Strategic policy analysis and forecasting of Renewable Energy in India using Machine Learning

Submitted To

Dr. M. K. Deshmukh

BITS Pilani

Submitted By :

Name	BITS ID
Shreyas Ashok Meher	2023AAPS0646G
Shruti Soumya Rout	2022AAPS1144G
Harsh Kumar Dixit	2021AAPS2935G

Birla Institute of Technology and Science, KK Birla Goa Campus, India

Data Description

The dataset used for this project contains monthly renewable energy generation data from the state of Gujarat, India, spanning from early 2015 to recent months. It includes generation figures from a variety of sources including wind, solar, biomass, bagasse, small and large hydroelectric plants, and other renewable technologies such as waste-to-energy.

The dataset is structured as a time-series, with each row corresponding to a month. It offers a valuable basis for building forecasting models using machine learning, with the goal of predicting future renewable energy output.

Preliminary cleaning was performed to address missing and inconsistent values, and also to correct formatting issues. The dataset also exhibited seasonal and yearly trends, making it suitable for time-series forecasting using machine learning techniques. Features such as wind and solar power demonstrated high variability and dominance in overall generation, highlighting their importance in Gujarat's renewable energy landscape.

For this project, we have only predicted energy outputs for wind energy and solar energy.

Dataset Overview

- The dataset spans a total of 139 rows and 9 columns.
- Out of these, about 115 rows have non-null energy values.

Feature name	Description	Data type
Month, Year	Monthly time period	Object/String
Wind Power	Energy generated from wind sources	Float
Solar	Energy generated from solar sources	Float
Biomass	Energy generated from biomass	Float
Bagasse	Energy from sugarcane byproduct	Float
Small Hydro	Energy from small hydroelectric projects	Float
Large Hydro	Large hydroelectric energy	Float
Other(Waste to energy etc.)	Miscellaneous renewable energy	Float
Total generation	Total electricity generation	Float

- The dataset follows a monthly time-series format, useful for temporal modeling.
- Starting period is from 2015, with consistent data from around April 2015 onwards.
- Missing values are present in the first few months.
- Large Hydro column contains numerous zero values in the initial months.
- Rest of the energy sources have comparatively normal values.

Missing values

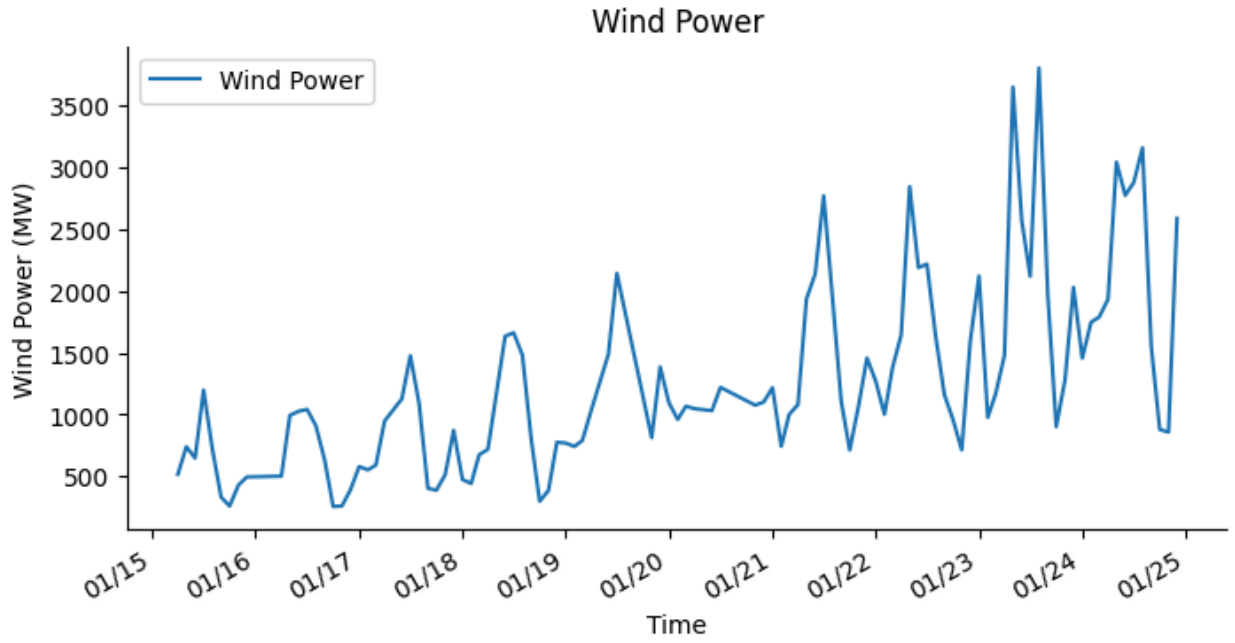
Feature	Missing values
Wind Power	15
Solar	15
Biomass	15
Bagasse	15
Small Hydro	15
Large Hydro	15
Other (Waste to energy etc)	15
Total generation	15

Data Distribution

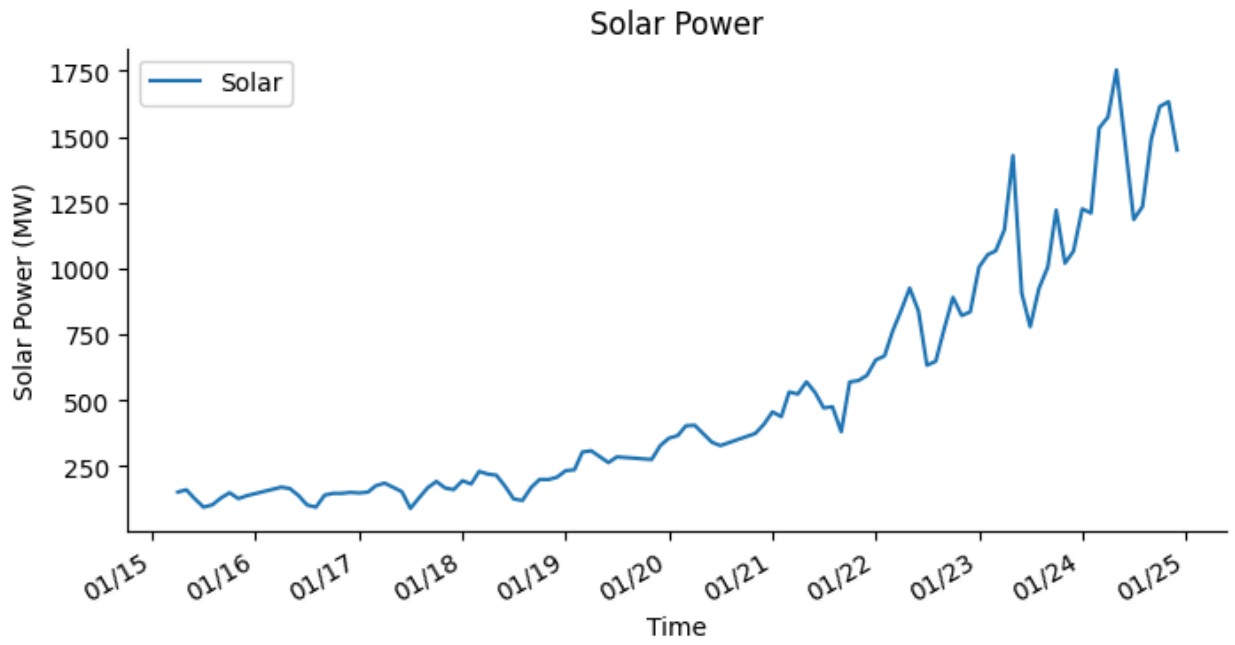
Feature	Mean	Min	Max	Std dev
Wind Power	2305.60	255.02	24644.66	4026.85
Solar	992.22	90.24	17374.38	2194.85
Biomass	5.14	0.00	68.97	11.70
Bagasse	1.14	0.00	20.15	3.07
Small Hydro	20.92	0.00	213.46	38.66
Large Hydro	187.09	0.00	5979.86	733.95
Other	0.55	0.00	16.08	2.27
Total Generation	3511.03	407.56	48245.43	6739.34

Month, Year varies from Jan, 2015 till Dec, 2024.

Wind Power



Solar Power



Where 1 corresponds to Jan 2015, 2 corresponds to Feb 2015 and so on.

Preprocessing and cleaning

For the purpose of predicting solar and wind energy outputs, separate had to be created for both the energy sources from the above dataset. The dataset for both was created by choosing the Month, Year column and Wind Power column(for wind energy prediction) and Solar column(for solar energy production) from the dataset.

Cleaning of the 2 datasets was done by removing the rows where solar or wind power value was either 0 or not a number.

Month, Year feature was modified by splitting the feature column into two separate columns, namely Month and Year. Months were assigned numeric values, with January being assigned value 1, February assigned 2 and so on, till December begin assigned 12. Year column remain unchanged.

In order to capture the effect of monthly and temporal variation on the energy generated **cyclical feature encoding** was used. In this 2 new features were included in the dataset, **month_sin** and **month_cos.** The formula for both is given below:

- **month_sin = sin(2π × Month / 12)**
- **month_cos = cos(2π × Month / 12)**

Since months in a year are cyclical — January and December are temporally close, yet numerically distant (1 vs 12). If we directly use month numbers in a machine learning model, the model may interpret December and January as being far apart, which is misleading.

To preserve this **cyclical relationship**, we map the months onto a **unit circle** using sine and cosine transformations. This way:

- Months like December and January will have similar sine and cosine values.
- The model can learn seasonal patterns more effectively, such as higher solar energy in summer months or more wind generation during monsoons.

To normalize the year values, the following transformation was applied:

- **Year_scaled = Year − min(Year)**

Where:

- **Year** is the numeric year extracted from the date column (e.g., 2015, 2016, ...).
- **`min(Year)`** is the earliest year present in the dataset, which in this case is 2015.

Machine learning models often perform better when input features are scaled or normalized, especially when features have large numerical ranges. This helps the model learn long-term trends in energy generation across years without being influenced by the absolute year values.

To analyze the impact of policies on energy generated, various policy features were also added in the dataset. Namely, these policies are -

1. **Wind Power**
    - Wind Power Policy - 2002
    - GERC Regulation on Wind Power (August 11, 2006)
    - Wind Power Policy - 2007
    - Wind Power Policy - 2009 (Gujarat Energy Development gency, GEDA)
    - Wind Power Policy (2013)
    - Gujarat Wind Power Policy - 2016
    - GERC - Wind Tariff Orders
2. **Solar Power**
    - Gujarat Solar Power Policy 2015
    - Gujarat Solar Power Policy 2021
    - Gujarat Wind-Solar Hybrid Power Policy 2018
    - Gujarat Rooftop Solar Subsidy Scheme (Surya Scheme)

Furthermore, the year in which policies were introduced changes in month numbering were made. July was assigned the value 1 and rest of the months were made 0.

Feature selection

- For Wind power prediction, features selected for training were **Year_scaled, month_sin, month_cos, Wind_policy_2002, GERC_policy_2006, Wind_policy_2007, Wind_policy_2009, Wind_policy_2013, Wind_policy_2016 and GERC_wind_tariffs_2020.**
- For Solar power prediction, features selected for training were **Year_scaled, month_sin, month_cos, solar_policy_2015, solar_policy_2021, wind_solar_hybrid_policy_2018 and rooftop_solar_scheme_2019.**

Target variable for both the sources was taken as wind energy generation and solar energy generation respectively.

The dataset for both was then split into training and testing data with 20% of the data being used for testing and the remaining 80% being used for training.

The primary reason for splitting the dataset is to **evaluate the model's ability to generalize to unseen data**. When building a machine learning model, we want it to not just memorize the training data but also perform well on new, real-world data that it hasn't encountered during training.

The 80/20 split is a common and often effective choice for several reasons:

1. **Sufficient Training Data:** 80% of the data provides a substantial amount of information for the model to learn from, allowing it to develop a good understanding of the underlying patterns.
2. **Reliable Evaluation:** 20% of the data is usually enough to provide a reliable estimate of the model's performance on unseen data. It gives a reasonable sense of how well the model generalizes.
3. **Balance:** This split strikes a good balance between providing enough data for training and having enough data for evaluation.

Methodology

After data processing and cleaning, the next step was to choose the suitable model and algorithm for training and predicting renewable energy generation.

Initial attempts were made using **Random Forest Regression**, which specializes in regression tasks and is quite flexible in terms of feature addition, implying that more features could be added for training later on.

The model was able to predict values but it only showed monthly variation. All the values varied on a monthly basis and remained constant for subsequent years. For eg. wind energy generated during Aug 2025 came out to be same as that generated during Aug 2026 and Aug 2027.

This was a major problem as the model was not able to utilize yearly variation during energy prediction. Many changes were made in the dataset to try to make the model understand more about yearly variation in the training data, but it showed no promising results.

Model was changed to **Gradient Boost Regression** and then again to **XGBoost Regression**. Both these models gave the same results as Random Forest.

It was evident that a single one off the 3 models used could not give valid results, thus it was decided to use all 3 of them simultaneously for energy prediction. Using **Ensemble Modelling** and a tool called **Voting Regressor**, predictions were made using the 3 models, which finally yielded suitable results.

Ensemble modeling is like getting opinions from multiple experts instead of just one. Each "expert" in this case is a different machine learning model. The code uses three different types of models:

1. **Gradient Boosting Regressor (`model1`)**: This model learns by sequentially building small decision trees, where each tree tries to correct the mistakes of the previous ones.
2. **Random Forest Regressor (`model2`)**: This model builds many decision trees independently and then combines their predictions to make a final decision. It's like having a committee of trees voting on the outcome.
3. **XGBoost Regressor (`model3`)**: This is another powerful tree-based model that is similar to Gradient Boosting but with some optimizations for performance and accuracy.

Instead of relying on just one of these models, the code combines them using a technique called **Voting Regressor (`ensemble_model`)**.

Here's how it works:

1. **Individual Model Training**: Each of the three models (`model1`, `model2`, `model3`) is trained separately on the training data (`X_train`, `y_train`). This means each model learns its own way of predicting wind power based on the input features.
2. **Voting for Predictions**: When it's time to make a prediction, the `ensemble_model` asks each of the three trained models to make a prediction independently. It's like having each expert give their opinion.
3. **Combining Predictions**: The `ensemble_model` then combines the predictions from the individual models. In this case, it uses a simple averaging approach to determine the final prediction.

Tools and libraries used

- Libraries
    1. Pandas
    2. Numpy
    3. Matplotlib
    4. Sklearn
    5. XGBoost
- Tools
    1. Train_test_split (from sklearn library)
    2. RandomForestRegressor (from sklearn library)
    3. GradientBoostingRegressor (from sklearn library)
    4. XGBRegressor (from XGBoost library)
    5. VotingRegressor (from sklearn library)
    6. Mean_squared_error, r2_score (from sklearn library)

Parameters and configurations

1. Random Forest Regressor (model1)
    - **`n_estimators=200`**:
        - This parameter determines the number of decision trees to be built in the forest.
        - In this case, the forest will consist of 200 trees.
    - **`random_state=42`**:
        - This parameter controls the randomness involved in the model's construction.
        - Setting a specific value (like 42 here) ensures that the results are reproducible.
    - **`criterion='squared_error'` (default)**: This specifies the function used to measure the quality of a split at each node of a decision tree. 'squared_error' is the default for regression and represents the mean squared error.
    - **`max_depth=None` (default)**: This determines the maximum depth of each decision tree. If None, the trees are grown until all leaves are pure or until all leaves contain less than `min_samples_split` samples.
    - **`min_samples_split=2` (default)**: The minimum number of samples required to split an internal node.
    - **`min_samples_leaf=1` (default)**: The minimum number of samples required to be at a leaf node.

- **`max_features='auto'` (default)**: The number of features to consider when looking for the best split. 'auto' uses the square root of the total number of features.

2. Gradient Boosting Regressor (model2)
   - **`n_estimators=500`:**
     - Specifies the number of boosting stages (trees) to be built.
     - Here, 500 trees will be sequentially added to the ensemble.
   - **`learning_rate=0.01`:**
     - Controls the contribution of each tree to the final prediction.
     - A lower learning rate slows down the learning process but can lead to better generalization.
   - **`max_depth=5`:**
     - Limits the maximum depth of each individual tree.
     - Controlling tree depth helps prevent overfitting.
   - **`min_samples_split=5`:**
     - Sets the minimum number of samples required to split an internal node.
     - Higher values can prevent the model from learning overly specific patterns.
   - **`min_samples_leaf=3`:**
     - Specifies the minimum number of samples required to be at a leaf node.
     - Similar to `min_samples_split`, it helps control overfitting.
   - **`subsample=0.85`:**
     - Indicates the fraction of samples to be used for fitting each tree.
     - Introducing randomness through subsampling can improve generalization.
   - **`random_state=42`:**
     - Ensures reproducibility of the model by fixing the random seed.
   - **`loss='squared_error'` (default):** The loss function to be optimized. 'squared_error' is commonly used for regression.
   - **`criterion='friedman_mse'` (default):** The function used to measure the quality of a split.

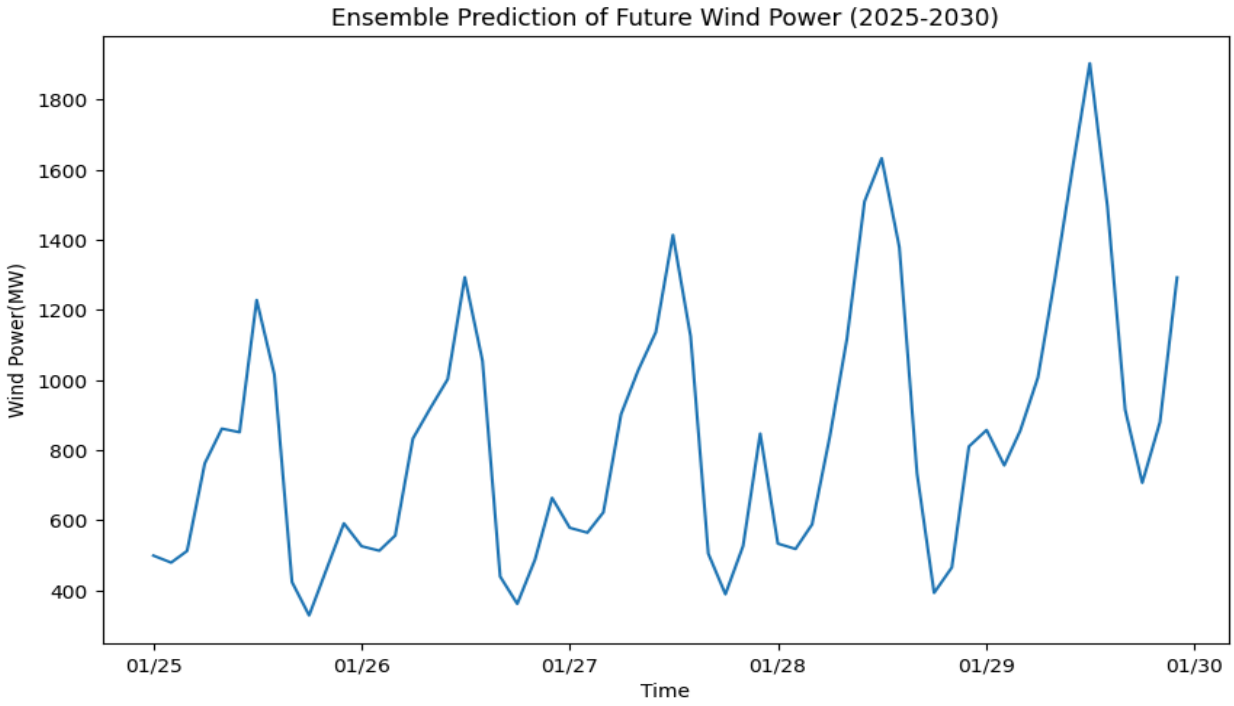
3. XGBoost Regressor (model3)
   - **`n_estimators=100`:**
     - Specifies the number of boosting rounds (trees) to be built.
     - Here, 100 trees will be added to the ensemble.
   - **`random_state=42`:**
     - Ensures reproducibility by fixing the random seed.
   - **`objective='reg:squarederror'`:**
     - Defines the learning objective, in this case, regression with squared error as the evaluation metric.
   - **`learning_rate=0.3` (default):** Similar to Gradient Boosting, controls the step size shrinkage used in update to prevents overfitting.
   - **`max_depth=6` (default):** Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.

- **`subsample=1` (default):** Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting.
- **`colsample_bytree=1` (default):** Subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
- **`gamma=0` (default):** Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be.

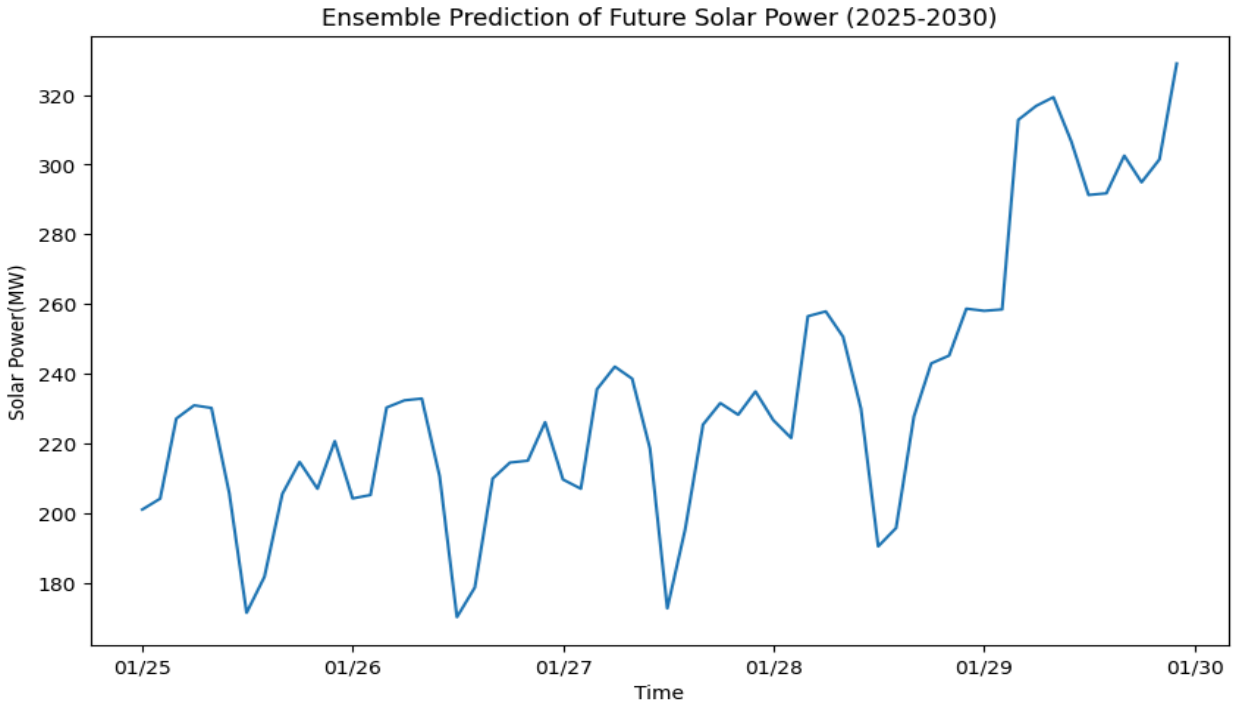
4. Voting Regressor
    - **`estimators`:**
        - This is the core parameter of `VotingRegressor`.
        - It is a list of tuples, where each tuple contains:
            - A string representing the name of the estimator (e.g., 'gb', 'rf', 'xgb').
            - The estimator object itself (e.g., `model1`, `model2`, `model3`).
    - **`voting='hard'` (default):**
        - This parameter determines how the final prediction is made.
        - With 'hard' voting (the default), the predicted class labels of the base estimators are used to determine the most frequent class, which becomes the final prediction. In regression this translates to an average of the predictions.
        - In the case of 'soft' voting, the base estimators need to predict probabilities (using `predict_proba` method), and the final prediction is based on the average of these probabilities. In regresion this parameter is ignored.

Results

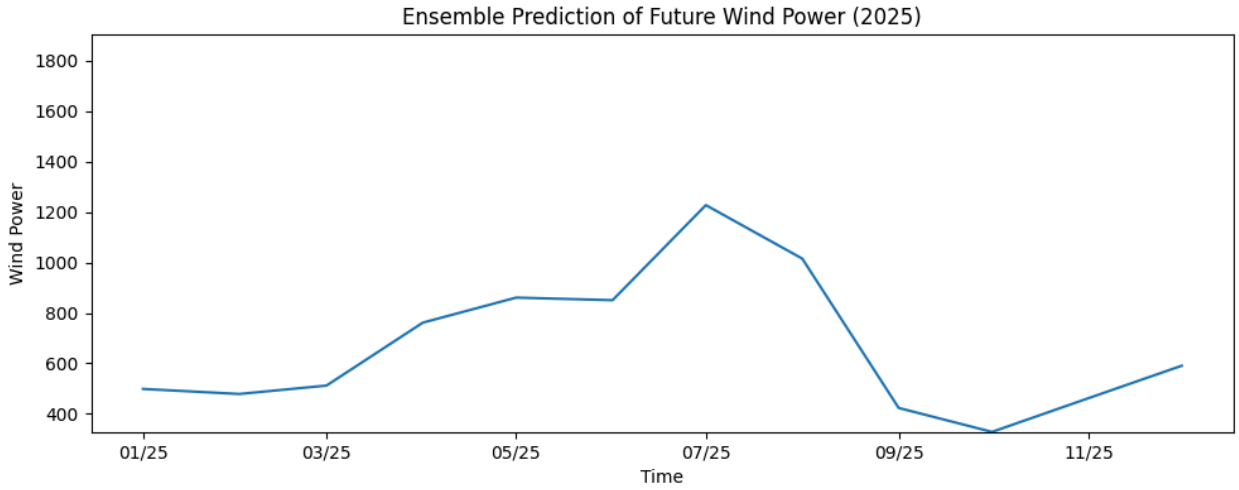
1. Wind Power prediction for years 2025-2030

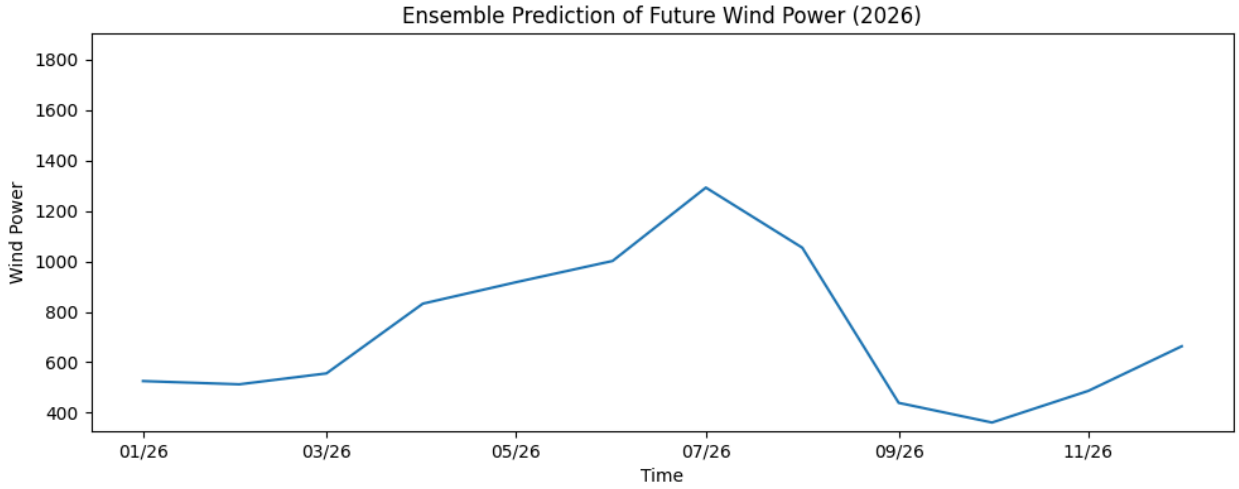


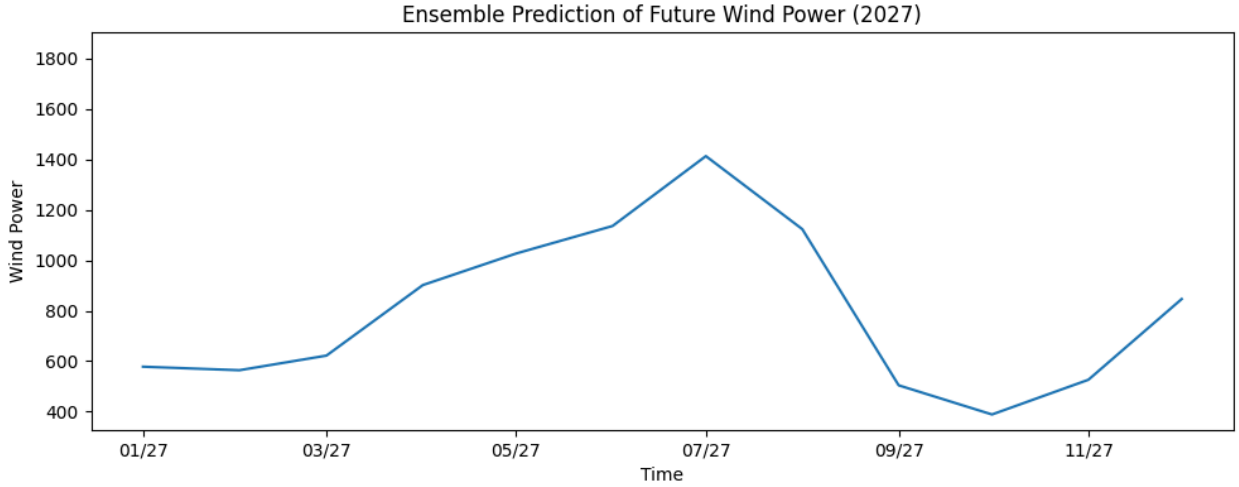
2. Solar Power prediction for years 2025-2030



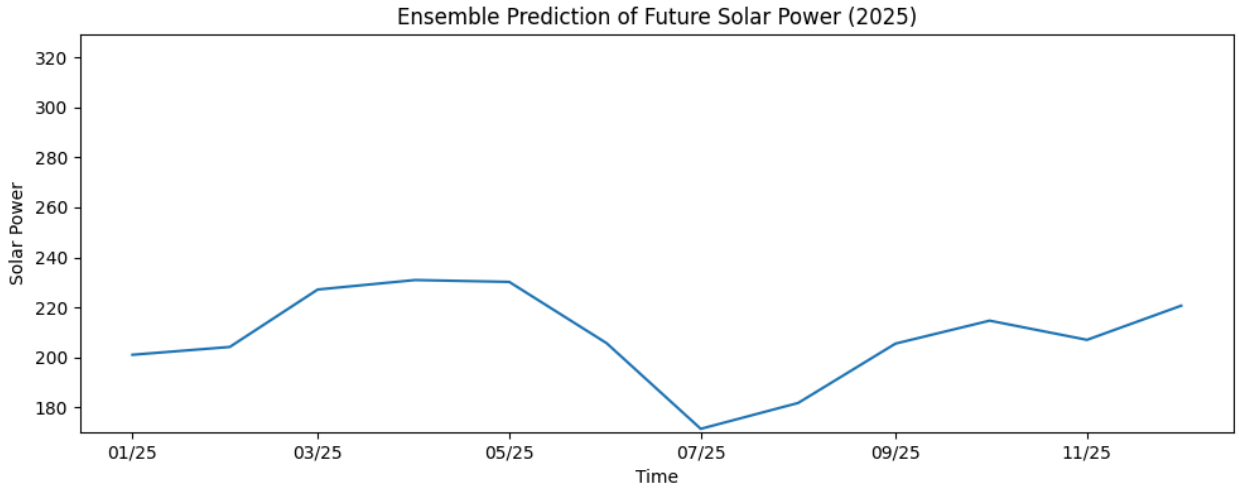
3. Wind Power prediction for 2025, 2026 and 2027

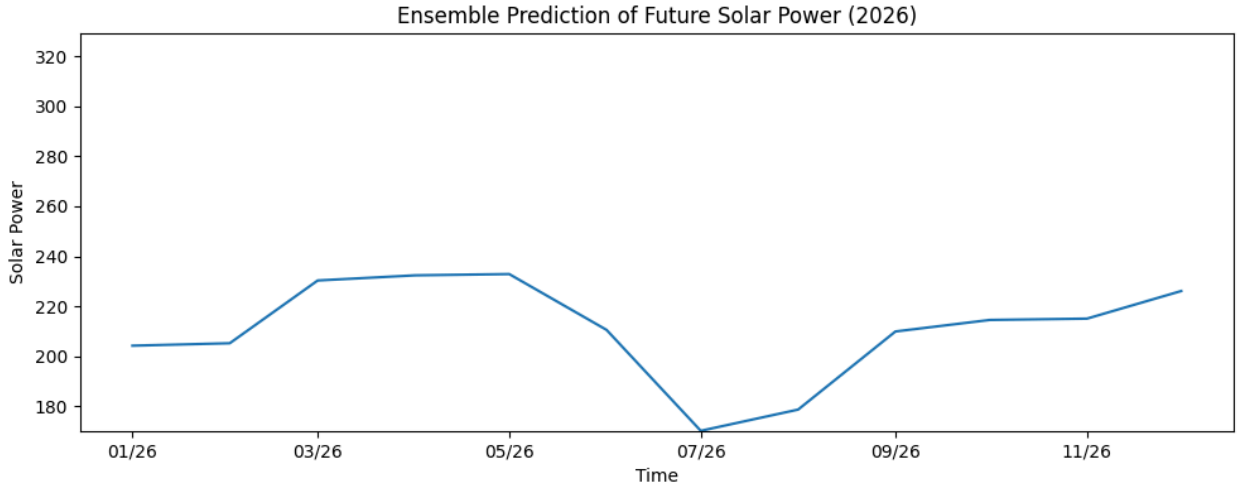


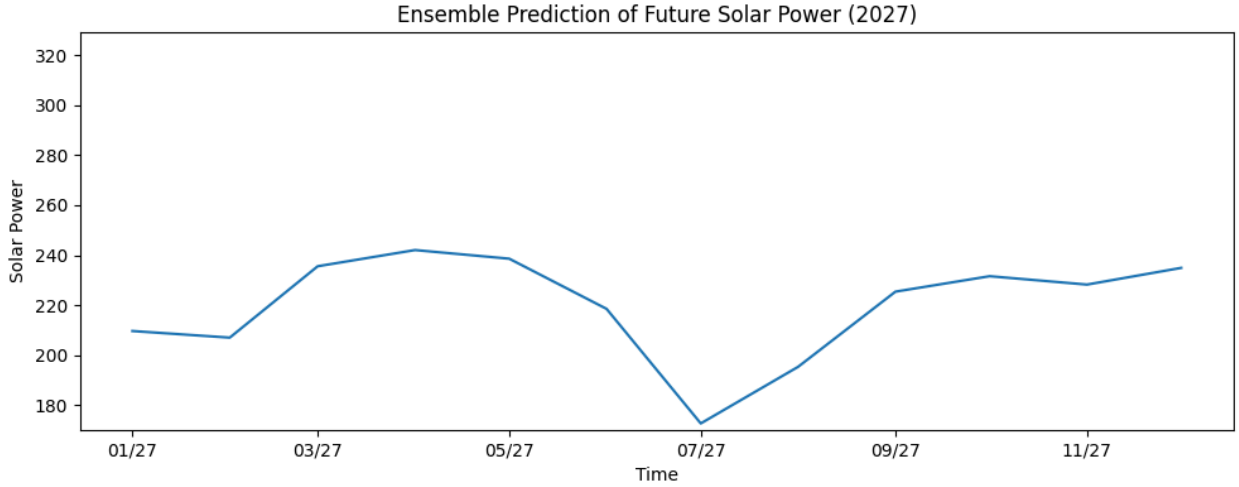




4. Solar Power prediction for years 2025, 2026 and 2027



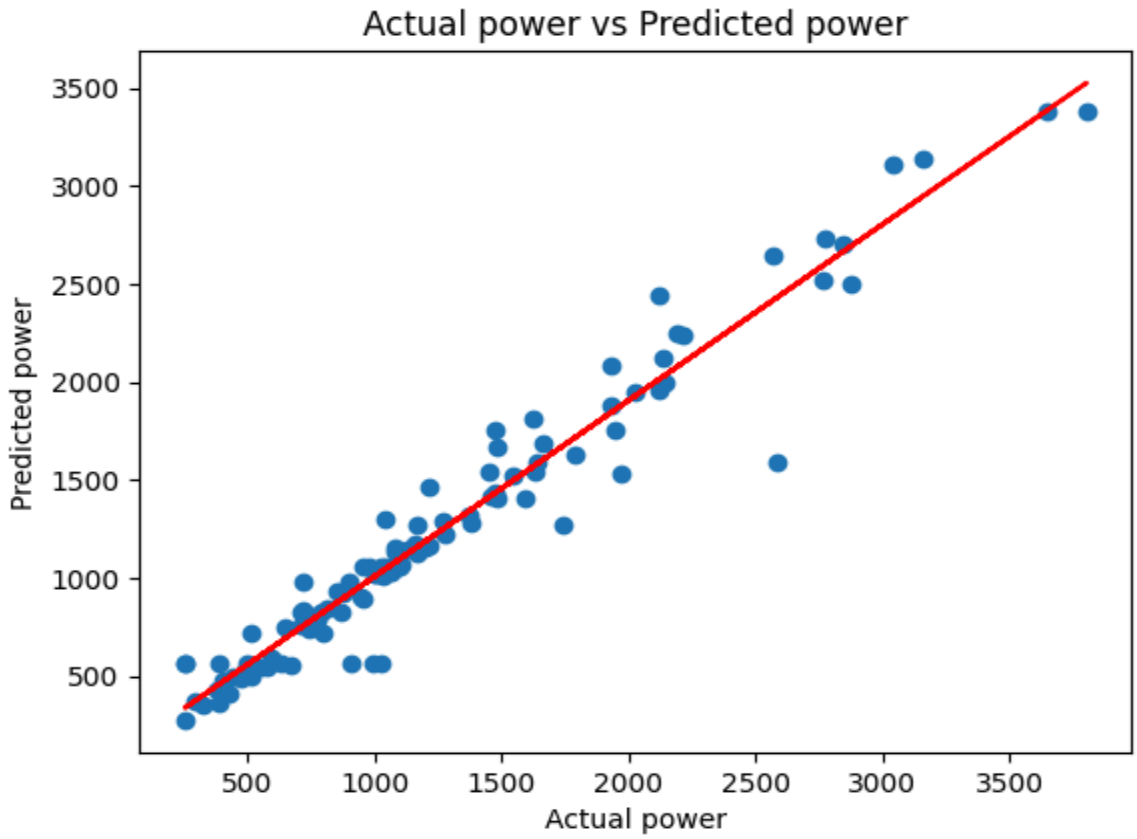




Performance metrics of ensemble model

1. **Wind power**

Result of predicted power vs actual power:



Accuracy on training data:

Mean Squared Error: 33582.15

R^2 Score: 0.94

Train Prediction Std Dev: 702.6257

Accuracy on testing data:

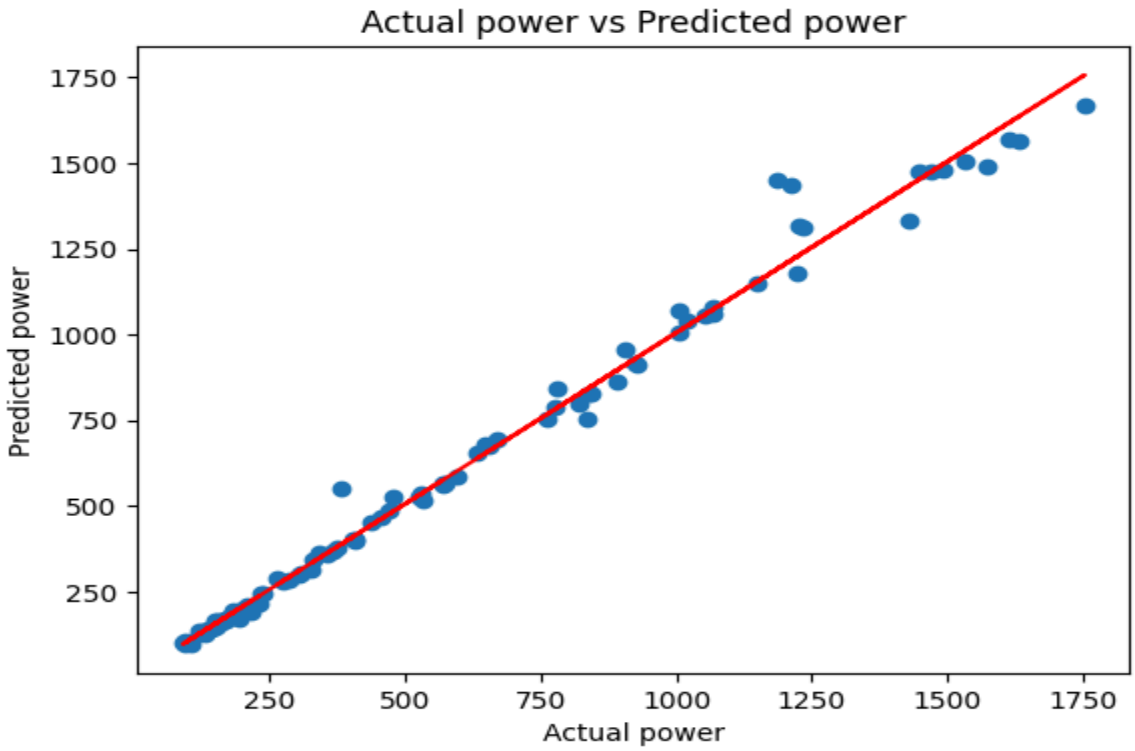
Mean Squared Error: 97402.89

R^2 Score: 0.79

Test Prediction Std Dev: 527.5062

2. Solar Power

Result of predicted power vs actual power:



Accuracy on training data:

Mean Squared Error: 2179.23

R^2 Score: 0.99

Train Prediction Std Dev: 453.6011

Accuracy on testing data:

Mean Squared Error: 8298.85

R^2 Score: 0.96

Train Prediction Std Dev: 499.2253

Model explanation

The Voting Regressor is an ensemble method that combines the predictions of multiple base regressors to produce a final prediction. It's like a democratic voting system where each model gets a "vote," and the final prediction is determined by aggregating those votes.

Statistical Working:

- **Averaging:** In its simplest form, the Voting Regressor averages the predictions of the base regressors. This approach is called **"hard voting"** and is suitable when the base models have similar performance levels.
- **Weighted Averaging:** To give more importance to better-performing models, the Voting Regressor can use **"soft voting."** In this case, it averages the predictions weighted by the models' individual weights. These weights can be assigned based on factors like cross-validation scores or domain expertise.

Benefits:

- **Reduced Variance:** By combining predictions from multiple models, the Voting Regressor helps to reduce the variance in the final prediction, making it more stable and less prone to overfitting.
- **Improved Accuracy:** Ensembles often outperform individual models, especially when the base models are diverse and have different strengths and weaknesses.

Base Regressors

Now let's explore the three base regressors used in this ensemble:

1. Gradient Boosting Regressor

Gradient Boosting is a technique where multiple weak learners (typically decision trees) are trained sequentially, with each learner focusing on correcting the errors made by the previous ones.

Statistical Working:

- **Additive Modeling:** Gradient Boosting builds an additive model, meaning it combines the predictions of multiple trees by summing them up.

- **Gradient Descent:** It uses gradient descent optimization to minimize a loss function (e.g., mean squared error) by iteratively adjusting the weights of the trees.
- **Boosting:** Each new tree is trained to predict the residuals (errors) of the previous trees, gradually improving the overall prediction accuracy.

Statistical Interpretation:

- Gradient Boosting can be viewed as a non-parametric regression method that approximates a complex function by combining simpler functions (trees).
- It's effective in capturing non-linear relationships and interactions between features.

2. Random Forest Regressor

Random Forest is an ensemble method that combines multiple decision trees trained on different subsets of the data and features.

Statistical Working:

- **Bagging:** Random Forest uses bagging (bootstrap aggregating) to create diverse training sets for each tree. This involves randomly sampling data points with replacement.
- **Feature Randomness:** It also introduces randomness in the feature selection process for each tree, further increasing diversity.
- **Averaging:** The final prediction is obtained by averaging the predictions of all the trees in the forest.

Statistical Interpretation:

- Random Forest reduces overfitting by combining predictions from multiple trees, each trained on a slightly different view of the data.
- It's known for its robustness and ability to handle high-dimensional data.

3. XGBoost Regressor

XGBoost (Extreme Gradient Boosting) is an optimized implementation of Gradient Boosting that offers improved performance and scalability.

Statistical Working:

- **Regularization:** XGBoost incorporates regularization techniques to prevent overfitting and improve generalization.

- **Parallel Processing:** XGBoost is designed for efficient parallel processing, making it suitable for large datasets.

Statistical Interpretation:

- XGBoost builds on the statistical principles of Gradient Boosting but with enhancements that lead to better accuracy and computational efficiency.
- It's widely used in machine learning competitions and real-world applications.

Advantage of Random forest over other models

Random forest is a collection of decision trees, as the name suggests. A single decision tree cannot be used for predicting energy production as it is very prone to overfitting, which means it will be biased towards the data used to train the model and will not give correct outputs for new and unfamiliar data.

Random forest improves upon this by averaging predictions from many diverse decision trees, each trained on a random subset of the data. This diversity helps reduce the impact of individual tree errors.

Random forest also boasts significant advantages as compared to xgboost and gradient boost regressor.

- Random Forest is known for handling noisy data effectively. By randomly selecting features for creating each tree, it is less likely to be influenced by irrelevant or noisy features.
- Random Forest typically requires less hyperparameter tuning than boosting methods. While it has hyperparameters to adjust, the default values often work well. Hence, it reduces the load of choosing optimal parameters.

Accuracy comparison between models (for wind power)

1. **XGBoost Regressor**
    - `XGBoost Regressor - Mean Squared Error: 132576.08`
    - `XGBoost Regressor - R^2 Score: 0.71`
2. **Gradient Boost Regressor**
    - `Gradient Boosting Regressor - Mean Squared Error: 84222.99`
    - `Gradient Boosting Regressor - R^2 Score: 0.82`
3. **Random Forest Regressor**
    - `Random Forest Regressor - Mean Squared Error : 115968.76`
    - `Random Forest Regressor - R^2 Score : 0.75`

The less accuracy of Random Forest as compared to Gradient Boost Regressor comes from the fact that it is robust to overfitting and thus makes better predictions for new data.

Tree-based models like XGBoost, Gradient Boosting, and Random Forest generally perform better with larger datasets. Since the dataset used was not large enough to capture the complex patterns effectively, the accuracy of these models was affected.

XGBoost and Gradient Boosting, in particular, are prone to overfitting, especially with a large number of trees or deep trees. Overfitting occurs when the model learns the training data too well, including noise, and fails to generalize well to unseen data, leading to lower accuracy on the test set.

Random Forest vs. Gradient Boosting

- **Sequential Learning:** Gradient Boosting builds trees sequentially, where each tree learns from the mistakes of the previous trees. This iterative approach allows it to focus on difficult-to-predict samples and potentially achieve higher accuracy. Random Forest, on the other hand, builds trees independently and averages their predictions. While this makes it robust and less prone to overfitting, it might not capture complex relationships as effectively as Gradient Boosting.
- **Bias-Variance Trade-off:** Random Forest generally has a higher bias but lower variance compared to Gradient Boosting. This means that Random Forest might make simpler assumptions about the data, which can limit its accuracy for complex relationships. Gradient Boosting, with its sequential learning, can capture more complex patterns but is also more prone to overfitting (higher variance).

Random Forest vs. XGBoost

- **Regularization:** XGBoost incorporates regularization techniques, such as L1 and L2 regularization, to prevent overfitting. Random Forest typically doesn't have explicit regularization, which might make it more susceptible to overfitting, especially with complex datasets. This explains why Random Forest might perform better than XGBoost in your case if XGBoost is overfitting more significantly.
- **Feature Importance:** XGBoost has a built-in mechanism for calculating feature importance, which helps it focus on the most relevant features for prediction. Random Forest also provides feature importance, but it might not be as sophisticated as XGBoost's approach. If XGBoost's selection is too aggressive it could miss important features.

In Summary

The Voting Regressor leverages the statistical strengths of Gradient Boosting, Random Forest, and XGBoost to create a more robust and accurate prediction model. By combining the predictions of these diverse base regressors, it reduces variance, improves accuracy, and provides a powerful tool for predicting wind power and forecasting future values. Each base regressor has its unique statistical approach, but they all contribute to the overall performance of the ensemble.