A Multi Model Machine Learning Analysis of Wind Turbine Power Output Prediction

Kishor S
Kumuaraguru College of Technology
B.Tech Information Technology
kishor.24it@kct.ac.in
RegID: 24BIT051

Guidance:
Sugunanthan P
Kumuaraguru College of Technology
B.E Electronics and Communication Engineering
sugunanthan.22ec@kct.ac.in
RegID: 22BEC172

Sujitha V
Kumuaraguru College of Technology
Assistant Professor III
Dept. Information Technology
sujitha.v.it@kct.ac.in

Abstract

This study applies machine learning models to predict wind turbine power output using real-world SCADA data. The dataset incorporates operational features like wind speed, direction, temperature, and rotor RPM. After exploratory analysis and outlier removal, multiple algorithms (linear regression, polynomial regression, transformer networks, and XGBoost) were trained and compared. The research evaluates each model's performance metrics to identify their strengths and limitations in accurate power output forecasting in smart grid applications.
Index Keywords: machine learning, prediction, wind turbine, XGBoost

Introduction

The integration of machine learning (ML) techniques into the renewable energy sector has opened new avenues for improving the efficiency and reliability of combining different power systems. In particular, wind energy, being highly dependent on fluctuating atmospheric conditions, presents a unique challenge in accurately forecasting power output. Effective prediction models can significantly enhance energy scheduling, reduce costs, and support grid stability.

Data Set Information

In this study, we investigate the use of multiple machine learning models for predicting the power output of wind turbines. The foundation of our work is a SCADA (Supervisory Control and Data Acquisition) dataset collected over a span of several years. The dataset contains over **653,000 observations**, with **22 distinct columns**, each representing a physical or operational parameter of the turbine.

Description of the Data Set:
The data set contains a multi-annual, 10-minute interval SCADA dataset collected from a Vestas V52 wind turbine operated by Dundalk Institute of Technology, Ireland (GPS: 53.98352°N, -6.39139°W). The data spans from **30 January 2006 to 12 March 2020**, encompassing over 14 years of continuous operation in a peri-urban environment. The turbine functions as a **behind-the-meter** system, with a **hub height of 60 meters** and a **rotor diameter of 52 meters**.

Note from the Data Set Provider:
A notable period of inactivity in power generation occurred between **04 October 2018** and **27 July 2019** due to a **gearbox replacement**, during which **no positive electrical power** was recorded.

Key Features: Wind Speed(m/s), Wind Direction, Rotor RPM, Generator RPM, Environment Temperature, Generator Temperature, Individual Phase(1,2,3) Temperature, Power(KWt), Minimum Power(KWt), Maximum Power(KWt).

Analysis and Visualization of the Data Set

To explore and understand various features of the data set, we employed **Python** along with the widely used data analysis and visualization libraries **numpy, pandas, matplotlib, seaborn.** These tools facilitated efficient data cleaning, statistical summarization, and feature-level exploration. While libraries like **sklearn, PyTorch, XGBoost** help us for deeper analysis and predictive modeling.

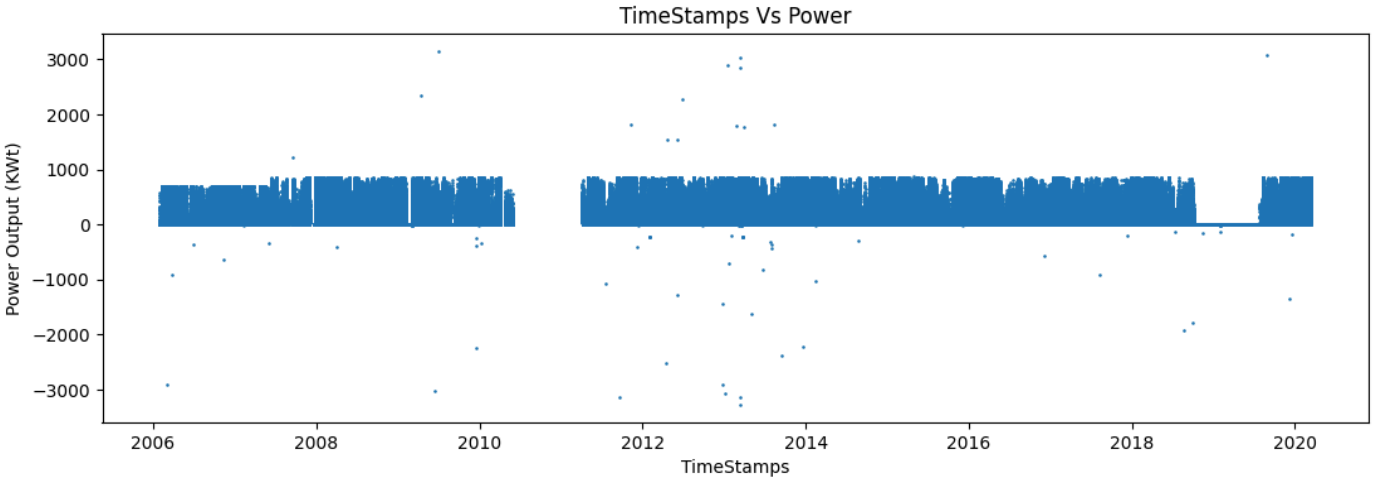


Figure 1.1 *Time Vs Power*

Inferences from visualization of the Data Set:

1. **No positive electrical output** is recorded during the period **04 October 2018 to 27 July 2019**, coinciding with a **gearbox replacement** event.

2. The dataset contains **no readings** between **01 June 2010 and 01 April 2011**, indicating a **data gap,** but not a critical issue.

3. Several **power readings exceed 1000 kilowatts**, which is beyond the **rated capacity** of the Vestas V52 turbine and are likely **outliers**.

4. Some records show **negative power output**, which may reflect **braking behavior** during maintenance or turbine anomalies. However, values **below -10 kilowatts** are classified as **outliers** due to their unrealistic nature.

5. The **wind speed box plot** (Figure 1.2) shows values **exceeding 25 m/s**, which is near or above the **cut-off limit** for most turbines, indicating the presence of

high-end outliers.

6. The **power output box plot** (Figure 1.2) further confirms the **impact of extreme outliers**, especially those associated with **negative and abnormally high power values**.

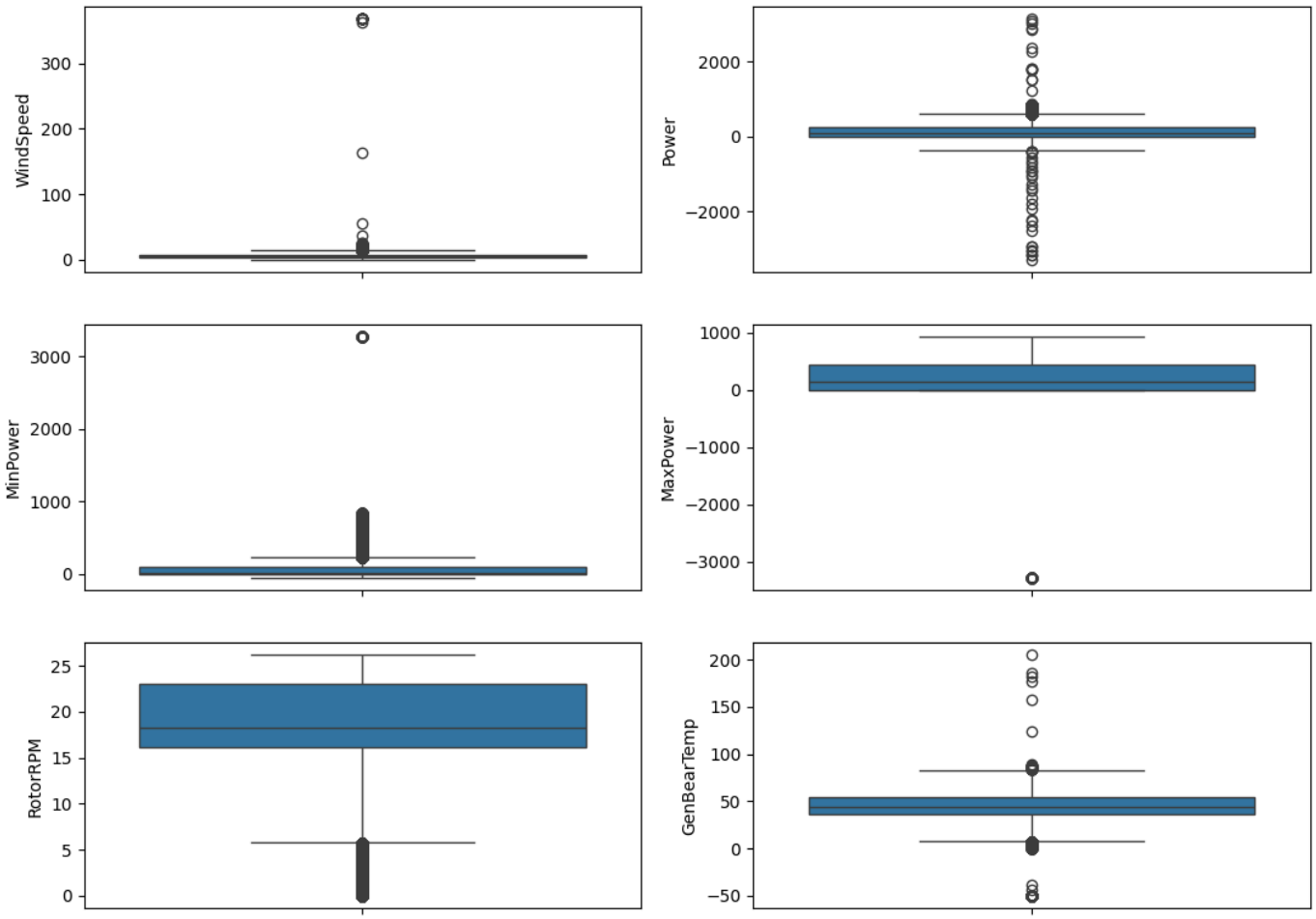


Figure 1.2 *Box Plot Analysis*

Filtering out the Outliers using the Inter Quartile Range(IQR) Method:

The Inter Quartile Range(IQR) method is a robust statistical method to identify and remove outliers in a data set. This uses the middle 50% of the values, defined as

Q1(First Quartile): 25th percentile of the data

Q3(Third Quartile): 75th percentile of the data

IQR = Q3 - Q1

Any data points that lie **outside** the below range are considered **outliers**.

Low = Q1 - 1.5*IQR

High = Q3 + 1.5*IQR

Filtered Data = data > LOW & data < HIGH (OR) **LOW < data < HIGH**

Where the **integer 1.5** in the equation is the **multiplicative factor.** Which decides the **outliers influence.(1 denotes no influence of data outside the IQR)**

After several iterations of trial and error in the process of filtering and removing outliers using the **Interquartile Range (IQR) method**, it was observed that a **custom IQR range using the 15th and 99th percentiles** with a **multiplicative factor of 1.5** effectively removed outliers without discarding valid operational data. This refined approach helps preserve meaningful variations while eliminating extreme and anomalous values.

The following Python code demonstrates the implementation of this method on the dataset:

```
# Where,
# df -> Pandas DataFrame Object
# features -> List of features to perform IQR Outlier removal
def remove_outliers_iqr(df, features):
    for feature in features:
        Q_Low = df[feature].quantile(0.15)
        Q_High = df[feature].quantile(0.99)

        IQR = Q_High - Q_Low
        low = Q_Low - 1.5*IQR
        high = Q_High + 1.5*IQR

        df = df[(df[feature] > low) & (df[feature] < high)]
    return df
```

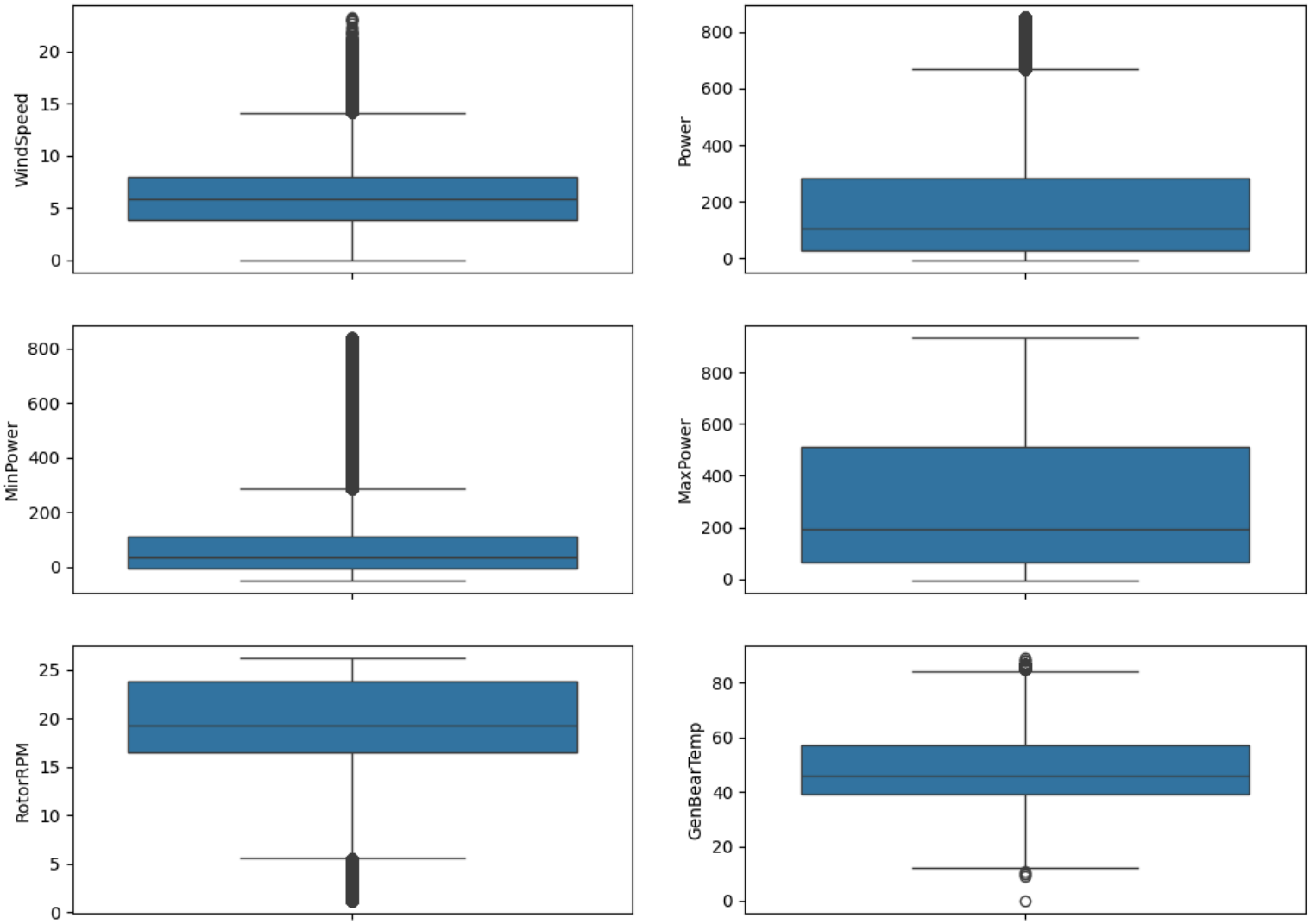


Figure 1.3 *Box Plot Analysis of Filtered Data*

Box plot analysis before and after outlier removal using the **Interquartile Range (IQR)** method clearly demonstrates the improvement in data quality. As shown in Figure 1.2, the original dataset contains several outliers, particularly in power output and wind speed, which can distort model performance. After applying the IQR method with quartile limits at the **15th and 99th percentiles**, these anomalies are effectively removed, as illustrated in Figure 1.3. The cleaned data now shows a tighter and more accurate distribution, enhancing the overall reliability of the dataset for machine learning tasks.

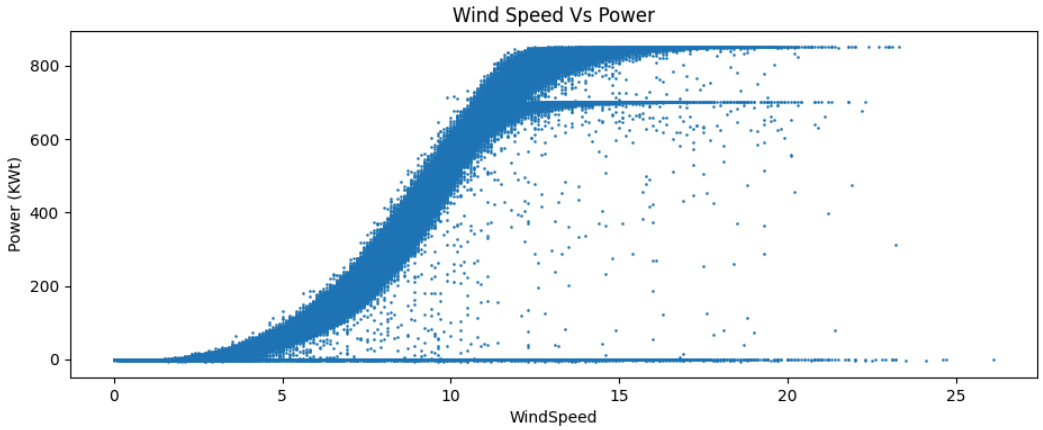


Figure 1.4 *Wind Speed vs Power after IQR*

The above image (Figure 1.4) illustrates the **Wind Speed vs Power Output** relationship after applying initial outlier removal. From the graph, it is evident that extreme values—such as **negative power outputs**, **power values exceeding 1000 kW**, and **wind speeds above 25 m/s**—have been successfully filtered out. However, a notable flat line persists near zero power for wind speeds exceeding **4 m/s**, indicating periods where the turbine received sufficient wind but still failed to generate power. This anomaly is most likely due to **maintenance or shutdown events**, such as **gearbox changeouts** or **operational halts**, during which the turbine was non-functional. Since these data points do not reflect normal operational behavior and can negatively influence the performance and accuracy of forecasting models, they are **manually excluded** from the dataset to ensure the model is trained on valid and meaningful data.

This manual removal process is performed in two key steps to clean the dataset by eliminating data points that are likely to distort the model's predictions due to known operational issues and anomalies.

First, data within the date range from **04 October 2018 to 27 July 2019** is removed. This period corresponds to a **gearbox replacement** event, during which the turbine was non-operational. As a result, no positive power output is recorded during this time, and including such data would bias the model towards lower or zero output inappropriately.

Second, the code filters out any data points where **power output is negative (Power < 0)** while the **wind speed is greater than 4 m/s**. Under normal conditions, the turbine should generate positive power at such wind speeds. The presence of negative power readings in this range typically indicates faulty measurements, maintenance downtime, or system malfunctions.

```
filtered_df = df[~((df['Timestamps'] > "2018-10-04 00:00:00") &
(df['Timestamps'] < "2019-07-27 00:00:00"))]

filtered_df = filtered_df[~((filtered_df['Power'] < 0) &
(filtered_df['WindSpeed'] > 4))]
```

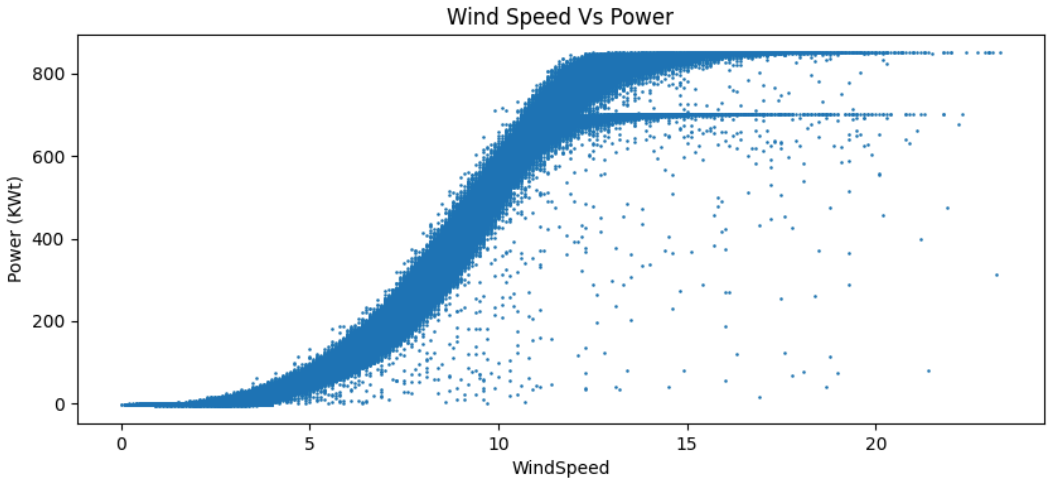


Figure 1.5 *Wind Speed Vs Power after IQR and Manual Filtering*

Exploring Different Machine Learning Models for Forecasting:

To accurately predict wind turbine power output, it is essential to explore a variety of machine learning models, each with their own strengths and assumptions. Different models capture patterns in data in different ways—some are more suited for linear relationships, while others excel in modeling complex nonlinear interactions. In this study, we begin by implementing simple regression models like **Linear Regression** to establish a baseline, and then progressively move towards more advanced and powerful models such as **Random Forests**, **Gradient Boosting (XGBoost)**, and **Neural Networks**. This comparative approach not only helps in understanding the modeling capabilities of

each algorithm but also in selecting the most effective one based on accuracy, robustness, and computational efficiency. The goal is to build a forecasting model that generalizes well to unseen data while handling the inherent variability and nonlinearity in wind power generation.

Train, Test Split of the Data Set:

The **train_test_split** function from the **scikit-learn** library is used to divide the dataset into training and testing subsets, typically to evaluate the model's performance on unseen data. This ensures that the model learns patterns from one part of the data (training set) and is then tested on a separate part (test set) to assess its generalization ability.
The python code to split the input data is as follows:

```
from sklearn.model_selection import train_test_split# Define features
and target
features = [
    'WindSpeed', 'StdDevWindSpeed', 'WindDirRel', 'RotorRPM',
    'GenRPM', 'GearOilTemp', 'GenPh1Temp', 'GenPh2Temp',
    'GenPh3Temp', 'GearBearTemp'
]
target = 'Power'

X = filtered_df[features]
y = filtered_df[target]

# Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=42)
```

Linear Regression:

Linear Regression models the relationship between input features and output by fitting a straight line, represented as:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

where $\hat{y}$ is the predicted output, β_0 is the intercept, and β_i are the coefficients for each feature x_i.

```
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score,
mean_squared_error

# Initialize and train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Plotting
plt.scatter(X_test['WindSpeed'], y_test, label = "Actual Values", s =
0.7)
plt.scatter(X_test['WindSpeed'], y_pred, label = "Predicted Values",
color = "red", s = 0.7)
plt.xlabel("Wind Speed")
plt.ylabel("Power (KWt)")
plt.title("Linear Regression")
plt.legend()

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Linear Regression Performance:")
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Linear Regression Performance:
MAE: 47.4001
MSE: 3763.9589
```

```
R² Score: 0.9215
```

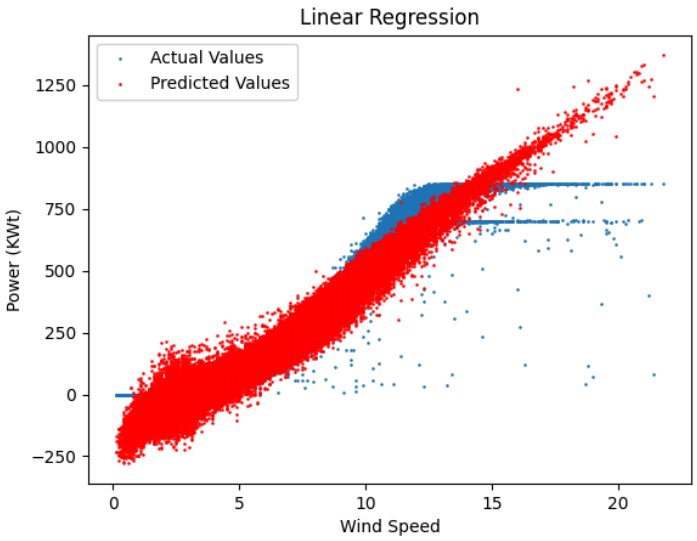


Figure 2.1 *Linear Regression Prediction Graph*

Inference:

The **Linear Regression model** demonstrates a **moderate to high performance** in predicting wind turbine power output, as indicated by an **R^2 score of 0.9215**, meaning approximately **92.15% of the variance** in power output is explained by the input features. However, the **MAE of 47.40** and **MSE of 3763.96** suggest there are still **significant absolute prediction errors** in certain cases, indicating that while the model captures the general trend, it may not perform optimally for all instances—especially where the data is nonlinear or more complex relationships exist.

Polynomial Regression:

Polynomial Regression is a type of regression that models the relationship between the independent variable and the dependent variable as a n-degree polynomial, suitable for capturing nonlinear patterns.

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n$$

Where:

- $\hat{y}$ = predicted output
- x = input variable
- $\beta_0, \beta_1, \ldots, \beta_n$ = coefficients for each term
- x^n = input raised to the power of n

```
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Transform the features to polynomial features of degree 3
poly = PolynomialFeatures(degree=3)
X_poly_train = poly.fit_transform(X_train)
X_poly_test = poly.transform(X_test)

# Initialize and train the polynomial regression model
model = LinearRegression()
model.fit(X_poly_train, y_train)

# Predict on the test set
y_pred = model.predict(X_poly_test)

# Plotting (using WindSpeed as x-axis reference for visualization)
plt.scatter(X_test['WindSpeed'], y_test, label="Actual Values", s=0.7)
plt.scatter(X_test['WindSpeed'], y_pred, label="Predicted Values",
color="red", s=0.7)
plt.xlabel("Wind Speed")
plt.ylabel("Power (KWt)")
plt.title("Polynomial Regression (Degree 3)")
plt.legend()

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
```

```
r2 = r2_score(y_test, y_pred)

print("Polynomial Regression (Degree 3) Performance:")
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Polynomial Regression (Degree 3) Performance:
MAE: 11.2462
MSE: 335.7254
R² Score: 0.9930
```

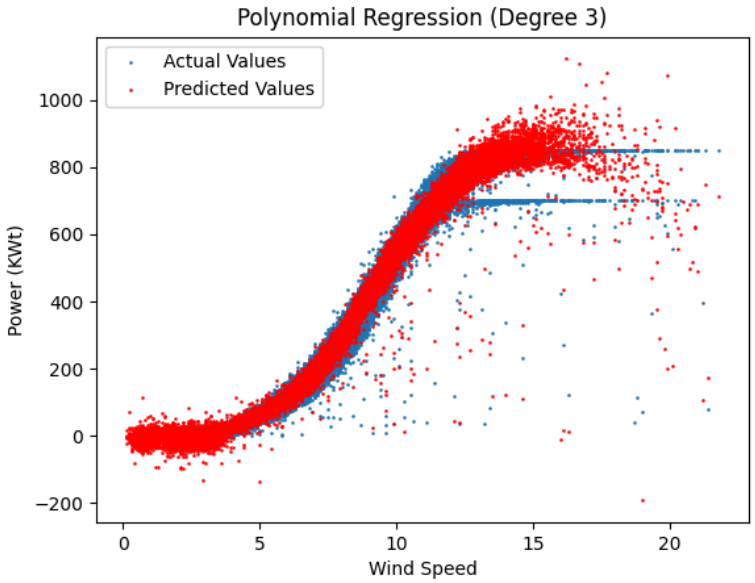


Figure 2.2 *Polynomial Regression Prediction Graph*

Inference:

The **polynomial regression model of degree 3** shows a **significant improvement** in performance compared to linear regression. The **MAE of 11.25** and **MSE of 335.73** indicate **low prediction errors**, while the **R^2 score of 0.9930** suggests that the model explains **99.3% of the variance** in the power output.
This means the model fits the nonlinear pattern in the wind speed–power relationship **very well**, making it **highly suitable** for this dataset.

Random Forest:

Random Forest is a powerful ensemble machine learning algorithm that combines multiple decision trees to produce more accurate and stable predictions. Each tree is trained on a random subset of the data and features, and their individual outputs are averaged (for regression) or voted (for classification) to form the final prediction. This method reduces overfitting and improves generalization by leveraging the wisdom of the crowd. Random Forest is especially effective for complex, nonlinear relationships and high-dimensional datasets.

```
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score


# Initialize and train the Random Forest model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Plotting
plt.scatter(X_test['WindSpeed'], y_test, label="Actual Values", s=0.7)
plt.scatter(X_test['WindSpeed'], y_pred, label="Predicted Values",
color="red", s=0.7)
plt.xlabel("Wind Speed")
plt.ylabel("Power (KWt)")
plt.title("Random Forest Regression")
plt.legend()

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
```

```
r2 = r2_score(y_test, y_pred)

print("Random Forest Regression Performance:")
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Random Forest Regression Performance:
MAE: 11.2468
MSE: 333.0407
R² Score: 0.9931
```

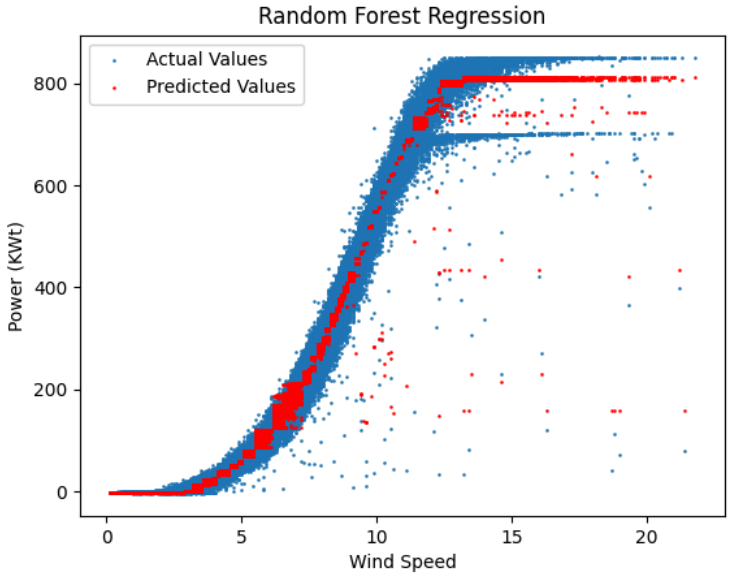


Figure 2.3 *Random Forest Prediction Graph*

Inference:

This high R^2 score (~0.9931) indicates that the model explains over 99% of the variance in the power output based on wind speed, which is **comparable to the performance of the polynomial regression model**.

However, **training time for Random Forests can be significantly high**, especially with large datasets or default hyperparameters. To mitigate this, I **reduced the number of**

decision trees (n_estimators) to 20 and **limited their depth (max_depth) to 6**. This helped **accelerate the training process** while still retaining excellent prediction accuracy.

XGBoost:

XGBoost (Extreme Gradient Boosting) is a fast and regularized ensemble learning algorithm based on gradient boosting, known for its high accuracy and scalability. It builds additive decision trees sequentially, optimizing errors and handling overfitting with built-in regularization.

```
from xgboost import XGBRegressor
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Initialize the model
xgb_model = XGBRegressor(n_estimators=1000, max_depth=10,
learning_rate=0.1, random_state=42)

# Train the model
xgb_model.fit(X_train, y_train)

# Predict on the test set
y_pred = xgb_model.predict(X_test)

# Plotting
plt.scatter(X_test['WindSpeed'], y_test, label="Actual Values", s=0.7)
plt.scatter(X_test['WindSpeed'], y_pred, label="Predicted Values",
color="red", s=0.7)
plt.xlabel("Wind Speed")
plt.ylabel("Power (KWt)")
plt.title("XGBoost Regression")
plt.legend()

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
```

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("XGBoost Regression Performance:")
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
XGBoost Regression Performance:
MAE: 7.7519
MSE: 161.9130
R² Score: 0.9966
```

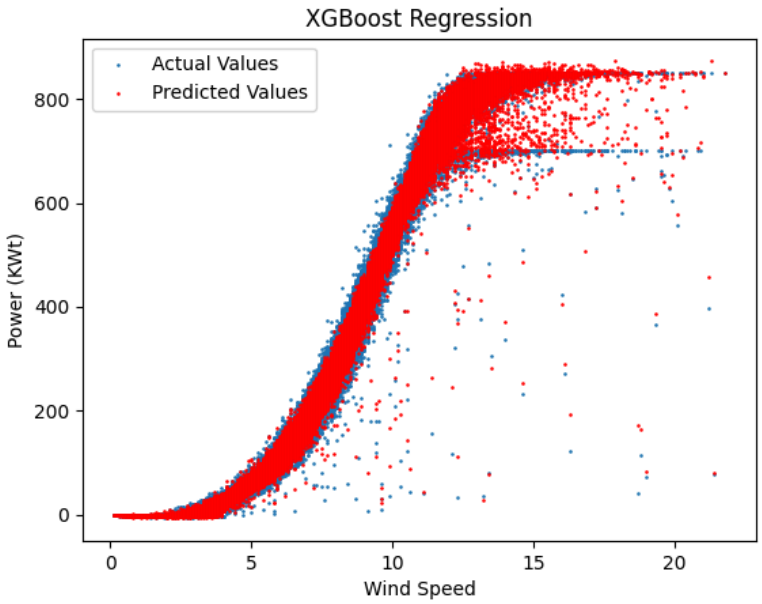


Figure 2.4 *XGBoost Prediction Graph*

Inference:

The XGBoost Regression model has shown exceptional predictive accuracy with a **Mean Absolute Error (MAE)** of **7.7519**, **Mean Squared Error (MSE)** of **161.9130**, and an **R^2 Score** of **0.9966**, indicating a nearly perfect fit to the data.

One of the standout advantages of XGBoost is its **high speed and efficiency**, even on large datasets. Due to its faster training time compared to models like Random Forest, I increased the number of `n_estimators` to enhance performance without compromising training time. This makes XGBoost an excellent choice for both precision and scalability.

Transformer (Neural Net):

The Transformer is a deep learning architecture originally designed for sequence modeling tasks, particularly in natural language processing. Unlike traditional recurrent models, it uses self-attention mechanisms to capture complex dependencies across the entire input sequence in parallel. Its ability to model temporal patterns makes it suitable for sequential data like time-series wind turbine readings.

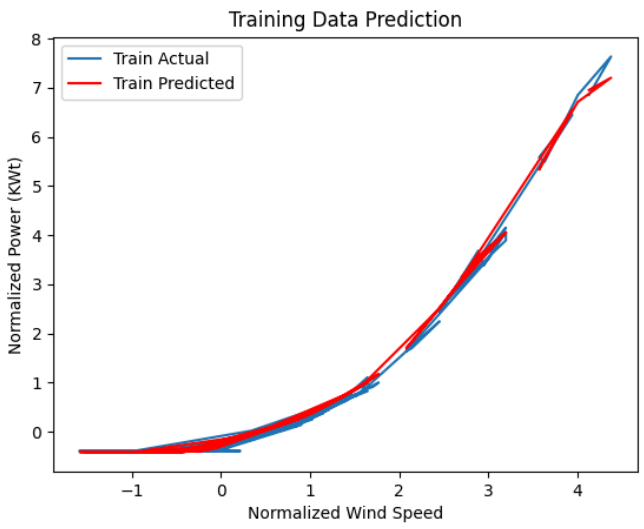
```
# Simple Transformer Model
import torch
from torch import nn
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score,
mean_squared_error

# Prepare Training Data
train_x = filtered_df['WindSpeed'].to_numpy()[:500]
train_y = filtered_df['Power'].to_numpy()[:500]

# Normalize
train_xt = (train_x - train_x.mean()) / train_x.std()
train_yt = (train_y - train_y.mean()) / train_y.std()

# Convert to Tensors
x_train_tensor = torch.tensor(train_xt,
dtype=torch.float32).unsqueeze(1)
y_train_tensor = torch.tensor(train_yt,
dtype=torch.float32).unsqueeze(1)

# Add sequence dimension
x_train_seq = x_train_tensor.unsqueeze(1)
```

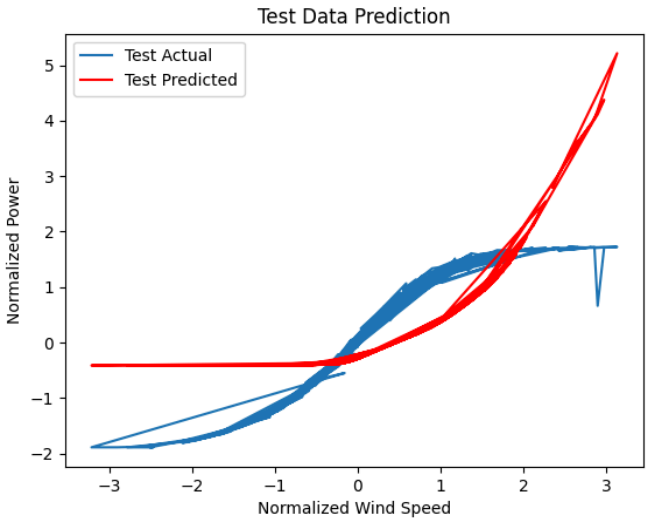
```
y_train_seq = y_train_tensor.unsqueeze(1)

# Define Transformer Model
class WSTransformer(nn.Module):
    def __init__(self, d_model=16, nhead=4, num_layers=2):
        super().__init__()
        self.input_layer = nn.Linear(1, d_model)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model,
nhead=nhead)
        self.transformer = nn.TransformerEncoder(encoder_layer,
num_layers=num_layers)
        self.output_layer = nn.Linear(d_model, 1)

    def forward(self, x):
        x = self.input_layer(x)
        x = self.transformer(x)
        x = self.output_layer(x)
        return x

# Instantiate model, loss, optimizer
tmodel = WSTransformer()
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(tmodel.parameters(), lr=0.001)

# Train the model
for epoch in range(1050):
    tmodel.train()
    output = tmodel(x_train_seq)
    loss = loss_fn(output, y_train_seq)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

# Evaluation on Training Data
```

```
tmodel.eval()
with torch.no_grad():
    train_predicted = tmodel(x_train_seq).squeeze().numpy()

train_actual = y_train_seq.squeeze().numpy()

plt.plot(x_train_tensor.squeeze().numpy(), train_actual, label="Train
Actual")
plt.plot(x_train_tensor.squeeze().numpy(), train_predicted,
label="Train Predicted", color="red")
plt.xlabel("Normalized Wind Speed")
plt.ylabel("Normalized Power (KWt)")
plt.legend()
plt.title("Training Data Prediction")
plt.show()

print("TRAIN MAE:", mean_absolute_error(train_actual,
train_predicted))
print("TRAIN R2:", r2_score(train_actual, train_predicted))
print("TRAIN MSE:", mean_squared_error(train_actual, train_predicted))

# Prepare Test Data (last 500 points)
test_x = new_df['WindSpeed'].to_numpy()[-500:]
test_y = new_df['Power'].to_numpy()[-500:]

test_xt = (test_x - test_x.mean()) / test_x.std()
test_yt = (test_y - test_y.mean()) / test_y.std()

x_test_tensor = torch.tensor(test_xt,
dtype=torch.float32).unsqueeze(1)
y_test_tensor = torch.tensor(test_yt,
dtype=torch.float32).unsqueeze(1)

x_test_seq = x_test_tensor.unsqueeze(1)
y_test_seq = y_test_tensor.unsqueeze(1)

# Evaluation on Test Data
tmodel.eval()
```

```
with torch.no_grad():
    test_predicted = tmodel(x_test_seq).squeeze().numpy()

test_actual = y_test_seq.squeeze().numpy()

plt.plot(x_test_tensor.squeeze().numpy(), test_actual, label="Test
Actual")
plt.plot(x_test_tensor.squeeze().numpy(), test_predicted, label="Test
Predicted", color="red")
plt.xlabel("Normalized Wind Speed")
plt.ylabel("Normalized Power")
plt.legend()
plt.title("Test Data Prediction")
plt.show()

print("TEST MAE:", mean_absolute_error(test_actual, test_predicted))
print("TEST R2:", r2_score(test_actual, test_predicted))
print("TEST MSE:", mean_squared_error(test_actual, test_predicted))
```

```
TRAIN MAE: 0.0401352159678936
TRAIN R2: 0.9963955283164978
TRAIN MSE: 0.0036044795997440815

TEST MAE: 0.636644721031189
TEST R2: 0.41068750619888306
TEST MSE: 0.5893124938011169
```





Inference:

The Transformer model demonstrates **strong training performance** with metrics indicating an excellent fit:

- **Train MAE:** 0.0401
- **Train MSE:** 0.0036
- **Train R^2 Score:** 0.9964

However, during evaluation on unseen test data (last 500 points), the performance significantly drops:

- **Test MAE:** 0.6366
- **Test MSE:** 0.5893
- **Test R^2 Score:** 0.4107

This gap suggests that the model, although effective during training, struggles to generalize — likely due to:

- **Single-feature training (WindSpeed only)** due to computational limitations
- **Training conducted on only a single batch**, which is insufficient for learning robust temporal patterns

Given its batch-based training mechanism, **increasing the number of training batches or expanding the training to the full dataset** would likely improve generalization and bring the performance **closer to that of Random Forest or XGBoost**, which currently outperform the Transformer in prediction accuracy.

Comparative Analysis of Predictive Algorithms for Wind Turbine Power Output

To assess the forecasting performance of various machine learning algorithms for wind turbine power output prediction, a comparative analysis was conducted across multiple models including Linear Regression, Polynomial Regression, Random Forest, XGBoost, and a simple Transformer-based neural network. Each model was evaluated based on three key performance metrics—Mean Absolute Error (MAE), Mean Squared Error (MSE), and the Coefficient of Determination (R^2 Score). Additionally, observations regarding training time and inference behavior are noted to offer a holistic perspective on their applicability in real-world scenarios. The summarized performance results are presented in Table [X], highlighting the trade-offs between model complexity, accuracy, and computational efficiency.

ML Models	MAE	MSE	R^2	Training Time	Model Inference
Linear Regression	47.4008	3763.9589	0.9215	Very Fast	Underfit Likely
Polynomial Regression	11.2462	335.7254	0.9930	Fast	Overfit Risk
Random Forest	11.2468	333.0407	0.9931	Moderate	Stable and Reliable
XGBoost	7.7519	161.9130	0.9966	Fast and Scalable	High Accuracy
Simple Transformer (Training)	0.0401	0.0036	0.9963	Slow	Very Accurate
Simple Transformer (Testing)	0.6366	0.5893	0.4206	-	Poor Generalization

Conclusion

This study explored multiple machine learning approaches—including Linear Regression, Polynomial Regression, Random Forest, XGBoost, and Transformer-based neural networks—for predicting wind turbine power output from SCADA data. Each model was evaluated using standard performance metrics such as MAE, MSE, and R^2, across both training and testing datasets. While the Transformer model exhibited excellent training performance ($R^2 \approx 0.996$), its generalization to unseen data remained suboptimal due to limited batch-wise training and computational constraints. This suggests that although deep learning models can fit the data very well, their performance is highly sensitive to training strategies, data volume, and resource availability.

In contrast, **XGBoost demonstrated a compelling balance between accuracy, training speed, and computational efficiency**, outperforming other models on test data without requiring extensive tuning or GPU acceleration. Its ability to handle non-linear patterns and outliers effectively, while delivering robust and interpretable results, makes it particularly suitable for wind power forecasting tasks.

Given these findings, **XGBoost is identified as the most practical and high-performing model for the wind turbine power prediction problem**—especially in resource-constrained environments or real-time deployment scenarios. However, with further refinement and full-scale training, deep learning models like Transformers may still prove useful in cases requiring sequential modeling or temporal awareness.

Future work will involve expanding the dataset, incorporating temporal embeddings or attention-based sequence encoders, and evaluating hybrid architectures that combine the interpretability of tree-based models with the flexibility of neural networks.

Reference:

[1]: DataSet: https://data.mendeley.com/datasets/tm988rs48k/2
[2]: XGBoost Tutorial: https://www.kaggle.com/code/dansbecker/xgboost
[3]: Linear Regression:
https://www.geeksforgeeks.org/machine-learning/python-linear-regression-using-sklearn/
[4]: Polynomial Regression:
https://data36.com/polynomial-regression-python-scikit-learn/
[5]: Standardization and Normalization:
https://youtu.be/sxEqtjLC0aM?si=u40N1_CjFTtwW88l
[6]: XGBoost Visualization: https://youtu.be/TyvYZ26alZs?si=SQIIAheYV29kMtXq
[7]: Tools and Libraries Used: Jupyter Notebook, Python, Numpy, Pandas, Matplotlib, Seaborn, PyTorch, XGBoost, SciKit-Learn.