Phase-4

Measure energy consumption

Development part 2

By,

R.vikram

S.karmegam

M.gokul

R.vinayaprasath

G.Dhanalakshmi.

Topic:

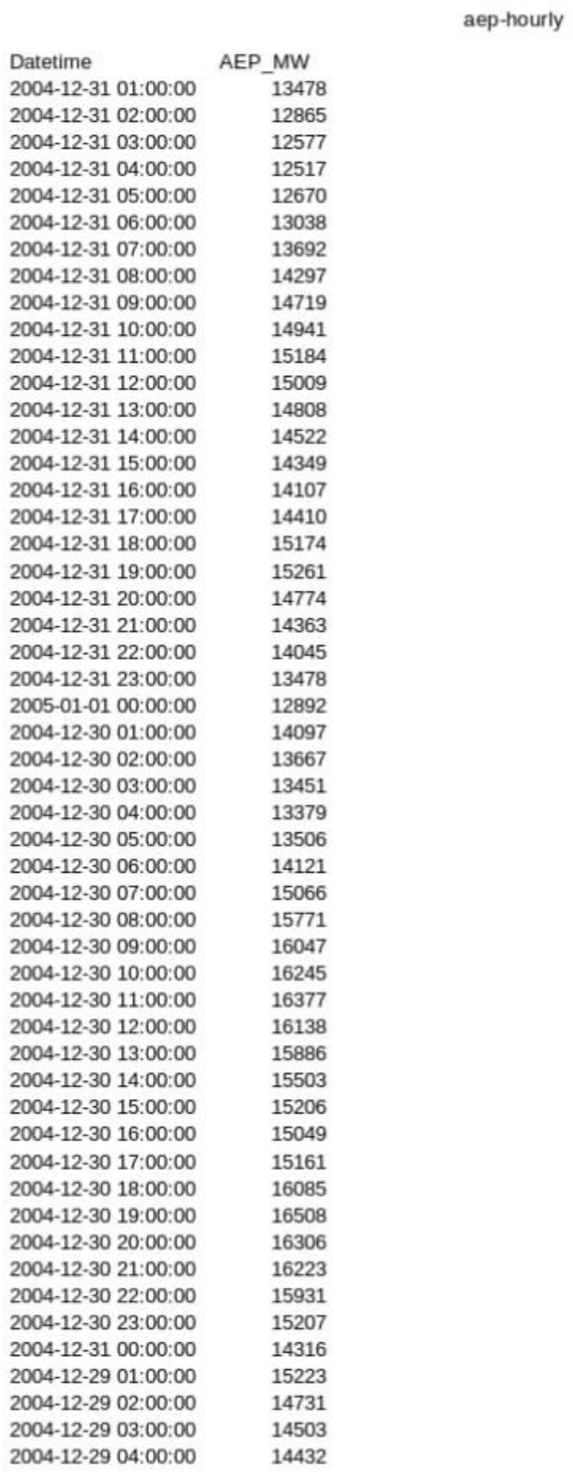
building the project by performing different activities like feature engineering, model training and evaluation .

Feature engineering:

Introduction:

- Feature engineering is a crucial aspect of measuring energy consumption effectively. It involves selecting, creating, and transforming variables to extract meaningful information from energy data. By carefully crafting features, we can uncover patterns, trends, and insights that help optimize energy usage, identify inefficiencies, and make informed decisions. In this context, feature engineering plays a pivotal role in enhancing the accuracy and efficiency of energy consumption analysis and management systems.

Dataset:



aep-hourly

Datetime	AEP_MW
2004-12-31 01:00:00	13478
2004-12-31 02:00:00	12865
2004-12-31 03:00:00	12577
2004-12-31 04:00:00	12517
2004-12-31 05:00:00	12670
2004-12-31 06:00:00	13038
2004-12-31 07:00:00	13692
2004-12-31 08:00:00	14297
2004-12-31 09:00:00	14719
2004-12-31 10:00:00	14941
2004-12-31 11:00:00	15184
2004-12-31 12:00:00	15009
2004-12-31 13:00:00	14808
2004-12-31 14:00:00	14522
2004-12-31 15:00:00	14349
2004-12-31 16:00:00	14107
2004-12-31 17:00:00	14410
2004-12-31 18:00:00	15174
2004-12-31 19:00:00	15261
2004-12-31 20:00:00	14774
2004-12-31 21:00:00	14363
2004-12-31 22:00:00	14045
2004-12-31 23:00:00	13478
2005-01-01 00:00:00	12892
2004-12-30 01:00:00	14097
2004-12-30 02:00:00	13667
2004-12-30 03:00:00	13451
2004-12-30 04:00:00	13379
2004-12-30 05:00:00	13506
2004-12-30 06:00:00	14121
2004-12-30 07:00:00	15066
2004-12-30 08:00:00	15771
2004-12-30 09:00:00	16047
2004-12-30 10:00:00	16245
2004-12-30 11:00:00	16377
2004-12-30 12:00:00	16138
2004-12-30 13:00:00	15886
2004-12-30 14:00:00	15503
2004-12-30 15:00:00	15206
2004-12-30 16:00:00	15049
2004-12-30 17:00:00	15161
2004-12-30 18:00:00	16085
2004-12-30 19:00:00	16508
2004-12-30 20:00:00	16306
2004-12-30 21:00:00	16223
2004-12-30 22:00:00	15931
2004-12-30 23:00:00	15207
2004-12-31 00:00:00	14316
2004-12-29 01:00:00	15223
2004-12-29 02:00:00	14731
2004-12-29 03:00:00	14503
2004-12-29 04:00:00	14432

:

Feature Engineering for Time Series :

That is something that looks like:

Time 1, value 1

Time 2, value 2

Time 3, value 3

To something that looks like:

Input 1, output 1

Input 2, output 2

Input 3, output 3

- Input variables are also called features in the field of machine learning, and the task before us is to create or invent new input features from our time series dataset. Ideally, we only want input features that best help the learning methods model the relationship between the inputs (X) and the outputs (y) that we would like to predict.

Features:

- **Date Time Features:** these are components of the time step itself for each observation.

- **Lag Features:** these are values at prior time steps.

- **Window Features:** these are a summary of values over a fixed window of prior time steps.

Goal of Feature Engineering:

- The goal of feature engineering is to provide strong and ideally simple relationships between new input features and the output feature for the supervised learning algorithm to model.

- In effect, we are are moving complexity.

- Complexity exists in the relationships between the input and output data. In the case of time series, there is no concept of input and output variables; we must invent these too and frame the supervised learning problem from scratch.

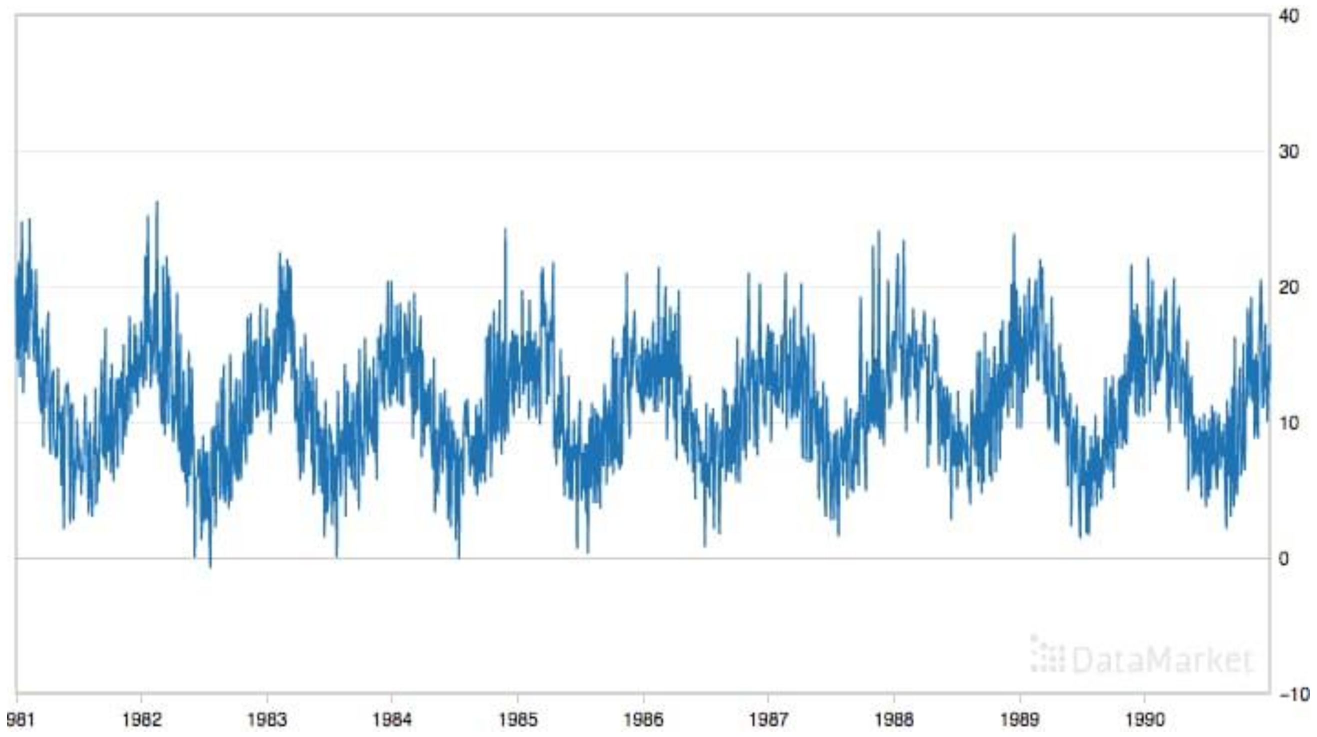
- We may lean on the capability of sophisticated models to decipher the complexity of the problem. We can make the job for these models easier (and even use simpler models) if we can better expose the inherent relationship between inputs and outputs in the data.

- The difficulty is that we do not know the underlying inherent functional relationship between inputs and outputs that we're trying to expose. If we did know, we probably would not need machine learning.

- performance of models developed on the supervised learning datasets or "views" of the problem

- Only focus on a univariate (one variable) time series dataset in the examples, but these methods are just as applicable to multivariate time series problems.

```
"Date","Temperature"
"1981-01-01",20.7
"1981-01-02",17.9
"1981-01-03",18.8
"1981-01-04",14.6
"1981-01-05",15.8
```



Value(t-1), Value(t+1)

Value(t-1), Value(t+1)

Value(t-1), Value(t+1)

The Pandas library provides the shift() function to help create these shifted or lag features from a time series dataset. Shifting the dataset by 1 creates the t-1 column, adding a NaN (unknown) value for the first row. The time series dataset without a shift represents the t+1.

```
Shifted, Original
NaN, 20.7
20.7, 17.9
17.9, 18.8
```

- To concatenate the shifted columns together into a new DataFrame using the concat() function along the column axis (axis=1).

- below is an example of creating a lag feature for our daily temperature dataset. The values are extracted from the loaded series and a shifted and unshifted list of these values is created. Each column is also named in the DataFrame

```
from pandas import read_csv
 from pandas import DataFrame
 from pandas import concat
 series = read_csv('daily-min-temperatures.csv', header=0, index_col=0)
 temps = DataFrame(series.values)
dataframe = concat([temps.shift(1), temps], axis=1)
 dataframe.columns = ['t-1', 't+1']
  print(dataframe.head(5))
```

Running the example prints the first 5 rows of the new dataset with the lagged feature.

```
t-1 t+1

0 NaN 20.7

1 20.7 17.9

2 17.9 18.8

3 18.8 14.6

4 14.6 15.8
```

- **The addition of lag features is called the sliding window method, in this case with a window width of 1. It is as though we are sliding our focus along the time series for each observation with an interest in only what is within the window width.**

expand the window width and include more lagged features. For example, below is the above case modified to include the last 3 observed values to predict the value at the next time step.

```
from pandas import read_csv

from pandas import DataFrame

from pandas import concat

series = read_csv('daily-min-temperatures.csv', header=0, index_col=0)

temps = DataFrame(series.values)

dataframe = concat([temps.shift(3), temps.shift(2), temps.shift(1), temps], axis=1)

dataframe.columns = ['t-3', 't-2', 't-1', 't+1']

print(dataframe.head(5))
```

Running this example prints the first 5 rows

```
t-3 t-2 t-1 t+1

0 NaN NaN NaN 20.7

1 NaN NaN 20.7 17.9

2 NaN 20.7 17.9 18.8

3 20.7 17.9 18.8 14.6

4 17.9 18.8 14.6 15.8
```

Another example:

mean(t-2, t-1), t+1

mean(20.7, 17.9), 18.8

19.3, 18.8

- Pandas provides a rolling() function that creates a new data structure with the window of values at each time step. We can then perform statistical functions on the window of values collected for each time step, such as calculating the mean.First, the series must be shifted. Then the rolling dataset can be created and the mean values calculated on each window of two values.

Here are the values in the first three rolling windows:

#, Window Values

1, NaN

2, NaN, 20.7

3, 20.7, 17.9

The example below demonstrates how to do this with Pandas with a window size of 2.

```
from pandas import read_csv

from pandas import DataFrame

from pandas import concat

series = read_csv('daily-min-temperatures.csv', header=0, index_col=0)

temps = DataFrame(series.values)

shifted = temps.shift(1)

window = shifted.rolling(window=2)

means = window.mean()

dataframe = concat([means, temps], axis=1)

dataframe.columns = ['mean(t-2,t-1)', 't+1']

print(dataframe.head(5))
```

- Running the example prints the first 5 rows of the new dataset. We can see that the first two rows are not useful.The first NaN was created by the shift of the series.The second because NaN cannot be used to calculate a mean value.Finally, the third row shows the expected value of 19.30 (the mean of 20.7 and 17.9) used to predict the 3rd value in the series of 18.8.

```
mean(t-2,t-1) t+1

0 NaN 20.7

1 NaN 17.9

2 19.30 18.8

3 18.35 14.6

4 16.70 15.8
```

```
#, Window Values
1, NaN
2, NaN, NaN
3, NaN, NaN, 20.7
4, NaN, 20.7, 17.9
5, 20.7, 17.9, 18.8
```

This suggests that we would not expect usable data until at least the 5th row (array index 4)

```
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-min-temperatures.csv', header=0, index_col=0)
temps = DataFrame(series.values)
width = 3
shifted = temps.shift(width - 1)
window = shifted.rolling(window=width)
dataframe = concat([window.min(), window.mean(), window.max(), temps], axis=1)
dataframe.columns = ['min', 'mean', 'max', 't+1']
print(dataframe.head(5))
```

- Running the code prints the first 5 rows of the new dataset.

- check the correctness of the values on the 5th row (array index 4). We can see that indeed 17.9 is the minimum and 20.7 is the maximum of values in the window of [20.7, 17.9, 18.8].

```
min mean max t+1
0 NaN NaN NaN 20.7
1 NaN NaN NaN 17.9
2 NaN NaN NaN 18.8
3 NaN NaN NaN 14.6
4 17.9 19.133333 20.7 15.8
```

Expanding Window Statistics:

- Another type of window that may be useful includes all previous data in the series.This is called an expanding window and can help with keeping track of the bounds of observable data. Like the rolling() function on DataFrame, Pandas provides an expanding() function that collects sets of all prior values for each time step.

These lists of prior numbers can be summarized and included as new features. For example, below are the lists of numbers in the expanding window for the first 5 time steps of the series:

```
#, Window Values
1, 20.7
2, 20.7, 17.9,
3, 20.7, 17.9, 18.8
4, 20.7, 17.9, 18.8, 14.6
5, 20.7, 17.9, 18.8, 14.6, 15.8
```

- Again, shift the series one-time step to ensure that the output value we wish Therefore the input windows look as follows:

```
#, Window Values
1, NaN
2, NaN, 20.7
3, NaN, 20.7, 17.9,
4, NaN, 20.7, 17.9, 18.8
5, NaN, 20.7, 17.9, 18.8, 14.6
```

Below is an example of calculating the minimum, mean, and maximum values of the expanding window on the daily temperature dataset.

```
# create expanding window features
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-min-temperatures.csv', header=0, index_col=0)
temps = DataFrame(series.values)
window = temps.expanding()
dataframe = concat([window.min(), window.mean(), window.max(), temps.shift(-
1)], axis=1)
dataframe.columns = ['min', 'mean', 'max', 't+1']
print(dataframe.head(5))
```

Running the example prints the first 5 rows of the dataset.

```
   min      mean   max   t+1
0  20.7  20.700000  20.7  17.9
1  17.9  19.300000  20.7  18.8
2  17.9  19.133333  20.7  14.6
3  14.6  18.000000  20.7  15.8
4  14.6  17.560000  20.7  15.8
```

Modeling energy consumption:

Objectives:

- **Create a stationary time-series.**
- **Test for autocorrelation of time series values.**

Introduction:

- we used metrics to forecast one or more timesteps of a time-series dataset. These forecasts help demonstrate some of the characteristic features of time-series, but as we saw when we evaluated the results they may not make very accurate forecasts. There are some types of random time-series data for which using a baseline metric to forecast a single timestamp ahead is the only option. Since that doesn't apply to the smart meter data - that is, energy consumption values are not random - we will pass over that topic for now., the smart meter data have characteristics that make them good candidates for methods that account for trends, autoregression, and one or more types of seasonality. We will develop these concepts over the next several lessons, beginning here with autocorrelation and the use of moving averages to make forecasts using autocorrelated data.

Create a subset to demonstrate autocorrelation

- First ,import the necessary libraries. In additional to Pandas, Numpy, and Matplotlib we are also importing modules from **`statsmodels`** and **`sklearn`**. These are Python libraries that come with many methods for modeling and machine learning.

Program:

```
Import pandas as pd

Import numpy as np

Import matplotlib.pyplot as plt

From statsmodels.tsa.stattools import adfuller

From statsmodels.graphics.tsaplots import plot_acf

From statsmodels.tsa.statespace.sarimax import SARIMAX

From sklearn.metrics import mean_squared_error

From sklearn.metrics import mean_absolute_error
```

Read the data. In this case we are using just a single smart meter.

```
Df = pd.read_csv("../../data/ladpu_smart_meter_data_01.csv")

Print(df.info())
```

Output:

```
<class ‘pandas.core.frame.DataFrame’>

RangeIndex: 105012 entries, 0 to 105011

Data columns (total 5 columns):

 #   Column        Non-Null Count   Dtype

---  ------        --------------   -----

 0   INTERVAL_TIME  105012 non-null  object
 1    1   METER_FID      105012 non-null  int64

 2   START_READ     105012 non-null  float64

 3   END_READ       105012 non-null  float64

 4   INTERVAL_READ  105012 non-null  float64

Dtypes: float64(3), int64(1), object(1)

Memory usage: 4.0+ MB

None
```

Set the datetime index and resample to a daily frequency.

```
Df.set_index(pd.to_datetime(df[“INTERVAL_TIME”]), inplace=True)

Df.sort_index(inplace=True)

Print(df.info())
```

OUTPUT:

```
<class ‘pandas.core.frame.DataFrame’>

DatetimeIndex: 105012 entries, 2017-01-01 00:00:00 to 2019-12-31 23:45:00

Data columns (total 5 columns):

 #   Column        Non-Null Count   Dtype

---  ------        --------------   -----

 0   INTERVAL_TIME  105012 non-null  object
 1    1   METER_FID      105012 non-null  int64

 2   START_READ     105012 non-null  float64

 3   END_READ       105012 non-null  float64

 4   INTERVAL_READ  105012 non-null  float64

Dtypes: float64(3), int64(1), object(1)

Memory usage: 4.8+ MB

None
```

Python:

```
Daily_data = pd.DataFrame(df.resample(“D”)[“INTERVAL_READ”].sum())

Print(daily_data.head())
```

OUTPUT:

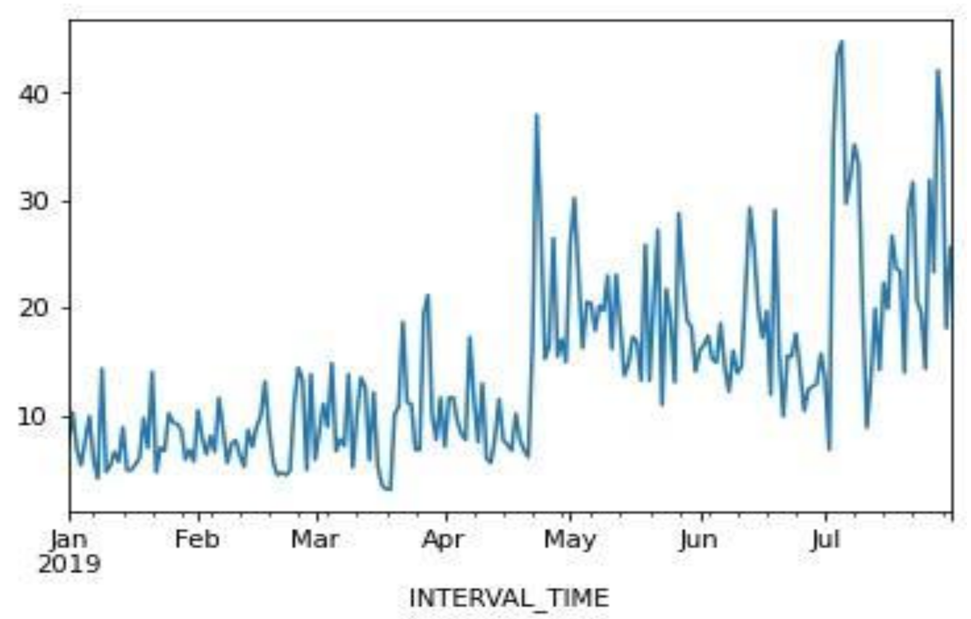
```
             INTERVAL_READ

INTERVAL_TIME

2017-01-01          11.7546

2017-01-02          15.0690

2017-01-03          11.6406

2017-01-04          22.0788
```

2017-01-05   12.8070

Subset to January – July, 2019 and plot the data.

Jan_july_2019 = daily_data.loc[“2019-01”: “2019-07”].copy()

Jan_july_2019[“INTERVAL_READ”].plot()



Differencing and autocorrelation

- In order to make a forecast using the moving average model, however, the data need to be stationary. That is, we need to remove trends from the data. We can test for stationarity using the adfuller function from statsmodels.

PYTHON:

Adfuller_test = adfuller(jan_july_2019[“INTERVAL_READ”])

Print(f’ADFuller result: {adfuller_test[0]}’)

Print(f’p-value: {adfuller_test[1]}’)

OUTPUT:

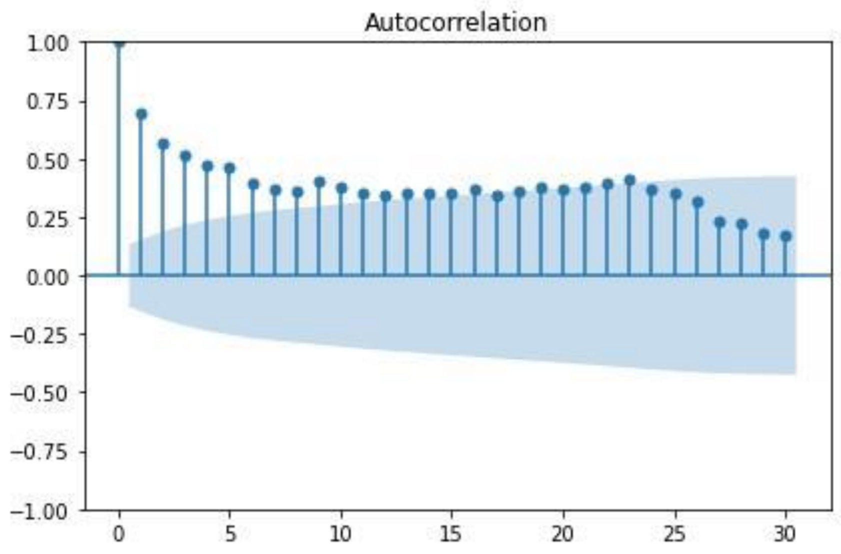
```
ADFuller result: -2.533089941397639

p-value: 0.10762933815081588
```

- The p-value above is greater than 0.05, which in this case indicates that the data are not stationary. That is, there is a trend in the data.

test for autocorrelation:

PYTHON:

```
Plot_acf(jan_july_2019["INTERVAL_READ"], lags=30)

Plt.tight_layout()
```



- Differencing data this way creates a Numpy array of values that represent the difference between one INTERVAL_READ value and the next. We can see this by comparing the head of the jan_july_2019 dataframe with the first five differenced values.

PYTHON:

```
Jan_july_2019_differenced = np.diff(jan_july_2019[“INTERVAL_READ”], n=1)
Print(“Head of dataframe:”, jan_july_2019.head())
Print(“\nDifferenced values:”, jan_july_2019_differenced[:5])
```

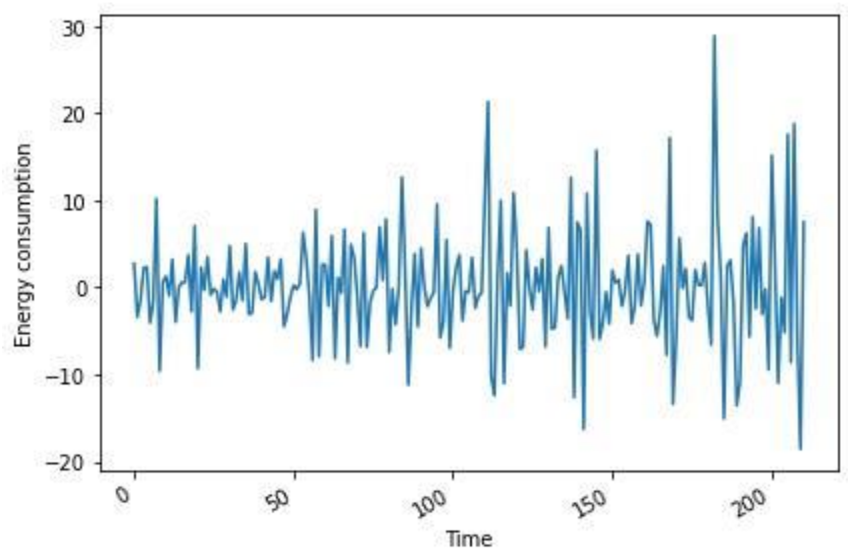
OUTPUT:

```
Head of dataframe:             INTERVAL_READ
INTERVAL_TIME
2019-01-01          7.5324
2019-01-02         10.2534
2019-01-03          6.8544
2019-01-04          5.3250
2019-01-05          7.5480
Differenced values: [ 2.721  -3.399  -1.5294  2.223   2.3466]
```

Plotting the result shows that there are no obvious trends in the differenced data.

PYTHON:

```
Fig, ax = plt.subplots()
Ax.plot(jan_july_2019_differenced)
Ax.set_xlabel(‘Time’)
Ax.set_ylabel(‘Energy consumption’)
Fig.autofmt_xdate()
Plt.tight_layout()
```



Moving average forecast:

- training data, we will use the 90% of the dataset. The remaining 10% of the data will be used to evaluate the performance of the moving average forecast in comparison with a baseline forecast.
- Since the differenced data is a numpy array, we also need to convert it to a dataframe.

PYTHON:

Jan_july_2019_differenced = pd.DataFrame(jan_july_2019_differenced,

Columns=[“INTERVAL_READ”])

Train = jan_july_2019_differenced[:int(round(len(jan_july_2019_differenced) * .9, 0))] # ~90% of data

Test = jan_july_2019_differenced[int(round(len(jan_july_2019_differenced) * .9, 0)):] # ~10% of data

Print(“Training data length:”, len(train))

Print(“Test data length:”, len(test))

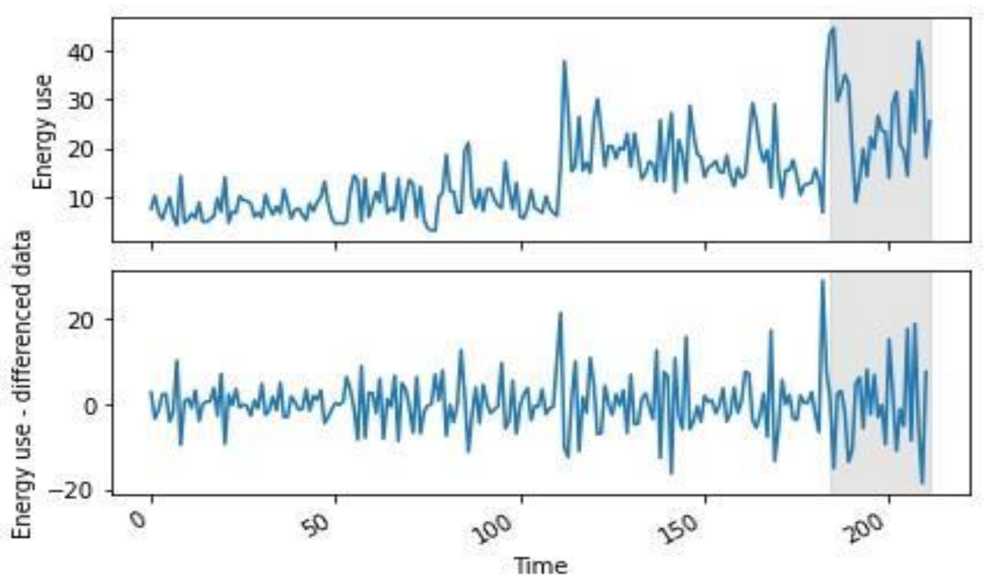
OUTPUT:

```
Training data length: 190

Test data length: 21
```

- **plot the original and differenced data together. The shaded area is the date range for which we will be making and evaluating forecasts.**

PYTHON:

```
Fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, sharex=True)


Ax1.plot(jan_july_2019["INTERVAL_READ"].values)

Ax1.set_xlabel('Time')

Ax1.set_ylabel('Energy use')

Ax1.axvspan(184, 211, color='#808080', alpha=0.2)


 Ax2.plot(jan_july_2019_differenced["INTERVAL_READ"])

Ax2.set_xlabel('Time')

Ax2.set_ylabel('Energy use – differenced data')

Ax2.axvspan(184, 211, color='#808080', alpha=0.2)


 Fig.autofmt_xdate()

Plt.tight_layout()
```



- The moving_average() function is an implementation of the seasonal auto-regressive moving average model that is included in the statsmodels library.

PYTHON:

```
Def last_known(data, training_len, horizon, window):
   Total_len = training_len + horizon
   Pred_last_known = []
   For I in range(training_len, total_len, window):
      Subset = data[:i]
      Last_known = subset.iloc[-1].values[0]
      Pred_last_known.extend(last_known for v in range(window))
  Return pred_last_known
Def moving_average(data, training_len, horizon, ma_order, window):
   Total_len = training_len + horizon
```

```
 Pred_MA =horizol
  For I in range(training_len, total_len, window):
      Model = SARIMAX(data[:i], order=(0,0, ma_order))
      Res = model.fit(disp=False)
      Predictions = res.get_prediction(0, I + window – 1)
      Oos_pred = predictions.predicted_mean.iloc[-window:]
      Pred_MA.extend(oos_pred)
     Return pred_MA
```

- Both functions take the differenced dataframe as input and return a list of predicted values that is equal to the length of the test dataset.

PYTHON:

```
Pred_df = test.copy()
 TRAIN_LEN = len(train)
HORIZON = len(test)
ORDER = 2
WINDOW = 2
Pred_last_value = last_known(jan_july_2019_differenced, TRAIN_LEN, HORIZON,
WINDOW)
Pred_MA = moving_average(jan_july_2019_differenced, TRAIN_LEN, HORIZON,
ORDER, WINDOW)
Pred_df[‘pred_last_value’] = pred_last_value
Pred_df[‘pred_MA’] = pred_MA

Print(pred_df.head())
```

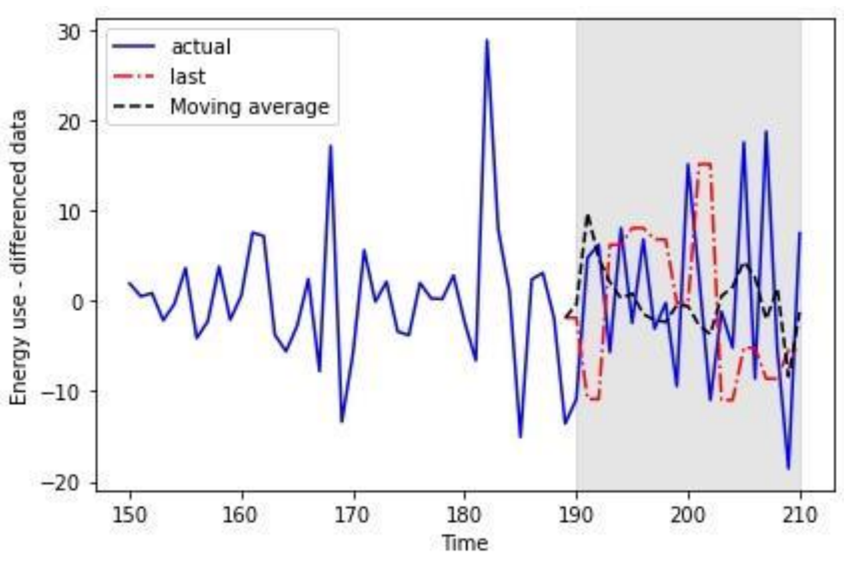
OUTPUT:

```
    INTERVAL_READ  pred_last_value   pred_MA
189      -13.5792          -1.863 -1.870535
190      -10.8660          -1.863 -0.379950
191        4.8054         -10.866  9.760944
192        6.2280         -10.866  4.751856
193       -5.6718           6.228  2.106354
```

- **Plotting the data allows for a visual comparison of the forecasts.**

PYTHON:

```
Fig, ax = plt.subplots()
 Ax.plot(jan_july_2019_differenced[150:][‘INTERVAL_READ’], ‘b-‘, label=’actual’)
Ax.plot(pred_df[‘pred_last_value’], ‘r-.’, label=’last’)
Ax.plot(pred_df[‘pred_MA’], ‘k—‘, label=’Moving average’)
 Ax.axvspan(190, 210, color=’#808080’, alpha=0.2)
Ax.legend(loc=2)
 Ax.set_xlabel(‘Time’)
Ax.set_ylabel(‘Energy use – differenced data’)
Plt.tight_layout()
```



- This time we will use the mean_squared_error function from the sklearn library to evaluate the results.

PYTHON:

```
Mse_last = mean_squared_error(pred_df['INTERVAL_READ'],
pred_df['pred_last_value'])
Mse_MA = mean_squared_error(pred_df['INTERVAL_READ'], pred_df['pred_MA'])
 Print("Last known forecast, mean squared error:", mse_last)
Print("Moving average forecast, mean squared error:", mse_MA)
```

OUTPUT:

```
Last known forecast, mean squared error: 185.5349359527273
Moving average forecast, mean squared error: 86.16289030738947
```

Transform the forecast to original scale:

- the numpy cumsum() method to calculate the cumulative sums of the values in the differenced dataset. We then map these sums to their corresponding rows of the original data.The transformed data are only being applied to the rows of the source data that were used for the test dataset, so we can use the tail() function to inspect the result.

PYTHON:

```
Jan_july_2019['pred_usage'] = pd.Series()

Jan_july_2019['pred_usage'][190:] = jan_july_2019['INTERVAL_READ'].iloc[190] +
pred_df['pred_MA'].cumsum()

Print(jan_july_2019.tail())
```

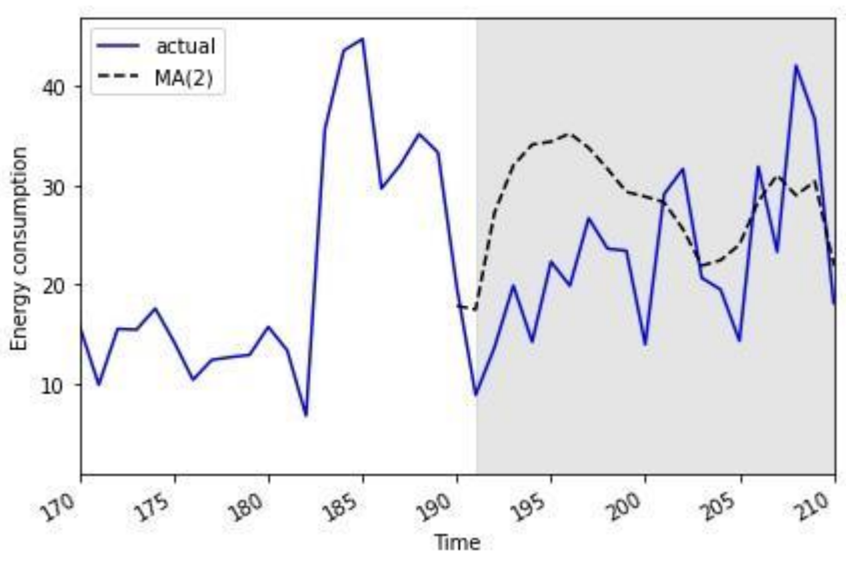
OUTPUT:

```
            INTERVAL_READ  pred_usage

INTERVAL_TIME

2019-07-27         23.2752   31.008305

2019-07-28         42.0504   28.974839

2019-07-29         36.6444   30.387655

2019-07-30         18.0828   22.025803

2019-07-31         25.5774   20.781842
```

- **We can plot the result to compare the transformed forecasts against the actual daily power consumption.**

PYTHON:

```
Fig, ax = plt.subplots()
 Ax.plot(jan_july_2019['INTERVAL_READ'].values, 'b-', label='actual')
Ax.plot(jan_july_2019['pred_usage'].values, 'k—', label='MA(2)')
 Ax.legend(loc=2)
 Ax.set_xlabel('Time')
Ax.set_ylabel('Energy consumption')
Ax.axvspan(191, 210, color='#808080', alpha=0.2)
Ax.set_xlim(170, 210)
 Fig.autofmt_xdate()
Plt.tight_layout()
```



- Finally, to evaluate the performance of the moving average forecast against the actual values in the undifferenced data, we use the mean_absolute_error from the sklearn library.

PYTHON:

```
Mae_MA_undiff =
mean_absolute_error(jan_july_2019[‘INTERVAL_READ’].iloc[191:],
                              Jan_july_2019[‘pred_usage’].iloc[191:])

Print(“Mean absolute error of moving average forecast”, mae_MA_undiff)
```

OUTPUT:

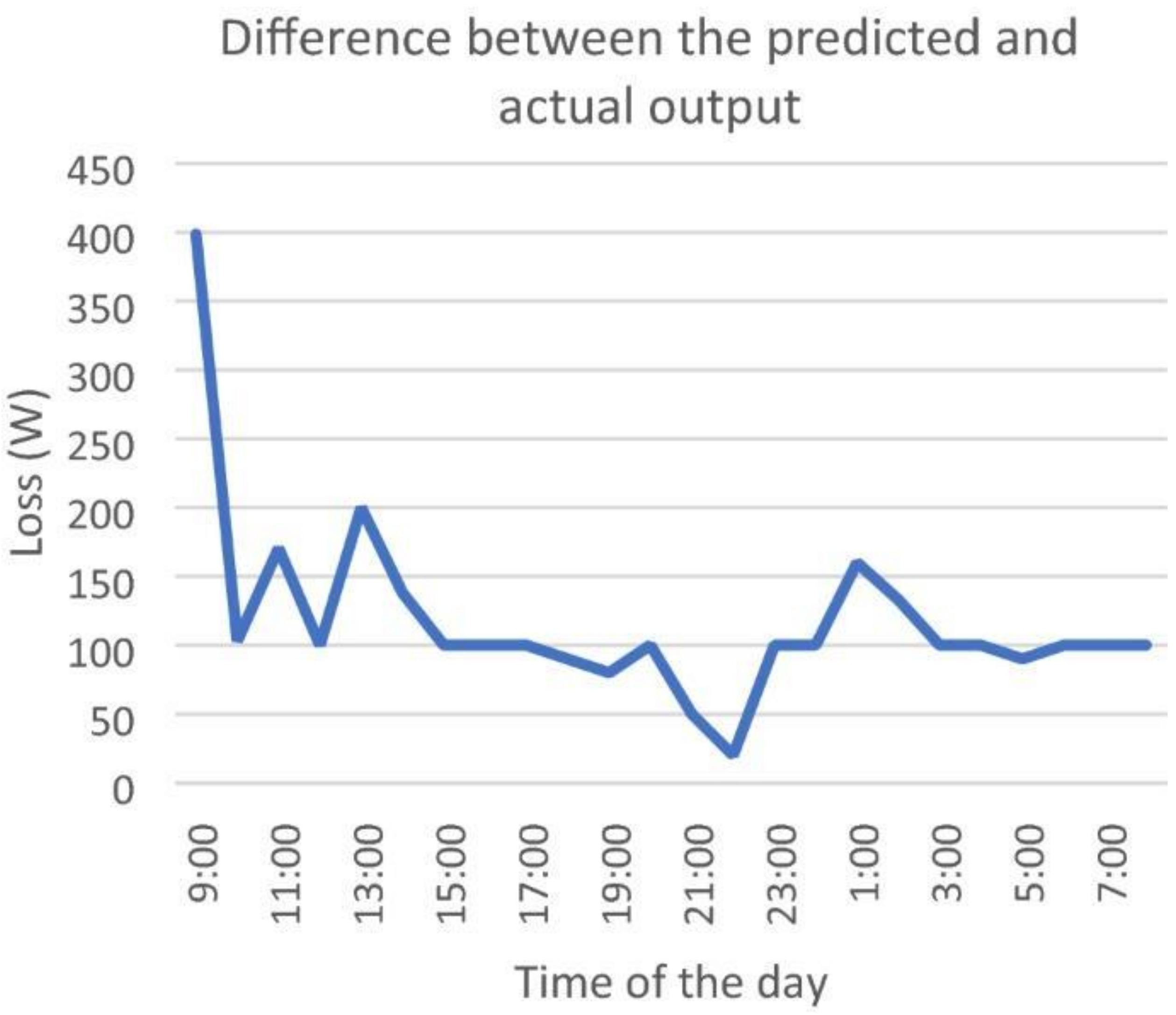
```
Mean absolute error of moving average forecast 8.457690692889582
```

Visualization:

- visualize the actual vs. predicted values for further analysis:

Python:

```
Import matplotlib.pyplot as plt
Plt.scatter(y_test, y_pred)
Plt.xlabel(“Actual Energy Consumption”)
Plt.ylabel(“Predicted Energy Consumption”)
Plt.title(“Actual vs. Predicted Energy Consumption”)
Plt.show()
```



Conclusion:

- In conclusion, measuring energy consumption is a crucial practice for promoting sustainability, reducing costs, and minimizing environmental impact. By accurately monitoring and analyzing energy usage, individuals, businesses, and governments can make informed decisions to conserve energy, reduce their carbon footprint, and ensure a more sustainable future. Effective energy management strategies can lead to significant energy savings, financial benefits, and a cleaner, more sustainable environment.