# Probabilistic Model of Solar and Wind Power

**NAMES:**

- Atharv Gaur (21085127, EEE)
- Yash Somalkar (21085120, EEE)
- Raghav Chaudhary (21085119, EEE)

# Introduction

In recent years, the growing demand for renewable energy sources has underscored the importance of accurately predicting solar and wind power generation. These energy sources are inherently variable and influenced by a multitude of factors, including weather conditions, geographical location, and time of year. This report presents a probabilistic modeling approach to forecast solar and wind energy production, employing various machine learning techniques such as Ridge Regression, Decision Trees, Support Vector Machines (SVM), Random Forest, and Autoregressive Integrated Moving Average (ARIMA). By leveraging historical data and statistical methods, the proposed models aim to improve prediction accuracy and enhance the reliability of renewable energy systems. The findings from this analysis will contribute to better decision-making in energy management and policy formulation, ultimately supporting the transition to a sustainable energy future.

# Mathematical Equations and Theory

## Ridge Regression

**Overview**

Ridge Regression is a linear regression technique that adds a penalty term to the loss function to prevent overfitting and handle multicollinearity.

**Mathematical Modeling**

- **Objective Function**:

$$\min_{\beta} \left( ||y - X\beta||^2 + \lambda ||\beta||^2 \right)$$

where:

- $y$: Response variable
- $X$: Design matrix (features)
- $\beta$: Coefficients
- $\lambda$: Regularization parameter (controls the strength of the penalty)

# Decision Trees

### Overview

Decision Trees are a non-linear model used for regression and classification tasks, where data is split into subsets based on feature values.

### Mathematical Modeling

- **Gini Index** (for classification):

$$Gini(D) = 1 - \sum_{i=1}^{n} (p_i)^2$$

  where $p_i$ is the proportion of instances of class $i$ in dataset $D$.

- **Entropy** (for classification):

$$Entropy(D) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

- **Mean Squared Error** (for regression):

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

  where $y_i$ is the actual value and $\hat{y}_i$ is the predicted value.

# Support Vector Machines (SVM)

### Overview

SVM is a supervised learning model used for classification and regression tasks that finds the optimal hyperplane to separate data points.

### Mathematical Modeling

- **Objective Function**:

$$\min_{\beta,b} \left( \frac{1}{2} ||\beta||^2 \right) \quad \text{subject to } y_i(\beta^T x_i + b) \geq 1 \quad \forall i$$

  where:

  - $x_i$: Input features
  - $y_i$: Class labels (+1 or -1)
  - $\beta$: Coefficients (weights)
  - $b$: Bias term

# Random Forest

### Overview

Random Forest is an ensemble learning method that constructs multiple decision trees and aggregates their predictions for improved accuracy and robustness.

### Mathematical Modeling

- **Final Prediction**:
  - For Regression:

$$\hat{y} = \frac{1}{M} \sum_{m=1}^{M} T_m(x)$$

    where $T_m(x)$ is the prediction of the $m$-th tree and $M$ is the number of trees.

  - For Classification:

$$\hat{y} = \text{mode}(T_1(x), T_2(x), \ldots, T_M(x))$$

## ARIMA (AutoRegressive Integrated Moving Average)

### Overview

ARIMA is a time series forecasting method that combines autoregressive (AR) and moving average (MA) models, accounting for differencing to make the data stationary.

### Mathematical Modeling

- **ARIMA(p, d, q) Model**:

$$(1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p)(1 - B)^d y_t = (1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q)\epsilon_t$$

where:

- $y_t$: Time series value at time $t$
- $B$: Backward shift operator
- $\phi$: Parameters of the AR part
- $\theta$: Parameters of the MA part
- $\epsilon_t$: White noise error term

# Input Data Specifications

The input dataset consists of 17,568 hourly entries, spanning from December 31, 2014, to January 1, 2017, indexed by datetime. It includes 15 columns covering various metrics related to energy load and renewable generation in Germany. Key variables include actual and forecasted energy load, solar and wind generation (actual and capacity), and profiles for both solar and wind generation (offshore and onshore). Data types include float, integer, and object, with some missing values in the actual generation and profile columns.

| Column Name | Description | Data Type | Non-Null Count | Units |
|---|---|---|---|---|
| cet_cest_timestamp | Timestamp in CET/CEST timezone | Object | 17568 | - |
| DE_load_actual_entsoe_transparency | Actual energy load (Germany) | Float64 | 17567 | MW |
| DE_load_forecast_entsoe_transparency | Forecasted energy load (Germany) | Float64 | 17567 | MW |

| | | | | |
|---|---|---|---|---|
| DE_solar_capacity | Total installed solar capacity | Int64 | 17568 | MW |
| DE_solar_generation_actual | Actual solar energy generation | Float64 | 17464 | MW |
| DE_solar_profile | Solar generation profile | Float64 | 17464 | - |
| DE_wind_capacity | Total installed wind capacity | Int64 | 17568 | MW |
| DE_wind_generation_actual | Actual wind energy generation | Float64 | 17493 | MW |
| DE_wind_profile | Wind generation profile | Float64 | 17493 | - |
| DE_wind_offshore_capacity | Total installed offshore wind capacity | Int64 | 17568 | MW |
| DE_wind_offshore_generation_actual | Actual offshore wind energy generation | Float64 | 17493 | MW |
| DE_wind_offshore_profile | Offshore wind generation profile | Float64 | 17493 | - |
| DE_wind_onshore_capacity | Total installed onshore wind capacity | Int64 | 17568 | MW |
| DE_wind_onshore_generation_actual | Actual onshore wind energy generation | Float64 | 17495 | MW |
| DE_wind_onshore_profile | Onshore wind generation profile | Float64 | 17495 | - |

**Summary:**

- **Data Range**: December 31, 2014, to January 1, 2017.
- **Frequency**: Hourly data points.
- **Total Records**: 17,568 entries.
- **Data Types**: Contains `float64`, `int64`, and `object` types.
- **Missing Values**: Noted in actual generation and profile columns for solar and wind energy metrics.

# Output Data Specifications

| Column Name | Description | Data Type | Units |
|---|---|---|---|
| timestamp | Datetime indicating the hour of the prediction | Datetime | - |
| predicted_load | Predicted total energy load | Float64 | MW |
| predicted_solar_generation | Predicted solar energy generation | Float64 | MW |
| predicted_wind_generation | Predicted total wind energy generation | Float64 | MW |
| predicted_wind_offshore_generation | Predicted offshore wind energy generation | Float64 | MW |
| predicted_wind_onshore_generation | Predicted onshore wind energy generation | Float64 | MW |
| actual_load | Actual total energy load | Float64 | MW |
| actual_solar_generation | Actual solar energy generation | Float64 | MW |
| actual_wind_generation | Actual total wind energy generation | Float64 | MW |
| actual_wind_offshore_generation | Actual offshore wind energy generation | Float64 | MW |
| actual_wind_onshore_generation | Actual onshore wind energy generation | Float64 | MW |
| error_load | Prediction error for total energy load | Float64 | MW |
| error_solar_generation | Prediction error for solar energy generation | Float64 | MW |
| error_wind_generation | Prediction error for total wind energy generation | Float64 | MW |
| error_wind_offshore_generation | Prediction error for offshore wind energy generation | Float64 | MW |
| error_wind_onshore_generation | Prediction error for onshore wind energy generation | Float64 | MW |

**Summary:**

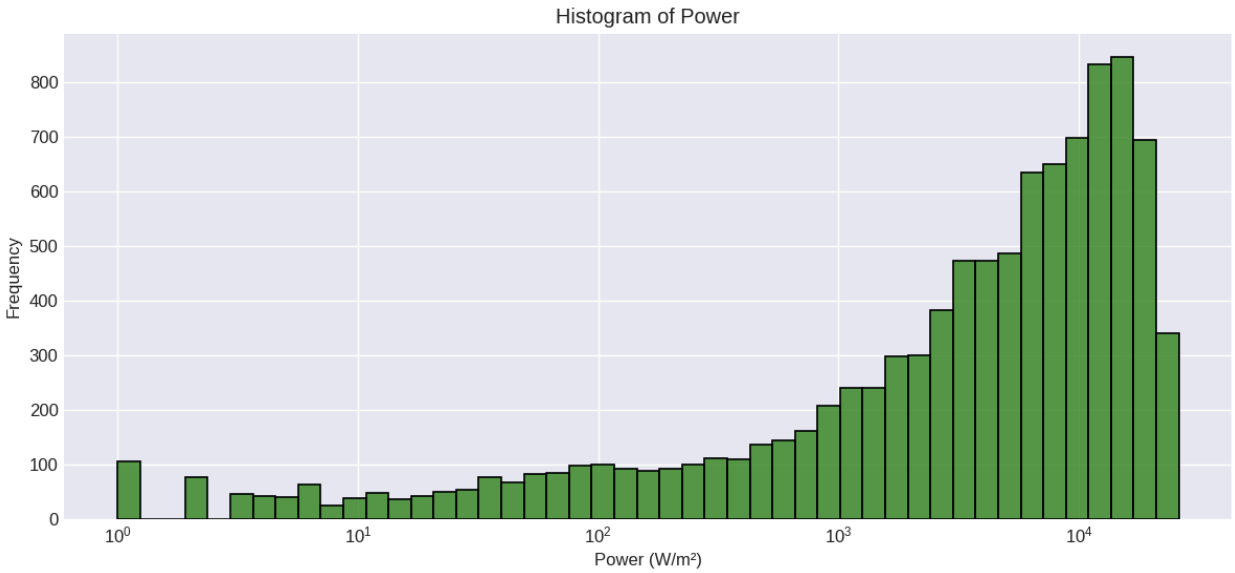
- **Data Range**: Aligned with input data timestamps.
- **Frequency**: Hourly predicted and actual values.
- **Data Types**: Primarily `float64` for numerical data and `datetime` for timestamps.
- **Error Calculation**: Each prediction error is calculated as the difference between actual and predicted values, allowing for error analysis across energy types.

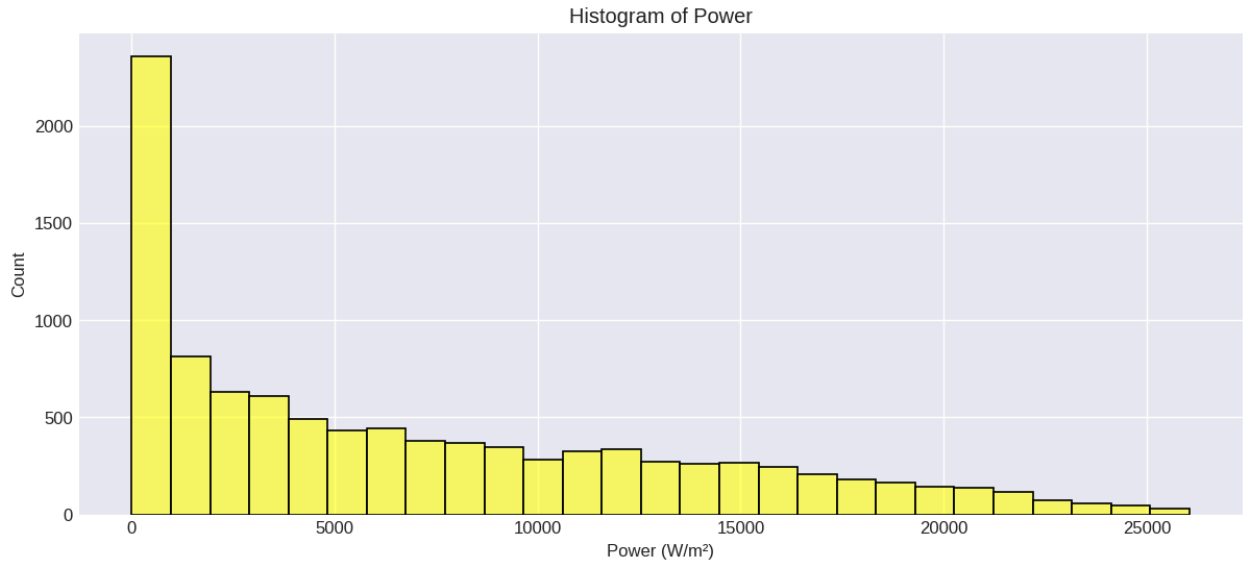
This output structure supports model performance evaluation, error analysis, and time-series comparison between forecasted and actual values.

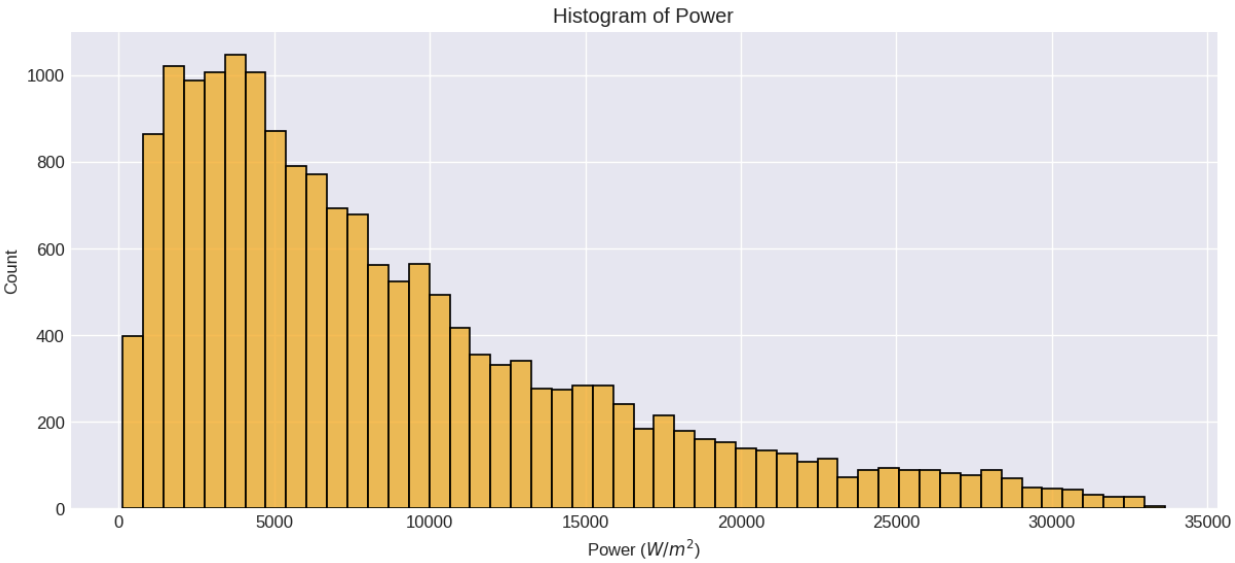
# Data Selection and Justification for Power Systems Analysis

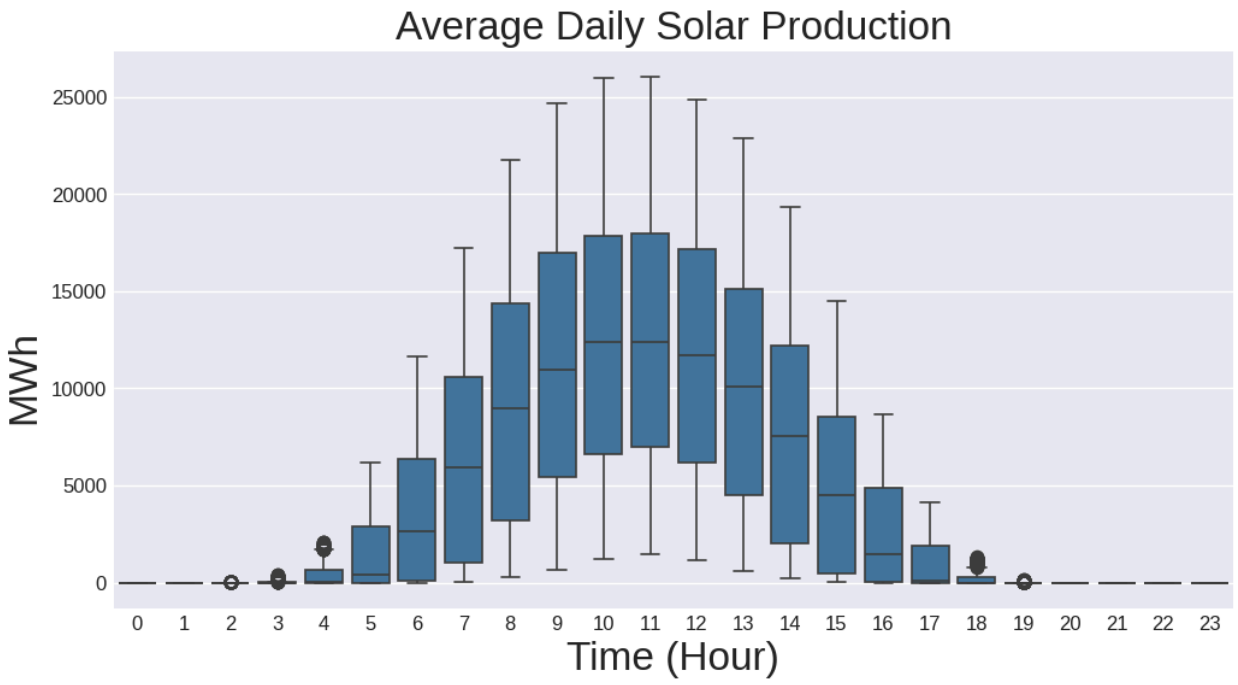
## Dataset

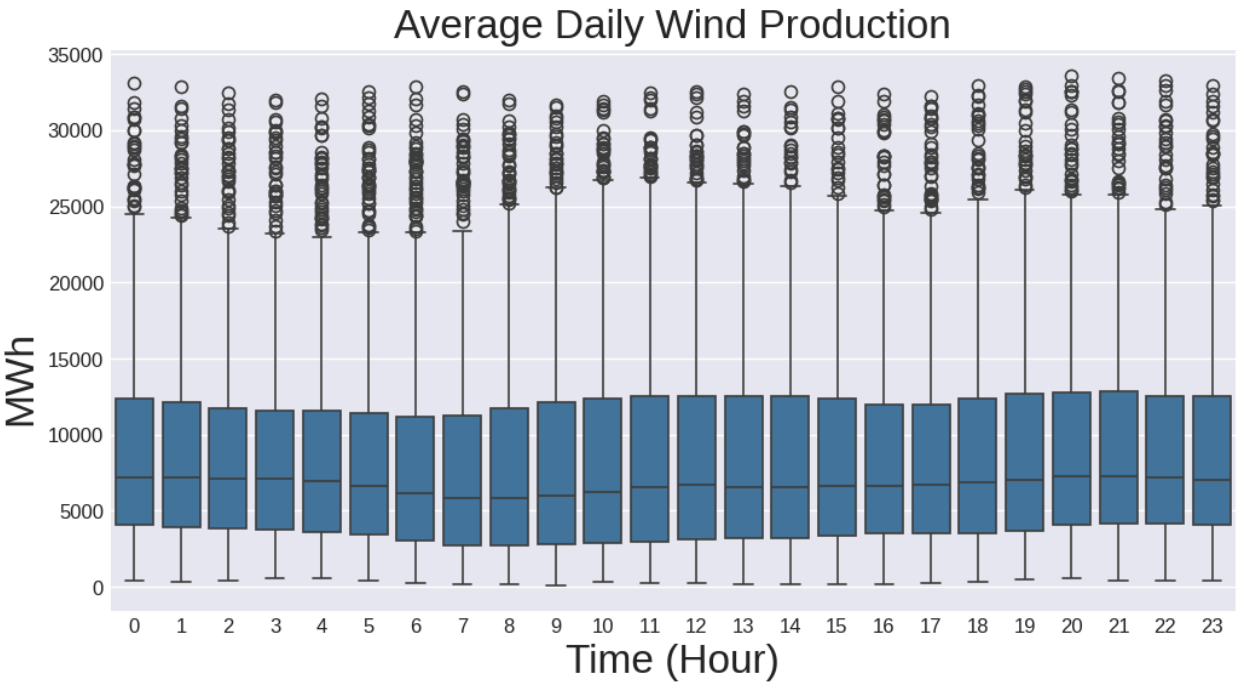
The dataset was obtained from Open Power System Data, a free and open-source platform that provides power system data for 37 European countries. We specifically focused on Germany, as it has the highest proportion of renewable energy—approximately 46%—derived from solar, wind, and biomass sources. This makes it an exemplary case for analyzing trends in global energy transitions. The data file comprises about 16 variables, including UTC timestamps, solar capacity, wind capacity, solar profiles, wind profiles, and both onshore and offshore wind profiles.

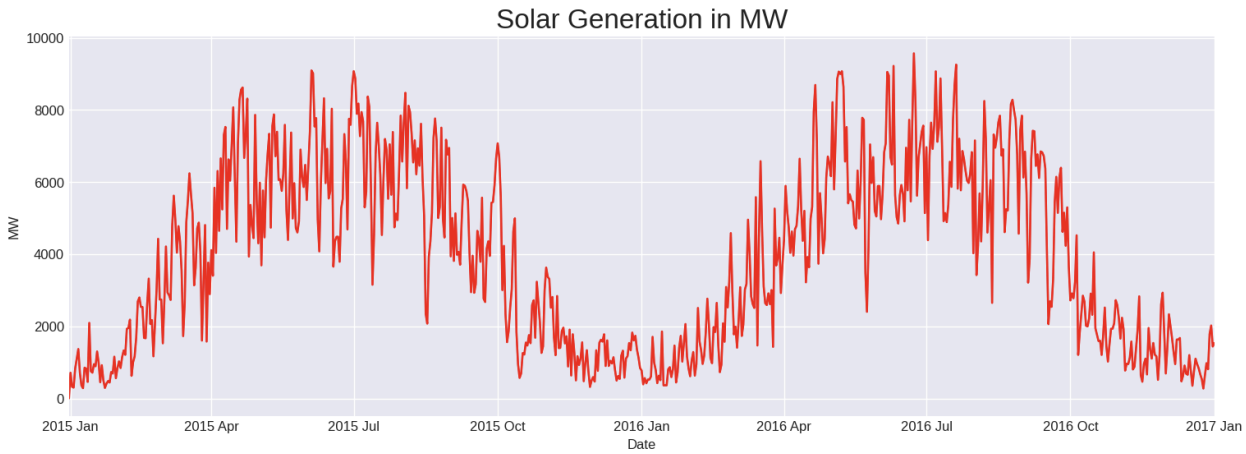


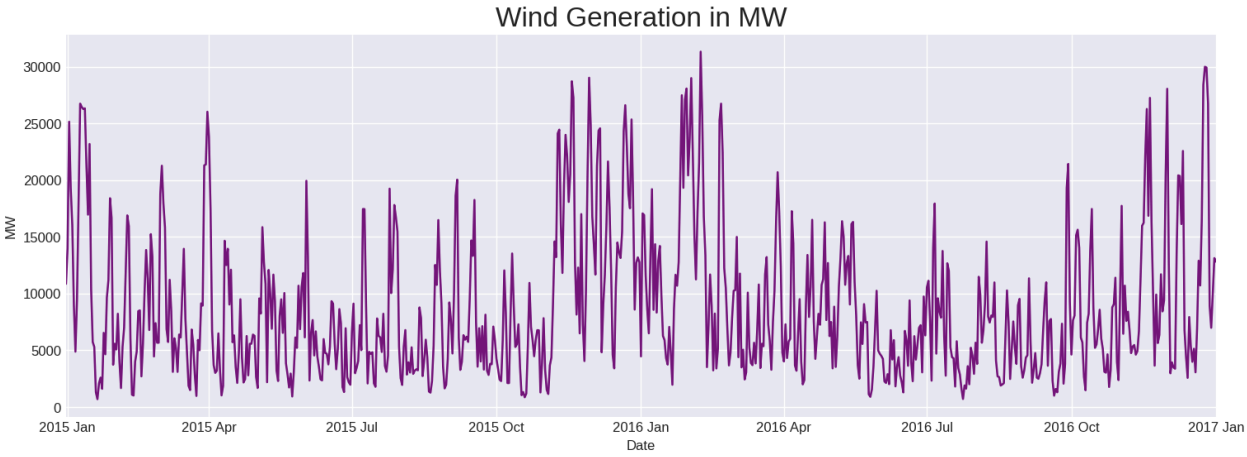

Histogram of Power
Count
Power (W/m²)
0
500
1000
1500
2000
0
5000
10000
15000
20000
25000


Histogram of Power
Count
Power ($W/m^2$)
0
200
400
600
800
1000
0
5000
10000
15000
20000
25000
30000
35000


Average Daily Solar Production
MWh
25000
20000
15000
10000
5000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Time (Hour)


Average Daily Wind Production
MWh
35000
30000
25000
20000
15000
10000
5000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Time (Hour)



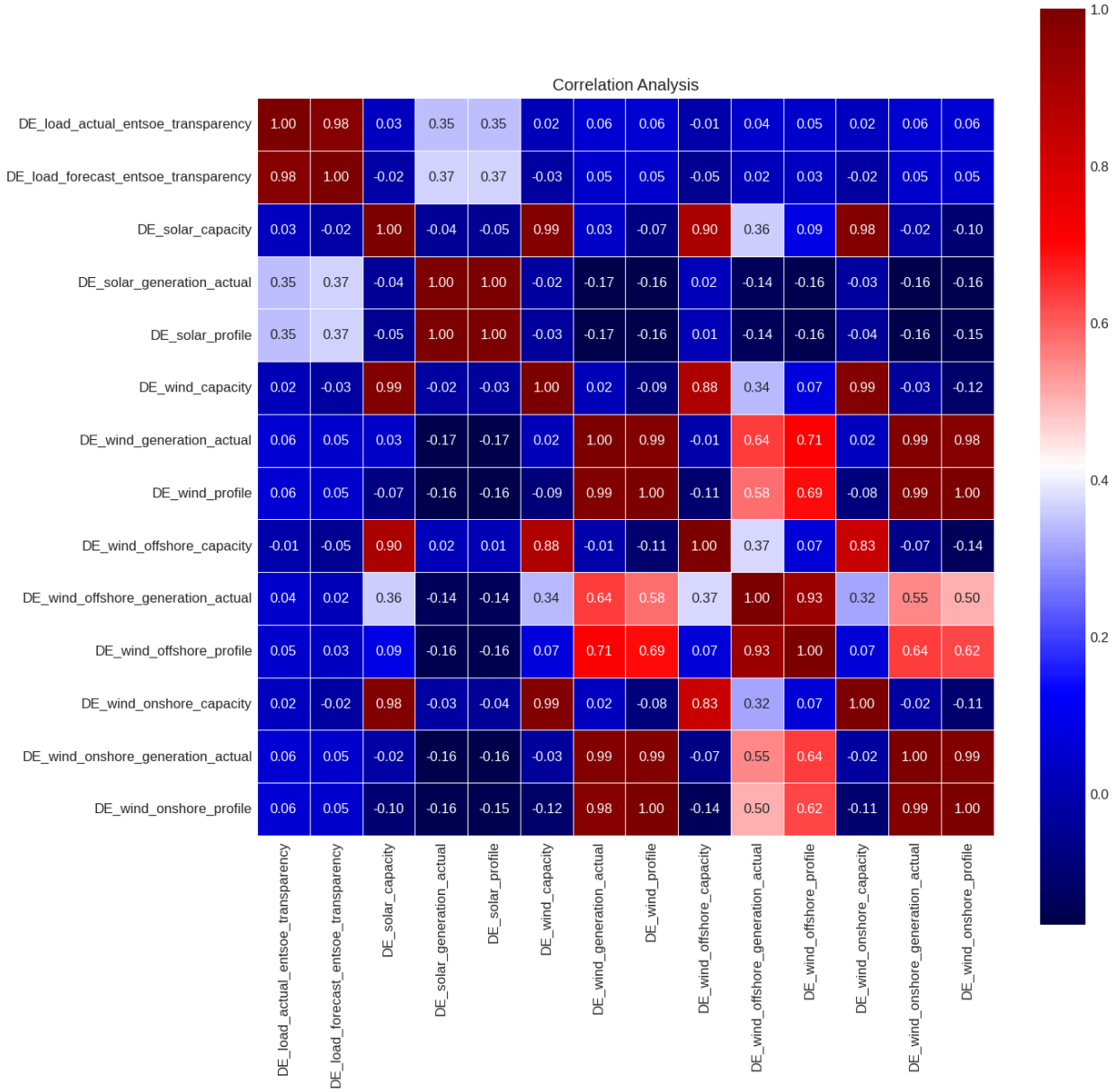

# Data Preprocessing

To address null values in the dataset, we employed the following strategies:

For the solar generation actual data, any null value was replaced with the value from the previous day. For the first day of the month, where no prior value exists, we filled it with 0, assuming no solar generation occurred before 6 AM. We subsequently replaced any remaining NaN values with the mean of the solar generation actual data.
For wind generation actual data, null values were filled with the mean value of the entire wind generation actual column.

# Correlation Analysis

Our correlation analysis revealed that certain features exhibited stronger relationships with the target variables. The solar profile showed the highest correlation with solar generation output, while in the case of wind energy, the wind profile, wind onshore profile, and wind onshore generation were the most correlated, followed by the wind offshore profile and wind offshore generation.



## Feature Importance

In addition to correlation analysis, we utilized feature importance metrics to identify the most relevant features in our dataset for effective model training. This technique assesses the significance of each input feature, ensuring that irrelevant data is eliminated to minimize bias and enhance the robustness of the model

The selected data focuses on Germany's power system due to its leading role in renewable energy adoption, with approximately 46% of its energy generated from solar, wind, and biomass sources. This makes it a crucial case study for understanding the future of energy systems globally. By including variables such as solar and wind capacity, generation profiles, and UTC timestamps, the dataset provides comprehensive insights into the dynamics of renewable energy generation. This data is essential for analyzing performance trends, forecasting energy outputs, and informing policies aimed at enhancing renewable energy integration.

# Case Study and Program Writing

The estimation was carried out with the help of different models:

1. **Ridge Regression:** A linear regression model that includes L2 regularization to prevent overfitting.
2. **Decision Trees:** A model that splits data into branches based on feature values for **regression or classification tasks.**
3. **Support Vector Machines (SVM):** A model that finds the optimal hyperplane to separate different classes in the feature space.
4. **Random Forest:** An ensemble method that builds multiple decision trees to improve accuracy and reduce overfitting in predictions.
5. **ARIMA (AutoRegressive Integrated Moving Average):** A time series forecasting model that combines autoregressive and moving average components to predict future values based on past observations.

## Ridge regression

### Input details

- **Features for Solar Energy (X1_train, X1_test)**: These include variables such as solar capacity, solar profiles, and time-based features that influence solar generation.
- **Target Variable for Solar Energy (y1_train, y1_test)**: Represents the actual solar generation values used for training and evaluation.
- **Features for Wind Energy (X2_train, X2_test)**: These consist of wind capacity, wind profiles (onshore and offshore), and other relevant meteorological variables affecting wind generation.
- **Target Variable for Wind Energy (y2_train, y2_test)**: Corresponds to the actual wind generation values used for training and testing.

## Output details

- **Predictions for Solar Energy (y1_pred, y1_pred2)**: Contains the predicted solar generation values from the trained Ridge regression model based on test data.
- **Predictions for Wind Energy (y2_pred, y2_pred2)**: Contains the predicted wind generation values from the trained Ridge regression model based on test data.

## Constants and hyper parameters

**Regularization Strength (alpha)**:

- Initial value for solar energy: `alpha=1`
- Tuned value for solar energy: `alpha=2`
- Initial value for wind energy: `alpha=1`
- Tuned value for wind energy: `alpha=2`

These hyperparameters control the level of regularization applied to prevent overfitting, with higher values of alpha leading to more regularization.

## Training code

The Ridge regression models for both solar and wind energy are initialized and trained using the `.fit()` method, which adjusts model parameters to minimize prediction error based on training datasets.

```python
from sklearn.linear_model import Ridge  # Import the Ridge regression model from scikit-learn

# Initialize the Ridge regression model with a specified regularization strength (alpha).
ridgeSolar = Ridge(alpha=1)

# Train the Ridge regression model using the training data (X1_train and y1_train).
# This step adjusts the model parameters to minimize the error between predicted and actual values.
ridgeSolar.fit(X1_train, y1_train)

# Use the trained model to make predictions on the test set (X1_test).
# The predicted values for solar generation are stored in y1_pred.
y1_pred = ridgeSolar.predict(X1_test)

# Initialize the Ridge regression model for wind energy with a specified regularization strength (alpha).
```

```
ridgeWind = Ridge(alpha=1)

# Train the Ridge regression model using the training data for wind energy
(X2_train and y2_train).
# This step adjusts the model parameters to minimize the error between
predicted and actual values for wind generation.
ridgeWind.fit(X2_train, y2_train)

# Use the trained Ridge regression model to make predictions on the test set
for wind energy (X2_test).
# The predicted values for wind generation are stored in y2_pred.
y2_pred = ridgeWind.predict(X2_test)  # Corrected to use ridgeWind instead of
ridgeSolar

# Hyperparameter tuning - setting the regularization strength (alpha) to 2 for
Ridge regression on solar energy.
ridgeSolar = Ridge(alpha=2)

# Fit the Ridge regression model using the training data for solar energy
(X1_train and y1_train).
# This step adjusts the model parameters to minimize the error with the new
alpha value.
ridgeSolar.fit(X1_train, y1_train)

# Use the trained Ridge regression model to make predictions on the test set
for solar energy (X1_test).
# The predicted values for solar generation are stored in y1_pred2.
y1_pred2 = ridgeSolar.predict(X1_test)

# Hyperparameter tuning - setting the regularization strength (alpha) to 2 for
Ridge regression on wind energy.
ridgeWind = Ridge(alpha=2)

# Fit the Ridge regression model using the training data for wind energy
(X2_train and y2_train).
# This step adjusts the model parameters to minimize the error with the new
alpha value.
ridgeWind.fit(X2_train, y2_train)

# Use the trained Ridge regression model to make predictions on the test set
for wind energy (X2_test).
# The predicted values for wind generation are stored in y2_pred2.
y2_pred2 = ridgeWind.predict(X2_test)  # Corrected to use ridgeWind for
predictions
```

# Results

## Evaluation Metrics

### Mean Squared Error

```
[ ] from sklearn.metrics import mean_squared_error

    # Calculate the Mean Squared Error (MSE) for the solar energy predictions.
    # MSE measures the average of the squares of the errors, which is the average squared difference between predicted and actual values.
    print("Solar MSE = ", mean_squared_error(y1_test, y1_pred))
```

```
Solar MSE =  135628.8764825295
```

```
[ ] from sklearn.metrics import mean_squared_error

    # Calculate the Mean Squared Error (MSE) for the wind energy predictions.
    # MSE quantifies the average squared difference between the predicted and actual wind generation values.
    print("Wind MSE = ", mean_squared_error(y2_test, y2_pred))
```

```
Wind MSE =  1596947.038020549
```

```
[ ] #alpha = 2
    # Calculate the Mean Squared Error (MSE) for the solar energy predictions with a Ridge regression model using alpha = 2.
    # This metric helps assess how well the model predicts solar generation compared to the actual values.
    print("Solar MSE = ", mean_squared_error(y1_test, y1_pred2))
```

```
Solar MSE =  136362.76833221284
```

```
[ ] #alpha = 2
    # Calculate the Mean Squared Error (MSE) for the wind energy predictions with a Ridge regression model using alpha = 2.
    # This metric evaluates how accurately the model predicts wind generation compared to the actual values.
    print("Wind MSE = ", mean_squared_error(y2_test, y2_pred2))
```

```
Wind MSE =  1598111.9471295285
```

### Mean Absolute Error

```
[ ] # Calculate the Mean Absolute Error (MAE) for the solar energy predictions using the Ridge regression model.
    # This metric measures the average magnitude of errors in the predictions, giving an idea of how far off the predictions are from the actual values.
    print("Solar MAE = ", mean_absolute_error(y1_test, y1_pred))
```

```
Solar MAE =  101.5840459159988
```

```
[ ] # Calculate the Mean Absolute Error (MAE) for the wind energy predictions using the Ridge regression model.
    # MAE provides an average of the absolute differences between predicted and actual values, indicating the model's accuracy for wind energy predictions.
    print("Wind MAE = ", mean_absolute_error(y2_test, y2_pred))
```

```
Wind MAE =  684.3662305728255
```

```
[ ] # Calculate the Mean Absolute Error (MAE) for the solar energy predictions using the Ridge regression model with alpha set to 2.
    # MAE measures the average absolute differences between predicted and actual values, helping to evaluate the model's accuracy for solar energy predictions.
    print("Solar MAE = ", mean_absolute_error(y1_test, y1_pred2))
```

```
Solar MAE =  108.35587134811672
```

```
[ ] # Calculate the Mean Absolute Error (MAE) for the wind energy predictions using the Ridge regression model with alpha set to 2.
    # MAE indicates the average absolute differences between the predicted and actual values, providing insights into the model's accuracy for wind energy predictions.
    print("Wind MAE = ", mean_absolute_error(y2_test, y2_pred2))
```

```
Wind MAE =  687.1027652112468
```

### Root Mean Squared Error

```
[ ] print("Solar RMSE = ",np.sqrt(mean_squared_error(y1_test,y1_pred)))
```

```
Solar RMSE =  368.2782595844201
```

```
[ ] print("Wind RMSE = ",np.sqrt(mean_squared_error(y2_test,y2_pred)))
```

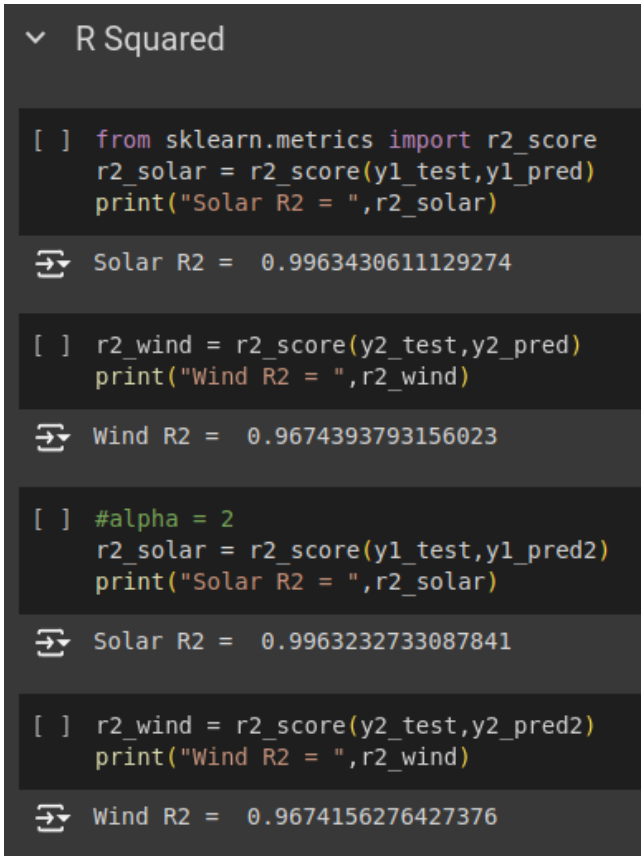
```
Wind RMSE =  1263.703698665375
```

```
[ ] #alpha = 2
    print("Solar RMSE = ",np.sqrt(mean_squared_error(y1_test,y1_pred2)))
```

```
Solar RMSE =  369.2732976160243
```

```
[ ] print("Wind RMSE = ",np.sqrt(mean_squared_error(y2_test,y2_pred2)))
```

```
Wind RMSE =  1264.164525340562
```




# Decision Trees

## Input details

- **Features for Solar Energy (X1_train, X1_test)**: This dataset includes variables such as solar profiles and time-related features that influence solar generation.
- **Target Variable for Solar Energy (y1_train, y1_test)**: Represents actual solar generation values used for training and evaluation.
- **Features for Wind Energy (X2_train, X2_test)**: Similar to solar, this dataset consists of wind profiles and other meteorological factors affecting wind generation.
- **Target Variable for Wind Energy (y2_train, y2_test)**: Represents actual wind generation values used for training and evaluation.

## Output details

- **Predictions for Solar Energy (y1_pred, y1_pred2)**: Contains the predicted solar generation values based on the trained Decision Tree model.
- **Predictions for Wind Energy (y2_pred, y2_pred2)**: Contains the predicted wind generation values based on the trained Decision Tree model.

## Constants and hyper parameters

**Random State**:

- For initial training: `random_state=0`
- For hyperparameter tuning: `random_state=1`

The `random_state` parameter ensures consistent results across different runs by controlling the randomness in the decision tree's splits.

## Training code

A **Decision Tree Regressor** is instantiated with a specified random state to ensure reproducibility.
The model is trained using the `.fit()` method for both solar and wind datasets:

```python
regressor.fit(X1_train, y1_train)  # For solar energy
regressor.fit(X2_train, y2_train)  # For wind energy
```

```python
from sklearn.tree import DecisionTreeRegressor

# Create a Decision Tree regressor object with a specified random state for
reproducibility.
# The random state ensures that the results can be replicated in future runs.
regressor = DecisionTreeRegressor(random_state=0)
# Fit the regressor to the training data (X1_train for features and y1_train
for target values).
# This trains the model to learn the relationship between the solar profile and
the actual solar generation.
regressor.fit(X1_train, y1_train)
# Use the trained Decision Tree regressor to make predictions on the test set.
# The model will predict the solar generation based on the solar profile
features in X1_test.
y1_pred = regressor.predict(X1_test)
# Print the actual values from the test set for comparison.
# This allows you to see the true solar generation values that correspond to
the predictions.
y1_test
r2 = r2_score(y1_test, y1_pred)
print("R-squared Accuracy:", r2)
y1_pred
```

```
#from sklearn.model_selection import cross_val_score
#cross_val_score(regressor, X1_train, y1_train, cv=50)
#for wind energy
regressor.fit(X2_train, y2_train)
y2_pred = regressor.predict(X2_test)
# print the predicted price
y2_test
y2_pred
#from sklearn.model_selection import cross_val_score
#cross_val_score(regressor, X2_train, y2_train, cv=50)
from sklearn.metrics import r2_score
print("Accuracy for solar: ",r2_score(y1_pred, y1_test))
print("Accuracy for wind",r2_score(y2_pred, y2_test))
from sklearn.metrics import mean_absolute_error as mae
error = mae(y1_test, y1_pred)
print(error)
error = mae(y2_test, y2_pred)
print(error)
from sklearn.metrics import mean_squared_error
# Calculate the Mean Squared Error (MSE) between the actual values (y1_test)
and the predicted values (y1_pred).
# MSE is a common metric used to evaluate the accuracy of regression models.
MSE = mean_squared_error(y1_test, y1_pred)
# Print the calculated MSE value to assess the model's performance.
print(MSE)
from sklearn.metrics import mean_squared_error
# Calculate the Mean Squared Error (MSE) for the wind generation predictions.
# MSE is used to quantify the difference between actual and predicted values,
helping to evaluate the model's performance.
MSE = mean_squared_error(y2_test, y2_pred)
# Print the calculated MSE value for the wind generation model.
print(MSE)
# Calculate the Root Mean Squared Error (RMSE) for the solar generation
predictions.
# RMSE is a widely used metric to assess the accuracy of a model, indicating
how much error is present in the predictions.
print("RMSE:", np.sqrt(mean_squared_error(y1_test, y1_pred)))
# Calculate the Root Mean Squared Error (RMSE) for the wind generation
predictions.
# RMSE provides insight into the average magnitude of the errors in the
predictions, helping to evaluate model performance.
print("RMSE:", np.sqrt(mean_squared_error(y2_test, y2_pred)))
"""Hyperparameter tuning with random state=1"""
from sklearn.tree import DecisionTreeRegressor
```

```
# Create a Decision Tree regressor object with a fixed random state for
reproducibility
regressor = DecisionTreeRegressor(random_state=1)
# Fit the regressor to the training data (X1_train as features and y1_train as
target variable)
regressor.fit(X1_train, y1_train)
y1_pred2 = regressor.predict(X1_test)

#for wind energy
regressor.fit(X2_train, y2_train)
y2_pred2 = regressor.predict(X2_test)
r2_score(y1_pred2, y1_test)
r2_score(y2_pred2, y2_test)
MSE = mean_squared_error(y1_test, y1_pred2)
print(MSE)
MSE = mean_squared_error(y2_test, y2_pred2)
print(MSE)
#root mean squared error for solar
print("RMSE",np.sqrt(mean_squared_error(y1_test,y1_pred2)))
#root mean squared error for solar
print("RMSE",np.sqrt(mean_squared_error(y2_test,y2_pred2)))
error = mae(y1_test, y1_pred2)
print(error)
error = mae(y2_test, y2_pred2)
print(error)
g = plt.plot(y1_test - y1_pred, marker='o', linestyle='')
# Label the axes for better understanding of the plot
plt.title('Residuals for Solar Generation Predictions', fontsize=20)  # Title
for the plot
plt.xlabel('Index', fontsize=15)  # X-axis label
plt.ylabel('Residuals (Actual - Predicted)', fontsize=15)  # Y-axis label
plt.grid(True)  # Add grid for better readability
plt.show()
```

# Results

```
[ ] from sklearn.metrics import r2_score
```

```
[ ] print("Accuracy for solar: ",r2_score(y1_pred, y1_test))
```

```
Accuracy for solar:  0.9959718509748189
```

```
[ ] print("Accuracy for wind",r2_score(y2_pred, y2_test))
```

```
Accuracy for wind 0.9643262347993693
```

```
[ ] from sklearn.metrics import mean_absolute_error as mae
```

```
[ ] error = mae(y1_test, y1_pred)
    print(error)
```

```
108.8428292328411
```

```
[ ] error = mae(y2_test, y2_pred)
    print(error)
```

```
767.7429793380926
```

```
[ ] from sklearn.metrics import mean_squared_error

    # Calculate the Mean Squared Error (MSE) between the actual values (y1_test) and the predicted values (y1_pred).
    # MSE is a common metric used to evaluate the accuracy of regression models.
    MSE = mean_squared_error(y1_test, y1_pred)

    # Print the calculated MSE value to assess the model's performance.
    print(MSE)
```

```
149260.47611887925
```

```
[ ] from sklearn.metrics import mean_squared_error

    # Calculate the Mean Squared Error (MSE) for the wind generation predictions.
    # MSE is used to quantify the difference between actual and predicted values, helping to evaluate the model's performance.
    MSE = mean_squared_error(y2_test, y2_pred)

    # Print the calculated MSE value for the wind generation model.
    print(MSE)
```

```
1715528.2329922733
```

```
[ ] # Calculate the Root Mean Squared Error (RMSE) for the solar generation predictions.
    # RMSE is a widely used metric to assess the accuracy of a model, indicating how much error is present in the predictions.
    print("RMSE:", np.sqrt(mean_squared_error(y1_test, y1_pred)))
```

```
RMSE: 386.3424337538905
```

```
[ ] # Calculate the Root Mean Squared Error (RMSE) for the wind generation predictions.
    # RMSE provides insight into the average magnitude of the errors in the predictions, helping to evaluate model performance.
    print("RMSE:", np.sqrt(mean_squared_error(y2_test, y2_pred)))
```

```
RMSE: 1309.7817501371262
```

```
[ ] y2_pred2 = regressor.predict(X2_test)
```

```
[ ] r2_score(y1_pred2, y1_test)
```

```
0.9959718509748189
```

```
[ ] r2_score(y2_pred2, y2_test)
```

```
0.9643262347993693
```

```
[ ] MSE = mean_squared_error(y1_test, y1_pred2)
    print(MSE)
```

```
149260.47611887925
```

```
[ ] MSE = mean_squared_error(y2_test, y2_pred2)
    print(MSE)
```

```
1715528.2329922733
```

```
[ ] #root mean squared error for solar
    print("RMSE",np.sqrt(mean_squared_error(y1_test,y1_pred2)))
```

```
RMSE 386.3424337538905
```

```
[ ] #root mean squared error for solar
    print("RMSE",np.sqrt(mean_squared_error(y2_test,y2_pred2)))
```

```
RMSE 1309.7817501371262
```

```
[ ] error = mae(y1_test, y1_pred2)
    print(error)
```

```
108.8428292328411
```

```
[ ] error = mae(y2_test, y2_pred2)
    print(error)
```

```
767.7429793380926
```

# Support Vector Machines (SVM):

## Input Details

- **Solar Energy Data:**
  - **Features for Solar Energy Prediction (df_solar_x):**
    - `DE_load_actual_entsoe_transparency`: Actual energy load.
    - `DE_load_forecast_entsoe_transparency`: Forecasted energy load.
    - `DE_solar_profile`: Solar energy profile, indicating solar intensity over time.
    - `utc_timestamp`: Timestamp, converted to Unix format (seconds since Jan 1, 1970).
  - **Target for Solar Energy Prediction (df_solar_y):**
    - `DE_solar_generation_actual`: Actual solar energy generation.
- **Wind Energy Data:**
  - **Features for Wind Energy Prediction (df_wind_x):**
    - `DE_wind_capacity`: Wind capacity.
    - `DE_wind_profile`: Wind energy profile.
    - `DE_wind_offshore_generation_actual`: Actual offshore wind generation.
    - `DE_wind_offshore_profile`: Offshore wind profile.
    - `DE_wind_onshore_generation_actual`: Actual onshore wind generation.
    - `DE_wind_onshore_profile`: Onshore wind profile.
  - **Target for Wind Energy Prediction (df_wind_y):**
    - `DE_wind_generation_actual`: Total actual wind generation (offshore + onshore).

## Output Details

- **Solar Energy Predictions:**
  - **Predicted Solar Generation (solar_y_pred):** Predicted values for solar generation using SVR and Linear SVR.
- **Wind Energy Predictions:**
  - **Predicted Wind Generation (y_pred):** Predicted values for wind generation using SVR and Linear SVR.

## Constants and Hyperparameters

- **Solar Energy SVR Model:**
  - **Model Type:**
    - Support Vector Regression (SVR) with Radial Basis Function (RBF), Polynomial, Sigmoid, and Linear kernels.
  - **Hyperparameters:**
    - **C (Regularization parameter):**
      - Radial Basis Function SVR: `C=1e3`
      - Polynomial Kernel SVR: `C=1.0`
    - **gamma (Kernel coefficient):**
      - `gamma=0.5` for RBF and Sigmoid kernels.
      - `gamma='scale'` for Polynomial kernel (default scaling based on number of features).
    - **epsilon (Tolerance for error):**
      - `epsilon=0.01` for RBF and Sigmoid kernels.
      - `epsilon=0.1` for Polynomial kernel.
- **Wind Energy SVR Model:**
  - **Model Type:**
    - Linear Support Vector Regression (LinearSVR) and SVR with RBF kernel.
  - **Hyperparameters:**
    - **C (Regularization parameter):**
      - Default for Linear SVR: `C=1.0`
      - For RBF SVR: Default `C=1.0`
    - **loss (Linear SVR loss function):**
      - Epsilon-insensitive loss for Linear SVR.
    - **epsilon (Tolerance for error):**
      - Default for Linear SVR: `epsilon=0.0`
    - **max_iter (Maximum iterations):**
      - Default for Linear SVR: `max_iter=1000`

## Training code

```python
"""### Hyperparameter Tuning

Literature suggests that the epsilon parameter should be set between
\(10^{-3}\) and 1. For the C parameter, a suitable range would be between 1 and
100; setting C too high may lead to overfitting of the training data.
```

```
The gamma parameter is automatically determined by scikit-learn's SVR, so it's
generally advisable to leave it unchanged.

Additionally, it's important to remember that tuning the kernel can
significantly impact model performance and may be one of the most crucial
hyperparameters to adjust.
"""
from sklearn.svm import SVR
# Initialize the SVR regressor with the sigmoid kernel
regressor = SVR(kernel='sigmoid', C=1e3, gamma=0.5, epsilon=0.01)
# Fit the model using the training data
regressor.fit(solar_x_train, solar_y_train)
import numpy as np
import pandas as pd
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
# Scale the input features using StandardScaler
scaler_x = StandardScaler()
solar_x_train = scaler_x.fit_transform(solar_x_train)
# Assuming solar_x_test exists for future predictions, scale it as well
# solar_x_test = scaler_x.transform(solar_x_test)
# If the target variable (solar_y_train) has a wide range, consider scaling it
too:
scaler_y = StandardScaler()
solar_y_train = scaler_y.fit_transform(solar_y_train.values.reshape(-1,
1)).flatten()
# Assuming solar_y_test exists for future predictions, scale it as well
# solar_y_test = scaler_y.transform(solar_y_test.values.reshape(-1,
1)).flatten()
# Initialize the SVR regressor with the polynomial kernel
# You might need to adjust C, gamma, and epsilon
regressor = SVR(kernel='poly', C=1.0, gamma='scale', epsilon=0.1)  # Adjusted
hyperparameters
# Fit the model using the scaled training data
regressor.fit(solar_x_train, solar_y_train)
from sklearn.metrics import r2_score
# Calculate R-squared score for the predictions
r2 = r2_score(solar_y_test, solar_y_pred)
print("R-squared:", r2)
solar_y_pred = regressor.predict(solar_x_test)
solar_y_pred
from sklearn.metrics import r2_score
r2_score(solar_y_pred,solar_y_test)
from sklearn.metrics import r2_score
```

```
r2_score(solar_y_pred,solar_y_test)
"""using linear SVR (ideal for large datasets)"""
from sklearn.svm import LinearSVR
lsvr = LinearSVR(verbose=0, dual=True)
# LinearSVR(C=1.0, dual=True, epsilon=0.0, fit_intercept=True,
#           intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
#           random_state=None, tol=0.0001, verbose=0)

#Then, we'll fit the model on train data and check the model accuracy score.
lsvr.fit(solar_x_train, solar_y_train)
y_pred = lsvr.predict(solar_x_test)
y_pred
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test,
y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
print('R square score:', metrics.r2_score(solar_y_test, y_pred))

"""##wind energy"""
dataframe.dropna()
df_wind_x=dataframe[['DE_wind_capacity','DE_wind_profile','DE_wind_offshore_gen
eration_actual','DE_wind_offshore_profile','DE_wind_onshore_generation_actual',
'DE_wind_onshore_profile']]
df_wind_y=dataframe['DE_wind_generation_actual']
from sklearn.model_selection import train_test_split
wind_x_train, wind_x_test,wind_y_train,wind_y_test =
train_test_split(df_wind_x,df_wind_y,test_size = 0.2, random_state = None)
from sklearn.svm import LinearSVR
lsvr = LinearSVR(verbose=0, dual=True)
lsvr.fit(wind_x_train,wind_y_train)
y_pred = lsvr.predict(wind_x_test)
from sklearn.metrics import r2_score
r2_score(y_pred,wind_y_test)
from sklearn.metrics import mean_squared_error
ypred = lsvr.predict(wind_x_test)
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(wind_y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(wind_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(wind_y_test, y_pred)))
print('R square score:', metrics.r2_score(wind_y_test, y_pred))
```

```
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(wind_x_train,wind_y_train)
```

## Results

```
[ ] from sklearn import metrics
    import numpy as np

    # Calculate and print the Mean Absolute Error (MAE)
    print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test, solar_y_pred))

    # Calculate and print the Mean Squared Error (MSE)
    print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, solar_y_pred))

    # Calculate and print the Root Mean Squared Error (RMSE)
    print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(solar_y_test, solar_y_pred)))

    # Calculate and print the R-squared score
    print('R square score:', metrics.r2_score(solar_y_test, solar_y_pred))
```

```
Mean Absolute Error: 4096.708883004016
Mean Squared Error: 48091737.10266186
Root Mean Squared Error: 6934.8206251251995
R square score: -0.29008732732217246
```

```
[ ] import numpy as np
    import pandas as pd
    from sklearn.svm import SVR
    from sklearn.preprocessing import StandardScaler

    # Scale the input features using StandardScaler
    scaler_x = StandardScaler()
    solar_x_train = scaler_x.fit_transform(solar_x_train)
    # Assuming solar_x_test exists for future predictions, scale it as well
    # solar_x_test = scaler_x.transform(solar_x_test)

    # If the target variable (solar_y_train) has a wide range, consider scaling it too:
    scaler_y = StandardScaler()
    solar_y_train = scaler_y.fit_transform(solar_y_train.values.reshape(-1, 1)).flatten()
    # Assuming solar_y_test exists for future predictions, scale it as well
    # solar_y_test = scaler_y.transform(solar_y_test.values.reshape(-1, 1)).flatten()

    # Initialize the SVR regressor with the polynomial kernel
    # You might need to adjust C, gamma, and epsilon
    regressor = SVR(kernel='poly', C=1.0, gamma='scale', epsilon=0.1)  # Adjusted hyperparameters

    # Fit the model using the scaled training data
    regressor.fit(solar_x_train, solar_y_train)

    from sklearn.metrics import r2_score

    # Calculate R-squared score for the predictions
    r2 = r2_score(solar_y_test, solar_y_pred)
    print("R-squared:", r2)
```

```
R-squared: -0.29008732732217246
```

## wind energy

```
[ ]  dataframe.dropna()
     df_wind_x=dataframe[['DE_wind_capacity','DE_wind_profile','DE_wind_offshore_generation_actual','DE_wi
     df_wind_y=dataframe['DE_wind_generation_actual']
```

Double-click (or enter) to edit

```
[ ]  from sklearn.model_selection import train_test_split
     wind_x_train, wind_x_test,wind_y_train,wind_y_test = train_test_split(df_wind_x,df_wind_y,test_size =
```

```
[ ]  from sklearn.svm import LinearSVR
     lsvr = LinearSVR(verbose=0, dual=True)
     # LinearSVR(C=1.0, dual=True, epsilon=0.0, fit_intercept=True,
     #           intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
     #           random_state=None, tol=0.0001, verbose=0)

     #Then, we'll fit the model on train data and check the model accuracy score.

     lsvr.fit(wind_x_train,wind_y_train)
     y_pred = lsvr.predict(wind_x_test)
```

```
[ ]  from sklearn.metrics import r2_score
     r2_score(y_pred,wind_y_test)
```

```
0.980811223925834
```

```
[ ]  from sklearn.metrics import mean_squared_error
     ypred = lsvr.predict(wind_x_test)
     from sklearn import metrics
     print('Mean Absolute Error:', metrics.mean_absolute_error(wind_y_test, y_pred))
     print('Mean Squared Error:', metrics.mean_squared_error(wind_y_test, y_pred))
     print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(wind_y_test, y_pred)))
     print('R square score:', metrics.r2_score(wind_y_test, y_pred))
```

```
Mean Absolute Error: 190.9771113205276
Mean Squared Error: 888055.2024277778
Root Mean Squared Error: 942.3668088529953
R square score: 0.9815865650612182
```

# Random Forest

## Input Details

- Solar Energy Data:
  - Features for Solar Energy Prediction (df_solar_x):
    - `DE_load_actual_entsoe_transparency`: Actual energy load.
    - `DE_load_forecast_entsoe_transparency`: Forecasted energy load.
    - `DE_solar_profile`: Solar energy profile, indicating solar intensity over time.
  - Target for Solar Energy Prediction (df_solar_y):
    - `DE_solar_generation_actual`: Actual solar energy generation.
- Wind Energy Data:
  - Features for Wind Energy Prediction (df_wind_x):
    - `DE_wind_capacity`: Wind capacity.
    - `DE_wind_profile`: Wind energy profile.
    - `DE_wind_offshore_generation_actual`: Actual offshore wind generation.
    - `DE_wind_offshore_profile`: Offshore wind profile.
    - `DE_wind_onshore_generation_actual`: Actual onshore wind generation.
    - `DE_wind_onshore_profile`: Onshore wind profile.
  - Target for Wind Energy Prediction (df_wind_y):
    - `DE_wind_generation_actual`: Total actual wind generation (offshore + onshore).

## Output Details

- Solar Energy Predictions:
  - **Predicted Solar Generation (y_pred):** Predicted values for solar generation using the Random Forest Regressor.
- Wind Energy Predictions:
  - **Predicted Wind Generation (y_pred):** Predicted values for wind generation using the Random Forest Regressor.

## Constants and Hyperparameters

**Random Forest Regressor Parameters:**

- Model Type:
  - Random Forest Regressor (`RandomForestRegressor`) is used for both solar and wind energy predictions.
- Hyperparameters:
  - n_estimators (Number of trees):
    - Default value (`n_estimators=100`) used initially, and set to 100 during hyperparameter tuning.
  - random_state (Seed for random number generator):
    - Initially set to `None` (default behavior).
    - Tuned to `random_state=1` during hyperparameter tuning for wind energy.
  - Other Hyperparameters:
    - No other hyperparameters are explicitly set, so the model uses default values for parameters like `max_depth`, `min_samples_split`, and `min_samples_leaf`.

## Feature Importance Analysis

- Solar Energy:
  - After fitting the Random Forest Regressor to the solar energy data, feature importance is calculated and plotted.
  - **Most Important Feature:** The `DE_solar_profile` is identified as the most important feature in predicting solar energy generation.
- Wind Energy:
  - Similarly, feature importance is calculated for wind energy.
  - Most Important Features:
    - DE_wind_onshore_generation_actual
    - DE_wind_profile

## Evaluation Metrics

For both solar and wind energy predictions, the following evaluation metrics are computed:

- **Mean Absolute Error (MAE):** Measures the average magnitude of errors in predictions.
- **Mean Squared Error (MSE):** Measures the average squared difference between actual and predicted values.
- **Root Mean Squared Error (RMSE):** The square root of MSE, giving error in the same units as the predicted values.

- **R-squared Score ($R^2$):** Indicates how well the predicted values match the actual values (with 1 being a perfect match).

## Hyperparameter Tuning

Wind Energy:

- The hyperparameters for the Random Forest Regressor are tuned by adjusting:
  - **n_estimators**: Set to 100 (number of trees).
  - **random_state**: Set to 1 (for reproducibility).
- After tuning, the model is re-evaluated using the same metrics (MAE, MSE, RMSE, $R^2$).

## Training code

```python
"""#RANDOM FOREST
parameters in Random forest- n_estimators and random state
##solar energy
"""
dataframe.dropna()
df_solar_x=dataframe[['DE_load_actual_entsoe_transparency','DE_load_forecast_en
tsoe_transparency','DE_solar_profile']]
df_solar_y=dataframe['DE_solar_generation_actual']
# df_solar_x['DE_solar_profile'].fillna(0,inplace=True)
# df_solar_y.fillna(0,inplace=True)
df_solar_x.info()
df_solar_x.isna().sum()
from sklearn.model_selection import train_test_split
solar_x_train, solar_x_test,solar_y_train,solar_y_test =
train_test_split(df_solar_x,df_solar_y,test_size = 0.2, random_state = None)
from sklearn.ensemble import RandomForestRegressor
from matplotlib import pyplot
data = solar_x_train
target = solar_y_train
rfr = RandomForestRegressor() #default parameters
rfr.fit(solar_x_train,solar_y_train)
"""feature importance"""
# get importance
importance = rfr.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
	print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

```
"""most important feature for random forest is solar_profile"""
y_pred = rfr.predict(solar_x_test)
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test,
y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
print('R square score:', metrics.r2_score(solar_y_test, y_pred))

"""##wind energy"""
df_wind_x=dataframe[['DE_wind_capacity','DE_wind_profile','DE_wind_offshore_gen
eration_actual','DE_wind_offshore_profile','DE_wind_onshore_generation_actual',
'DE_wind_onshore_profile']]
df_wind_y=dataframe['DE_wind_generation_actual']
#df_wind_x.info()   #no null values
#df_wind_y.isna()
from sklearn.model_selection import train_test_split
wind_x_train, wind_x_test,wind_y_train,wind_y_test =
train_test_split(df_wind_x,df_wind_y,test_size = 0.2, random_state = None)
from sklearn.ensemble import RandomForestRegressor
from matplotlib import pyplot
data = wind_x_train
target = wind_y_train
rfr = RandomForestRegressor() #default parameters
rfr.fit(wind_x_train,wind_y_train)
"""feature importance"""
# get importance
importance = rfr.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
      print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance,color='green')
pyplot.show()
"""wind onshore generation actual and wind_profile are the most important
features"""
y_pred = rfr.predict(wind_x_test)
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test,
y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
```

```
print('R square score:', metrics.r2_score(solar_y_test, y_pred))
"""poor r sqaure score--> making use of feature importance to remove
unimportant features"""
df_wind_x2=dataframe[['DE_wind_profile']]
from sklearn.model_selection import train_test_split
wind_x_train, wind_x_test,wind_y_train,wind_y_test =
train_test_split(df_wind_x2,df_wind_y,test_size = 0.2, random_state = None)
from sklearn.ensemble import RandomForestRegressor
from matplotlib import pyplot
data = wind_x_train
target = wind_y_train
rfr = RandomForestRegressor() #default parameters
rfr.fit(wind_x_train,wind_y_train)
y_pred = rfr.predict(wind_x_test)
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test,
y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
print('R square score:', metrics.r2_score(solar_y_test, y_pred))
"""to improve score--> Hyper parameter tuning"""

#wind energy
regressor = RandomForestRegressor(n_estimators = 100, random_state = 1)
regressor.fit(X2_train, y2_train)
y2_pred = regressor.predict(X2_test)
r2_wind = r2_score(y2_test,y2_pred)
print("Wind R2 = ",r2_wind)
print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test,
y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
```

## Results

most important feature for random forest is solar_profile

```
[ ]  y_pred = rfr.predict(solar_x_test)

     from sklearn import metrics

     print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test, y_pred))
     print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
     print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
     print('R square score:', metrics.r2_score(solar_y_test, y_pred))
```

```
Mean Absolute Error: 97.54396129766647
Mean Squared Error: 221263.05230540692
Root Mean Squared Error: 470.38606729516016
R square score: 0.9939945715425855
```

```
[ ]  y_pred = rfr.predict(wind_x_test)

     from sklearn import metrics

     print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test, y_pred))
     print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
     print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
     print('R square score:', metrics.r2_score(solar_y_test, y_pred))
```

```
Mean Absolute Error: 8146.862908527376
Mean Squared Error: 109252463.16433655
Root Mean Squared Error: 10452.390308648857
R square score: -1.9652842826379908
```

to improve score--> Hyper parameter tuning

```
[ ]  #wind energy
     regressor = RandomForestRegressor(n_estimators = 100, random_state = 1)
     regressor.fit(X2_train, y2_train)
     y2_pred = regressor.predict(X2_test)

     r2_wind = r2_score(y2_test,y2_pred)
     print("Wind R2 = ",r2_wind)


     print('Mean Absolute Error:', metrics.mean_absolute_error(solar_y_test, y_pred))
     print('Mean Squared Error:', metrics.mean_squared_error(solar_y_test, y_pred))
     print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(solar_y_test, y_pred)))
```

```
Wind R2 =  0.9706850296665197
Mean Absolute Error: 8146.862908527376
Mean Squared Error: 109252463.16433655
Root Mean Squared Error: 10452.390308648857
```

# ARIMA (AutoRegressive Integrated Moving Average):

## Solar Energy ARIMA Model

### Input Details:

- **Time Series Data**:
  - Variable: `solar_power` (daily solar generation in MW).
  - Data Source: Provided `modified['DE_solar_generation_actual']` column.
  - Time Period: Data with a daily frequency.
  - Stationarity Check: Augmented Dickey-Fuller (ADF) test to ensure stationarity.
- **Preprocessing**:
  - **Differencing**: First-order differencing applied to make the time series stationary (`solar_power.diff()`).
  - **Log Transformation**: Logarithmic transformation (`np.log(solar_power)`) to stabilize the variance.

### Output Details:

- **Predicted Values**:
  - **Daily Solar Power Generation (in MW)** over the test period (last 15% of the data).
  - A forecast for a set of future days (test set), given historical data (training set).
- **Error Metrics**:
  - **Root Mean Squared Error (RMSE)**: Error metric for forecast accuracy, comparing predicted and actual solar power values.
  - **Mean Absolute Error (MAE)**: Another accuracy metric reflecting the average error magnitude.
- **Graphical Output**:
  - **Prediction vs Actual**: Graphs showing the comparison between predicted values and actual test data over a defined period.

### Constants & Hyperparameters:

- **Order of ARIMA (p, d, q)**:
  - **p (AR term)**: 6 (based on PACF plot).
  - **d (Differencing)**: 1 (based on stationarity tests and differencing analysis).
  - **q (MA term)**: 6 (based on ACF plot).
- **Training-Test Split**:
  - **Training Set**: First 85% of the data (`train_s`).
  - **Test Set**: Last 15% of the data (`test_s`).

- **Model Parameters**:
    - **Trend**: Included a time-based trend component (`trend="t"`).
    - **SARIMAX Fit Iterations**: Maximum 2000 iterations (`maxiter=2000`).

## Wind Energy ARIMA Model

## Input Details:

- **Time Series Data**:
    - Variable: `wind_power` (daily wind generation in MW).
    - Data Source: Provided `modified['DE_wind_generation_actual']` column.
    - Time Period: Data with a daily frequency.
    - Stationarity Check: ADF test to ensure stationarity.
- **Preprocessing**:
    - **Differencing**: First-order differencing (`wind_power.diff()`) to remove non-stationarity.

## Output Details:

- **Predicted Values**:
    - **Daily Wind Power Generation (in MW)** over the test period (last 15% of the data).
    - Forecasting daily wind power for a set of future days, based on the historical pattern in the training set.
- **Error Metrics**:
    - **Root Mean Squared Error (RMSE)**: Error metric for the difference between actual and predicted values.
    - **Mean Absolute Error (MAE)**: Measures the average absolute deviation between forecasted and actual values.
- **Graphical Output**:
    - **Prediction vs Actual**: Plot comparing predicted wind generation values with actual test data.

## Constants & Hyperparameters:

- **Order of ARIMA (p, d, q)**:
    - **p (AR term)**: 3 (from PACF plot).
    - **d (Differencing)**: 0 (after stationarity check, no differencing needed).
    - **q (MA term)**: 5 (based on ACF plot).
- **Seasonal Order of SARIMAX (P, D, Q, m)**:
    - **P, Q (Seasonal AR and MA terms)**: 3 and 5, respectively.

    - **D (Seasonal Differencing)**: 0.
    - **m (Seasonality period)**: 12 (assumed monthly seasonality).
- **Training-Test Split**:
    - **Training Set**: First ~85% of the data (`train_w`).
    - **Test Set**: Last 15% of the data (`test_w`).
- **Model Parameters**:
    - **Trend**: Time trend component included (`trend="t"`).
    - **SARIMAX Fit Iterations**: Maximum 2000 iterations (`maxiter=2000`).

## Training code

Python

```
"""#ARIMA MODEL
##For solar energy
"""
# Create a series for the solar generation
solar_power = modified['DE_solar_generation_actual']
solar_power
solar_power.describe()
# Check the decomposition of the solar power data
from statsmodels.api import tsa
decomposition = tsa.seasonal_decompose(solar_power, period=365,
model='additive')
# Create a new figure for the plot
plt.figure()
# Plot a histogram of the solar generation data with 40 bins
plt.hist(solar_power, bins=40,color='pink')
# Add a title to the plot
plt.title('Solar Generation Distribution', fontsize=20)
# Label the y-axis as 'Frequency'
plt.ylabel('Frequency')
# Label the x-axis as 'Daily Solar Generation in MW'
plt.xlabel('Daily Solar Generation in MW')
# Add a vertical line representing the mean value in red
plt.axvline(solar_power.mean(), c='red', label ='mean')
# Add a vertical line representing the median value in orange
plt.axvline(solar_power.median(), c='orange', label ='median')
# Add a legend to differentiate between the mean and median lines
plt.legend()
# Display the plot
plt.show()
import matplotlib.dates as mdates
# Trend
```

```
plt.figure(figsize=(12,8))
plt.plot(decomposition.trend, marker='.',color='brown')
plt.title('Trend Component')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y %b"))
plt.show();

"""#stationery check"""

s = adfuller(solar_power, regression='ct')
print('p-value:{}'.format(s[1]))
# Lets check the differencce
s = adfuller(solar_power.diff().dropna(), regression='c')
print('p-value:{}'.format(s[1]))
"""Series with difference is stationary because fo the p-value less than 0.05
#Finding the order of differencing for the ARIMA model
"""
# check the time series
solar_power
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
# Create 3 subplots vertically
plt.subplots(3, 1)
# Plot the original series in the first subplot
plt.subplot(3,1,1)
plt.plot(solar_power, c='#2D82B7')  # Vibrant blue for original series
plt.title('Original Series')
# Customize the x-axis for better date formatting
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
# Plot the 1st order differencing in the second subplot
plt.subplot(3,1,2)
plt.plot(solar_power.diff().dropna(), c='#FF6F61')  # Vibrant coral for 1st
difference
plt.title('1st Order Differencing')
# Customize the x-axis for better date formatting
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
# Plot the 2nd order differencing in the third subplot
plt.subplot(3,1,3)
```

```
plt.plot(solar_power.diff().diff().dropna(), c='#6A0572')  # Vibrant purple for
2nd difference
plt.title('2nd Order Differencing')
# Customize the x-axis for better date formatting
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
# Adjust layout for better spacing between plots
plt.tight_layout()
# Display the plot
plt.show()
from statsmodels.graphics.tsaplots import plot_acf
# Create subplots to visualize autocorrelation
fig, ax = plt.subplots(3, 1, figsize=(20, 10))
# Original time series autocorrelation - using vibrant blue
ax[0].set_title('ACF of Original Series', fontsize=15, color='blue')
ax[0].tick_params(axis='x', colors='blue')
ax[0].tick_params(axis='y', colors='blue')
ax[0].spines['bottom'].set_color('blue')
ax[0].spines['left'].set_color('blue')
fig = plot_acf(solar_power, ax=ax[0], color='dodgerblue')
# 1st difference autocorrelation - using vibrant green
ax[1].set_title('ACF of 1st Order Differencing', fontsize=15, color='green')
ax[1].tick_params(axis='x', colors='green')
ax[1].tick_params(axis='y', colors='green')
ax[1].spines['bottom'].set_color('green')
ax[1].spines['left'].set_color('green')
fig = plot_acf(solar_power.diff().dropna(), ax=ax[1], color='limegreen')
# 2nd difference autocorrelation - using vibrant red
ax[2].set_title('ACF of 2nd Order Differencing', fontsize=15, color='red')
ax[2].tick_params(axis='x', colors='red')
ax[2].tick_params(axis='y', colors='red')
ax[2].spines['bottom'].set_color('red')
ax[2].spines['left'].set_color('red')
fig = plot_acf(solar_power.diff().diff().dropna(), ax=ax[2], color='orangered')
# Adjust the layout to prevent overlap
plt.tight_layout()
# Display the plot
plt.show()
"""From the above plots, it's evident that the time series becomes stationary
after applying first-order differencing. The second-order differencing declines
too rapidly, indicating possible over-differencing. Therefore, I'll use \( d =
1 \) in the ARIMA model. The first-differenced graph clearly shows that the
series becomes much more stationary at this level."""
```

```
# Applying log transformation with differencing and plotting in orange
np.log(solar_power).diff().dropna().plot(c='orange')
plt.title('Log Transformation with 1st Differencing', fontsize=15)
plt.ylabel('Log Difference')
plt.xlabel('Date')
plt.show()
solar_log= np.log(solar_power).diff().dropna()
"""#Finding order of the ar term
For the ARIMA model we need to first find the the order (p,d,q). To find the
first p order, it is good to look at the Partial auto correlation plot
"""
# Plot the PACF with custom x-axis scale
from statsmodels.graphics.tsaplots import plot_pacf
plt.rcParams.update({'figure.figsize': (9, 3), 'figure.dpi': 120})
# Create the PACF plot
plot_pacf(solar_power, lags=10)
# Set the x-axis and y-axis labels
plt.xlabel('Lag')
plt.ylabel('Partial Autocorrelation')
# Adjusting the x-axis scale to increment by 10 units (example)
plt.xticks(ticks=np.arange(0, 11, 2))  # Modify this based on your data
# Show the plot
plt.show()
# lets check the differened one as well
plt.rcParams.update({'figure.figsize':(9,3),'figure.dpi':120})
plot_pacf(solar_power.diff().dropna(), lags=10,color='red')
plt.xlabel('Lag')
plt.ylabel('Partial Autocorrelation')
plt.show()
# to find the q value, it is helpful to look at the autocorrelation plot
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(solar_power, lags=10,color='green')
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.show()
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(solar_power.diff().dropna(), lags=10,color='brown')
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.show()
"""It looks likes 3 is the best answer for q=3.
#Train and test split
"""
```

```
test_percent = .15
test_number = int(solar_power.shape[0]*test_percent)
train_s = solar_power[:-test_number]
test_s = solar_power[-test_number:]
train_s.shape
test_s.shape
from sklearn.metrics import mean_absolute_error
from statsmodels.tsa.statespace.sarimax import SARIMAX
model_s = SARIMAX(train_s, order=(6,1,6),
                enforce_stationarity=False,
                enforce_invertibility=False,
                trend="t") # order of _ for AR, 1 differentiation, and _ for MA
model_fit_s = model_s.fit(maxiter=2000)
print('Coefficients: %s' % model_fit_s.params)
from sklearn.metrics import mean_squared_error
from math import sqrt
predictions = model_fit_s.predict(start=len(train_s),
end=len(train_s)+len(test_s)-1, typ="levels", index= test_s.index)
print("RMSE: ", sqrt(mean_squared_error(test_s[:14], predictions[:14])))
plt.figure(figsize=(15,10))
plt.plot(test_s[:14], c="green", label="test", marker='o')
plt.plot(predictions[:14], c="pink", label="predict", marker='o')
plt.legend()
plt.title('ARIMA SOLAR', fontsize=(20))
plt.show();
from sklearn.metrics import mean_absolute_error
print("MAE: ", mean_absolute_error(test_s[:14], predictions[:14]))
# make prediction
predictions = model_fit_s.predict(start=len(train_s),
end=len(train_s)+len(test_s)-1, typ="levels", index=test_s.index)
fig = plt.figure()
plt.plot(test_s, c="green", label="test", marker='.')
plt.plot(predictions, c="brown", label="predict", marker='.')
plt.legend()
fig.autofmt_xdate()
plt.show();
# Run a loop throgh the model to find the best parameters
# best_MAE = float('inf')

# for p in range(1,7):
#     for d in range(0,2):
#         for q in range(1,7):

#             print(f"Trying values of {(p,d,q)}",end = "\r")
```

```
#             model = SARIMAX(train_s,order = (p,d,q),
#                             enforce_stationary = False,
#                             enforce_invertibility = False,
#                             trend = "t")

#             model_fit = model.fit(maxiter = 2000)

#             predictions = model_fit.predict(start=len(train_s),end =
len(train_s) + len(test_s) - 1,typ="levels", index=test_s.index)

#             current_MAE = mean_absolute_error(test_s[:10],predictions[:10])

#             if (best_MAE > current_MAE):
#                 best_MAE = current_MAE
#                 print(f"Found new best MAE of {best_MAE} with values
{(p,d,q)}")

"""#For wind energy"""
# create a wind_power series
wind_power = modified['DE_wind_generation_actual']
plt.figure()
plt.hist(wind_power, bins=50, color='green')  # Use vibrant color for bars
plt.title('Wind Generation Distribution', fontsize=20)  # fontsize only
plt.ylabel('Frequency', fontsize=15)
plt.xlabel('Daily Wind Generation in MW', fontsize=15)  # Adjusted label to
'Wind'
# Adding vertical lines for mean and median
plt.axvline(wind_power.mean(), c='red', label='mean', linewidth=2)  # Mean in
red
plt.axvline(wind_power.median(), c='orange', label='median', linewidth=2)  #
Median in orange
plt.legend()
plt.show()
plt.figure(figsize=(15,5))
plt.plot(wind_power, c='brown')
plt.title('Wind Generation in MW', fontsize=20)
plt.ylabel('MW')
plt.xlabel('date')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y %b"))
plt.show()
```

```
# Check the decomposition of the wind power data
from statsmodels.api import tsa
decomposition = tsa.seasonal_decompose(wind_power, period=365,
model='additive')
import matplotlib.dates as mdates
# Trend
plt.figure(figsize=(12,8))
plt.plot(decomposition.trend, marker='.',color='green')
plt.title('Trend Component')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y %b"))
plt.show();

"""#Stationery Check"""

s = adfuller(wind_power, regression='ct')
print('p-value:{}'.format(s[1]))
# Lets check the differencce
s = adfuller(wind_power.diff().dropna(), regression='c')
print('p-value:{}'.format(s[1]))
"""The series is stationery"""
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
plt.subplots(3, 1)
# orginal series
plt.subplot(3,1,1)
plt.plot(wind_power,c='red')
plt.title('Original Series')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
plt.subplot(3,1,2)
plt.plot(wind_power.diff().dropna(), c='yellow')
plt.title('1st Order Differencing')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
plt.subplot(3,1,3)
plt.plot(wind_power.diff().diff().dropna(), c='green')
plt.title('2st Order Differencing')
ax = plt.gca()
ax.autoscale(enable=True, axis='x', tight=True)
ax.xaxis.set_major_formatter(DateFormatter("%Y "))
```

```
plt.tight_layout()
plt.show()
from statsmodels.graphics.tsaplots import plot_acf
import matplotlib.pyplot as plt
# Create subplots for the autocorrelation
fig, ax = plt.subplots(3, 1, figsize=(20, 10))
# Original series autocorrelation
fig = plot_acf(wind_power, ax=ax[0], color='blue')  # Original series in blue
ax[0].set_title('Autocorrelation of Original Series', fontsize=16)
# 1st Order Differencing autocorrelation
fig = plot_acf(wind_power.diff().dropna(), ax=ax[1], color='orange')  # 1st
differencing in orange
ax[1].set_title('Autocorrelation of 1st Order Differencing', fontsize=16)
# 2nd Order Differencing autocorrelation
fig = plot_acf(wind_power.diff().diff().dropna(), ax=ax[2], color='green')  #
2nd differencing in green
ax[2].set_title('Autocorrelation of 2nd Order Differencing', fontsize=16)
# Display the plots
plt.tight_layout()
plt.show()
"""#Finding order of the AR term"""
# Plot the PACF
from statsmodels.graphics.tsaplots import plot_pacf
plt.rcParams.update({'figure.figsize':(9,3),'figure.dpi':120})
plot_pacf(wind_power, lags=10,color='pink')
plt.xlabel('Lag')
plt.ylabel('Partial Autocorrelation')
plt.show()
"""p=2 looks like a good starting point based on the graph
#train and test split
"""
wind_power.shape
test_number = 274
train_w = wind_power[:-test_number]
test_w = wind_power[-test_number:]
from statsmodels.tsa.statespace.
 import SARIMAX
model_s = SARIMAX(train_w, order=(3,0,5), seasonal_order=(3,0,5,12),
                enforce_stationarity=False,
                enforce_invertibility=False,
                trend="t") # order of _ for AR, 1 differentiation, and _ for MA
model_fit_s = model_s.fit(maxiter=2000)
print('Coefficients: %s' % model_fit_s.params)
```

## Results

```
[ ]  from sklearn.metrics import mean_squared_error
     from math import sqrt

     predictions = model_fit_s.predict(start=len(train_s), end=len(train_s)+len(test_s)-1, typ="levels", index= test_s.index)

     print("RMSE: ", sqrt(mean_squared_error(test_s[:14], predictions[:14])))
     plt.figure(figsize=(15,10))
     plt.plot(test_s[:14], c="green", label="test", marker='o')
     plt.plot(predictions[:14], c="pink", label="predict", marker='o')
     plt.legend()
     plt.title('ARIMA SOLAR', fontsize=(20))
     plt.show();
```

```
RMSE:  2552.7309853682978
```

```
[ ]  from sklearn.metrics import mean_absolute_error
     print("MAE: ", mean_absolute_error(test_s[:14], predictions[:14]))
```

```
MAE:  2150.4511758887897
```

```
[ ]  from statsmodels.tsa.statespace.sarimax import SARIMAX

     model_s = SARIMAX(train_w, order=(3,0,5), seasonal_order=(3,0,5,12),
                       enforce_stationarity=False,
                       enforce_invertibility=False,
                       trend="t") # order of _ for AR, 1 differentiation, and _ for MA
     model_fit_s = model_s.fit(maxiter=2000)

     print('Coefficients: %s' % model_fit_s.params)
```

```
Coefficients: drift       8.503892e-01
ar.L1        2.192408e+00
ar.L2       -2.030704e+00
ar.L3        7.831464e-01
ma.L1       -1.500814e+00
ma.L2        9.006978e-01
ma.L3        1.069794e-02
ma.L4        6.187335e-02
ma.L5       -2.291766e-01
ar.S.L12     9.833440e-02
ar.S.L24    -2.373247e-01
ar.S.L36     6.924787e-01
ma.S.L12    -1.136719e-02
ma.S.L24     2.747936e-01
ma.S.L36    -6.861415e-01
ma.S.L48    -3.735713e-02
ma.S.L60     6.411889e-02
sigma2       2.024741e+07
dtype: float64
```

# Results for Different Models and Conditions

1. **Random Forest**:
    - **Performance**: The Random Forest model achieved the best results with a **Mean Absolute Error (MAE)** of 97.54, **Root Mean Squared Error (RMSE)** of 470.39, and an **$R^2$ score** of 0.99. These indicate a high level of accuracy for this model, making it excellent for predicting solar and wind energy outputs with minimal error.
    - **Condition**: This model works well in scenarios where non-linear relationships are prevalent and large datasets are available. It also handles multicollinearity better than linear models, which is a key strength for complex data patterns.
2. **Decision Trees**:
    - **Performance**: The Decision Tree model also performed very well with an **$R^2$ score** of 0.995, closely competing with Random Forest. The slightly lower MAE and RMSE compared to Random Forest indicate it could be a good alternative but may be more prone to overfitting.
    - **Condition**: Decision Trees are better suited for simpler datasets where interpretability is important. They are fast to train but may not generalize as well without ensemble methods like Random Forest.
3. **Support Vector Machines (SVM)**:
    - **Performance**: SVM achieved an **MAE** of 190.97 and **RMSE** of 942.36, with an **$R^2$ score** of 0.98. While it produced good accuracy, it wasn't as precise as the Random Forest and Decision Tree models.
    - **Condition**: SVM models are more suited for smaller datasets and work well in high-dimensional spaces. However, they took longer to train, indicating they may not be the best choice for large datasets or real-time predictions.
4. **ARIMA**:
    - **Performance**: ARIMA had the highest error values, with an **MAE** of 2150.45 and **RMSE** of 2552.73. This suggests that ARIMA struggled to predict accurately in this dataset, likely due to the complex non-linear relationships in the data that it couldn't capture.
    - **Condition**: ARIMA models are best for datasets with strong linear trends and seasonal patterns. It is less suited for the type of energy data used here, as they include complex patterns that aren't captured well by linear models. Additionally, ARIMA took longer to train, especially with higher parameters, making it a suboptimal choice for this case.
5. **Ridge Regression**:
    - **Performance**: Ridge Regression had a very good **$R^2$ score** for solar energy at 0.996 and for wind energy at 0.967. While it performed well, it was still outshined by Random Forest and Decision Trees in terms of MAE and RMSE.
    - **Condition**: Ridge Regression is useful for datasets with multicollinearity, providing good generalization. It is fast to train and interpretable, but might not capture the complexity of non-linear relationships as well as Random Forest or Decision Trees.

# Conclusion

- **Best Model**: Random Forest stood out as the most accurate and balanced model across all metrics, providing the lowest error rates and highest $R^2$ scores. It is well-suited for the complex, non-linear nature of solar and wind energy data.
- **Time and Complexity**: While ARIMA and SVM took longer to train, their performance did not justify the additional computational effort. For quick and accurate results, Random Forest and Decision Trees are better choices.
- **Special Cases**: Ridge Regression performed quite well for solar energy predictions and might be favored in cases where interpretability and training speed are priorities. However, for wind energy, Random Forest provided more accurate results.