Project Name - **Global Power Plant Database Project**

Name - Aman Mulla.

Batch - DS2307

Project Summary -

The Global Power Plant Database serves as an expansive, open repository consolidating information on approximately 35,000 power plants across 167 countries globally. It offers a comprehensive resource to streamline navigation, comparison, and in-depth analysis of power generation facilities worldwide. Covering an array of energy sources, it encompasses thermal plants—such as coal, gas, oil, nuclear, biomass, waste, geothermal—and renewable sources like hydro, wind, and solar.

Each entry in this database is geotagged, providing precise location data along with essential information like plant capacity, generation statistics, ownership details, and the primary fuel used for electricity generation or export. The dataset boasts key attributes like country codes, plant names, unique identifiers, geolocation coordinates, commissioning year, ownership, data sources, URLs for reference, and more. Notably, it continuously updates as fresh data becomes available, ensuring relevance and accuracy.

The database's attributes facilitate diverse analyses and insights into the global power landscape. For instance, utilizing machine learning algorithms or statistical models could predict primary fuel types based on historical data, elucidating trends or shifts in energy sources over time. Additionally, the dataset enables forecasting or estimating power plant capacities, aiding in future energy projections and infrastructure planning.

Key attributes of the database,

1. **country**: Three-character country code (ISO 3166-1 alpha-3) where the power plant is located.
2. **country_long**: The full name of the country where the power plant is situated.
3. **name**: Title or name of the power plant in Romanized form.
4. **gppd_idnr**: Unique identifier (10-12 characters) for the power plant.
5. **capacity_mw**: Electrical generating capacity in megawatts.
6. **latitude**: Geolocation in decimal degrees (WGS84 - EPSG:4326).
7. **longitude**: Geolocation in decimal degrees (WGS84 - EPSG:4326).
8. **primary_fuel**: Energy source primarily used in electricity generation or export.

9. **other_fuel1, other_fuel2, other_fuel3**: Additional energy sources used in electricity generation.
10. **commissioning_year**: Year of plant operation, weighted by unit capacity when available.
11. **owner**: Majority shareholder of the power plant in Romanized form.
12. **source**: Entity reporting the data, often an organization or document.
13. **url**: Web document corresponding to the source field.
14. **geolocation_source**: Attribution for geolocation information.
15. **wepp_id**: Reference to a unique plant identifier in the PLATTS-WEPP database.
16. **year_of_capacity_data**: Year when capacity information was reported.
17. **generation_gwh_2013 to generation_gwh_2019**: Electricity generation in gigawatt-hours reported for respective years.
18. **generation_data_source**: Attribution for reported generation information.
19. **estimated_generation_gwh_2013 to estimated_generation_gwh_2017**: Estimated electricity generation in gigawatt-hours for respective years.
20. **estimated_generation_note_2013 to estimated_generation_note_2017**: Label of the model/method used for estimated generation.

Problem Statement

In a rapidly evolving global energy landscape, understanding and analyzing power plant data is critical for informed decision-making and sustainable energy planning. However, the dispersed nature and diverse attributes of power plant information pose challenges in comprehensive analysis. The need arises for a streamlined approach to harness the vast repository of over 35,000 power plants from 167 countries, consolidating data on capacities, fuel types, locations, and historical generation figures. Developing predictive models to accurately determine primary fuel sources and forecast power plant capacities based on historical data becomes imperative. Addressing these challenges involves aggregating, organizing, and analyzing this expansive dataset to derive insights into shifting energy trends, aiding policymakers, researchers, and industry experts in making informed decisions and facilitating strategic planning for the global energy transition.

Knowing data and variable in dataset

```
# Importing Necessary Libraries

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

pd.set_option('display.max_columns', None)
```

```
pp_data = pd.read_csv('/content/drive/MyDrive/DataSets/database_IND.csv')
```

```
pp_data.head()
```

	country	country_long	name	gppd_idnr	capacity_mw	latitude	longitude	p
0	IND	India	ACME Solar Tower	WRI1020239	2.5	28.1839	73.2407	
1	IND	India	ADITYA CEMENT WORKS	WRI1019881	98.0	24.7663	74.6090	
2	IND	India	AES Saurashtra Windfarms	WRI1026669	39.2	21.9038	69.3732	
3	IND	India	AGARTALA GT	IND0000001	135.0	23.8712	91.3602	
4	IND	India	AKALTARA TPP	IND0000002	1800.0	21.9603	82.4091	

```
pp_data.columns
```

```
Index(['country', 'country_long', 'name', 'gppd_idnr', 'capacity_mw',
       'latitude', 'longitude', 'primary_fuel', 'other_fuel1', 'other_fuel2',
       'other_fuel3', 'commissioning_year', 'owner', 'source', 'url',
       'geolocation_source', 'wepp_id', 'year_of_capacity_data',
       'generation_gwh_2013', 'generation_gwh_2014', 'generation_gwh_2015',
       'generation_gwh_2016', 'generation_gwh_2017', 'generation_gwh_2018',
       'generation_gwh_2019', 'generation_data_source',
       'estimated_generation_gwh'],
      dtype='object')
```

```
# Will Check for shape of dataset

pp_data.shape
```

```
(907, 27)
```

We have total 907 rows and 27 column in our detaset.

Dataset Information

```
pp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 907 entries, 0 to 906
Data columns (total 27 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   country                   907 non-null    object 
 1   country_long              907 non-null    object 
 2   name                      907 non-null    object 
 3   gppd_idnr                 907 non-null    object 
 4   capacity_mw               907 non-null    float64
 5   latitude                  861 non-null    float64
 6   longitude                 861 non-null    float64
 7   primary_fuel              907 non-null    object 
 8   other_fuel1               198 non-null    object 
 9   other_fuel2               1 non-null      object 
 10  other_fuel3               0 non-null      float64
 11  commissioning_year        527 non-null    float64
 12  owner                     342 non-null    object 
 13  source                    907 non-null    object 
 14  url                       907 non-null    object 
 15  geolocation_source        888 non-null    object 
 16  wepp_id                   0 non-null      float64
 17  year_of_capacity_data     519 non-null    float64
 18  generation_gwh_2013       0 non-null      float64
 19  generation_gwh_2014       398 non-null    float64
 20  generation_gwh_2015       422 non-null    float64
 21  generation_gwh_2016       434 non-null    float64
 22  generation_gwh_2017       440 non-null    float64
 23  generation_gwh_2018       448 non-null    float64
 24  generation_gwh_2019       0 non-null      float64
 25  generation_data_source    449 non-null    object 
 26  estimated_generation_gwh  0 non-null      float64
dtypes: float64(15), object(12)
memory usage: 191.4+ KB
```

From .info(), we can observe that there were varaibles with datatype of object,float and int only.

```
# Will check for description of dataset

pp_data.describe()
```

	capacity_mw	latitude	longitude	other_fuel3	commissioning_year	wepp_id
count	907.000000	861.000000	861.000000	0.0	527.000000	0.0
mean	326.223755	21.197918	77.464907	NaN	1997.091082	NaN
std	590.085456	6.239612	4.939316	NaN	17.082868	NaN
min	0.000000	8.168900	68.644700	NaN	1927.000000	NaN
25%	16.725000	16.773900	74.256200	NaN	1988.000000	NaN
max	4760.000000	34.649000	95.408000	NaN	2018.000000	NaN

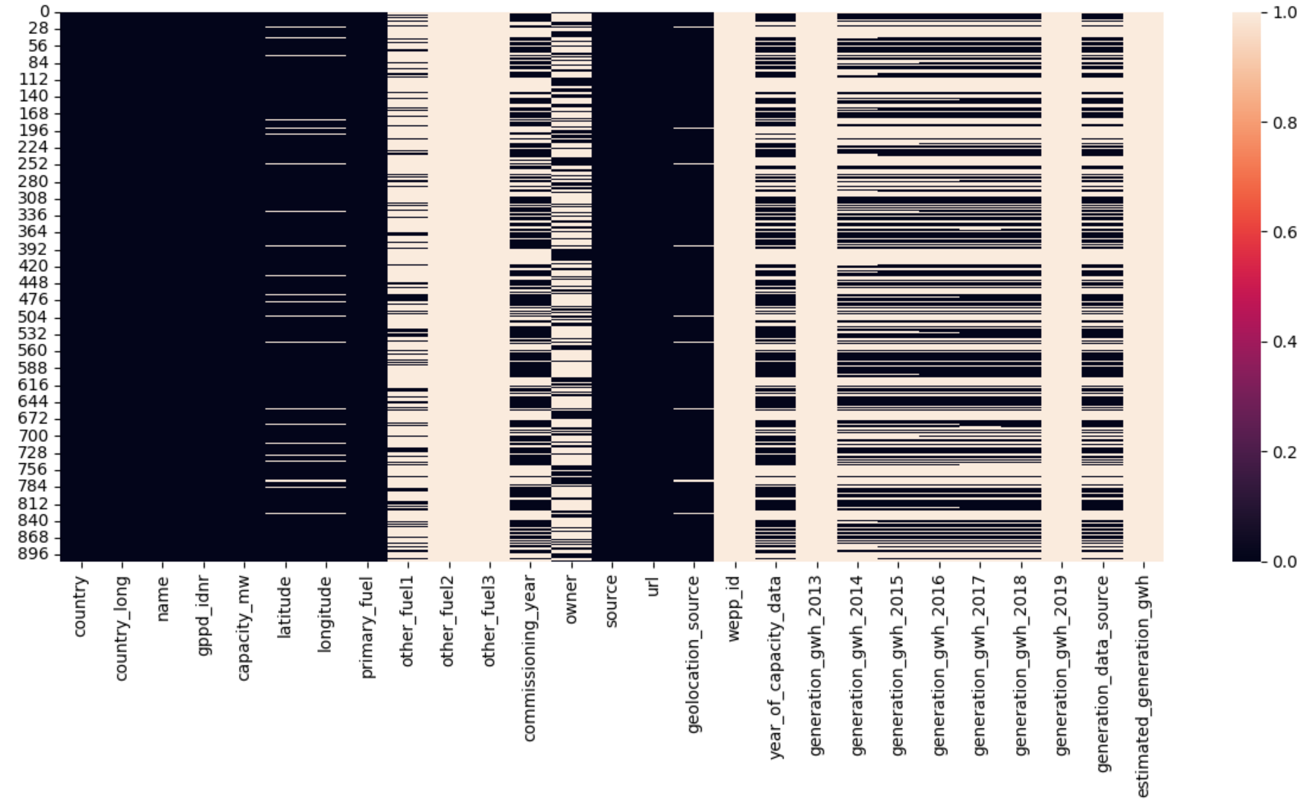
From .describe() we can get count, mean, minimum value, maximum values and quirtile value for each numerical column.

```
pp_data.isnull().sum()
```

```
country                     0
country_long                0
name                        0
gppd_idnr                   0
capacity_mw                 0
latitude                   46
longitude                  46
primary_fuel                0
other_fuel1               709
other_fuel2               906
other_fuel3               907
commissioning_year        380
owner                     565
source                      0
url                         0
geolocation_source         19
wepp_id                   907
year_of_capacity_data     388
generation_gwh_2013       907
generation_gwh_2014       509
generation_gwh_2015       485
generation_gwh_2016       473
generation_gwh_2017       467
generation_gwh_2018       459
generation_gwh_2019       907
generation_data_source    458
estimated_generation_gwh  907
dtype: int64
```

```
plt.figure(figsize=(15,6))
sns.heatmap(pp_data.isnull())
```

```
<Axes: >
```




```
pp_data['country'].value_counts()

pp_data['country_long'].value_counts()

# 'country' and 'country_long'. For this two column as cheecked with value counts.I

pp_data.drop(['country', 'country_long'], inplace= True, axis= 1)


# From .isnull().sum() We can obseve that for columns, other_fuel1,other_fuel2,othe

pp_data.drop(['other_fuel1', 'other_fuel2', 'other_fuel3', 'wepp_id', 'url',
              'geolocation_source', 'generation_data_source', 'estimated_generation
```

```
# We have some column which give some important information about the dataset, so i

pp_data['generation_gwh_2013'].fillna(value=pp_data['generation_gwh_2013'].median()
pp_data['generation_gwh_2014'].fillna(value=pp_data['generation_gwh_2014'].median()
pp_data['generation_gwh_2015'].fillna(value=pp_data['generation_gwh_2015'].median()
pp_data['generation_gwh_2016'].fillna(value=pp_data['generation_gwh_2016'].median()
pp_data['generation_gwh_2017'].fillna(value=pp_data['generation_gwh_2017'].median()
pp_data['generation_gwh_2018'].fillna(value=pp_data['generation_gwh_2018'].median()
```

```
# Similarly, for column 'latitude' and 'longitude' will filling with mean of column

pp_data['latitude'].fillna(value=pp_data['latitude'].mean(), inplace= True)
pp_data['longitude'].fillna(value=pp_data['longitude'].mean(), inplace= True)
```

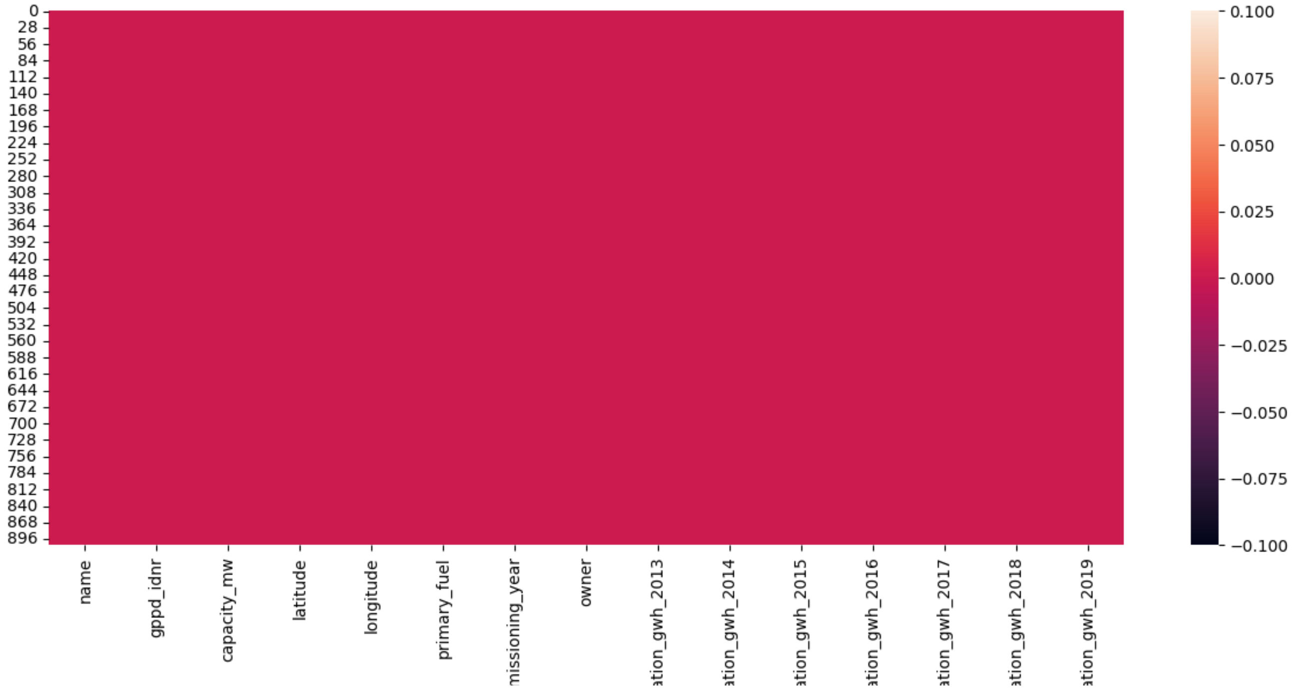
```
/usr/local/lib/python3.10/dist-packages/numpy/lib/nanfunctions.py:1215: RuntimeWarnir
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
```

```
# column 'generation_gwh_2013' and 'generation_gwh_2019'having all NaNs, as cannot

columns_to_fill = ['generation_gwh_2013', 'generation_gwh_2019']
pp_data[columns_to_fill] = pp_data[columns_to_fill].fillna(0)
```

```
# For Column 'commissioning_year' and 'owner' are seems important so filling with 0

columns_to_fill = ['commissioning_year']
pp_data[columns_to_fill] = pp_data[columns_to_fill].fillna(0)

columns_to_fill = ['owner']
pp_data[columns_to_fill] = pp_data[columns_to_fill].fillna('N/A')
```

```
plt.figure(figsize=(15,6))
sns.heatmap(pp_data.isnull())
```

```
<Axes: >
```



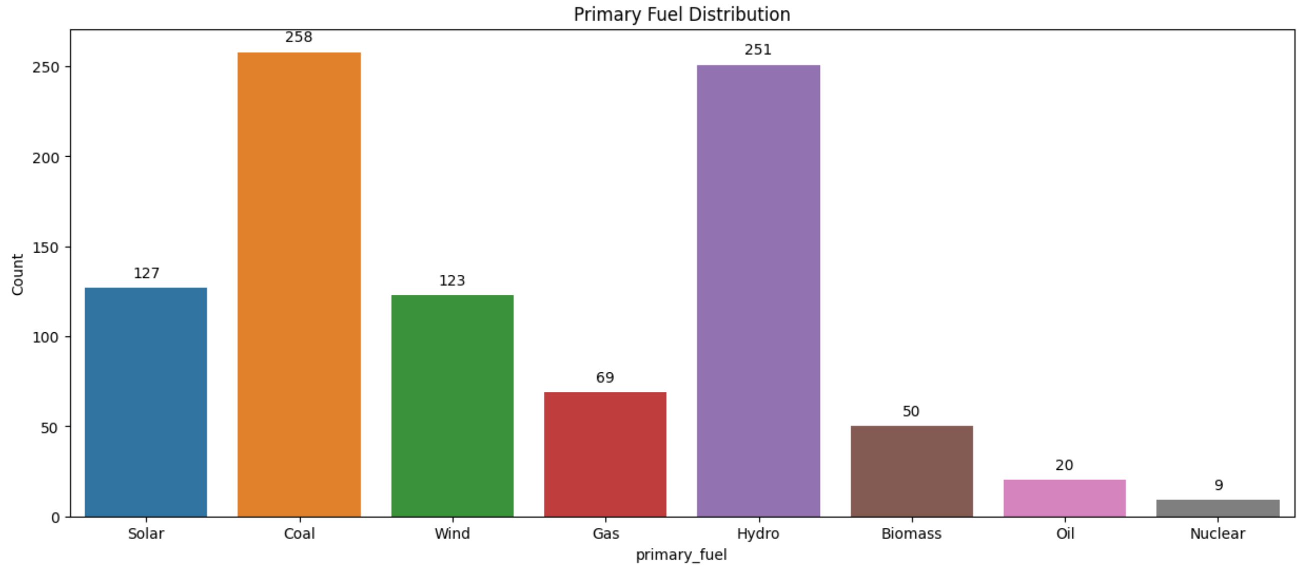

```
pp_data.columns
```

```
Index(['name', 'gppd_idnr', 'capacity_mw', 'latitude', 'longitude',
       'primary_fuel', 'commissioning_year', 'owner', 'generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019'],
      dtype='object')
```

Chart - 1

Primary Fuel Distribution

```
plt.figure(figsize=(15,6))
ax = sns.countplot(pp_data, x = 'primary_fuel')

# Adding data labels
for p in ax.patches:
    ax.annotate(format(p.get_height(), '.0f'),
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha = 'center', va = 'center',
                xytext = (0, 10),
                textcoords = 'offset points')

plt.xlabel('primary_fuel')
plt.ylabel('Count')
plt.title('Primary Fuel Distribution')
plt.show()
```



Insights from the above chart:

- The count plot displays the frequency or count of power plants for each primary fuel type.
- It helps identify the prevalence of different fuel types used across the global power plant dataset. For Fuel 'Coal' we have more number of power plant along with 'Hydro'. Have very less power plant for fuel 'Nuclear'.

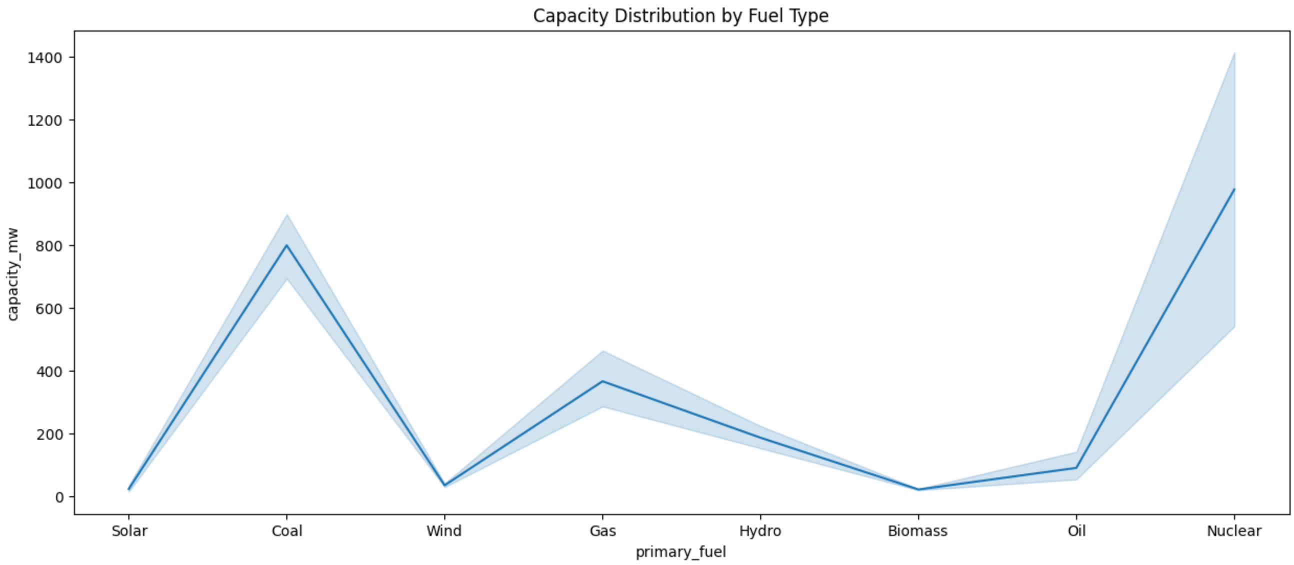
Possible Reasons:

- India has a long history of coal-based power generation. Coal has been a traditional and abundant energy source in the country, leading to a higher count of coal-based power plants.
- India possesses substantial coal reserves, making it a readily available and cost-effective fuel for electricity generation. This abundance contributes to the prevalence of coal-based plants.
- India also boasts significant hydroelectric potential due to its diverse river systems and topography. This availability of natural resources has led to the development of numerous hydroelectric power plants.
- Nuclear power plants require substantial investments, advanced technology, and specialized infrastructure. These factors might have contributed to a slower growth rate and lower count of nuclear plants compared to other sources in India.

Chart - 2

Capacity Distribution by Fuel Type

```
cross_tab = pd.crosstab(index=pp_data['primary_fuel'],columns=pp_data['capacity_mw'

cross_tab

plt.figure(figsize=(15,6))
sns.lineplot(data=pp_data,x='primary_fuel',y='capacity_mw',markers='=')
plt.xlabel('primary_fuel')
plt.ylabel('capacity_mw')
plt.title('Capacity Distribution by Fuel Type')
plt.show()
```



Insights from the above chart:

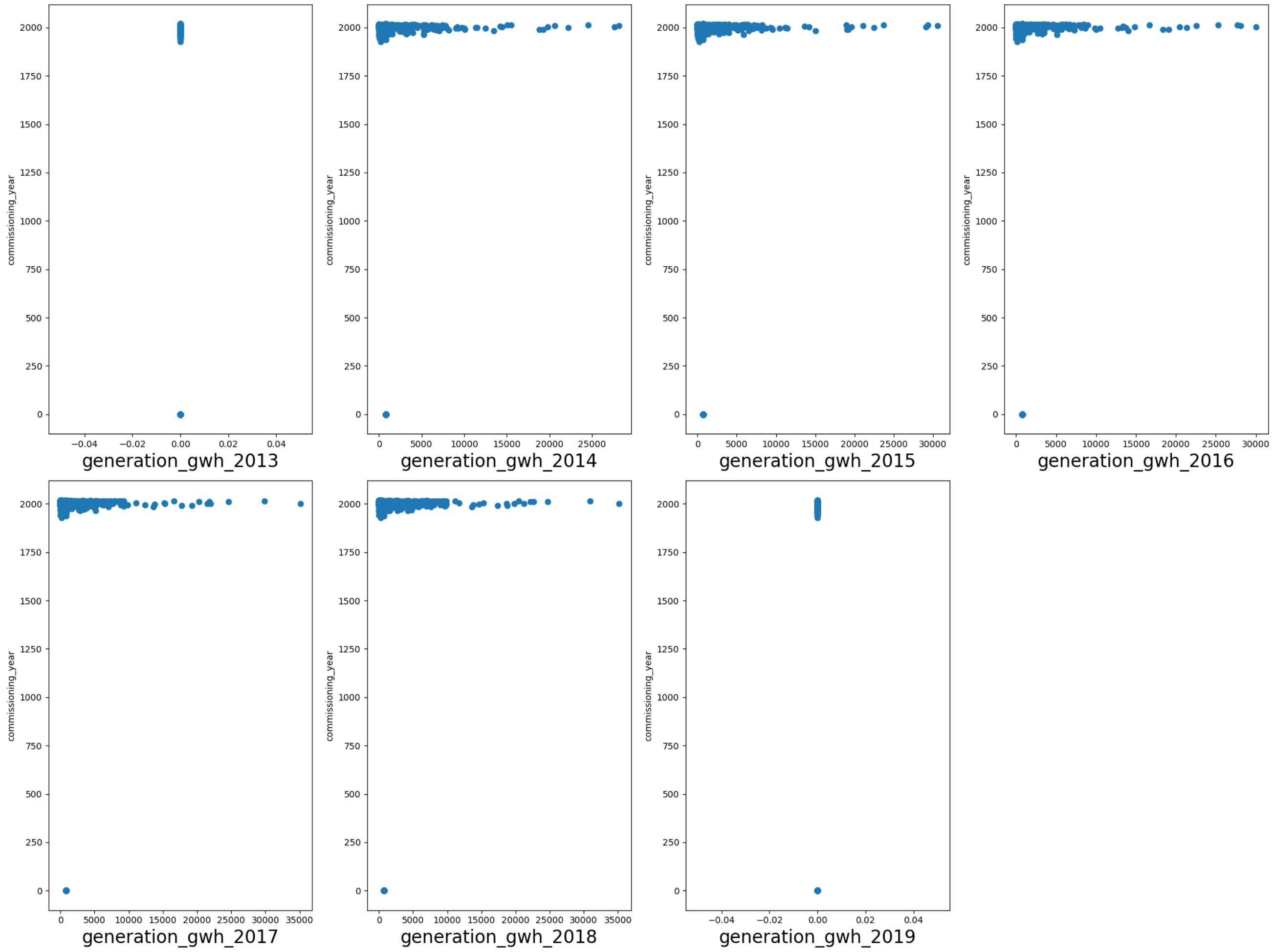
- Using a line plot to represent primary_fuel categories against capacity_mw might not convey the intended insights effectively, as line plots typically connect continuous data points along an axis, not categorical data.

- From linechart, it suggest that with nuclear type fuel generating more capacity in MW. While having biomass and Wind haveing less capacity.

Chart - 3

"Total Electricity Generation Over Years (2013-2019)

```
x = pp_data[['generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019']]

y = pp_data['commissioning_year']

plt.figure(figsize=(20,15))
plotnumber=1

for column in x:
  if plotnumber<=8:
    ax= plt.subplot(2,4,plotnumber)
    plt.scatter(x[column],y)
    plt.xlabel(column,fontsize=20)
    plt.ylabel('commissioning_year',fontsize=10)
  plotnumber+=1
plt.tight_layout()
```



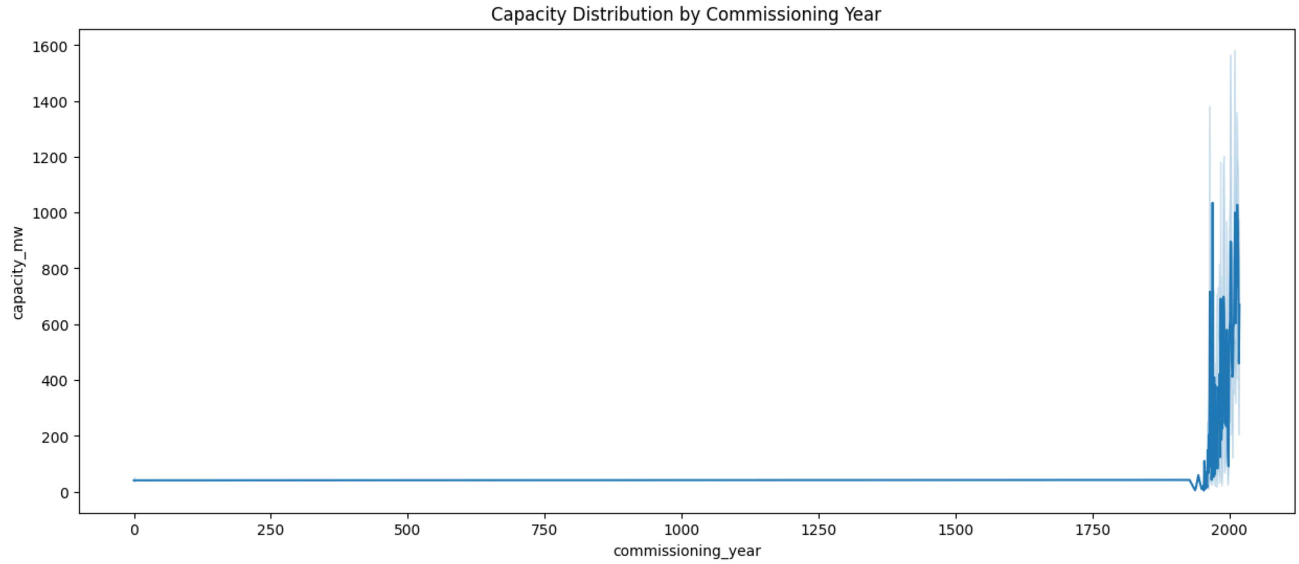
Insights from the above chart:

- The scatter plots showcase how electricity generation varies concerning the year the power plants were commissioned. It helps identify if newer plants tend to generate more electricity compared to older ones.
- Comparing these plots side by side can reveal patterns across different years of generation data, offering a comparative view of power plants commissioned in various periods.

Chart - 4

Capacity Distribution by Commissioning Year

```
plt.figure(figsize=(15,6))
sns.lineplot(data=pp_data, x='commissioning_year',y='capacity_mw')
plt.xlabel('commissioning_year')
plt.ylabel('capacity_mw')
plt.title('Capacity Distribution by Commissioning Year')
plt.show()
```



Insights from the above chart:

- The plot may reveal the trend of capacity additions over different years. Steeper upward slopes indicate years with significant capacity additions, potentially reflecting periods of high infrastructure development in the energy sector.

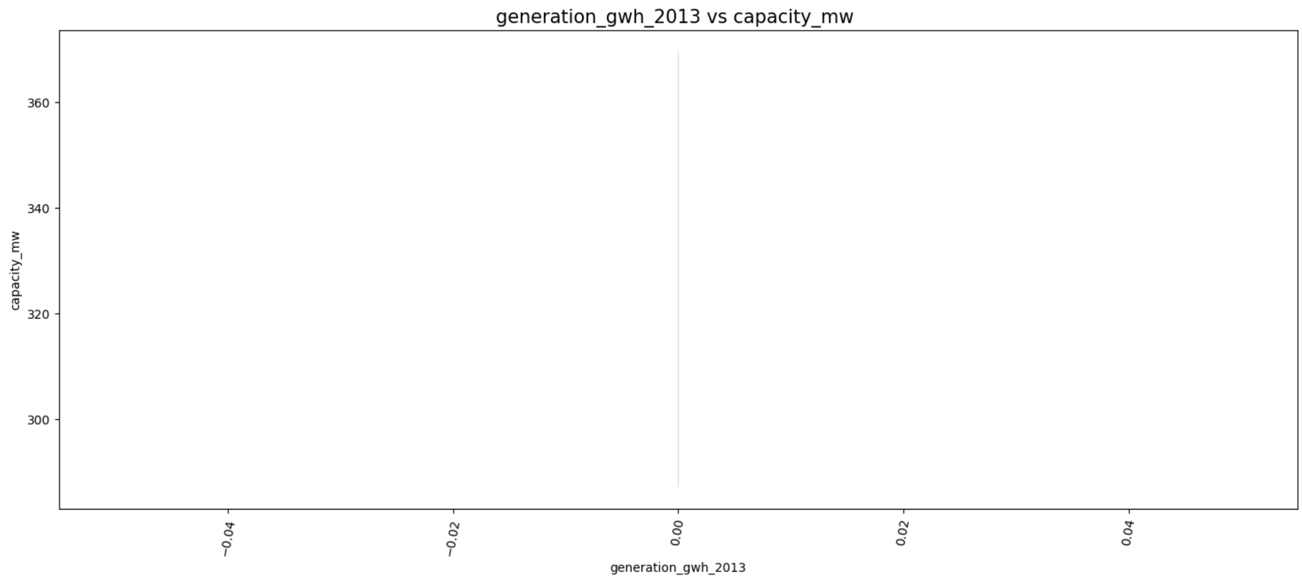
```
pp_data.columns
```

```
Index(['name', 'gppd_idnr', 'capacity_mw', 'latitude', 'longitude',
       'primary_fuel', 'commissioning_year', 'owner', 'generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019'],
      dtype='object')
```

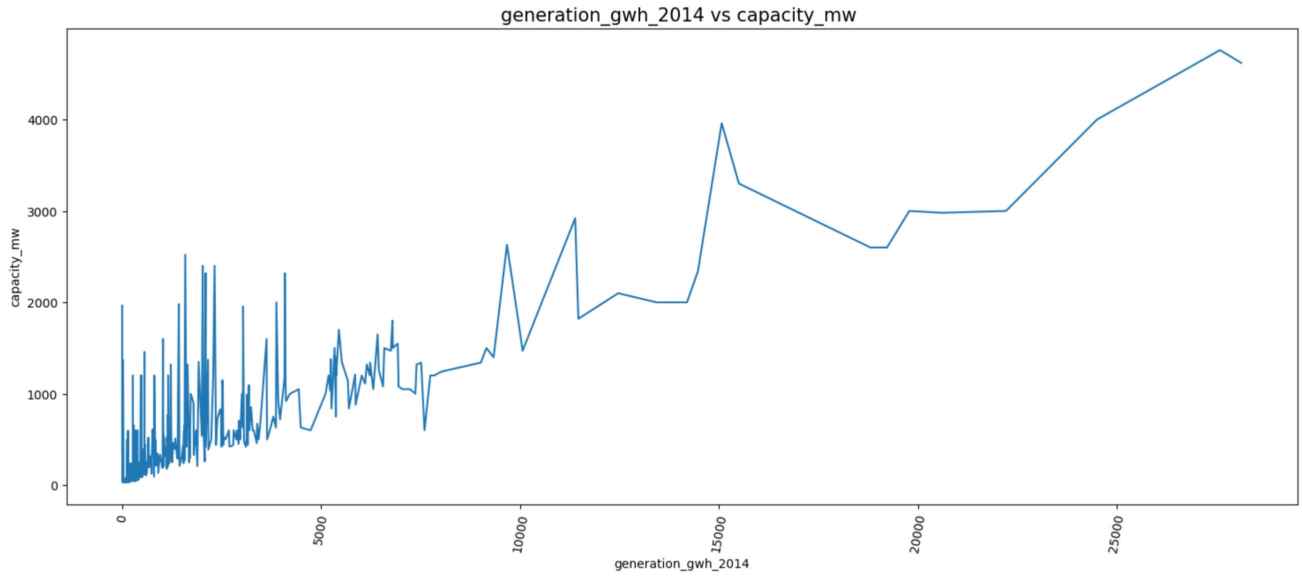
Chart - 5

generation_gwh_2013 vs capacity_mw

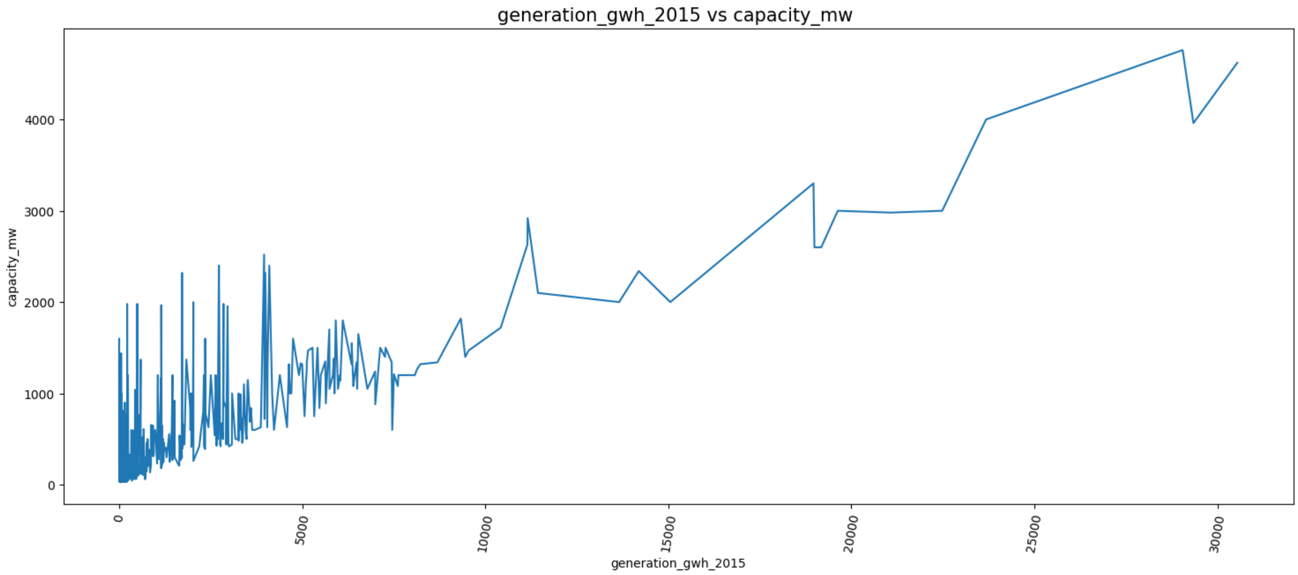
```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2013',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2013 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```



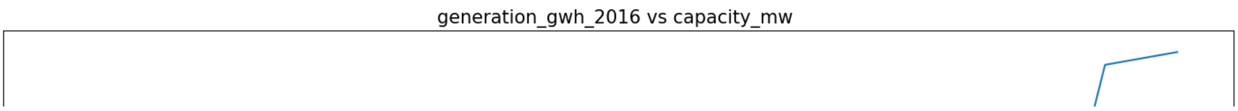

```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2014',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2014 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```



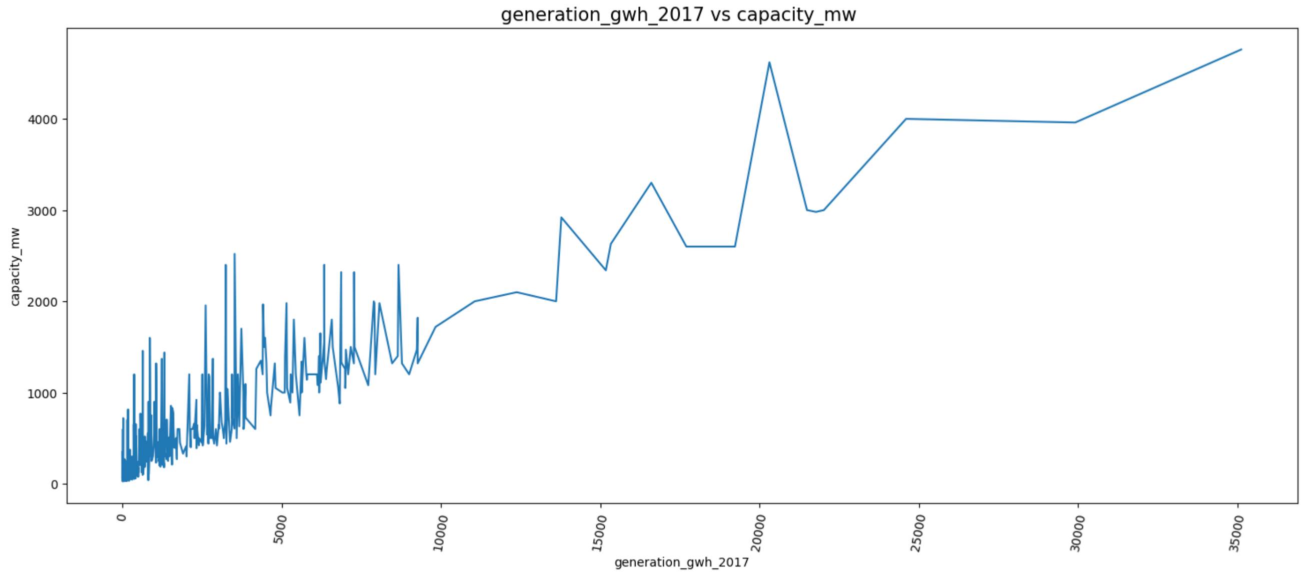

```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2015',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2015 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```



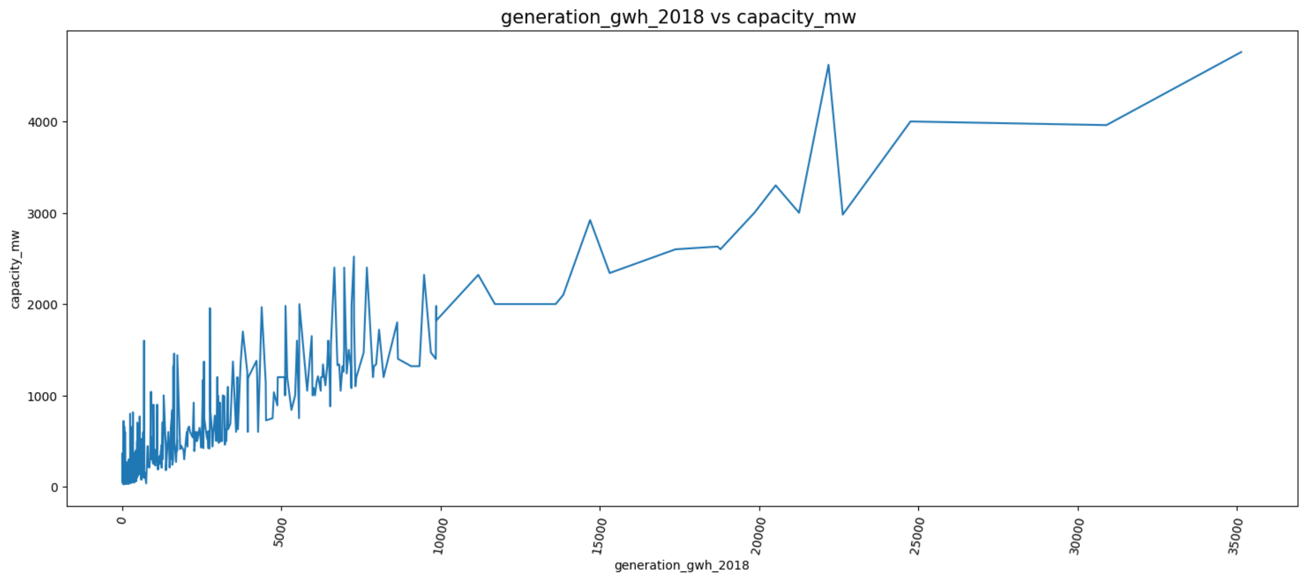

```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2016',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2016 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```



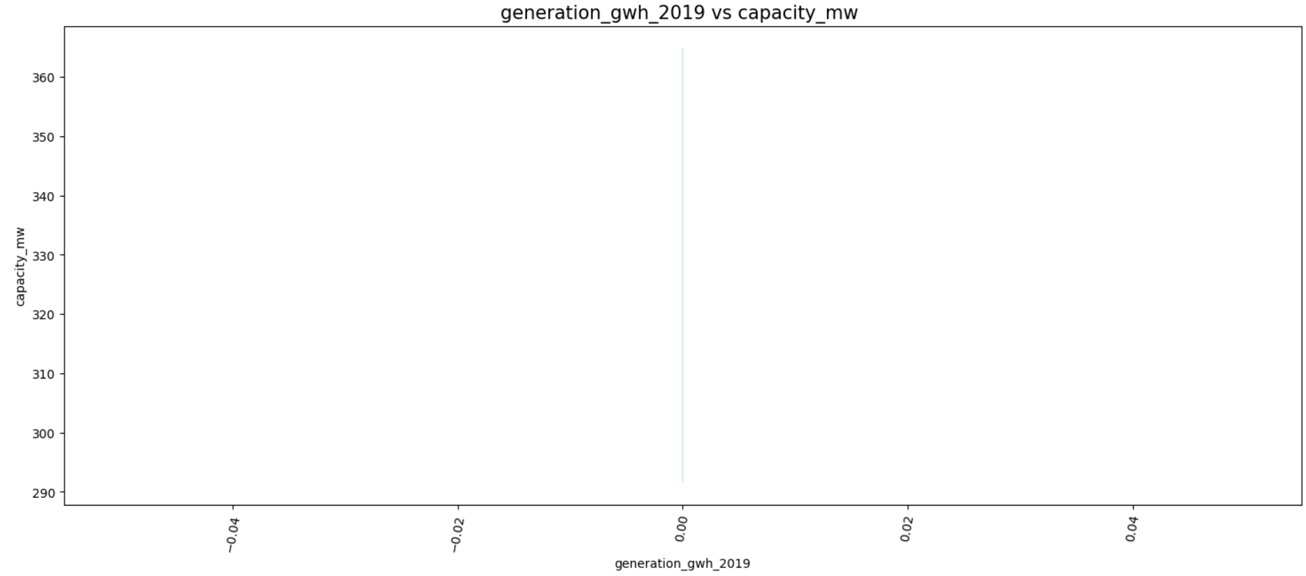

```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2017',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2017 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```




```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2018',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2018 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```




```
plt.figure(figsize=(18,7))
sns.lineplot(data=pp_data, x='generation_gwh_2019',y='capacity_mw')
locs, labels = plt.xticks()
plt.title('generation_gwh_2019 vs capacity_mw ', fontsize=15)
plt.setp(labels, rotation=80)
plt.show()
```



Insights from above charts:

- The plots depict the relationship between the capacity of power plants and their respective electricity generation for each year. A positive correlation would typically show that higher capacity plants tend to generate more electricity.
- Observing the trend lines across the years can unveil how the relationship between capacity and generation has evolved over time. Consistent upward trends might indicate a proportional increase in generation concerning capacity across the years.

From .info(), we have some columns with object datatype. Doing label encoding for the column

```
pp_data.head()
```

name	gppd_idnr	capacity_mw	latitude	longitude	primary_fuel	commission
ACME						

```
pp_data['primary_fuel'].unique()
```

```
array(['Solar', 'Coal', 'Wind', 'Gas', 'Hydro', 'Biomass', 'Oil',
       'Nuclear'], dtype=object)
```

WORKS

By observing the primary_fuel, will categorize fuel columns in 'Conventional Fuels' and 'Renewable Fuels'.

Conventional_Fuels=['Coal','Gas','Oil','Nuclear']

Renewable_Fuels=['Solar','Wind','Hydro','Biomass']

```
# Define the conditions for categorization

Conventional_Fuels=['Coal','Gas','Oil','Nuclear']

Renewable_Fuels=['Solar','Wind','Hydro','Biomass']

pp_data['categorized_fuel']= np.where(pp_data['primary_fuel'].isin(Conventional_Fue
                                 np.where(pp_data['primary_fuel'].isin(Renewable_Fue

# For categorized_fuel we have 2 unique entries, will encode as Conventional_Fuels

categorized_fuel_encoding ={"categorized_fuel":{"Conventional_Fuels":1,"Renewable_F

categorized_fuel_encoding

pp_data = pp_data.replace(categorized_fuel_encoding)
```

Chart - 5

Pairplot

```
sns.pairplot(pp_data)
```

```
<seaborn.axisgrid.PairGrid at 0x783857d08730>
```

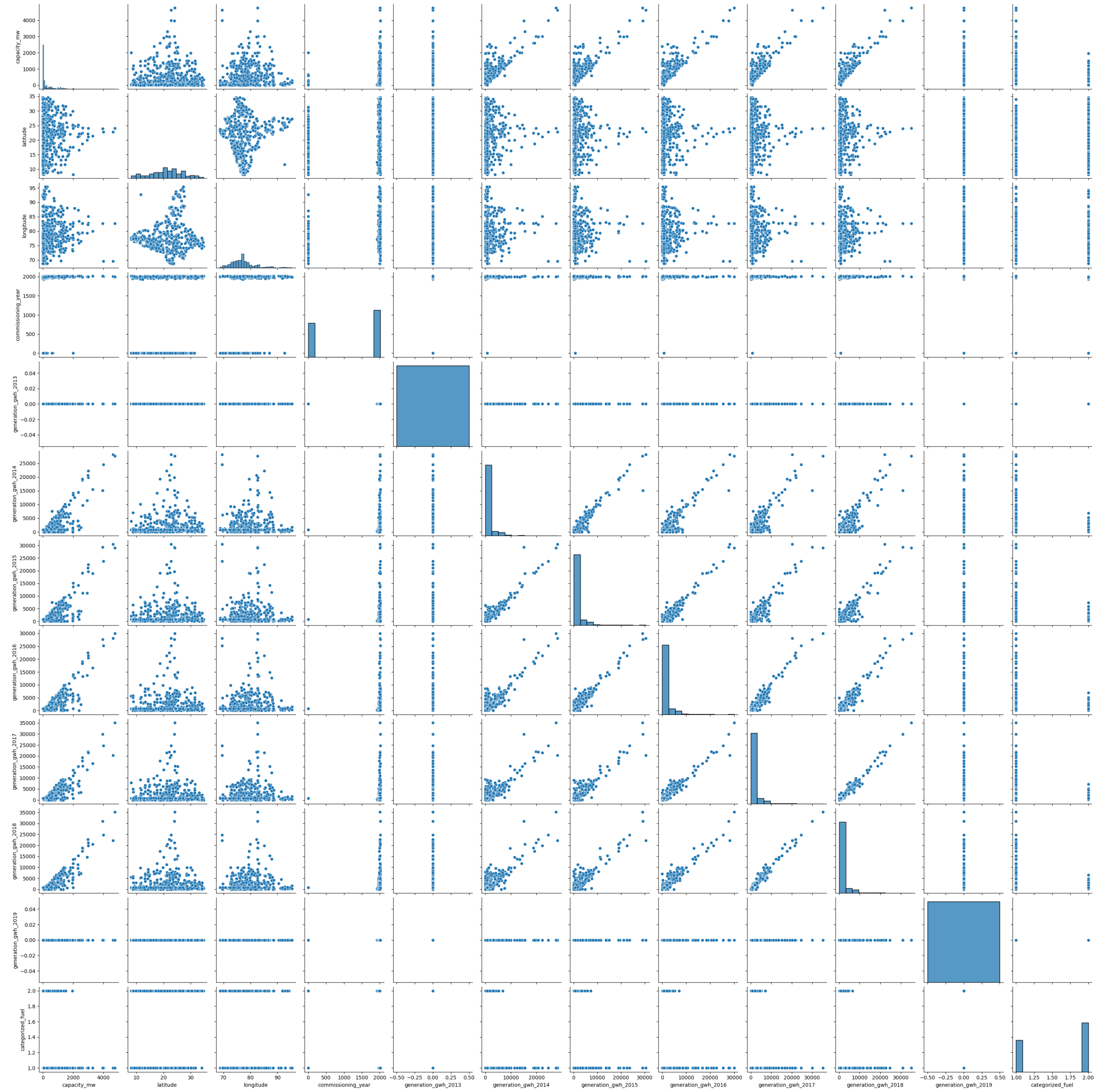
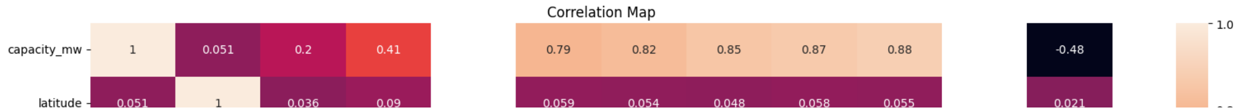


Chart - 12

Heatmap

```
correlation_data = pp_data

correlation_matrix = correlation_data.corr()

plt.figure(figsize=(20,10))

sns.heatmap(correlation_matrix,annot=True)
plt.title('Correlation Map')
plt.show()
```

```
<ipython-input-103-228cdddc9749>:3: FutureWarning: The default value of numeric_only
  correlation_matrix = correlation_data.corr()
```




```
pp_data.head()
```

	name	gppd_idnr	capacity_mw	latitude	longitude	primary_fuel	commission
0	ACME Solar Tower	WRI1020239	2.5	28.1839	73.2407	Solar	
1	ADITYA CEMENT WORKS	WRI1019881	98.0	24.7663	74.6090	Coal	
2	AES Saurashtra Windfarms	WRI1026669	39.2	21.9038	69.3732	Wind	
3	AGARTALA GT	IND0000001	135.0	23.8712	91.3602	Gas	
4	AKALTARA TPP	IND0000002	1800.0	21.9603	82.4091	Coal	

Will Check for VIF

```
x = pp_data.drop(columns=['name','gppd_idnr','owner','primary_fuel'])
y = pp_data['categorized_fuel']

from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor

scalar = StandardScaler()
x_scaled=scalar.fit_transform(x)

# VIF

vif = pd.DataFrame()

vif['vif']=[variance_inflation_factor(x_scaled,i) for i in range(x_scaled.shape[1])

vif['features'] = x.columns

vif
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/regression/linear_model.py:1783:
  return 1 - self.ssr/self.uncentered_tss
```

	vif	features
0	6.449581	capacity_mw
1	1.018072	latitude
2	1.261930	longitude
3	1.555926	commissioning_year
4	NaN	generation_gwh_2013
5	14.211052	generation_gwh_2014
6	34.419352	generation_gwh_2015
7	44.197927	generation_gwh_2016
8	52.520319	generation_gwh_2017
9	44.372170	generation_gwh_2018

Firsty will predict for primary_fuel attribute

ML Model - 1

Using all Variables for ML Model-1

```
# Importing Necessary Libraries

from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import accuracy_score,confusion_matrix,roc_curve,roc_auc_score
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
```

```
pp_data.head()
```

	name	gppd_idnr	capacity_mw	latitude	longitude	primary_fuel	commission
0	ACME Solar Tower	WRI1020239	2.5	28.1839	73.2407	Solar	
1	ADITYA CEMENT WORKS	WRI1019881	98.0	24.7663	74.6090	Coal	

```
pp_data.columns
```

```
Index(['name', 'gppd_idnr', 'capacity_mw', 'latitude', 'longitude',
       'primary_fuel', 'commissioning_year', 'owner', 'generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019',
       'categorized_fuel'],
      dtype='object')
```

```
# For ML Model 1 Using all variables from dataset

x = pp_data[['capacity_mw', 'latitude', 'longitude', 'commissioning_year','generati
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019']]

y = pp_data['categorized_fuel']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting Logistic Regression model to dataset

logistic_reg = LogisticRegression()

logistic_reg.fit(x_train,y_train)

# Make predictions on the test set
y_pred = logistic_reg.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For ML Model 1:", accuracy)
print("Confusion Matrix For ML Model 1:\n", confusion)
print("Classification Report for ML Model 1:\n", classification_report_str)

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'penalty': ['l1', 'l2'],  # Regularization penalty
    'solver': ['liblinear', 'lbfgs'],  # Solver for optimization
}

# Initialize GridSearchCV
```

```
grid_search = GridSearchCV(estimator=logistic_reg, param_grid=param_grid, scoring='

# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For ML Model 1(with Hyperparameter Tuning):", accuracy)
print("Confusion Matrix For ML Model 1(with Hyperparameter Tuning):\n", confusion)
print("Classification Report for ML Model 1(with Hyperparameter Tuning):\n", classi
```

```
Accuracy For ML Model 1: 0.7400881057268722
Confusion Matrix For ML Model 1:
 [[ 54  42]
 [ 17 114]]
Classification Report for ML Model 1:
               precision    recall  f1-score   support

           1       0.76      0.56      0.65        96
           2       0.73      0.87      0.79       131

    accuracy                           0.74       227
   macro avg       0.75      0.72      0.72       227
weighted avg       0.74      0.74      0.73       227

Best Hyperparameters: {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
Best Model: LogisticRegression(C=10, penalty='l1', solver='liblinear')
Accuracy For ML Model 1(with Hyperparameter Tuning): 0.7797356828193832
Confusion Matrix For ML Model 1(with Hyperparameter Tuning):
 [[ 62  34]
 [ 16 115]]
Classification Report for ML Model 1(with Hyperparameter Tuning):
               precision    recall  f1-score   support

           1       0.79      0.65      0.71        96
           2       0.77      0.88      0.82       131

    accuracy                           0.78       227
   macro avg       0.78      0.76      0.77       227
weighted avg       0.78      0.78      0.78       227
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.py:378: F
30 fits failed out of a total of 120.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score=

Below are more details about the failures:
--------------------------------------------------------------------------------
30 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.p
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py", l
    solver = _check_solver(self.solver, self.penalty, self.dual)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py", l
    raise ValueError(
ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:952: UserW
 0.63088235 0.62205882 0.71911765        nan 0.725      0.72941176
 0.77941176        nan 0.76323529 0.76176471 0.78235294        nan
 0.77205882 0.77352941 0.77647059        nan 0.77647059 0.77647059]
  warnings.warn(
```

Insigts from the ML Model 1 :

- **Accuracy**:The initial accuracy of the model was **74.01%**.
- **Confusion Matrix:**

  Model predicted 54 instances as Class 1 correctly, but misclassified 42 instances.

  For Class 2, it predicted 114 instances correctly and misclassified 17 instances.
- **Classification Report:**

  Precision for Class 1 (0.76) indicates that among the instances predicted as Class 1, 76% were actually Class 1.

  Recall for Class 1 (0.56) implies that of all the actual Class 1 instances, only 56% were predicted correctly.

  F1-score (a blend of precision and recall) for Class 1 was 0.65, and for Class 2, it was 0.79.

After Hyperparameter

- **Accuracy**:The accuracy increased to 77.97% after hyperparameter tuning.
- **Confusion Matrix:**

  There was an enhancement in predicting Class 1 instances (62 correct predictions) and a reduction in misclassifications (34 misclassified).

  For Class 2, there were 115 correct predictions and 16 misclassified instances.
- **Classification Report:**

Precision for Class 1 increased to 0.79, indicating a higher proportion of correctly predicted Class 1 instances among all predicted Class 1 instances.

Recall for Class 1 increased to 0.65, which means the model captured a higher percentage of actual Class 1 instances compared to before tuning.

F1-score for both Class 1 and Class 2 improved to 0.71 and 0.82, respectively.

** After tuning, there was a notable improvement in the precision of both classes, implying fewer false positives.The model became better at capturing actual instances of Class 1 after tuning, but there's room for further improvement in recall for this class.Overall, the harmonic mean of precision and recall (F1-score) increased for both classes, indicating an overall improvement in model performance.**

Feature Engineering

From above VIF result and heatmap,we have some strong positive correlations.

'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016', 'generation_gwh_2017', 'generation_gwh_2018': These variables seem to have considerably high VIF values, indicating potential multicollinearity issues among these variables.

'capacity_mw', 'latitude', 'longitude', 'commissioning_year', 'categorized_fuel': These variables have VIF values below 5, suggesting lower multicollinearity compared to the generation variables. They are relatively less correlated with each other and might be considered as more suitable predictors for the machine learning model.

ML Model - 2

cosidering columns with feature Engineering

```
pp_data.columns
```

```
Index(['name', 'gppd_idnr', 'capacity_mw', 'latitude', 'longitude',
       'primary_fuel', 'commissioning_year', 'owner', 'generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019',
       'categorized_fuel'],
      dtype='object')
```

```
# For ML Model 1 Using all variables from dataset

x = pp_data[['capacity_mw', 'latitude', 'longitude', 'commissioning_year']]

y = pp_data['categorized_fuel']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting Logistic Regression model to dataset

logistic_reg = LogisticRegression()

logistic_reg.fit(x_train,y_train)

# Make predictions on the test set
y_pred = logistic_reg.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For ML Model 1:", accuracy)
print("Confusion Matrix For ML Model 1:\n", confusion)
print("Classification Report for ML Model 1:\n", classification_report_str)

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'penalty': ['l1', 'l2'],  # Regularization penalty
    'solver': ['liblinear', 'lbfgs'],  # Solver for optimization
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=logistic_reg, param_grid=param_grid, scoring='
```

```
# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For ML Model 1(with Hyperparameter Tuning):", accuracy)
print("Confusion Matrix For ML Model 1(with Hyperparameter Tuning):\n", confusion)
print("Classification Report for ML Model 1(with Hyperparameter Tuning):\n", classi
```

```
Accuracy For ML Model 1: 0.748898678414097
Confusion Matrix For ML Model 1:
 [[ 59  37]
 [ 20 111]]
Classification Report for ML Model 1:
               precision    recall  f1-score   support

           1       0.75      0.61      0.67        96
           2       0.75      0.85      0.80       131

    accuracy                           0.75       227
   macro avg       0.75      0.73      0.73       227
weighted avg       0.75      0.75      0.74       227

Best Hyperparameters: {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
Best Model: LogisticRegression(C=10, penalty='l1', solver='liblinear')
Accuracy For ML Model 1(with Hyperparameter Tuning): 0.7709251101321586
Confusion Matrix For ML Model 1(with Hyperparameter Tuning):
 [[ 62  34]
 [ 18 113]]
Classification Report for ML Model 1(with Hyperparameter Tuning):
               precision    recall  f1-score   support

           1       0.78      0.65      0.70        96
           2       0.77      0.86      0.81       131

    accuracy                           0.77       227
   macro avg       0.77      0.75      0.76       227
weighted avg       0.77      0.77      0.77       227

/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.py:378: F
30 fits failed out of a total of 120.
```

```
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score=

Below are more details about the failures:
--------------------------------------------------------------------------------
30 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.p
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py", l
    solver = _check_solver(self.solver, self.penalty, self.dual)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py", l
    raise ValueError(
ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:952: UserW
 0.61764706 0.61764706 0.71911765        nan 0.71911765 0.725
 0.77941176        nan 0.76029412 0.76617647 0.78088235        nan
 0.77205882 0.77058824 0.77941176        nan 0.78088235 0.78088235]
  warnings.warn(
```

Insigts from the ML Model 2 :

- **Accuracy**:The initial accuracy of the model was 0.749, meaning it correctly predicted the class around 75% of the time
- **Confusion Matrix:**

  111 instances of class 2 were correctly predicted, and 59 instances of class 1 were correctly predicted.

  37 instances of class 1 were predicted as class 2, and 20 instances of class 2 were predicted as class 1.
- **Classification Report:**

  Precision for class 1 was 0.75, and for class 2, it was 0.75. Precision measures the accuracy of the positive predictions.

  Recall for class 1 was 0.61, and for class 2, it was 0.85. Recall measures the actual positive instances that were correctly predicted.

  F1-score for class 1 was 0.67, and for class 2, it was 0.80. F1-score is the harmonic mean of precision and recall.

Best Hyperparameters: The hyperparameters chosen after tuning were {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'} for a Logistic Regression model.

After Hyperparameter

- **Accuracy**: the accuracy increased to 0.771, indicating a slight enhancement in overall predictive performance.

- **Confusion Matrix:**

  TP for class 1 increased to 62, FP reduced to 34.

  TP for class 2 stayed the same, but FP reduced to 18.

- **Classification Report:**

  Precision, recall, and F1-scores improved slightly for both classes.

  Class 1 metrics (precision, recall, F1-score) showed improvement from 0.75, 0.61, 0.67 to 0.78, 0.65, 0.70 respectively.

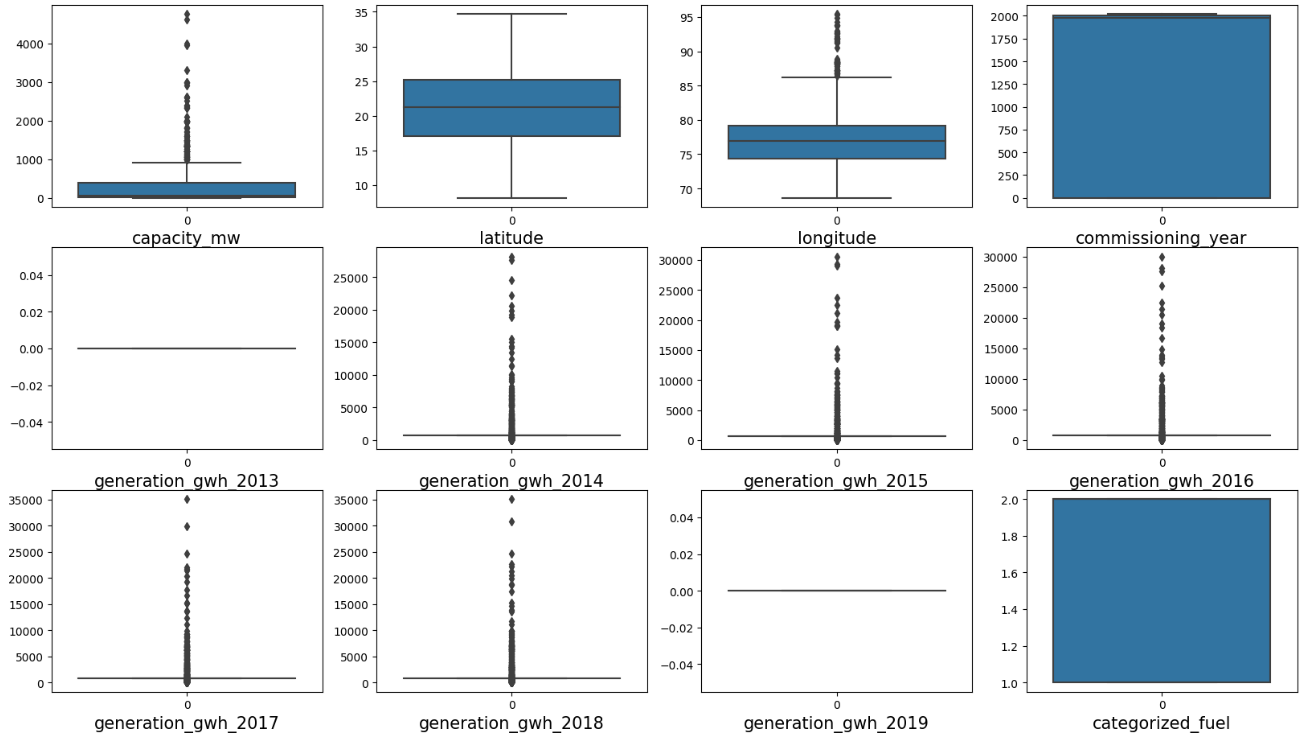
  Class 2 metrics also saw an enhancement from 0.75, 0.85, 0.80 to 0.77, 0.86, 0.81 respectively.

The model had decent overall accuracy but showed imbalanced performance between precision and recall for different classes.Class 2 had higher recall but slightly lower precision compared to class 1.Hyperparameter tuning led to a marginal improvement across metrics, refining the balance between precision and recall for both classes.The model shows a better capability to distinguish between classes, especially in class 1 where both precision and recall have increased.

Will check for outliers in each column using boxplot

```
x = pp_data.drop(columns=['name', 'gppd_idnr','primary_fuel','owner'])
```

```
plt.figure(figsize=(20,15))
graph = 1

for column in x:
  if graph<=16:
    plt.subplot(4,4,graph)
    ax=sns.boxplot(data= x[column])
    plt.xlabel(column,fontsize=15)
  graph+=1
plt.show()
```




```
pp_data.columns
```

```
Index(['name', 'gppd_idnr', 'capacity_mw', 'latitude', 'longitude',
       'primary_fuel', 'commissioning_year', 'owner', 'generation_gwh_2013',
       'generation_gwh_2014', 'generation_gwh_2015', 'generation_gwh_2016',
       'generation_gwh_2017', 'generation_gwh_2018', 'generation_gwh_2019',
       'categorized_fuel'],
      dtype='object')
```

```
pp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 907 entries, 0 to 906
Data columns (total 16 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   name                907 non-null    object 
 1   gppd_idnr           907 non-null    object 
 2   capacity_mw         907 non-null    float64
 3   latitude            907 non-null    float64
 4   longitude           907 non-null    float64
 5   primary_fuel        907 non-null    object 
 6   commissioning_year  907 non-null    float64
 7   owner               907 non-null    object 
```

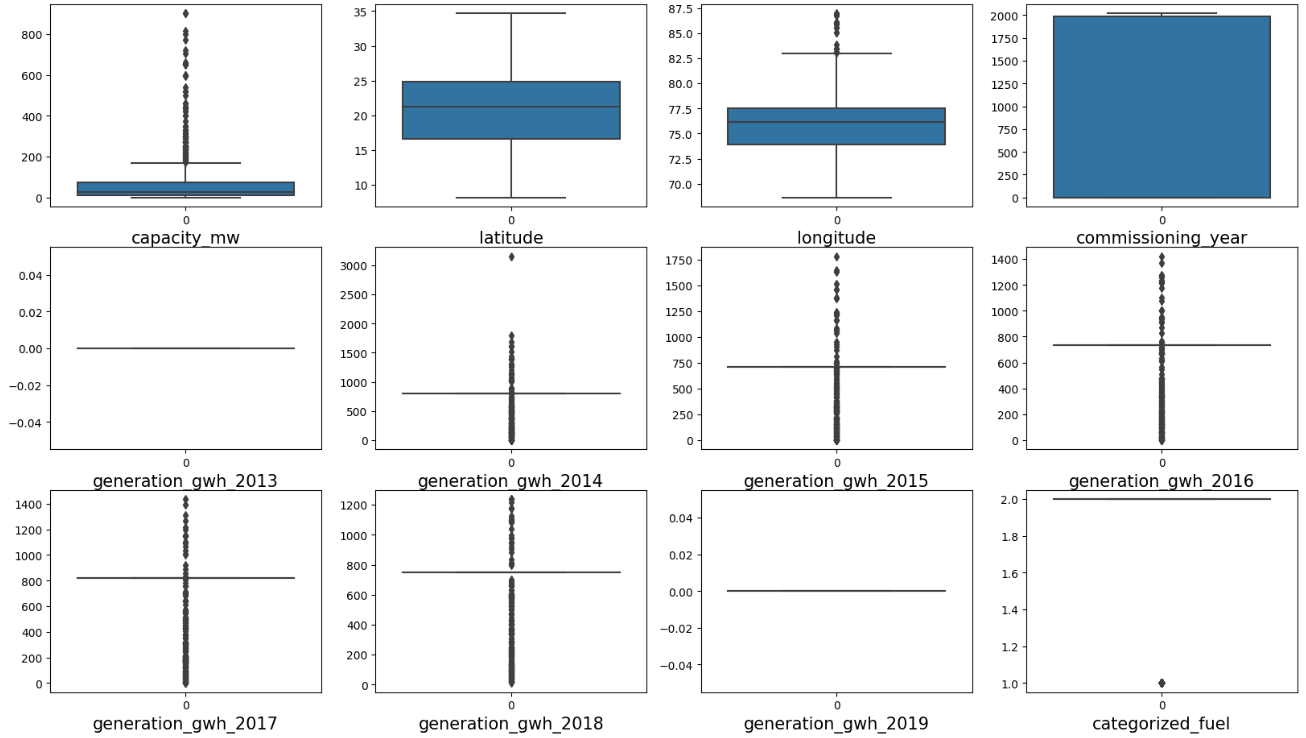
```
 8   generation_gwh_2013  907 non-null    float64
 9   generation_gwh_2014  907 non-null    float64
 10  generation_gwh_2015  907 non-null    float64
 11  generation_gwh_2016  907 non-null    float64
 12  generation_gwh_2017  907 non-null    float64
 13  generation_gwh_2018  907 non-null    float64
 14  generation_gwh_2019  907 non-null    float64
 15  categorized_fuel     907 non-null    int64
dtypes: float64(11), int64(1), object(4)
memory usage: 113.5+ KB
```

```
from scipy import stats

# Define a threshold for the Z-score
z_score_threshold = 2  # You can adjust this threshold based on your data and requi

# Select numerical columns where you want to detect and treat outliers
numerical_cols = ['capacity_mw','longitude','commissioning_year','generation_gwh_20
       'generation_gwh_2017', 'generation_gwh_2018']

# Create a copy of the dataset for outlier treatment
no_outliers = pp_data.copy()

# Loop through each numerical column and detect and remove outliers
for col in numerical_cols:
    z_scores = stats.zscore(no_outliers[col])
    no_outliers = no_outliers[(z_scores < z_score_threshold) & (z_scores > -z_score

# Display the shape of the dataset after removing outliers
print("Shape of data after outlier removal:", no_outliers.shape)
```

```
Shape of data after outlier removal: (627, 16)
```

```
x = no_outliers.drop(columns=['name', 'gppd_idnr','primary_fuel','owner'])
```

```
plt.figure(figsize=(20,15))
graph = 1

for column in x:
  if graph<=16:
    plt.subplot(4,4,graph)
    ax=sns.boxplot(data= x[column])
    plt.xlabel(column,fontsize=15)
  graph+=1
plt.show()
```



ML Model - 3

After Treating outliers

```
# For ML Model 1 Using all variables from dataset

x = no_outliers[['capacity_mw', 'latitude', 'longitude', 'commissioning_year','gene

y = no_outliers['categorized_fuel']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting Logistic Regression model to dataset

logistic_reg = LogisticRegression()

logistic_reg.fit(x_train,y_train)

# Make predictions on the test set
y_pred = logistic_reg.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For ML Model 1:", accuracy)
print("Confusion Matrix For ML Model 1:\n", confusion)
print("Classification Report for ML Model 1:\n", classification_report_str)

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'penalty': ['l1', 'l2'],  # Regularization penalty
    'solver': ['liblinear', 'lbfgs'],  # Solver for optimization
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=logistic_reg, param_grid=param_grid, scoring='
```

```
# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For ML Model 1(with Hyperparameter Tuning):", accuracy)
print("Confusion Matrix For ML Model 1(with Hyperparameter Tuning):\n", confusion)
print("Classification Report for ML Model 1(with Hyperparameter Tuning):\n", classi
```

Insigts from the ML Model 3 :

- **Accuracy**:The initial model had an accuracy of approximately 78.34%.
- **Confusion Matrix:**

  It showed that the model misclassified 33 instances of class 1 as class 2 and 3 instances of class 2 as class 1.
- **Classification Report:**

  Precision for class 1 was low (0.40), indicating a higher rate of false positives.

  Recall for class 1 was also low (0.06), suggesting a high rate of false negatives.

  F1-score for class 1 was particularly low (0.11), indicating a lack of balance between precision and recall.

Best Hyperparameters: The optimal hyperparameters were found to be {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}.

After Hyperparameter

- **Accuracy**: The accuracy improved slightly to around 80.25% after hyperparameter tuning.
- **Confusion Matrix:**

  There was a reduction in misclassifications, but still, 23 instances of class 1 were classified as class 2 and 8 instances of class 2 were misclassified as class 1.

- **Classification Report:**

  Precision for class 1 improved to 0.56, indicating a reduction in false positives after tuning.

  Recall for class 1 also improved to 0.30, suggesting a reduction in false negatives.

  F1-score for class 1 increased to 0.39, reflecting a better balance between precision and recall, though still relatively low.

The model's overall accuracy improved marginally after hyperparameter tuning.Hyperparameter tuning notably enhanced the performance for class 1 predictions, although it still lags behind class.While there were improvements, there's still room to enhance the model's ability to predict class 1 instances more accurately without compromising much on other metrics.

ML Model - 4

Decision Tree Model

```
# Importing decision tree classifier

from sklearn.tree import DecisionTreeClassifier

# For ML Model 3, using selected variable from previous model which gives more accu

x = no_outliers[['capacity_mw', 'latitude', 'longitude', 'commissioning_year','gene

y = no_outliers['categorized_fuel']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting Logistic Regression model to dataset

decision_tree = DecisionTreeClassifier()
decision_tree.fit(x_train, y_train)

# Make predictions on the test set
y_pred = decision_tree.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For Decision Tree Model:", accuracy)
print("Confusion Matrix Decision Tree Model:\n", confusion)
print("Classification Report Decision Tree Model:\n", classification_report_str)

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid

param_grid = {
    'criterion': ['gini','entropy'],
    'max_depth': range(10,15),
    'min_samples_leaf': range(2,6),
    'min_samples_split': range(3,8),
```

```
    'max_leaf_nodes': range(5,10)
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=decision_tree, param_grid=param_grid, scoring=

# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For Decision Tree Model (with Hyperparameter Tuning):", accuracy)
print("Confusion Matrix For Decision Tree Model (with Hyperparameter Tuning):\n", c
print("Classification Report for Decision Tree Model (with Hyperparameter Tuning):\
```

```
Accuracy For Decision Tree Model: 0.8280254777070064
Confusion Matrix Decision Tree Model:
 [[ 14  19]
 [  8 116]]
Classification Report Decision Tree Model:
              precision    recall  f1-score   support

           1       0.64      0.42      0.51        33
           2       0.86      0.94      0.90       124

    accuracy                           0.83       157
   macro avg       0.75      0.68      0.70       157
weighted avg       0.81      0.83      0.81       157

Best Hyperparameters: {'criterion': 'gini', 'max_depth': 10, 'max_leaf_nodes': 9, 'mi
Best Model: DecisionTreeClassifier(max_depth=10, max_leaf_nodes=9, min_samples_leaf=2
                       min_samples_split=3)
Accuracy For Decision Tree Model (with Hyperparameter Tuning): 0.8471337579617835
Confusion Matrix For Decision Tree Model (with Hyperparameter Tuning):
 [[ 19  14]
 [ 10 114]]
Classification Report for Decision Tree Model (with Hyperparameter Tuning):
              precision    recall  f1-score   support

           1       0.66      0.58      0.61        33
```

```
           2       0.89      0.92      0.90       124

    accuracy                           0.85       157
   macro avg       0.77      0.75      0.76       157
weighted avg       0.84      0.85      0.84       157
```

Insigts from the Decision Tree Model :

- **Accuracy**:The initial accuracy of the model was around 82.8%, which means it correctly predicted the class of the data nearly 83% of the time.
- **Confusion Matrix:**

  This matrix shows that there were 14 instances of Class 1 (predicted) that were actually Class 1 (actual) and 116 instances of Class 2 (predicted) that were actually Class 2.
- **Classification Report:**

  Precision: Precision for Class 1 (0.64) indicates that when the model predicted Class 1, it was correct about 64% of the time. For Class 2 (0.86), it was around 86% accurate.

  Recall: Recall for Class 1 (0.42) shows that it identified 42% of the actual Class 1 instances. For Class 2 (0.94), it was about 94%.

  F1-score: The harmonic mean of precision and recall. F1-score for Class 1 (0.51) and Class 2 (0.90).

Best Hyperparameters: The hyperparameters chosen after tuning were {'criterion': 'gini', 'max_depth': 10, 'max_leaf_nodes': 9, 'min_samples_leaf': 2, 'min_samples_split': 3}.

After Hyperparameter

- **Accuracy**: The accuracy improved slightly to around 84.7% after hyperparameter tuning.
- **Confusion Matrix:**

  There were 19 instances of Class 1 correctly predicted and 114 instances of Class 2 correctly predicted after tuning
- **Classification Report:**

  Precision: Precision for Class 1 (0.66) and Class 2 (0.89) improved slightly after tuning, especially for Class 1.

  Recall: Recall for both classes also saw improvements, notably for Class 1 (0.58) while maintaining a good value for Class 2 (0.92).

  F1-score: There were enhancements in F1-scores for both classes, especially for Class 1 (0.61).

Hyperparameter tuning marginally enhanced the model's accuracy.Tuning led to better balance in predicting both classes, seen in the improved precision, recall, and F1-scores for Class 1,

indicating better performance in identifying this class.

ML Model - 4

RandomForestClassifier

```
# Importing decision tree classifier

from sklearn.tree import DecisionTreeClassifier

# For ML Model RandomForestClassifier, using selected variable from previous model

x = no_outliers[['capacity_mw', 'latitude', 'longitude', 'commissioning_year','gene

y = no_outliers['categorized_fuel']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting RandomForest model, first will import from sklearn package

from sklearn.ensemble import RandomForestClassifier

random_for = RandomForestClassifier()

random_for.fit(x_train,y_train)

# Make predictions on the test set
y_pred = random_for.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For RandomForestClassifier:", accuracy)
print("Confusion Matrix RandomForestClassifier:\n", confusion)
print("Classification Report RandomForestClassifier:\n", classification_report_str)

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid

param_grid = {
    'criterion': ['gini','entropy'],
```

```
    'max_depth': range(10,15),
    'min_samples_leaf': range(2,6),
    'min_samples_split': range(3,8),
    'max_leaf_nodes': range(5,10)
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=random_for, param_grid=param_grid, scoring='ac

# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For RandomForestClassifier (with Hyperparameter Tuning):", accuracy
print("Confusion Matrix For RandomForestClassifier (with Hyperparameter Tuning):\n"
print("Classification Report for RandomForestClassifier (with Hyperparameter Tuning
```

```
Accuracy For RandomForestClassifier: 0.89171974522293
Confusion Matrix RandomForestClassifier:
 [[ 18  15]
 [  2 122]]
Classification Report RandomForestClassifier:
               precision    recall  f1-score   support

           1       0.90      0.55      0.68        33
           2       0.89      0.98      0.93       124

    accuracy                           0.89       157
   macro avg       0.90      0.76      0.81       157
weighted avg       0.89      0.89      0.88       157

Best Hyperparameters: {'criterion': 'gini', 'max_depth': 13, 'max_leaf_nodes': 9, 'mi
Best Model: RandomForestClassifier(max_depth=13, max_leaf_nodes=9, min_samples_leaf=2
                       min_samples_split=6)
Accuracy For RandomForestClassifier (with Hyperparameter Tuning): 0.8407643312101911
Confusion Matrix For RandomForestClassifier (with Hyperparameter Tuning):
 [[  9  24]
 [  1 123]]
```

```
Classification Report for RandomForestClassifier (with Hyperparameter Tuning):
              precision    recall  f1-score   support

           1       0.90      0.27      0.42        33
           2       0.84      0.99      0.91       124

    accuracy                           0.84       157
   macro avg       0.87      0.63      0.66       157
weighted avg       0.85      0.84      0.80       157
```

Insigts from the RandomForestClassifier:

- **Accuracy**: The initial model achieved an accuracy of approximately 89.17%.
- **Confusion Matrix:**

  It correctly predicted 122 instances of class 2 but struggled with class 1, correctly predicting only 18 instances.

  Misclassified 15 instances of class 1 and 2 instances of class 2.
- **Classification Report:**

  Precision for class 1 was 90% but recall was 55%, indicating it missed a significant number of actual class 1 instances.

  Class 2 had higher precision (89%) and recall (98%), showing better performance in predicting this class.

  The F1-scores were 68% for class 1 and 93% for class 2, indicating a significant imbalance in performance between the two classes.

After Hyperparameter

- **Accuracy**: The tuned model's accuracy decreased slightly to around 84.08%.
- **Confusion Matrix:**

  The tuned model improved prediction for class 1 instances, correctly predicting 9 instances compared to the initial 18.
- **Classification Report:**

  Precision for class 1 remained the same (90%), but recall dropped significantly to 27%, indicating the model missed a lot more class 1 instances after tunin

  Class 2 maintained a high precision (84%) and recall (99%), showing consistent performance in predicting this class.

  The F1-score for class 1 dropped to 42%, indicating a substantial decrease in overall performance for predicting this class, while class 2 maintained a high F1-score at 91%.

Initially, the model was biased towards the majority class (class 2) and showed poorer performance for the minority class (class 1).While tuning improved the performance of class 2 predictions, it significantly deteriorated the model's ability to predict class 1 instances.

ML Model - 5

KNeighborsClassifier

```
# Importing decision tree classifier

from sklearn.tree import DecisionTreeClassifier

# For ML Model 7,Using all KNeighborsClassifier

x = no_outliers[['capacity_mw', 'latitude', 'longitude', 'commissioning_year','gene

y = no_outliers['categorized_fuel']


# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.25,random_state=34

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting KNeighborsClassifier model, first will import from sklearn package

from sklearn.neighbors import KNeighborsClassifier

kNN = KNeighborsClassifier()

kNN.fit(x_train,y_train)

# Make predictions on the test set
y_pred = kNN.predict(x_test)

# Evaluate the model using various metrics
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy For KNeighborsClassifier:", accuracy)
print("Confusion Matrix for KNeighborsClassifier:\n", confusion)
print("Classification Report for KNeighborsClassifier:\n", classification_report_st

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data using the scaler
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define the hyperparameter grid

param_grid = {
```

```
    'n_neighbors': range(1, 21),
    'weights': ['uniform', 'distance'],
    'p': [1, 2],
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=kNN, param_grid=param_grid, scoring='accuracy'

# Fit the grid search to your training data
grid_search.fit(x_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Get the best model
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(x_test_scaled)

# Evaluate the model with the best hyperparameters
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Print the best hyperparameters, best model, and evaluation metrics
print("Best Hyperparameters:", best_params)
print("Best Model:", best_model)
print("Accuracy For KNeighborsClassifier (with Hyperparameter Tuning):", accuracy)
print("Confusion Matrix For KNeighborsClassifier (with Hyperparameter Tuning):\n",
print("Classification Report for KNeighborsClassifier (with Hyperparameter Tuning):
```

```
Accuracy For KNeighborsClassifier: 0.8535031847133758
Confusion Matrix for KNeighborsClassifier:
 [[ 18  15]
 [  8 116]]
Classification Report for KNeighborsClassifier:
              precision    recall  f1-score   support

           1       0.69      0.55      0.61        33
           2       0.89      0.94      0.91       124

    accuracy                           0.85       157
   macro avg       0.79      0.74      0.76       157
weighted avg       0.84      0.85      0.85       157

Best Hyperparameters: {'n_neighbors': 1, 'p': 1, 'weights': 'uniform'}
Best Model: KNeighborsClassifier(n_neighbors=1, p=1)
Accuracy For KNeighborsClassifier (with Hyperparameter Tuning): 0.89171974522293
Confusion Matrix For KNeighborsClassifier (with Hyperparameter Tuning):
 [[ 22  11]
 [  6 118]]
Classification Report for KNeighborsClassifier (with Hyperparameter Tuning):
              precision    recall  f1-score   support
```

```
           1       0.79      0.67      0.72        33
           2       0.91      0.95      0.93       124

    accuracy                           0.89       157
   macro avg       0.85      0.81      0.83       157
weighted avg       0.89      0.89      0.89       157
```

Insigts from the KNeighborsClassifier :

- **Accuracy**: The model correctly predicts the class of the samples 85.35% of the time.
- **Confusion Matrix:**

  33 instances of class 1 were predicted, out of which 18 were correctly classified, and 15 were misclassified as class 2.

  124 instances of class 2 were predicted, with 116 correctly classified and 8 misclassified as class 1.
- **Classification Report:**

  Precision for class 1 (low recall) suggests that when the model predicts class 1, it's correct only 69% of the time. It misses several actual class 1 instances.

  Recall for class 1 (low precision) indicates that out of all actual class 1 instances, the model only captures 55% of them.

  F1-score, the harmonic mean of precision and recall, is relatively lower for class 1 compared to class 2.

After Hyperparameter

- **Accuracy**: The accuracy improved significantly after hyperparameter tuning, increasing to 89.17%.
- **Confusion Matrix:**

  Class 1 predictions improved with 22 correct classifications and 11 misclassifications.

  Class 2 predictions remained largely accurate, with 118 correct classifications and 6 misclassifications.
- **Classification Report:**

  Precision, recall, and F1-score for class 1 improved noticeably after tuning, reflecting better performance for this class.

  Precision for class 1 increased to 79%, indicating that when the model predicts class 1, it's correct 79% of the time.

  Recall for class 1 also increased to 67%, capturing a higher proportion of actual class 1 instances.The F1-score for class 1 improved to 0.72, showcasing better balance between precision and recall compared to before tuning.

The initial model had an acceptable accuracy but struggled with precision and recall for class 1, indicating a bias towards class 2.Hyperparameter tuning led to significant improvement in the model's performance, especially in correctly identifying instances of class 1, as seen in the enhanced precision, recall, and F1-score for that class.

Conclusion

Based on the evaluation matrices for the K-Nearest Neighbors (KNN) classifier before and after hyperparameter tuning, the model showed substantial improvement after tuning. It achieved an accuracy of 89.17% after tuning compared to 85.35% before.

Will save the classification model with name "classification_Model"

```
import pickle

# Save the model to a file
with open('classification_Model.pkl', 'wb') as file:
    pickle.dump(kNN, file)


# Load the saved model from file
with open('classification_Model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)
```

Now will predict for capacity_mw

ML Model - 1

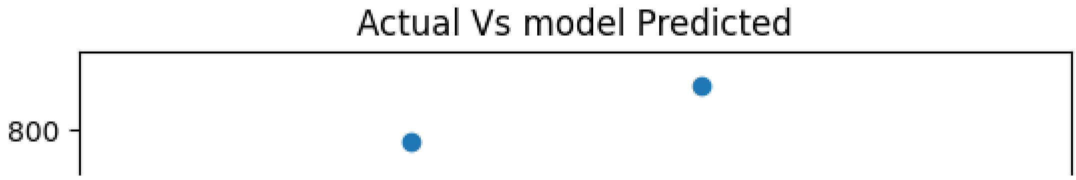
Using all varibales

```
# Importing Necessary Libraries

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import Ridge
import math
```

```
x = no_outliers[['latitude', 'longitude', 'commissioning_year','generation_gwh_2014

y = no_outliers['capacity_mw']

# splitting data into train and test set.

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.30,random_state=24

("Shape of x_train",x_train.shape)
("Shape of x_test",x_test.shape)
("Shape of y_train",y_train.shape)
("Shape of y_train",y_test.shape)

# Transforming data standardization

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# Fitting linear regressio to training set
LR = LinearRegression()
LR.fit(x_train, y_train)

# Predicting on test set results

y_pred = LR.predict(x_test)

y_pred

# Evaluate the Linear Regression model
LR_predictions = LR.predict(x_test)
LR_mse = mean_squared_error(y_test, LR_predictions)
LR_RMSE = math.sqrt(mean_squared_error(y_test,y_pred))
LR_r2 = r2_score(y_test, LR_predictions)
print("Linear Regression MSE For Model 1: ", LR_mse)
print("Linear Regression RMSE For Model 1: ", LR_RMSE)
print("Linear Regression R-squared For Model 1: ", LR_r2)


plt.scatter(y_test,y_pred)
plt.xlabel('Actual Demand')
plt.ylabel('Predicted Demand')
plt.title('Actual Vs model Predicted')
plt.show()

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression, Lasso, Ridge

# Lasso Regression model with hyperparameter tuning
lasso_model = Lasso()
lasso_param_grid = {'alpha': [0.01, 0.1, 1, 10]}
lasso_grid = GridSearchCV(lasso_model, lasso_param_grid, cv=5)
```

```
lasso_grid.fit(x_train, y_train)

# Evaluate the Lasso Regression model
lasso_predictions = lasso_grid.predict(x_test)
lasso_mse = mean_squared_error(y_test, lasso_predictions)
lasso_r2 = r2_score(y_test, lasso_predictions)
print("Lasso Regression MSE: ", lasso_mse)
print("Lasso Regression R-squared: ", lasso_r2)
print("Best Lasso Alpha: ", lasso_grid.best_params_['alpha'])


# Similarly for ridge regression


# Ridge Regression model with hyperparameter tuning
ridge_model = Ridge()
ridge_param_grid = {'alpha': [0.01, 0.1, 1, 10]}
ridge_grid = GridSearchCV(ridge_model, ridge_param_grid, cv=5)
ridge_grid.fit(x_train, y_train)

# Evaluate the Ridge Regression model
ridge_predictions = ridge_grid.predict(x_test)
ridge_mse = mean_squared_error(y_test, ridge_predictions)
ridge_r2 = r2_score(y_test, ridge_predictions)
print("Ridge Regression MSE: ", ridge_mse)
print("Ridge Regression R-squared: ", ridge_r2)
print("Best Ridge Alpha: ", ridge_grid.best_params_['alpha'])
```

```
Linear Regression MSE For Model 1:  60167.18455622816
Linear Regression RMSE For Model 1:  245.29000092997708
Linear Regression R-squared For Model 1:  -3.477055875372492
```



Insights from ML Model 1:

- **Linear Regression:**

  MSE: 60167.18

  RMSE: 245.29

  R-squared: -3.48

- **Lasso Regression:**

  MSE: 17275.98

  R-squared: -0.29

  Best Alpha: 1

- **Ridge Regression:**

  MSE: 50279.22

  R-squared: -2.74

  Best Alpha: 0.1

The linear regression model is performing poorly with a high MSE and RMSE, indicating significant errors in predicting temperatures. The negative R-squared value suggests the model fits the data worse than a horizontal line.Lasso regression, using a penalty term to shrink less important features to zero, performs better than linear regression but still struggles.Ridge regression, penalizing large coefficients to prevent overfitting, also displays better performance than linear regression but lags behind Lasso.

ML Model - 2

Decision Tree Regression Model

```
from sklearn.tree import DecisionTreeRegressor

x = no_outliers[['latitude', 'longitude', 'commissioning_year','generation_gwh_201

y = no_outliers['capacity_mw']

# Train and test set split
x_train,x_test,y_train,y_test = train_test_split(x,y,random_state=348)

# Fitting Model
clf = DecisionTreeRegressor()
clf.fit(x_train,y_train)

# defining function for evalution matrix
def metric_score(model, x_train, x_test, y_train, y_test, train=True):
    if train:
        y_pred = model.predict(x_train)
        mse = mean_squared_error(y_train, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_train, y_pred)
        print('\n======Train Result======')
        print(f'Mean Squared Error (MSE): {mse:.2f}')
        print(f'Root Mean Squared Error (RMSE): {rmse:.2f}')
        print(f'R-squared (R2): {r2:.2f}')
    else:
        y_pred = model.predict(x_test)
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)
        print('\n======Test Result======')
        print(f'Mean Squared Error (MSE): {mse:.2f}')
        print(f'Root Mean Squared Error (RMSE): {rmse:.2f}')
        print(f'R-squared (R2): {r2:.2f}')

# Calling above function and passing dataset to check train and test score

metric_score(clf,x_train,x_test,y_train,y_test,train=True) # for training score

metric_score(clf,x_train,x_test,y_train,y_test,train=False)  # for testing score

# Now doing Hypertuning

grid_param = {
    'criterion': ['squared_error'],
    'max_depth': range(5, 10),
    'min_samples_leaf': range(1, 3),
    'min_samples_split': range(1, 5),
    'max_leaf_nodes': range(3, 6)
}

grid_search = GridSearchCV(estimator=clf,
                           param_grid=grid_param,
```

```
                           cv=5,
                           n_jobs=-1,
                           error_score=np.nan)

grid_search.fit(x_train, y_train)

best_parameters = grid_search.best_params_
print(best_parameters)

# Using best_param for model

clf = DecisionTreeRegressor(criterion='squared_error',min_samples_split=3,max_dept

clf.fit(x_train,y_train)

metric_score(clf,x_train,x_test,y_train,y_test,train=True) # for training score
```

```
======Train Result======
Mean Squared Error (MSE): 10.10
Root Mean Squared Error (RMSE): 3.18
R-squared (R2): 1.00

======Test Result======
Mean Squared Error (MSE): 9119.24
Root Mean Squared Error (RMSE): 95.49
R-squared (R2): 0.44
{'criterion': 'squared_error', 'max_depth': 6, 'max_leaf_nodes': 4, 'min_samples_l

======Train Result======
Mean Squared Error (MSE): 4151.35
Root Mean Squared Error (RMSE): 64.43
R-squared (R2): 0.77

======Test Result======
Mean Squared Error (MSE): 7930.24
Root Mean Squared Error (RMSE): 89.05
R-squared (R2): 0.51
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.py:378
150 fits failed out of a total of 600.
```