plot(decomposition_add.trend, label='Additive Trend')
plt.title('Additive Trend')
plt.legend()

plt.subplot(3, 2, 2)
plt.plot(decomposition_add.seasonal, label='Additive Seasonal')
plt.title('Additive Seasonal')
plt.legend()

plt.subplot(3, 2, 3)
plt.plot(decomposition_add.resid, label='Additive Residual')
plt.title('Additive Residual')
plt.legend()

# Multiplicative model plots
plt.subplot(3, 2, 4)
plt.plot(decomposition_mul.trend, label='Multiplicative Trend')
plt.title('Multiplicative Trend')
plt.legend()

plt.subplot(3, 2, 5)
plt.plot(decomposition_mul.seasonal, label='Multiplicative Seasonal')
plt.title('Multiplicative Seasonal')
plt.legend()

plt.subplot(3, 2, 6)
plt.plot(decomposition_mul.resid, label='Multiplicative Residual')
plt.title('Multiplicative Residual')
plt.legend()


plt.suptitle(f'Decomposition of {column}', fontsize=16)
plt.show()
```

Additive Model Plots:

- Three subplots are dedicated to plotting the trend, seasonal, and residual components of the additive decomposition.
- For each subplot:
  - It uses **plt.subplot(3, 2, n)** to create a subplot grid with 3 rows and 2 columns. n specifies the position of the subplot within this grid.
  - **plt.plot()** is used to plot the corresponding component (trend, seasonal, or residual) obtained from the additive decomposition.
  - **plt.title()** sets the title of the subplot to indicate the type of component being plotted (e.g., 'Additive Trend').
  - **plt.legend()** adds a legend to the plot to label the plotted component.

Multiplicative Model Plots:

- Similarly, three subplots are dedicated to plotting the trend, seasonal, and residual components of the multiplicative decomposition.
- For each subplot:
  - It uses **plt.subplot(3, 2, n)** to create a new subplot in the same grid.
  - **plt.plot()** is used to plot the corresponding component (trend, seasonal, or residual) obtained from the multiplicative decomposition.
  - **plt.title()** sets the title of the subplot to indicate the type of component being plotted (e.g., 'Multiplicative Trend').
  - **plt.legend()** adds a legend to the plot to label the plotted component.

Suptitle: plt.suptitle() sets the main title of the entire plot, indicating the decomposition of the specific column. It uses string formatting (f'Decomposition of {column}') to include the name of the column being decomposed.

Show Plot: plt.show() is called to display the plot containing all the subplots.

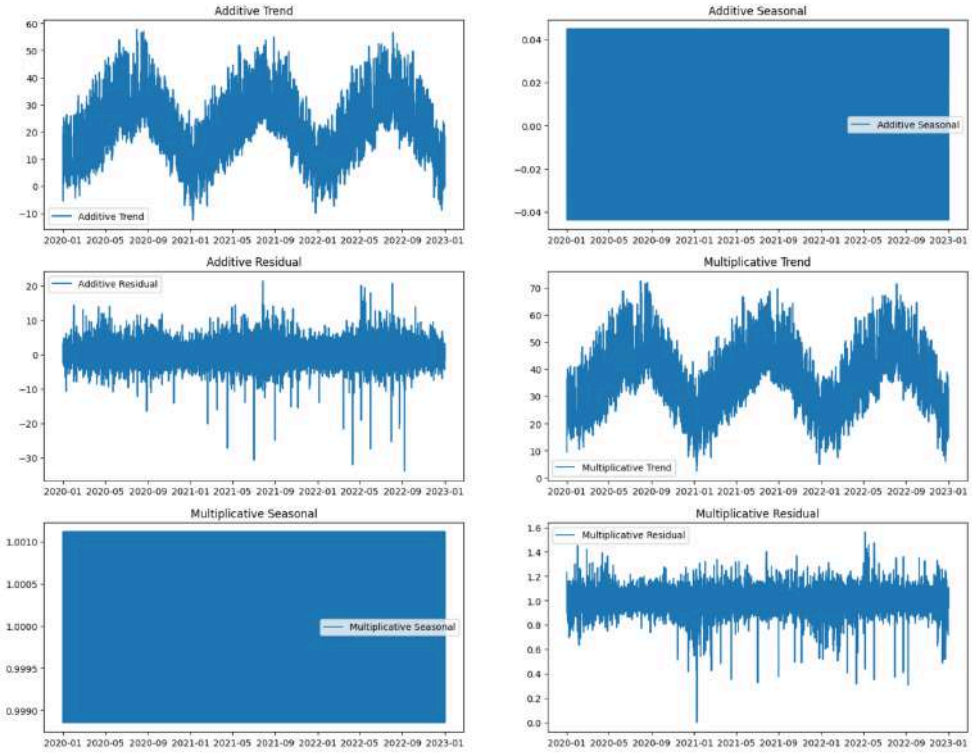


Figure 44. Decomposition of TMP_MODU (Additive and Multiplicative Model)

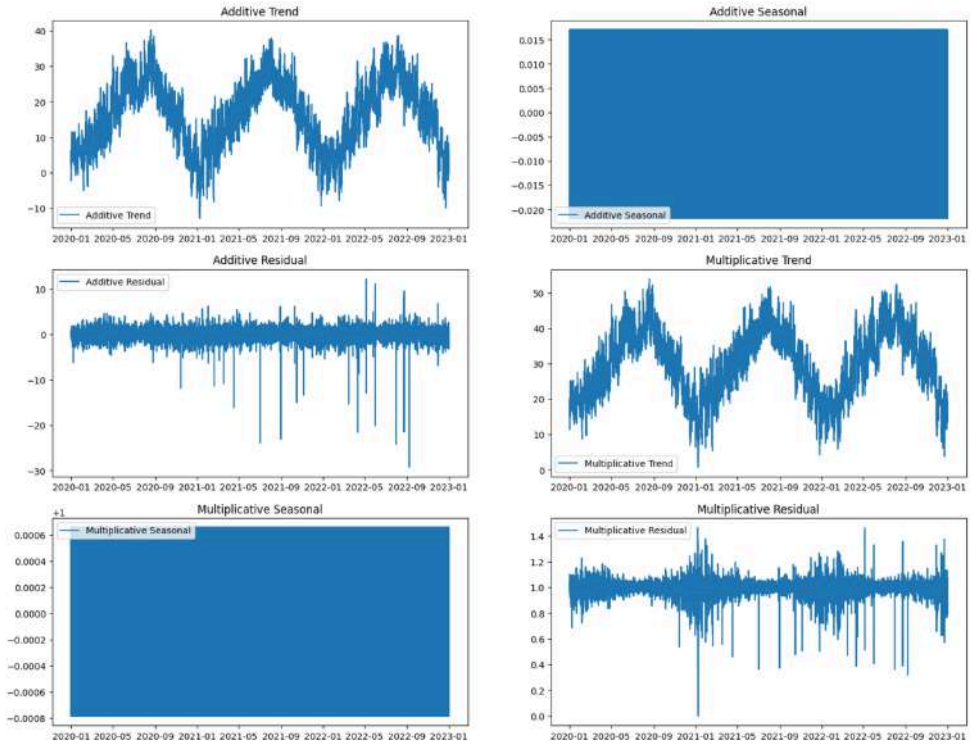


Figure 45. Decomposition of TMP_CLI(Additive and Multiplicative Model)

Question 2

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the autocorrelation lag (lag) using the Statsmodels acf() function and plot_acf(). Find .

This structure prepares the data for analysis by ensuring stationarity through the calculation of the first-order difference for the specified columns of interest.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Importing Libraries: It imports the pandas library as pd for data manipulation, matplotlib.pyplot as plt for plotting, and specific functions from the statsmodels library for time series analysis, including acf for autocorrelation and plot_acf for plotting autocorrelation functions.

Load Data: It reads the solar power generation data from the specified CSV file ('pv_2years_eng.csv') into a pandas DataFrame named 'data'.

Defining Columns of Interest:
It creates a list named 'columns_of_interest' containing the column names of interest, which are 'TMP_MODU' (Module temperature) and 'TMP_CLI' (Outdoor temperature).

Ensuring Stationarity of the Time Series:

- It iterates over each column of interest using a for loop.
- For each column, it calculates the first-order difference by subtracting the previous value from the current value using the diff() method in pandas.
- The resulting first-order difference series is stored in new columns with names suffixed by '_diff' and added to the 'data' DataFrame.

This structure facilitates the visualization and analysis of the first-order difference and autocorrelation of the specified columns, helping in understanding the stationarity and time dependency of the time series data.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate autocorrelation lag using acf() function
for column in columns_of_interest:
    acf_result = acf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Autocorrelation for {column}: {acf_result}')

# Plot autocorrelation using plot_acf() function
for column in columns_of_interest:
    plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Autocorrelation Plot for {column}')
    plt.show()
```

Plotting First-Order Difference:

- It creates a figure with a specified size using **plt.figure(figsize=(12, 6*len(columns_of_interest)))**, where the width is fixed and the height is proportional to the number of columns of interest.
- It iterates over each column of interest using a for loop.
- For each column, it creates a subplot using **plt.subplot(len(columns_of_interest), 1, i)**, where the subplot index 'i' increases in each iteration.
- It plots the first-order difference series against the index of the DataFrame using **plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')**.
- It sets the title of the subplot to '{column} First-order Difference' using **plt.title(f'{column} First-order Difference')**.
- It sets the x-axis label to 'Date' and y-axis label to 'Difference' using **plt.xlabel('Date')** and **plt.ylabel('Difference')**, respectively.
- It adds a legend to the subplot using **plt.legend()**.
- Finally, it adjusts the layout of the subplots to prevent overlap using **plt.tight_layout()** and displays the plot using **plt.show()**.

Calculating Autocorrelation:

- It calculates the autocorrelation for each column of interest using the acf() function from the statsmodels library.
- For each column, it calculates the autocorrelation up to lag 20 using **acf(data[column + '_diff'].dropna(), nlags=20)** and stores the result in the variable acf_result.
- It prints the autocorrelation result for each column using print(f'Autocorrelation for {column}: **{acf_result}')**.

Plotting Autocorrelation:

- It plots the autocorrelation function for each column of interest using the **plot_acf()** function from the statsmodels library.
- For each column, it plots the autocorrelation function up to lag 20 using **plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Autocorrelation Plot for {column}')** and displays the plot using **plt.show()**.

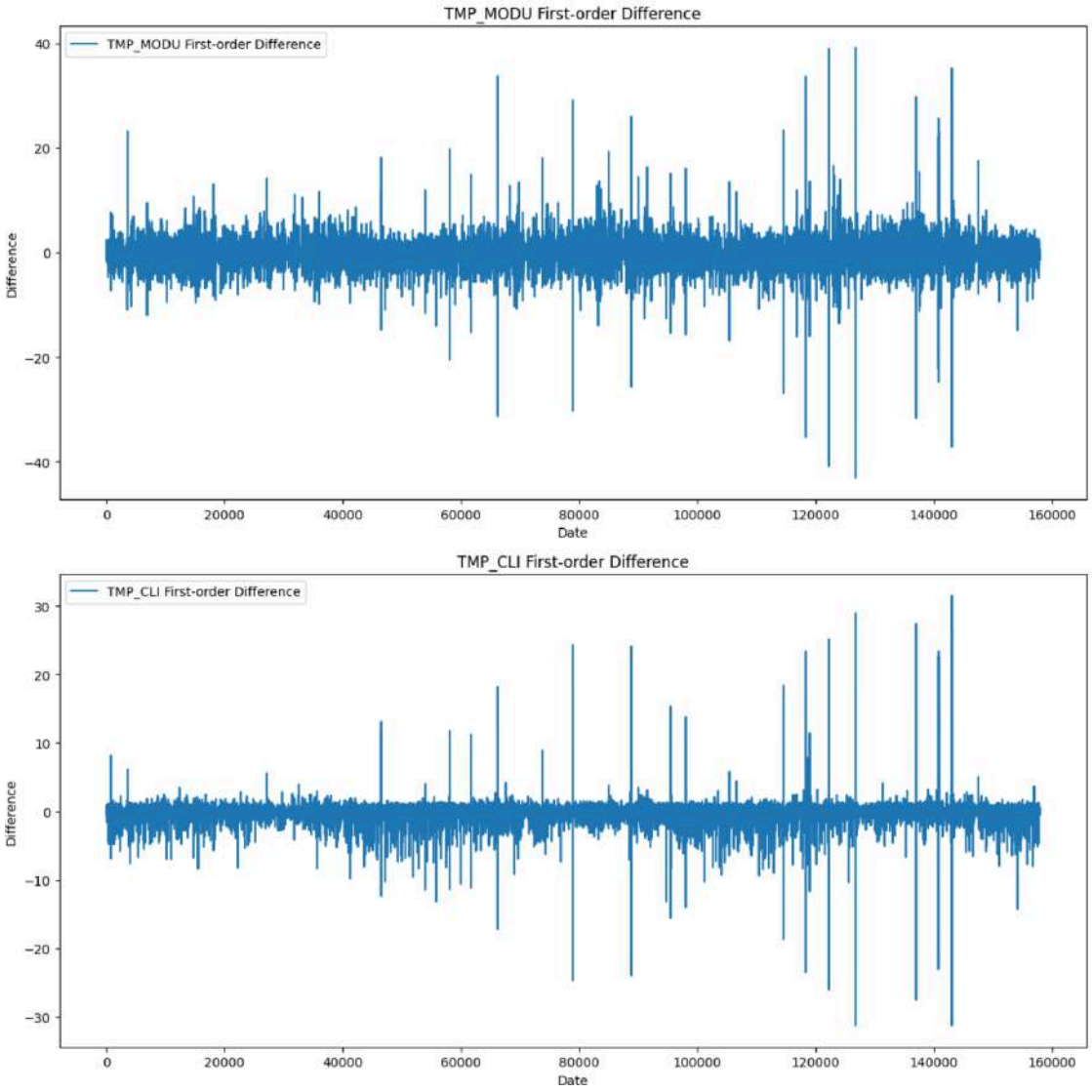


Figure 46. First-order difference of TMP_MODU and TMP_CLI

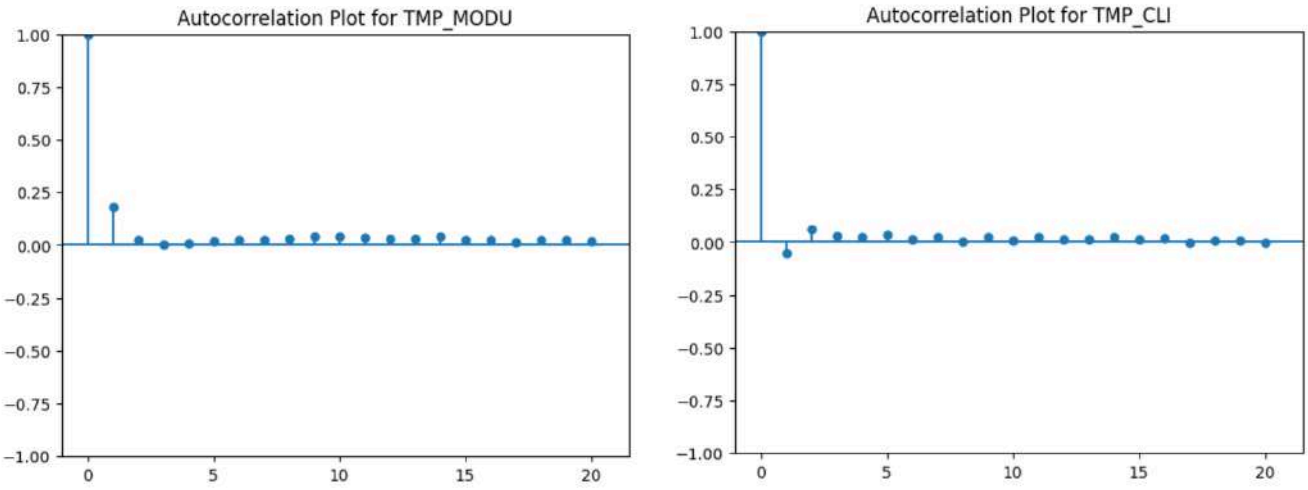


Figure 47. Autocorrelation of TMP_MODU and TMP_CLI

Question 3

Using solar power generation data, pv_2years_eng, to ensure stationarity of the time series, calculate the first-order difference and calculate the partial autocorrelation lag (lag) using the Statsmodels pacf() function and plot_acf().).

This code segment prepares the data by calculating the first-order difference for the specified columns to make them stationary, which is a common preprocessing step in time series analysis.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import pacf
from statsmodels.graphics.tsaplots import plot_acf

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']

# Ensure stationarity of the time series by calculating first-order difference
for column in columns_of_interest:
    data[column + '_diff'] = data[column].diff()
```

Importing Libraries: It imports necessary libraries such as pandas for data manipulation, matplotlib.pyplot for plotting, and functions for calculating partial autocorrelation (pacf) and plotting autocorrelation (plot_acf) from statsmodels.

Load Data: It reads the solar power generation data from the specified CSV file into a pandas DataFrame named **data.**

Defining Columns of Interest:
It defines the columns of interest from the loaded dataset. In this case, the columns of interest are 'TMP_MODU' (Module temperature) and 'TMP_CLI' (Outdoor temperature).

Ensuring Stationarity of the Time Series:
It ensures stationarity of the time series data by calculating the first-order difference for each column of interest. The first-order difference is computed using the **diff()** function in pandas, which calculates the difference between consecutive elements of the specified column.

This code visually inspects the first-order difference of the time series data and examines the partial autocorrelation to understand the relationship between observations at different time lags.

```python
# Plot the first-order difference
plt.figure(figsize=(12, 6*len(columns_of_interest)))
for i, column in enumerate(columns_of_interest, 1):
    plt.subplot(len(columns_of_interest), 1, i)
    plt.plot(data.index, data[column + '_diff'], label=f'{column} First-order Difference')
    plt.title(f'{column} First-order Difference')
    plt.xlabel('Date')
    plt.ylabel('Difference')
    plt.legend()
plt.tight_layout()
plt.show()

# Calculate partial autocorrelation lag using pacf() function
for column in columns_of_interest:
    pacf_result = pacf(data[column + '_diff'].dropna(), nlags=20)
    print(f'Partial Autocorrelation for {column}: {pacf_result}')

# Plot partial autocorrelation using plot_acf() function
for column in columns_of_interest:
    plot_acf(data[column + '_diff'].dropna(), lags=20, title=f'Partial Autocorrelation Plot for {column}')
    plt.xlabel('Lag')
    plt.ylabel('Partial Autocorrelation')
    plt.show()
```

Plotting First-Order Difference: It plots the first-order difference of the specified columns. It iterates through each column of interest, creates a subplot for each column, and plots the first-order difference against the index (which represents time). It also adds appropriate labels, including the column name and 'First-order Difference', to the plot.

Calculating Partial Autocorrelation:
It calculates the partial autocorrelation of the first-order difference for each column using the **pacf()** function from statsmodels. The **nlags** parameter specifies the number of lags to include in the calculation.

Plotting Partial Autocorrelation: It plots the partial autocorrelation of the first-order difference for each column using the **plot_acf()** function from statsmodels. It iterates through each column of interest, plots the partial autocorrelation with specified number of lags, and adds a title indicating the column name.

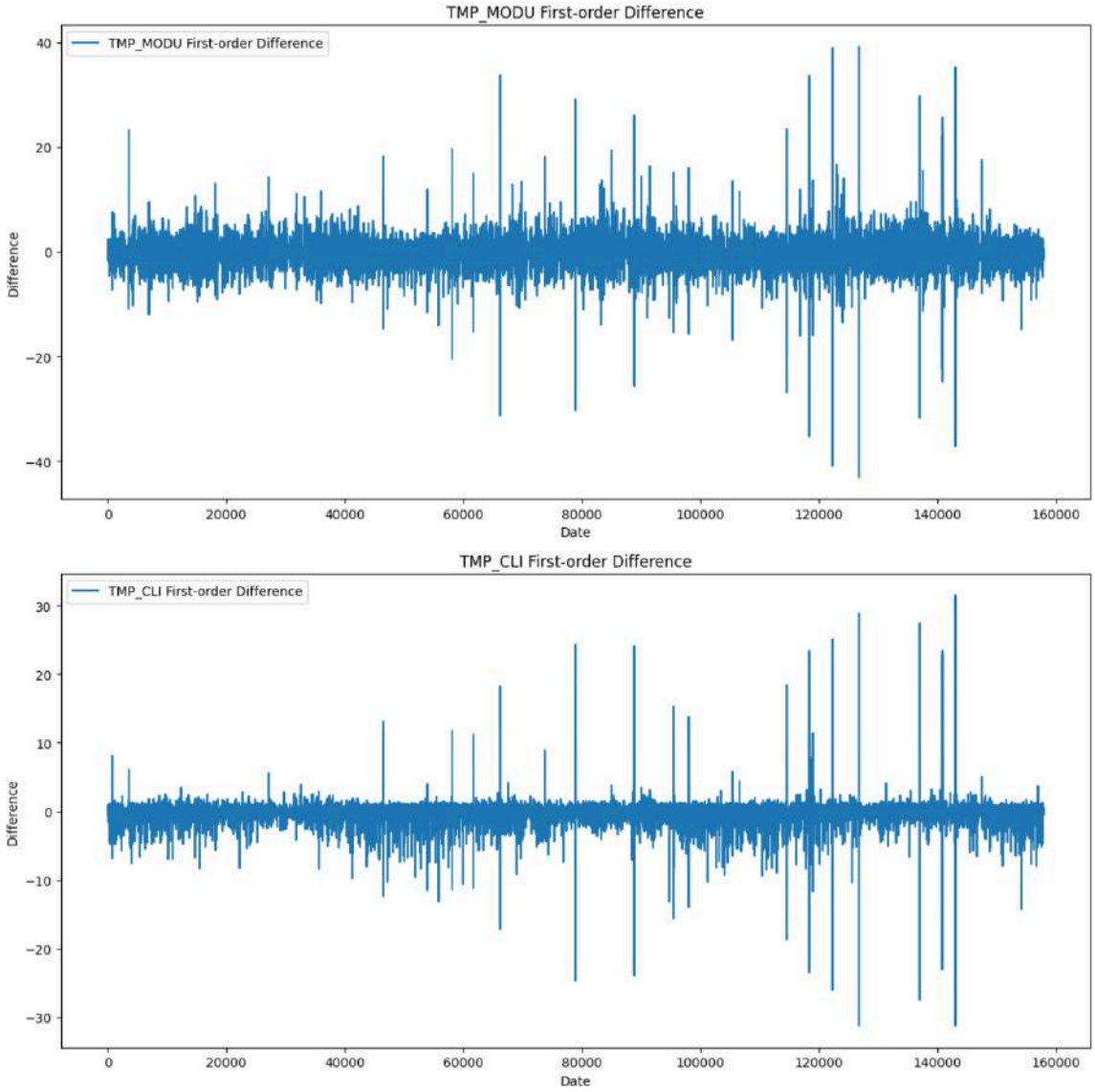


Figure 48. First-order difference of TMP_MODU and TMP_CLI

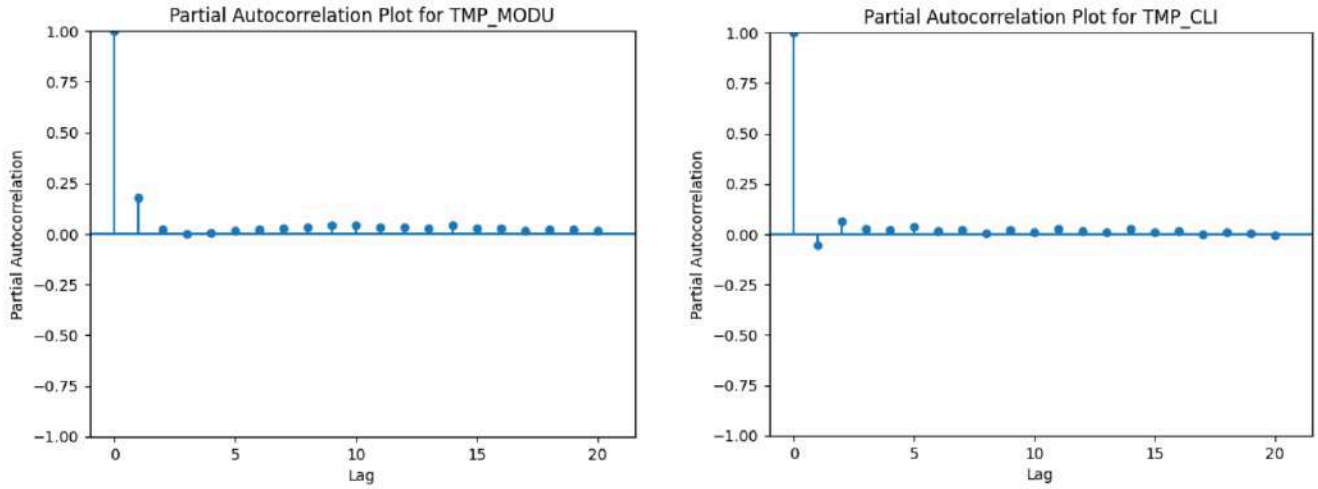


Figure 49. Partial Autocorrelation of TMP_MODU and TMP_CLI

Question 4

Using solar power generation data, pv_2years_eng, implement it as a moving average (MA), and display the moving average prediction results as plot().

This code generates separate plots for each column of interest, showing the original data and the corresponding moving average prediction results.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']

# Plot original data and moving average for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))

    # Plot original data
    plt.plot(data.index, data[column], label=f'Original {column}')

    # Implement moving average (MA)
    window_size = 10  # Specify the window size for the moving average
    ma = data[column].rolling(window=window_size).mean()
    plt.plot(data.index, ma, label=f'{column} MA ({window_size} periods)')

    plt.title(f'Moving Average Prediction Results for {column}')
    plt.xlabel('Date')
    plt.ylabel('Value')
    plt.legend()
    plt.show()
```

Import: It imports necessary libraries such as Pandas for data manipulation, Matplotlib for plotting, and specific functions.

Load Data: It imports the pandas library and reads the solar power generation data from the provided CSV file using the **pd.read_csv()** function. The loaded data is stored in the variable **data.**

Plot Original Data and Moving Average: For each column of interest, it iterates through the **specified columns (columns_of_interest)** and creates a separate plot using **plt.figure(figsize=(12, 6)).** Within each plot:

- It plots the original data against the index using **plt.plot(data.index, data[column], label=f'Original {column}').**
- It calculates the moving average (MA) with a window size of 10 periods using **data[column].rolling(window=window_size).mean()** and then plots the moving average on the same plot.
- It adds a title to the plot indicating the column name and the task being performed **(plt.title(f'Moving Average Prediction Results for {column}')).**
- It adds labels to the x-axis ('Date') and y-axis ('Value') using **plt.xlabel('Date')** and **plt.ylabel('Value'),** respectively.
- It adds a legend to distinguish between the original data and the moving average using **plt.legend().**
- Finally, it displays the plot using **plt.show().**

Defining Columns of Interest: It specifies the columns of interest for which moving averages will be calculated. In this case, it selects 'TMP_MODU' (Module temperature) and 'TMP_CLI' (Outdoor temperature).

This code calculates and plots the combined moving average of the specified columns of interest alongside their original data.

```python
# Calculate combined moving average
window_size = 10  # Specify the window size for the moving average
combined_ma = data[columns_of_interest].rolling(window=window_size).mean().mean(axis=1)

# Plot combined moving average
plt.figure(figsize=(12, 6))

# Plot original data for each column
for column in columns_of_interest:
    plt.plot(data.index, data[column], label=f'Original {column}')

# Plot combined moving average
plt.plot(data.index, combined_ma, label=f'Combined MA ({window_size} periods)', color='black', linestyle='--')

plt.title('Combined Moving Average Prediction Results')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

Calculate Combined Moving Average:

- It calculates the combined moving average for the columns of interest **(columns_of_interest)** by taking the mean of the rolling window over each column.
- The window size for the moving average is specified as **window_size = 10.**
- The result is stored in the variable **combined_ma.**

Plotting Combined Moving Average:

- It creates a new figure with a specified size using **plt.figure(figsize=(12, 6)).**
- For each column of interest, it iterates through the specified columns **(columns_of_interest)** and plots the original data against the index using **plt.plot(data.index, data[column], label=f'Original {column}').**
- After plotting the original data for each column, it plots the combined moving average against the index using **plt.plot(data.index, combined_ma, label=f'Combined MA ({window_size} periods)', color='black', linestyle='--').**
- It adds a title to the plot ('Combined Moving Average Prediction Results') using **plt.title('Combined Moving Average Prediction Results').**
- It adds labels to the x-axis ('Date') and y-axis ('Value') using **plt.xlabel('Date')** and **plt.ylabel('Value'),** respectively.
- It adds a legend to distinguish between the original data and the combined moving average using **plt.legend().**
- Finally, it displays the plot using **plt.show().**

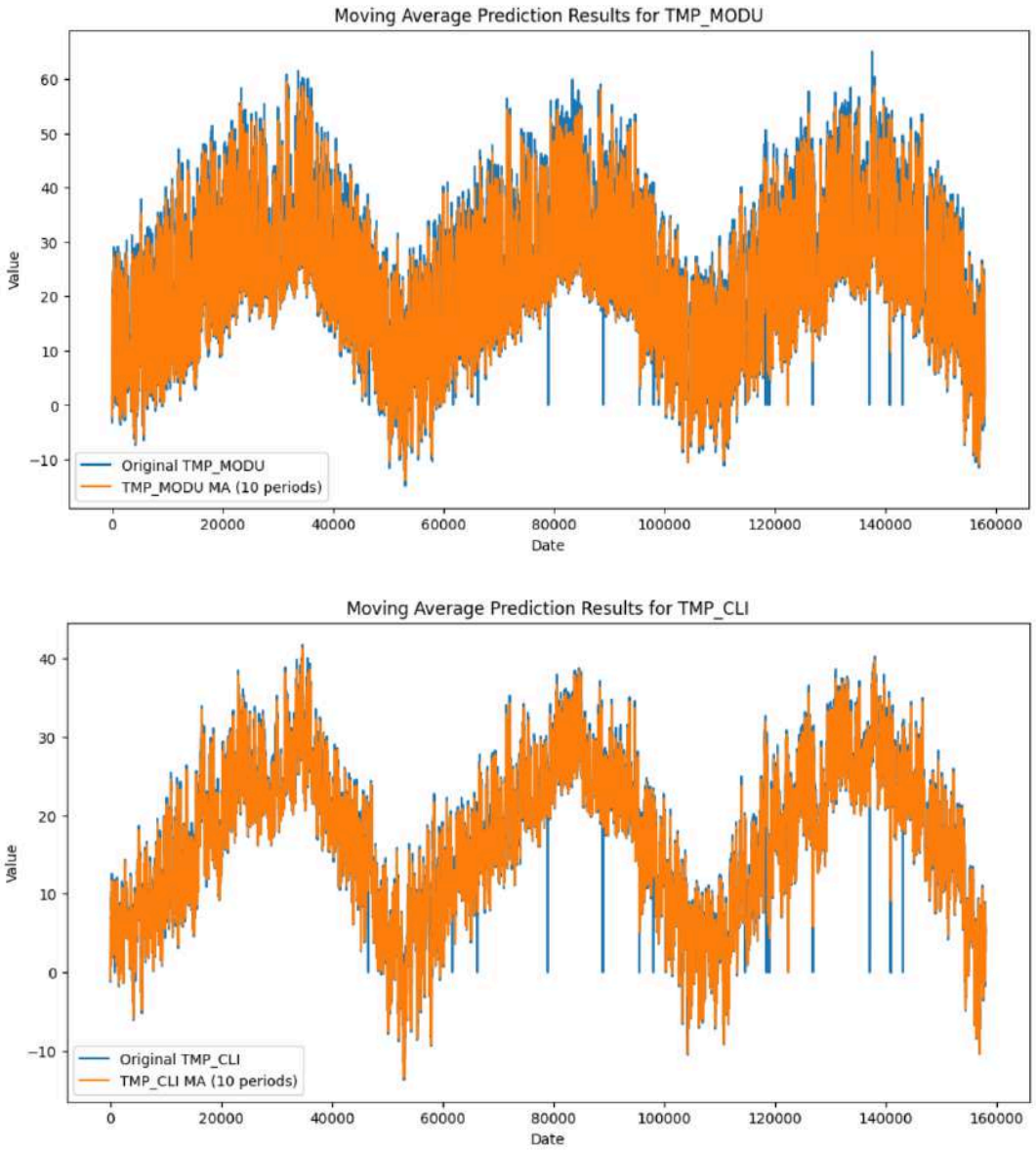


Figure 50. Moving Average of TMP_MODU and TMP_CLI

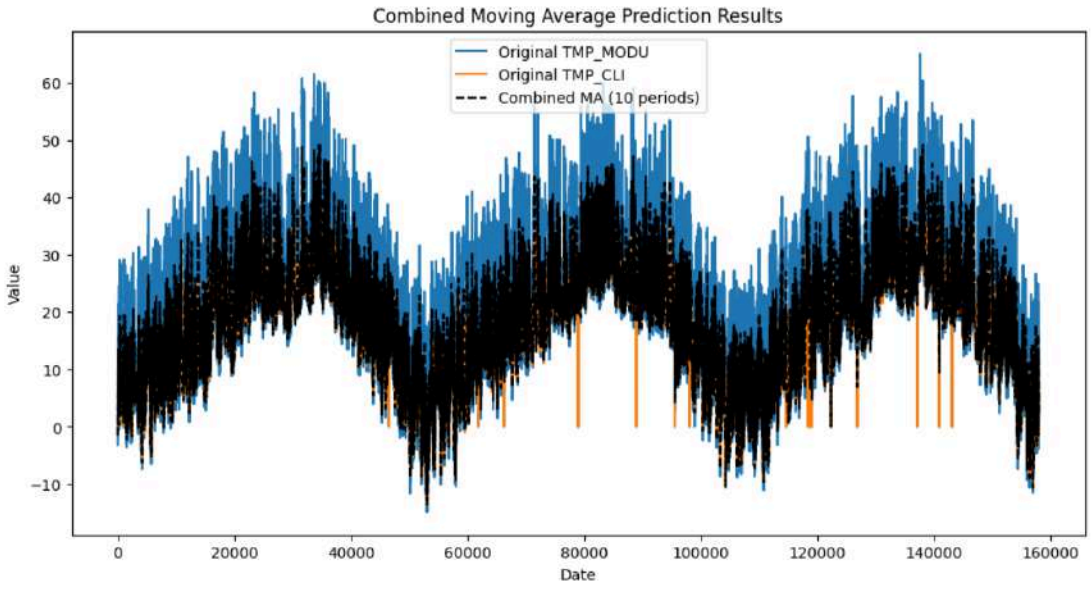


Figure 51. Combined Moving Average of TMP_MODU and TMP_CLI

Question 5

Using solar power generation data, pv_2years_eng, implement triple exponential smoothing and display the triple exponential smoothing results with plot().

This code segment essentially prepares the data, defines the columns of interest, and applies triple exponential smoothing to each specified column, storing the predictions for further analysis or visualization.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv', parse_dates=['date'])

# Prepare the data
data.set_index('date', inplace=True)

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']

# Implement triple exponential smoothing (Holt-Winters method) for each column
predictions = {}

for column in columns_of_interest:
    model = ExponentialSmoothing(data[column], trend='add', seasonal='add', seasonal_periods=12)
    result = model.fit()
    predictions[column] = result.predict(start=data.index[0], end=data.index[-1])
```

Importing Libraries: It imports necessary libraries:

- **pandas** for data manipulation.
- **matplotlib.pyplot** for plotting.
- **ExponentialSmoothing** from **statsmodels.tsa.holtwinters** for implementing triple exponential smoothing.

Loading the Data:

- It loads the solar power generation data from the specified CSV file path into a pandas DataFrame, **data.**
- The 'date' column is parsed as datetime during the data loading process using the **parse_dates** parameter.

Preparing the Data:

- The 'date' column is set as the index of the DataFrame to facilitate time-series analysis.

Defining Columns of Interest:

- It specifies the columns of interest for which triple exponential smoothing will be applied. In this case, it's 'TMP_MODU' (Module temperature) and 'TMP_CLI' (Outdoor temperature).

Implementing Triple Exponential Smoothing:

- For each column of interest, it creates a triple exponential smoothing model using **ExponentialSmoothing** from statsmodels.
- The model is fitted to the data using the **fit()** method, and the predictions are obtained for the entire date range of the dataset using the **predict()** method.

Storing Predictions:

- The predictions for each column are stored in a dictionary named **predictions**, where the keys are the column names.

This code segment allows for the visualization of the original data and the corresponding triple exponential smoothing predictions for each column of interest.

```python
# Plot the predictions for each column separately
for column in columns_of_interest:
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label=f'Original {column}')
    plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions')

    plt.title(f'Triple Exponential Smoothing Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Plotting Predictions:
- It iterates over each column of interest specified in columns_of_interest.

Creating Subplots:
- For each column, it creates a new figure with a specific size (12x6) using plt.figure(figsize=(12, 6)). This ensures that each plot is large enough for clear visualization.

Plotting Original Data:
- It plots the original data against the date index using plt.plot(data.index, data[column], label=f'Original {column}'). This line plots the actual values of the selected column over time.

Plotting Predictions:
- It plots the predicted values obtained from triple exponential smoothing against the date index using plt.plot(predictions[column].index, predictions[column], label=f'{column} Predictions'). These lines represent the forecasted values produced by the triple exponential smoothing model for the selected column.

Customizing Plot:
- It sets the title of the plot to indicate that it shows the triple exponential smoothing predictions for the specific column using plt.title(f'Triple Exponential Smoothing Predictions for {column}').
- The x-axis label is set to 'Date' using plt.xlabel('Date').
- The y-axis label is set to the name of the column being plotted using plt.ylabel(column).
- A legend is added to the plot to differentiate between the original data and the predictions using plt.legend().
- Grid lines are enabled on the plot using plt.grid(True) to aid in visualization.

Displaying the Plot:
- Finally, it displays the plot using plt.show().

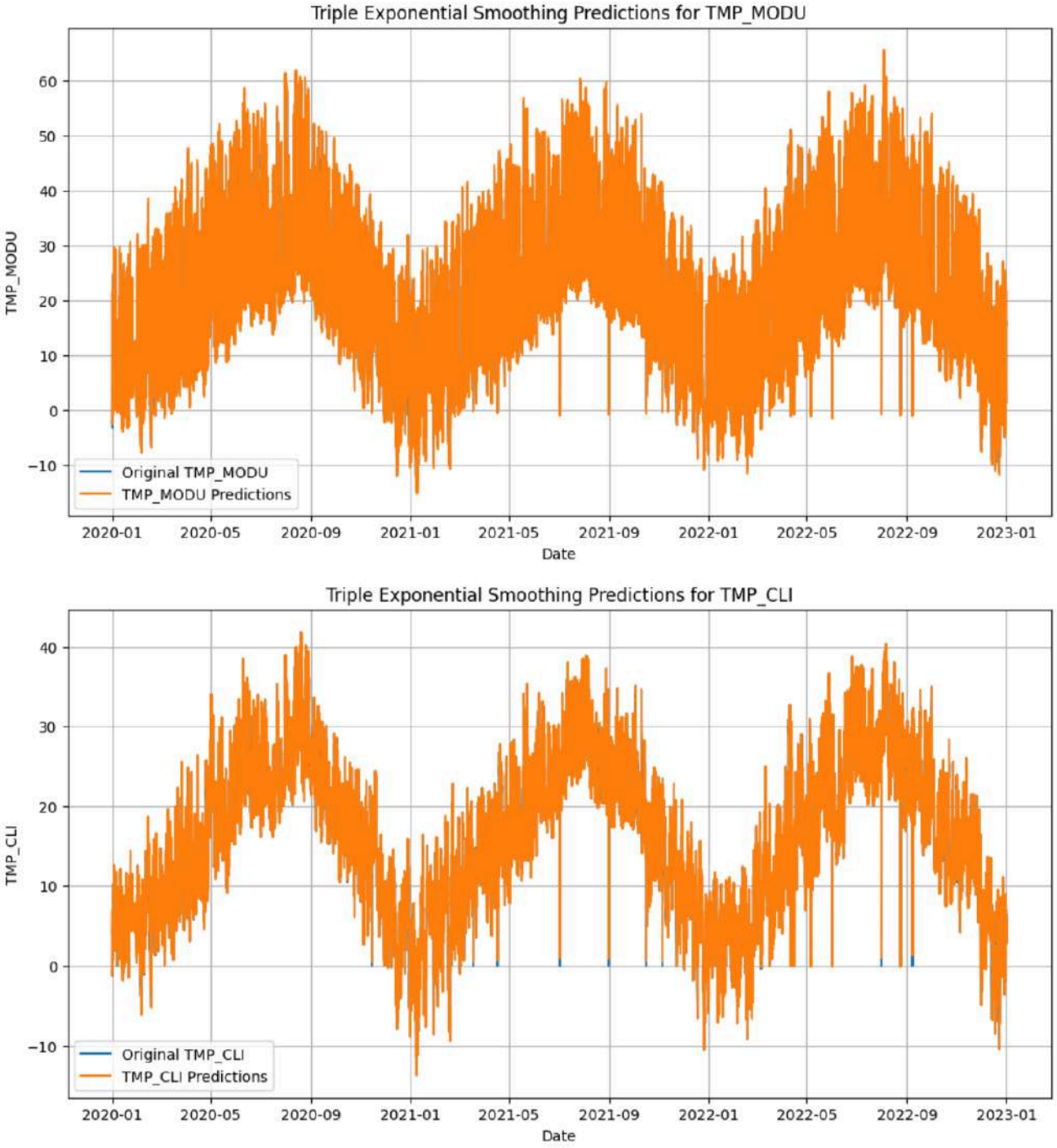


Figure 52. Triple Exponential Smooting Predictions of TMP_MODU and TMP_CLI

Question 6

Using solar power generation data, pv_2years_eng, implement autoregressive (AR) and display the autoregressive (AR) results in plot().

The structure of this code is designed to efficiently apply autoregressive modeling to the specified columns of interest in the solar power generation dataset.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AutoReg

# Load the data
data = pd.read_csv('/kaggle/input/solar-power-generation-data/pv_2years_eng.csv')

# Define the columns of interest
columns_of_interest = ['TMP_MODU', 'TMP_CLI']
```

Importing Libraries: It imports the necessary libraries:

- **pandas** as pd for data manipulation.
- **matplotlib.pyplot** as plt for plotting.
- **AutoReg** from **statsmodels.tsa.ar_model** for autoregressive modeling.

Loading Data:

- It loads the solar power generation data from the specified CSV file using **pd.read_csv().**
- The loaded data is stored in the DataFrame named **data.**

Defining Columns of Interest:

- It defines the columns of interest as **['TMP_MODU', 'TMP_CLI'].**
- These columns likely represent temperature data: 'Module (℃) temperature' and 'Outdoor (℃) temperature'.

This structure allows for the efficient visualization of autoregressive (AR) model predictions for each specified column in the dataset.

```python
# Plot autoregressive (AR) results for each column
for column in columns_of_interest:
    # Fit autoregressive (AR) model
    model = AutoReg(data[column].dropna(), lags=1)  # Using lag 1 for simplicity
    model_fit = model.fit()

    # Make predictions
    predictions = model_fit.predict(start=1, end=len(data[column]))

    # Plot original data and autoregressive (AR) predictions
    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data[column], label='Original Data')
    plt.plot(data.index, predictions, label='Autoregressive (AR) Predictions', color='orange')

    plt.title(f'Autoregressive (AR) Model Predictions for {column}')
    plt.xlabel('Date')
    plt.ylabel(column)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Making Predictions:

- After fitting the model, it generates predictions using the predict method of the fitted model.
- It specifies the start and end indices for predictions using **start=1** and **end=len(data[column]),** respectively. This ensures that predictions start from the second time step and continue until the end of the data.

Plotting Autoregressive (AR) Results:

- It loops over each column specified in **columns_of_interest.**
- For each column, it performs the following steps:

Fitting Autoregressive (AR) Model:

- It initializes an autoregressive (AR) model using the AutoReg function from **statsmodels.**
- The model is fitted to the non-missing values of the current column using **data[column].dropna().**
- It specifies **lags=1** to use a lag of 1 for simplicity. This means the model uses the value of the current time step to predict the next time step.

Plotting:

- It creates a new figure with a specified size of (12, 6) using **plt.figure(figsize=(12, 6)).**
- It plots the original data and the autoregressive (AR) predictions on the same plot using **plt.plot().**
- The original data is plotted against the index of the DataFrame **(data.index),** while the predictions are plotted against the same index.
- The original data is labeled as "Original Data", and the predictions are labeled as "Autoregressive (AR) Predictions" with the specified color 'orange'.
- It sets the title of the plot dynamically based on the current column being processed.
- The x-axis label is set to 'Date', and the y-axis label is set to the name of the current column.
- It adds a legend to differentiate between the original data and predictions.
- It displays a grid for better visualization using **plt.grid(True).**
- Finally, it shows the plot using **plt.show().**

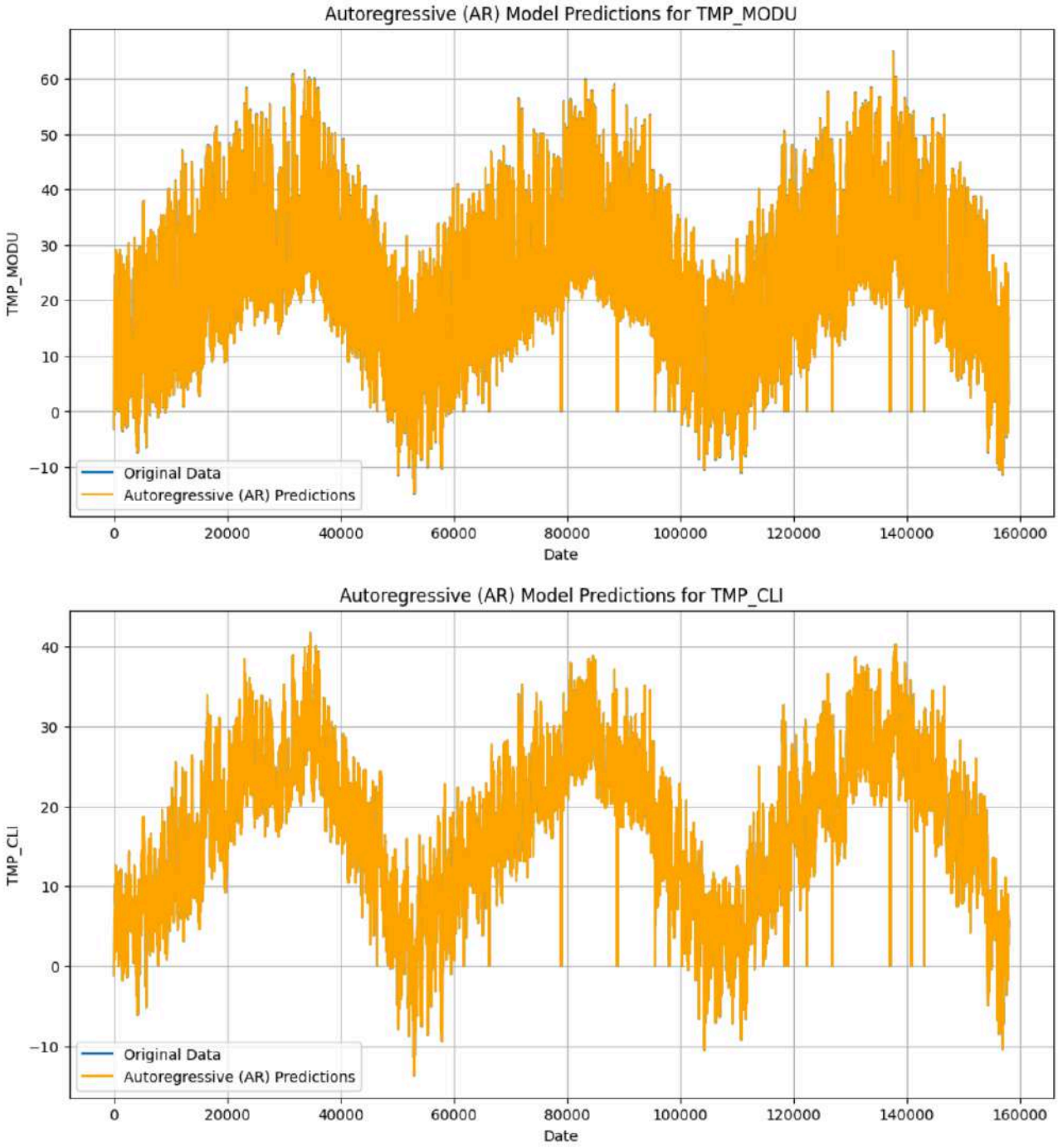


Figure 53. Autoregressive (AR) Model Predictions of TMP_MODU and TMP_CLI

VIEW THE ASSIGNMENT

CONTACT

Salaki Reynaldo Joshua
010 5865 5071
salakirjoshua@kangwon.ac.kr