# SOLAR POWER FORECASTING USING MACHINE LEARNING

PREDICTING DC POWER GENERATION WITH XGBOOST

**PRESENTED BY**: DONN BRYAN JULIAN

# OUTLINE

- **Executive Summary**
- **Introduction**
- **Data Collection and Wrangling**
- **Methodology**
  - Exploratory Data Analysis (EDA)
  - Feature Engineering
  - Model Selection
  - Hyperparameter Tuning
  - Cross-Validation
- **Results**
  - Model Performance
  - Feature Importance
- **Conclusion**
- **Appendix**

# EXECUTIVE SUMMARY

- **Objective**:

  Predict solar power generation using machine learning techniques to optimize energy production efficiency.

- **Key Findings**:

  **Best Model**: XGBoost, with a final $R^2$ of 0.9608 and MSE of 631,631.

  **Feature Importance**: AMBIENT_TEMPERATURE and DAILY_YIELD were the most critical predictors.

  **Cross-Validation**: The model performed well with a mean cross-validation MSE of 706,671, confirming its robustness.

- **Outcome**:

  The model provides reliable forecasts for solar power generation, suitable for real-time use.

# INTRODUCTION

- **Project Goal**:

  To forecast DC power generation using environmental factors like temperature and solar radiation.

- **Why Solar Power Forecasting?**:

  Improves energy distribution and grid management.

  Optimizes solar power plant operations, minimizing energy waste.

- **Data Used**:

  **Data Source**: Solar Power Generation Data

  **Why Focus on Plant 1?**: Plant 2 had inconsistent data with lower correlation to DC_POWER, reducing its predictive power.

# DATA COLLECTION AND WRANGLING

**Step-by-Step Process**

**Data Collection:**

Source: Solar Power Generation Data from Kaggle.

Data Structure: The dataset contains records from two solar plants, each with variables such as DC_POWER, AC_POWER, AMBIENT_TEMPERATURE, and IRRADIATION.

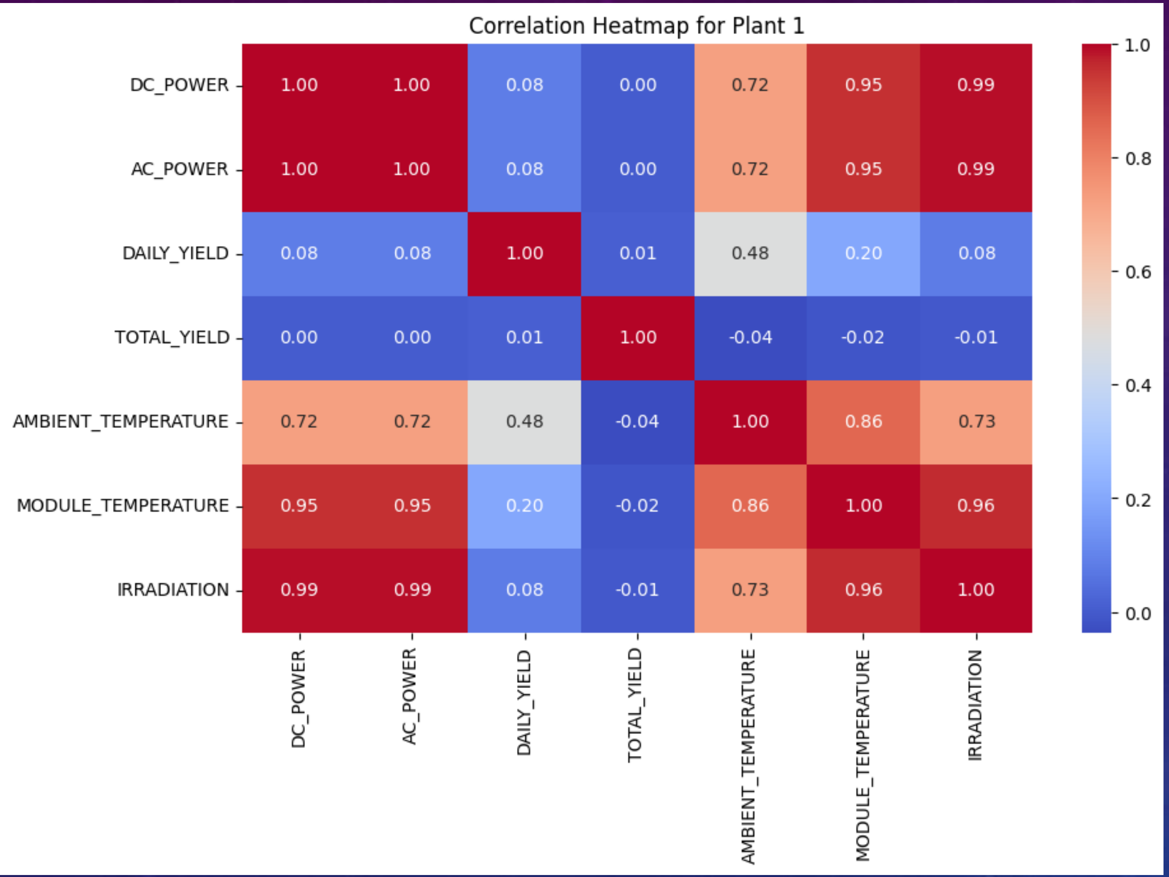
Reason for Focusing on Plant 1: Plant 2 had inconsistent and lower-quality data, leading to unreliable predictions, so we excluded it to ensure accuracy and reliability.

ETL Process Flowchart: Solar Power Generation Data

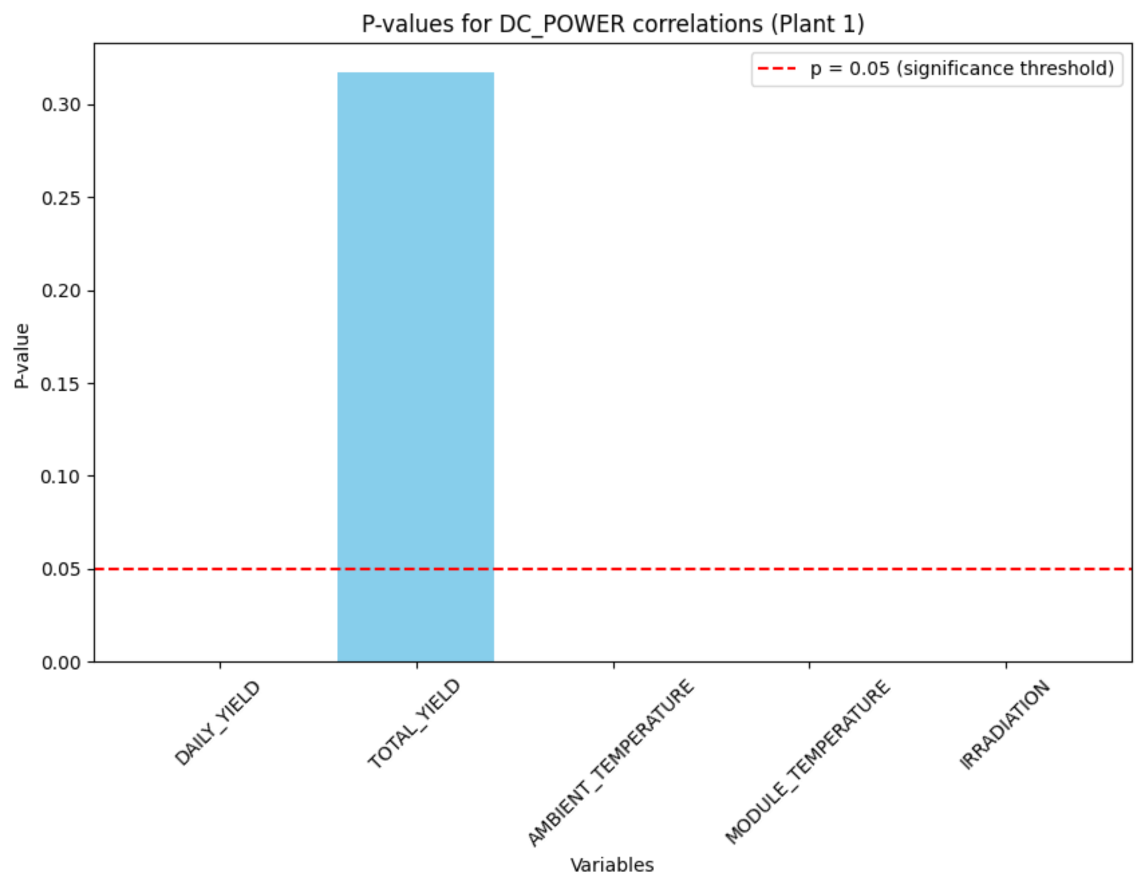
Extract:
Load dataset from Kaggle
↓
Check for missing values, duplicates, and outliers

Transform:
Remove irrelevant columns (e.g., AC_POWER)
↓
Feature engineering (add lagged variables for DAILY_YIELD and TOTAL_YIELD)
↓
Normalize continuous variables (e.g., AMBIENT_TEMPERATURE, IRRADIATION)

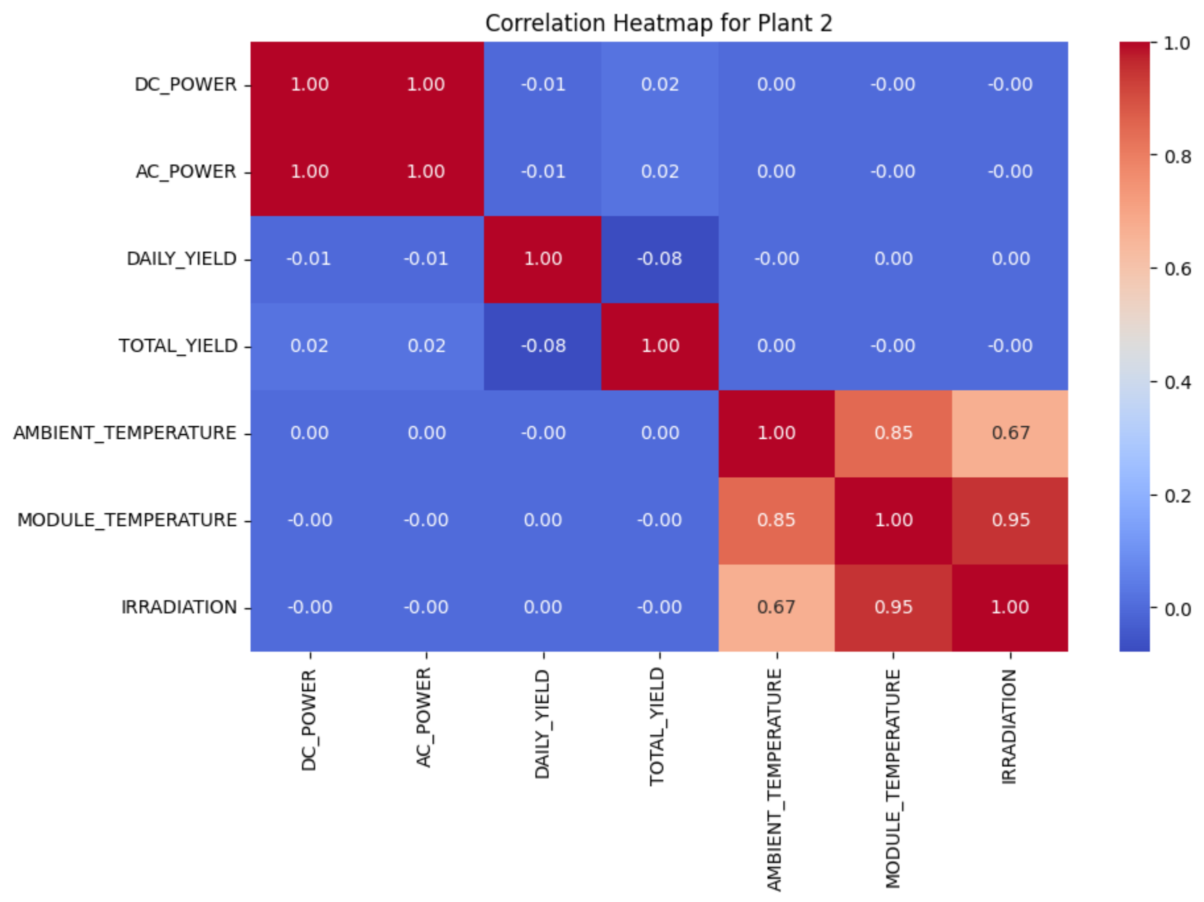
Load:
Prepare cleaned and transformed data for modeling
↓
Validate data quality and confirm readiness for analysis

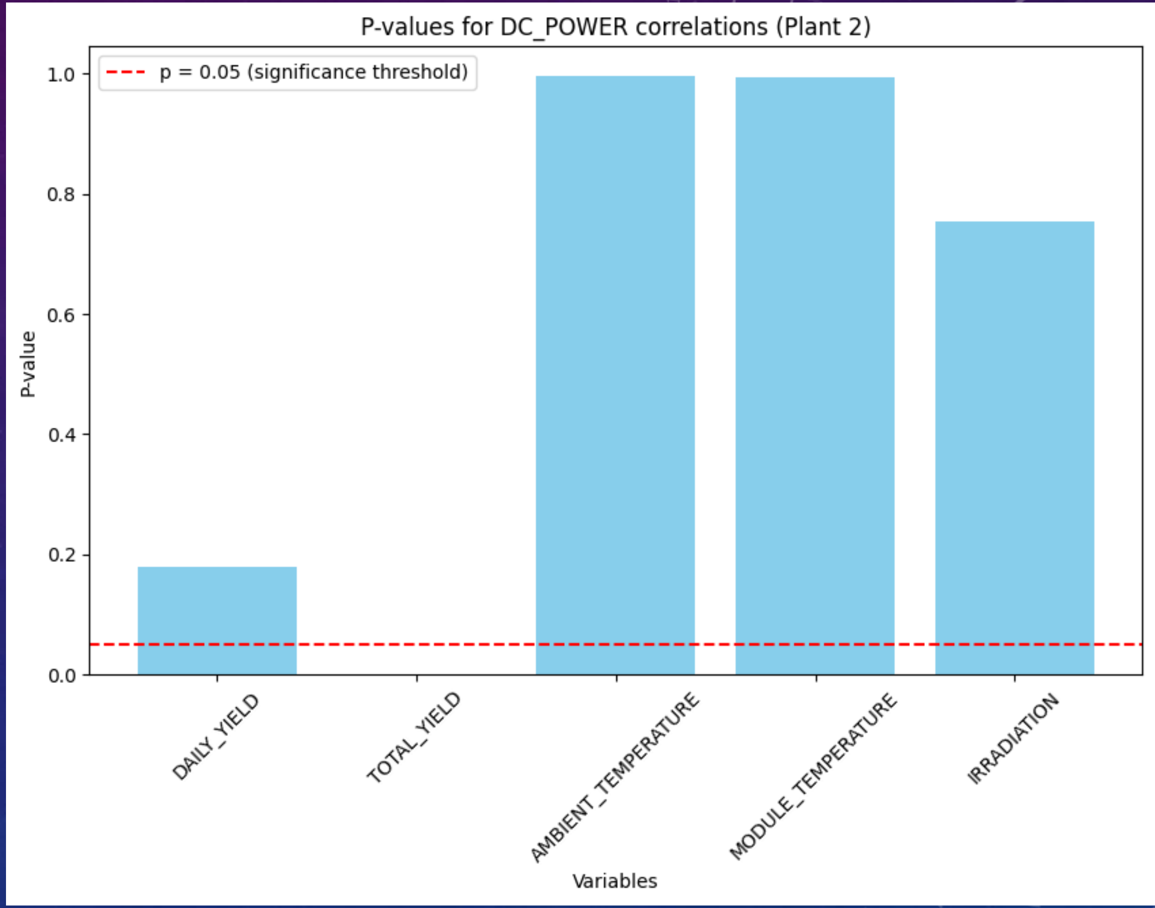


Correlation Heatmap for Plant 1
DC_POWER
AC_POWER
DAILY_YIELD
TOTAL_YIELD
AMBIENT_TEMPERATURE
MODULE_TEMPERATURE
IRRADIATION
1.00 1.00 0.08 0.00 0.72 0.95 0.99
1.00 1.00 0.08 0.00 0.72 0.95 0.99
0.08 0.08 1.00 0.01 0.48 0.20 0.08
0.00 0.00 0.01 1.00 -0.04 -0.02 -0.01
0.72 0.72 0.48 -0.04 1.00 0.86 0.73
0.95 0.95 0.20 -0.02 0.86 1.00 0.96
0.99 0.99 0.08 -0.01 0.73 0.96 1.00
1.0
0.8
0.6
0.4
0.2
0.0

P-values for DC_POWER correlations (Plant 1)
p = 0.05 (significance threshold)
P-value
0.30
0.25
0.20
0.15
0.10
0.05
0.00
DAILY_YIELD
TOTAL_YIELD
AMBIENT_TEMPERATURE
MODULE_TEMPERATURE
IRRADIATION
Variables


Correlation Heatmap for Plant 2
DC_POWER
AC_POWER
DAILY_YIELD
TOTAL_YIELD
AMBIENT_TEMPERATURE
MODULE_TEMPERATURE
IRRADIATION
1.00 1.00 -0.01 0.02 0.00 -0.00 -0.00
1.00 1.00 -0.01 0.02 0.00 -0.00 -0.00
-0.01 -0.01 1.00 -0.08 -0.00 0.00 0.00
0.02 0.02 -0.08 1.00 0.00 -0.00 -0.00
0.00 0.00 -0.00 0.00 1.00 0.85 0.67
-0.00 -0.00 0.00 -0.00 0.85 1.00 0.95
-0.00 -0.00 0.00 -0.00 0.67 0.95 1.00
1.0
0.8
0.6
0.4
0.2
0.0


P-values for DC_POWER correlations (Plant 2)
p = 0.05 (significance threshold)
P-value
1.0
0.8
0.6
0.4
0.2
0.0
DAILY_YIELD
TOTAL_YIELD
AMBIENT_TEMPERATURE
MODULE_TEMPERATURE
IRRADIATION
Variables

**Data Wrangling:**

Missing Values: While the dataset had no missing values, we ran validation checks to confirm data quality.

Feature Selection: Focused on key features like DAILY_YIELD, TOTAL_YIELD, and temperature-related metrics, which showed high relevance to DC_POWER.

Reasoning for Removing AC_POWER: It showed weak correlation with DC_POWER, which justified its exclusion to reduce model complexity.

# VARIABLES USED IN THE MODEL

**Key Variables**

- **Target Variable**:

    **DC_POWER**: The model predicts this based on other features.

- **Predictor Variables**:

    **DAILY_YIELD**: Daily energy output.

    **TOTAL_YIELD**: Cumulative energy produced.

    **AMBIENT_TEMPERATURE**: External temperature affecting efficiency.

    **MODULE_TEMPERATURE**: Solar panel temperature.

    **IRRADIATION**: Solar radiation received.

    **Lagged Variables**: Added DAILY_YIELD_LAG1, TOTAL_YIELD_LAG1 to enhance time-series prediction.

# METHODOLOGY - EDA & FEATURE ENGINEERING

**Exploratory Data Analysis (EDA):**

•**Correlation Heatmaps**: Revealed strong relationships between features and the target variable DC_POWER, particularly for temperature and irradiation.

•**Multicollinearity Check**: Handled through variance inflation factor (VIF), ensuring no multicollinearity issues in the features.

**Feature Engineering:**

•**Reasoning**: Added lagged features to capture time-series behavior in solar power generation. This helps the model account for temporal patterns.

# METHODOLOGY - MODEL DIAGNOSTICS

- **Model Diagnostics and Assumption Testing:**
- **Breusch-Pagan Test for Heteroskedasticity**:

  Ensures constant variance in residuals, which is essential for accurate predictions.

  **Result**: The Breusch-Pagan p-value was extremely small (4.99e-304), indicating the presence of heteroskedasticity. To address this, heteroscedasticity-robust standard errors were used.
- **Variance Inflation Factor (VIF) for Multicollinearity**:

  Measures collinearity among predictor variables. High VIF values indicate problematic multicollinearity.

  **Result**:

  - **DAILY_YIELD** and **TOTAL_YIELD** had acceptable VIF scores (<5), indicating low multicollinearity.
  - **AMBIENT_TEMPERATURE**, **MODULE_TEMPERATURE**, and **IRRADIATION** had high VIF scores, with **MODULE_TEMPERATURE** and **IRRADIATION** being the most problematic (VIF > 10). Multicollinearity needed attention.
- **Durbin-Watson Test for Autocorrelation**:
  - Checks if residuals are independent. A value close to 2 indicates no autocorrelation.
  - **Result**: The Durbin-Watson statistic was 1.32, which suggests slight positive autocorrelation in the residuals.

```python
# Ensure that the data contains only numeric values
X = merged_data_plant_1.drop(columns=['DC_POWER'])
X = X.select_dtypes(include=[np.number])  # Keep only numeric columns
X = sm.add_constant(X)  # Add intercept

y = merged_data_plant_1['DC_POWER']

# Fit the model
model = sm.OLS(y, X).fit()

# 1. Breusch-Pagan test for heteroskedasticity
_, pval_breusch, _, _ = het_breuschpagan(model.resid, model.model.exog)
print(f'Breusch-Pagan p-value: {pval_breusch}')

# 2. VIF for multicollinearity
vif_data = pd.DataFrame()
vif_data['Variable'] = X.columns
vif_data['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
print("\nVIF:")
print(vif_data)

# 3. Durbin-Watson test for autocorrelation
dw_stat = durbin_watson(model.resid)
print(f'\nDurbin-Watson statistic: {dw_stat}')
```

```
Breusch-Pagan p-value: 4.989168496217311e-304

VIF:
              Variable         VIF
0                const  485.661682
1          DAILY_YIELD    1.656402
2          TOTAL_YIELD    1.004786
3  AMBIENT_TEMPERATURE    9.172826
4   MODULE_TEMPERATURE   44.770729
5          IRRADIATION   24.786823

Durbin-Watson statistic: 1.3177611907434237
```

# METHODOLOGY - MODEL REFINEMENT

Model Refinement and Feature Engineering:

- Handling Multicollinearity:

    Removed MODULE_TEMPERATURE and IRRADIATION due to high VIF values, indicating strong multicollinearity.

    Rationale: Keeping multicollinear features would distort the model, so eliminating them improved clarity in feature impact.

- Introduction of Lagged Variables:

    Added a lagged feature (DAILY_YIELD_LAG1) to capture temporal effects on DC_POWER.

    Rationale: Lagged variables help improve the model by considering the impact of past energy yields on current power generation.

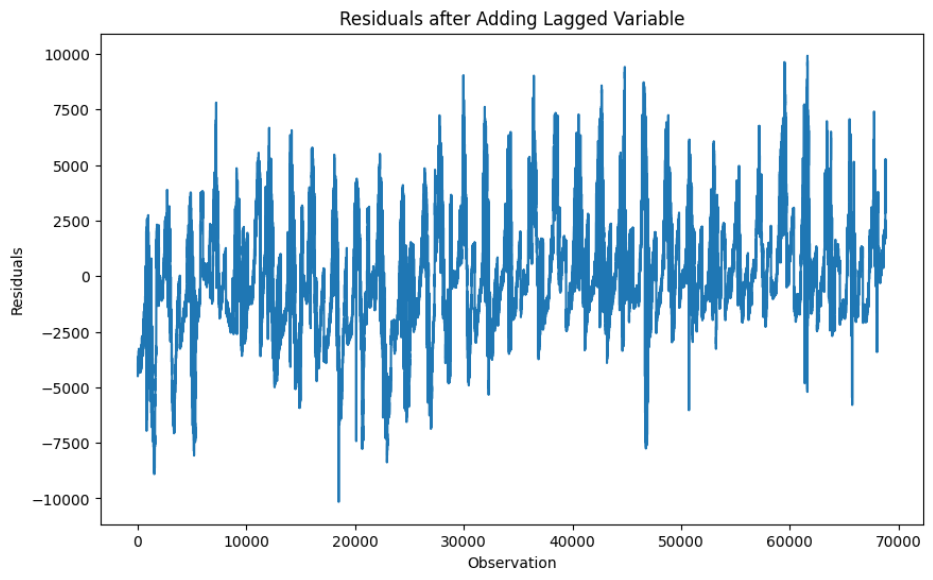
- Revised Model Fit:

    After making these changes, the model was refitted using heteroscedasticity-robust standard errors to account for any residual heteroskedasticity.

    Visual Representation: A plot of residuals after adding lagged variables shows improved consistency.

```python
X = merged_data_plant_1.drop(columns=['DC_POWER', 'MODULE_TEMPERATURE', 'IRRADIATION', 'DATE_TIME'])
y = merged_data_plant_1['DC_POWER']

X = X.apply(pd.to_numeric, errors='coerce')
X = X.dropna()
y = y.loc[X.index]

X = sm.add_constant(X)

model_robust = sm.OLS(y, X).fit(cov_type='HC0')
print("Initial Model Summary:")
print(model_robust.summary())

vif = pd.DataFrame()
vif["Feature"] = X.columns
vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
print("\nVariance Inflation Factors (VIF):")
print(vif)

X['DAILY_YIELD_LAG1'] = X['DAILY_YIELD'].shift(1)
X = X.dropna()
y = y.loc[X.index]

model_lag = sm.OLS(y, X).fit(cov_type='HC0')
print("\nModel Summary After Adding Lagged Variable:")
print(model_lag.summary())

plt.figure(figsize=(10, 6))
plt.plot(model_lag.resid)
plt.title('Residuals after Adding Lagged Variable')
plt.xlabel('Observation')
plt.ylabel('Residuals')
plt.show()
```

```
Initial Model Summary:
                            OLS Regression Results
==============================================================================
Dep. Variable:               DC_POWER   R-squared:                       0.617
Model:                            OLS   Adj. R-squared:                  0.617
Method:                 Least Squares   F-statistic:                 2.827e+04
Date:                Wed, 25 Sep 2024   Prob (F-statistic):               0.00
Time:                        02:25:41   Log-Likelihood:            -6.3588e+05
No. Observations:               68803   AIC:                         1.272e+06
Df Residuals:                   68799   BIC:                         1.272e+06
Df Model:                           3
Covariance Type:                  HC0
=======================================================================================
                          coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------------
const               -2.539e+04    182.127   -139.394      0.000   -2.57e+04    -2.5e+04
DAILY_YIELD            -0.4416      0.003   -149.237      0.000      -0.447      -0.436
TOTAL_YIELD             0.0004   2.27e-05     16.808      0.000       0.000       0.000
AMBIENT_TEMPERATURE  1069.1488      3.696    289.261      0.000    1061.905    1076.393
==============================================================================
Omnibus:                     1097.355   Durbin-Watson:                   0.133
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1791.038
Skew:                           0.149   Prob(JB):                         0.00
Kurtosis:                       3.732   Cond. No.                     1.33e+08
==============================================================================
...
Notes:
[1] Standard Errors are heteroscedasticity robust (HC0)
[2] The condition number is large, 1.33e+08. This might indicate that there are
strong multicollinearity or other numerical problems.
```



# METHODOLOGY - RESIDUAL ANALYSIS

- Residual Diagnostics for Final Model:
- Residual Analysis:

  Analyzed the residuals of the final ARIMA model to ensure they follow a normal distribution and do not exhibit patterns over time.

  Why: Checking residuals ensures that the model assumptions of independence, homoscedasticity, and normality hold, which is crucial for accurate forecasting.

- Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF):

  The ACF and PACF plots help detect autocorrelation in the residuals, which could indicate that the model hasn't captured all the information in the data.

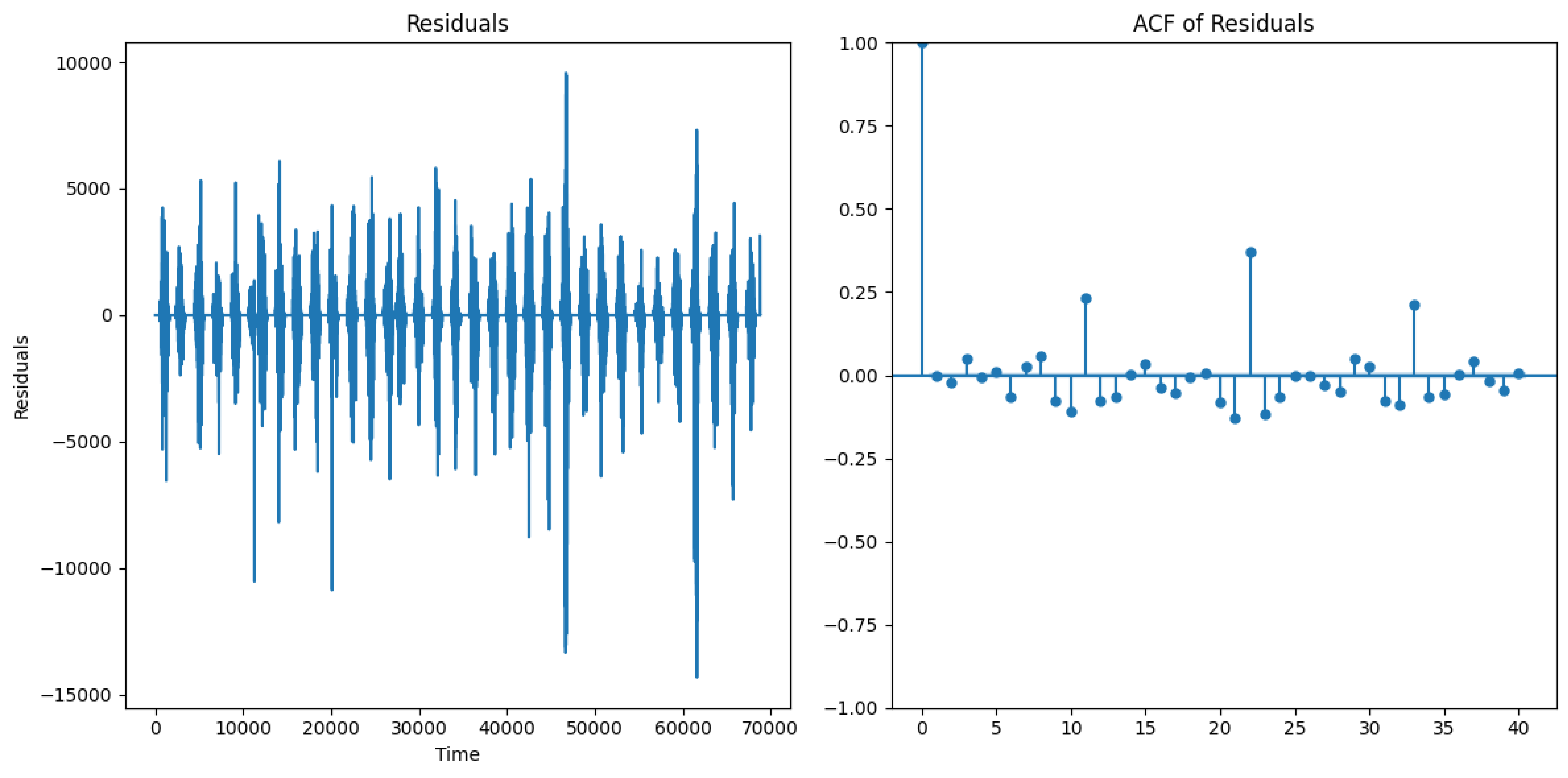
  Why: Residuals should ideally have no significant autocorrelation, confirming the model is well-specified.

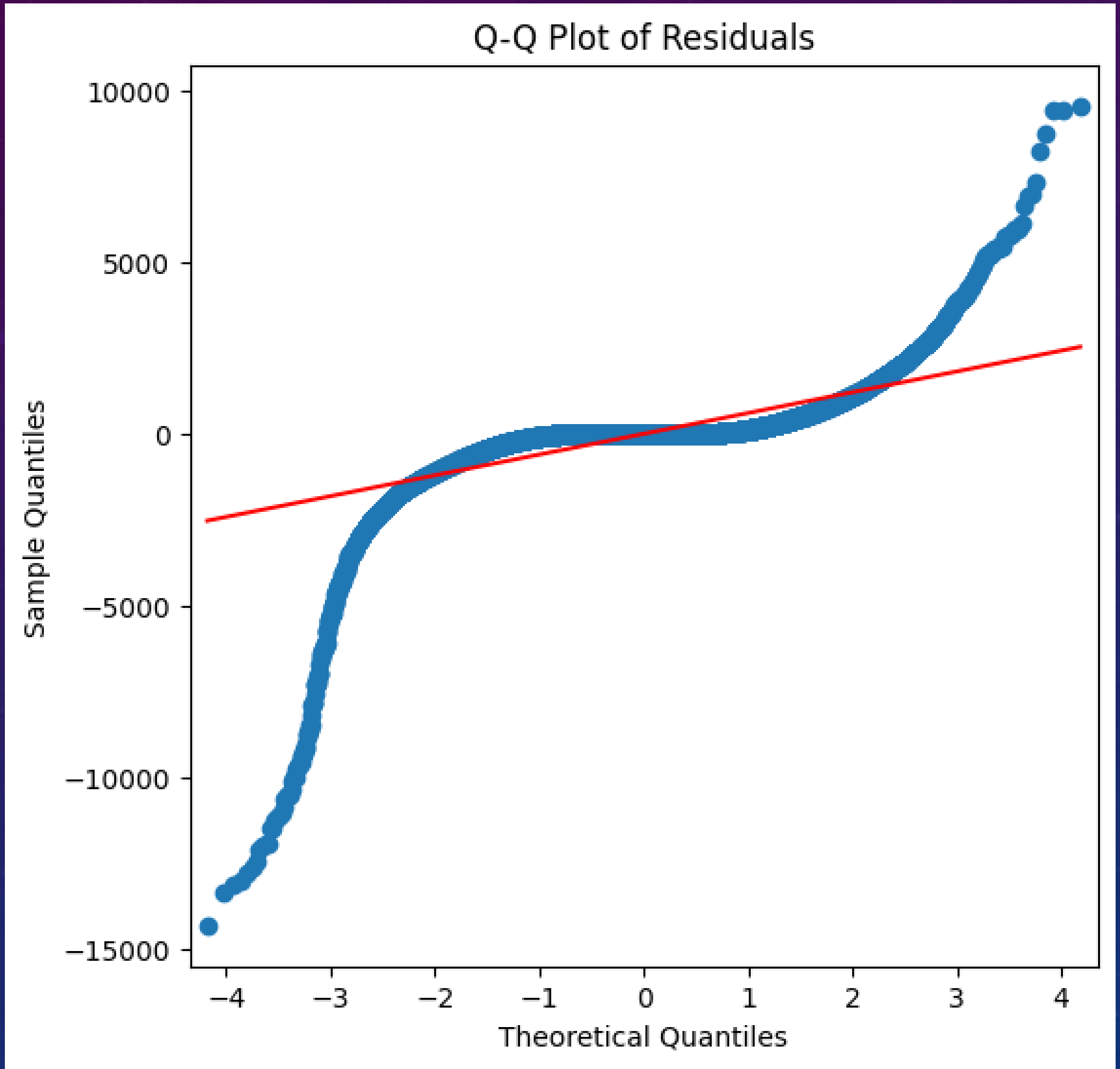
- Normality Check (Q-Q Plot and Jarque-Bera Test):

  Used a Q-Q plot and performed the Jarque-Bera test to assess whether residuals are normally distributed.

  Result: (Include Jarque-Bera result to explain if the residuals follow a normal distribution.)

  Why: Normality of residuals helps validate that the model is reliable and robust for inference.


Residuals
10000
5000
0
−5000
−10000
−15000
Residuals
0
10000
20000
30000
40000
50000
60000
70000
Time
ACF of Residuals
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30
35
40



Q-Q Plot of Residuals
10000
5000
0
−5000
−10000
−15000
Sample Quantiles
−4
−3
−2
−1
0
1
2
3
4
Theoretical Quantiles

# METHODOLOGY - MODEL REFINEMENT (LAGGED VARIABLES)

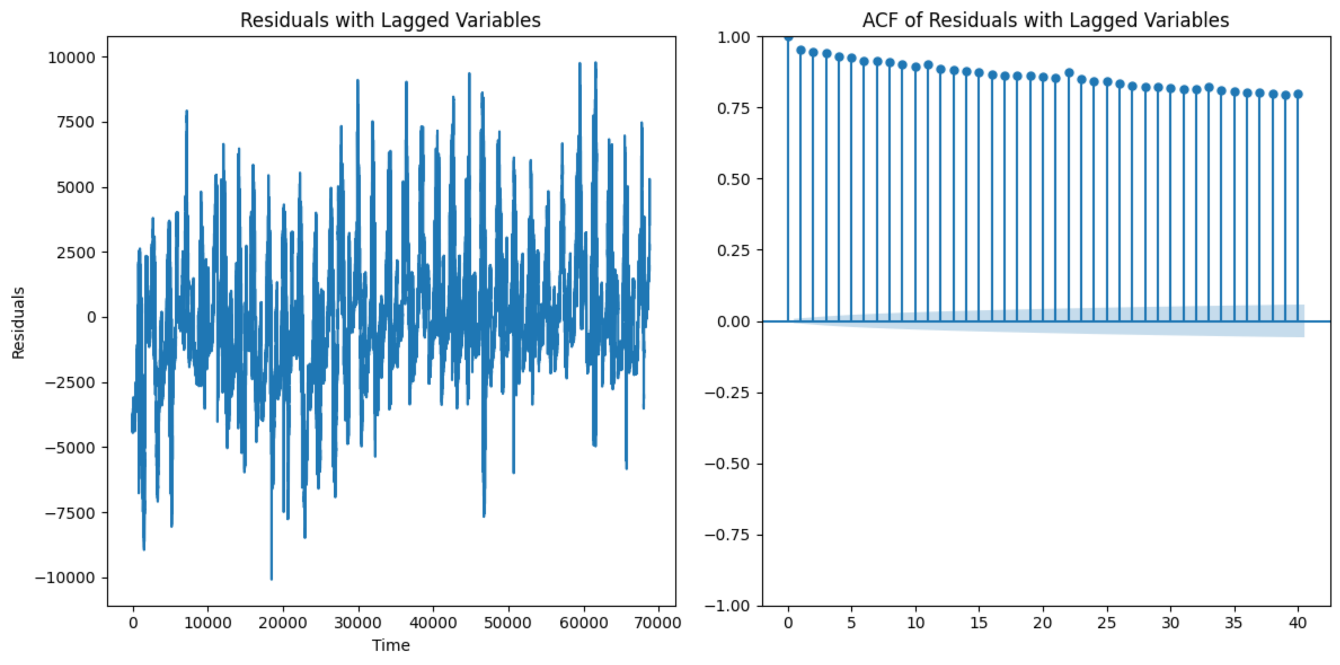
**Introducing Lagged Variables:**

- **Feature Engineering:**
  - Added lagged variables for DAILY_YIELD and TOTAL_YIELD to capture temporal dependencies in the data.
  - Why: Energy production often depends on past values, so incorporating lagged variables helps improve predictive power by capturing these relationships.
- **Model Refinement:**
  - Refit the model with the newly added lagged variables.
  - Why: The addition of lagged variables aims to improve the model's ability to predict DC_POWER by considering past values of key features.

**Residual Diagnostics After Lagging:**

- **Residuals with Lagged Variables:**
  - Residuals were analyzed again to check for patterns after adding the lagged variables.
  - Why: It's essential to ensure that adding lagged variables improves the model's performance without introducing new issues like autocorrelation.
- **Autocorrelation Check (ACF):**
  - ACF plot of residuals was used to confirm whether autocorrelation was still present.
  - Why: Ideally, residuals should have no significant autocorrelation, confirming that the model captures the data well.

```python
# Add lagged variables
X['DAILY_YIELD_LAG1'] = X['DAILY_YIELD'].shift(1)
X['DAILY_YIELD_LAG2'] = X['DAILY_YIELD'].shift(2)
X['TOTAL_YIELD_LAG1'] = X['TOTAL_YIELD'].shift(1)

# Drop rows with NaN values from lagging
X = X.dropna()

# Refit the model
model_lagged = sm.OLS(y.loc[X.index], X).fit(cov_type='HC0')
print(model_lagged.summary())

# Plot residuals and ACF
residuals_lagged = model_lagged.resid
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(residuals_lagged)
plt.title('Residuals with Lagged Variables')
plt.xlabel('Time')
plt.ylabel('Residuals')

# ACF of residuals
plt.subplot(1, 2, 2)
plot_acf(residuals_lagged, lags=40, ax=plt.gca())
plt.title('ACF of Residuals with Lagged Variables')
plt.tight_layout()
plt.show()
```



# METHODOLOGY - MODEL SELECTION

Models Tested:

1.Random Forest

2.Gradient Boosting

3.XGBoost

4.Support Vector Machines

5.Linear Regression

Selection Criteria:

Mean Squared Error (MSE): Measures prediction accuracy.

R-squared: Measures how well the model explains the variance in DC_POWER.

Best Model: XGBoost performed the best, providing a balance between accuracy and computational efficiency.

```python
# Initialize models
models = {
    'Random Forest': RandomForestRegressor(n_estimators=100, random_state=42),
    'Linear Regression': LinearRegression(),
    'Gradient Boosting': GradientBoostingRegressor(n_estimators=100, random_state=42),
    'Decision Tree': DecisionTreeRegressor(random_state=42)
}

# Store results
results = {}

# Train and evaluate each model
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Evaluate
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    results[name] = {'MSE': mse, 'R²': r2}

# Print results
for name, result in results.items():
    print(f'{name}:')
    print(f"   Mean Squared Error: {result['MSE']}")
    print(f"   R-squared: {result['R²']}")
```

```
Random Forest:
   Mean Squared Error: 544483.758569032
   R-squared: 0.9662284211825973
Linear Regression:
   Mean Squared Error: 6106786.111340532
   R-squared: 0.6212268865059101
Gradient Boosting:
   Mean Squared Error: 2241848.612025071
   R-squared: 0.8609494481586258
Decision Tree:
   Mean Squared Error: 1007078.7118486665
   R-squared: 0.9375359915566452
```

```python
# Set up the models to try
models = {
    'XGBoost': XGBRegressor(n_estimators=100, random_state=42),
    'LightGBM': LGBMRegressor(n_estimators=100, random_state=42),
    'SVM': SVR(kernel='rbf')  # Using rbf kernel to capture non-linear trends
}

# Track the performance
results = {}

# Train and evaluate each model
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

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

    results[name] = {'MSE': mse, 'R-squared': r2}

# Display results
for name, metrics in results.items():
    print(f"{name}:")
    print(f"   MSE: {metrics['MSE']}")
    print(f"   R-squared: {metrics['R-squared']}")
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000909 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1530
[LightGBM] [Info] Number of data points in the train set: 55040, number of used features: 6
[LightGBM] [Info] Start training from score 3151.818499
XGBoost:
   MSE: 1143139.5546690864
   R-squared: 0.9290968243547654
LightGBM:
   MSE: 1476544.7468729883
   R-squared: 0.9084173834174691
SVM:
   MSE: 23405959.469281018
   R-squared: -0.45175350517556057
```

# WHY WE CHOSE XGBOOST OVER RANDOM FOREST

1. Performance and Speed:
   - XGBoost is optimized for speed and performance, using gradient boosting to iteratively improve predictions.
   - Faster and more efficient for large datasets compared to Random Forest.
2. Built-in Regularization:
   - XGBoost incorporates L1 and L2 regularization, which helps prevent overfitting by penalizing model complexity.
   - Random Forest lacks this built-in feature, making XGBoost more robust when overfitting is a concern.
3. Handling Imbalanced Data:
   - XGBoost provides parameters like scale_pos_weight to better manage imbalanced datasets.
   - This improves the model's performance on datasets where one class dominates.
4. Advanced Hyperparameter Tuning:
   - XGBoost offers more customization and tuning options (e.g., learning_rate, max_depth, n_estimators).
   - These options allow for a more refined and optimized model compared to Random Forest.
5. Cross-Validation and Early Stopping:
   - XGBoost includes early stopping to halt training when performance stops improving, avoiding overfitting.
   - Random Forest doesn't have this built-in functionality, requiring more manual checks.
6. Handling Missing Data:
   - XGBoost can handle missing values automatically by learning how to manage missing data points.
   - Random Forest requires more manual data preprocessing to handle missing data effectively.

# METHODOLOGY - HYPERPARAMETER TUNING

Why Tuning?:

Avoiding Overfitting: Hyperparameter tuning helps prevent the model from becoming too complex.

Improving Accuracy: Tuned parameters like max_depth, learning_rate, and n_estimators led to a significant performance boost.

Tuned XGBoost Parameters:

Max Depth: 10, Learning Rate: 0.1, Estimators: 500, Subsample: 0.8.

# HYPERPARAMETER TUNING WITH GRIDSEARCHCV

Tuning XGBoost for Optimal Performance

•Why: Tuning hyperparameters allows us to optimize the model's performance by finding the best combination of parameters that minimize error.

Steps Taken:

•Grid Search:

We defined a parameter grid that included different values for n_estimators, max_depth, learning_rate, subsample, and colsample_bytree.

Why: These parameters control how the XGBoost model grows decision trees and combines them. By systematically testing combinations, we find the most effective setup.

•Cross-Validation:

Applied 3-fold cross-validation within the grid search to ensure the model performs well across different data splits.

Why: Cross-validation reduces the risk of overfitting by evaluating the model on multiple training/validation splits.

Results:

•Best Parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 500, 'subsample': 0.8}

These were the parameters that minimized the mean squared error (MSE) across the training data.

•Test Set Evaluation:

•After tuning, the final model was evaluated on the test set:

Tuned XGBoost MSE: 631631.9427001182

Tuned XGBoost R-squared: R-squared: 0.9608230592726087Conclusion:

•Why Tuning Matters: The hyperparameter tuning process helped improve the model’s performance by finding the best parameter configuration.

```python
# Define parameter grid for tuning
param_grid = {
    'n_estimators': [100, 300, 500],
    'max_depth': [3, 5, 10],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0]
}

# Initialize XGBoost model
xgb_model = XGBRegressor(random_state=42)

# Set up GridSearchCV for tuning
xgb_cv = GridSearchCV(estimator=xgb_model, param_grid=param_grid,
                      cv=3, scoring='neg_mean_squared_error', verbose=2, n_jobs=-1)

# Fit the model
xgb_cv.fit(X_train, y_train)

# Get the best parameters and performance
print(f"Best Parameters: {xgb_cv.best_params_}")

# Make predictions with the best estimator
best_xgb = xgb_cv.best_estimator_
y_pred_xgb = best_xgb.predict(X_test)

# Evaluate the tuned model
mse_xgb = mean_squared_error(y_test, y_pred_xgb)
r2_xgb = r2_score(y_test, y_pred_xgb)

print(f"Tuned XGBoost MSE: {mse_xgb}")
print(f"Tuned XGBoost R-squared: {r2_xgb}")
```

```
Best Parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 500, 'subsample': 0.8}
Tuned XGBoost MSE: 631631.9427001182
Tuned XGBoost R-squared: 0.9608230592726087
```

# METHODOLOGY - CROSS-VALIDATION

- **Why Cross-Validation?:**

**Reasoning**: Ensures the model's generalizability by testing its performance on multiple splits of the data.

- **Cross-Validation Results:**

**Mean MSE**: 706,671.

**Consistency**: The model performed well across all data splits, confirming its robustness.

```python
# Finalize the XGBoost model with the best parameters
final_xgb_model = XGBRegressor(
    colsample_bytree=0.8,
    learning_rate=0.1,
    max_depth=10,
    n_estimators=500,
    subsample=0.8,
    random_state=42
)

# Perform cross-validation with the finalized XGBoost model
cv_scores = cross_val_score(final_xgb_model, X_train, y_train, cv=5, scoring='neg_mean_squared_error')

# Convert negative MSE to positive
cv_mse_scores = -cv_scores

# Calculate the mean and standard deviation of MSE from cross-validation
mean_cv_mse = cv_mse_scores.mean()
std_cv_mse = cv_mse_scores.std()

# Output the results
print(f"Cross-Validation MSE scores: {cv_mse_scores}")
print(f"Mean Cross-Validation MSE: {mean_cv_mse}")
print(f"Standard Deviation of MSE: {std_cv_mse}")

# Fit the final model on the training data
final_xgb_model.fit(X_train, y_train)

# Evaluate on the test set
y_pred = final_xgb_model.predict(X_test)
test_mse = mean_squared_error(y_test, y_pred)
test_r2 = r2_score(y_test, y_pred)

# Output the test performance
print(f"Test MSE: {test_mse}")
print(f"Test R-squared: {test_r2}")
```

```
Cross-Validation MSE scores: [733209.55497024 745480.8475119  670336.1335878  678990.51639581
 705337.97210963]
Mean Cross-Validation MSE: 706671.0049150748
Standard Deviation of MSE: 29321.349958405808
Test MSE: 631631.9427001182
Test R-squared: 0.9608230592726087
```

# RESULTS - FEATURE IMPORTANCE

**Top Features:**

1. **AMBIENT_TEMPERATURE**: Most important predictor of solar power generation.

2. **DAILY_YIELD**: Also highly important, capturing the day-to-day variation in energy production.

3. **Lagged Variables**: Contributed significantly to improving model accuracy.

**Why We Kept All Features:**

- Removing low-importance features led to a drop in performance, so they were retained.

# RESULTS - MODEL PERFORMANCE

**Final Model Performance (XGBoost):**

•**Test Set MSE**: 631,631.

•**R-squared**: 0.9608, indicating that the model explained 96% of the variance in DC_POWER.

**Cross-Validation:**

•Consistently low MSE values across different data splits confirmed the model's robustness.

```python
# Finalize the XGBoost model with the best parameters
final_xgb_model = XGBRegressor(
    colsample_bytree=0.8,
    learning_rate=0.1,
    max_depth=10,
    n_estimators=500,
    subsample=0.8,
    random_state=42
)

# Perform cross-validation with the finalized XGBoost model
cv_scores = cross_val_score(final_xgb_model, X_train, y_train, cv=5, scoring='neg_mean_squared_error')

# Convert negative MSE to positive
cv_mse_scores = -cv_scores

# Calculate the mean and standard deviation of MSE from cross-validation
mean_cv_mse = cv_mse_scores.mean()
std_cv_mse = cv_mse_scores.std()

# Output the results
print(f"Cross-Validation MSE scores: {cv_mse_scores}")
print(f"Mean Cross-Validation MSE: {mean_cv_mse}")
print(f"Standard Deviation of MSE: {std_cv_mse}")

# Fit the final model on the training data
final_xgb_model.fit(X_train, y_train)

# Evaluate on the test set
y_pred = final_xgb_model.predict(X_test)
test_mse = mean_squared_error(y_test, y_pred)
test_r2 = r2_score(y_test, y_pred)

# Output the test performance
print(f"Test MSE: {test_mse}")
print(f"Test R-squared: {test_r2}")
```

```
Cross-Validation MSE scores: [733209.55497024 745480.8475119  670336.1335878  678990.51639581
 705337.97210963]
Mean Cross-Validation MSE: 706671.0049150748
Standard Deviation of MSE: 29321.349958405808
Test MSE: 631631.9427001182
Test R-squared: 0.9608230592726087
```

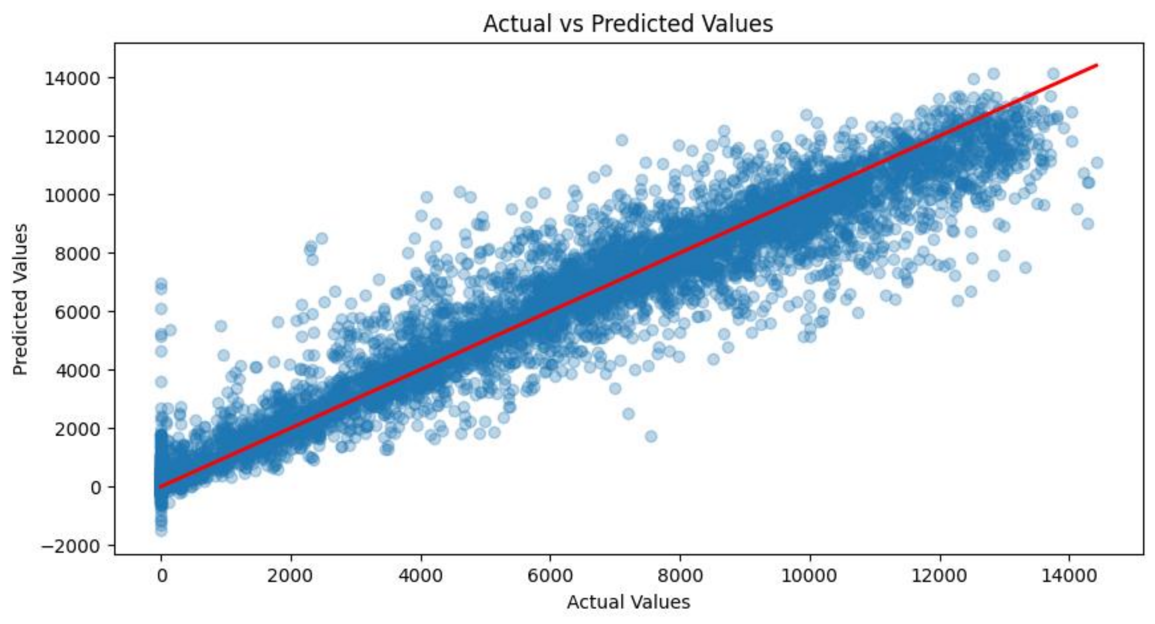
# VISUALIZATIONS

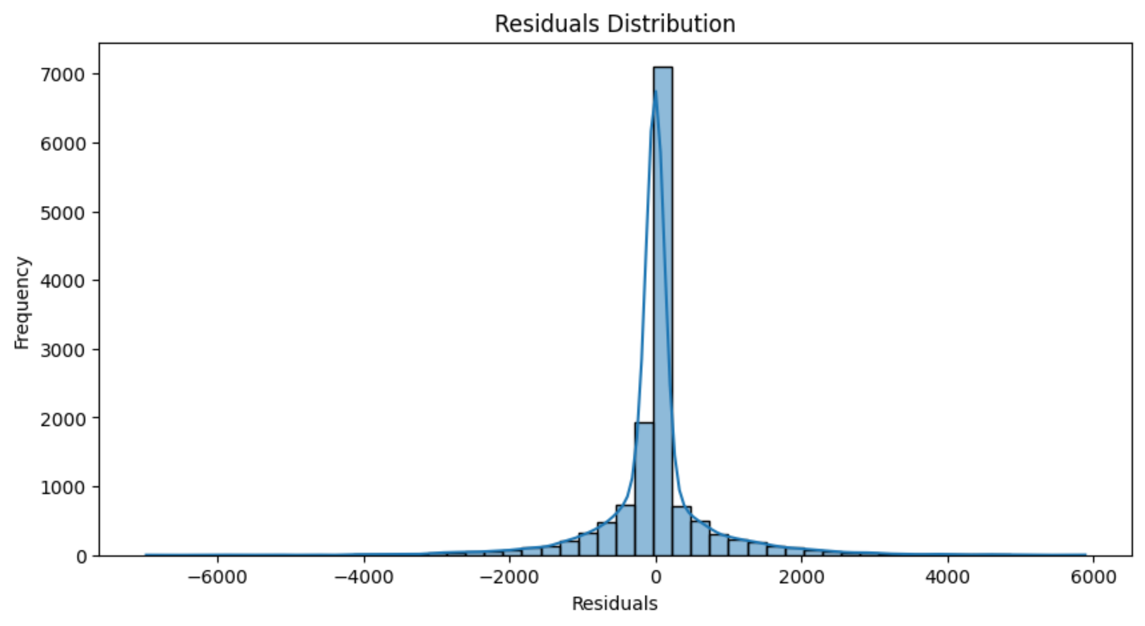
**Actual vs Predicted Plot:**

- **Interpretation**: The plot shows a close alignment between the actual and predicted values, indicating strong model performance.

**Residual Plot:**

- **Interpretation**: The residuals are evenly distributed around zero, which means the model is making accurate predictions without systematic errors.


Actual vs Predicted Values
Predicted Values
Actual Values
14000
12000
10000
8000
6000
4000
2000
0
−2000
0
2000
4000
6000
8000
10000
12000
14000


Residuals Distribution
Frequency
Residuals
7000
6000
5000
4000
3000
2000
1000
0
−6000
−4000
−2000
0
2000
4000
6000

# CONCLUSION

•**Summary**:

The XGBoost model provided accurate predictions with low error rates.

**Feature Engineering** and **Hyperparameter Tuning** were key to improving model performance.

**Lagged Variables** and temperature factors played a crucial role in predictive accuracy.

•**Challenges**:

Removing Plant 2 due to data inconsistencies.

Handling multicollinearity and ensuring data quality were critical steps.

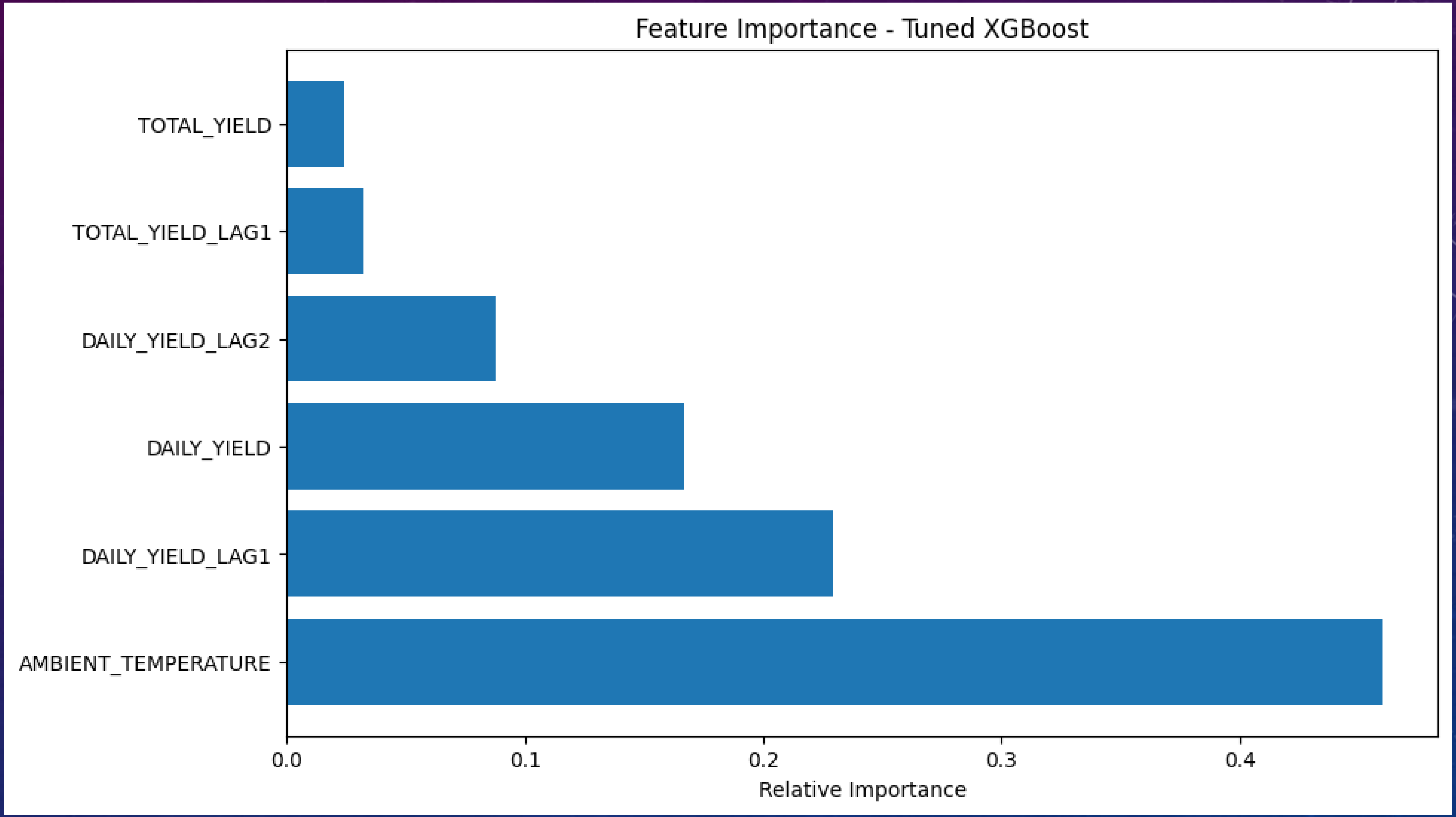
•**Next Steps**:

Deploy the model for real-time solar power forecasting.

Integrate weather forecasting data to further improve accuracy.

# APPENDIX

- **Additional Visuals**:
    - Heatmaps, feature importance plots, and residual analysis.
- **Hyperparameter Tuning Details**: Full tuning process for XGBoost and other models.
- **Data Source**: Solar Power Generation Data.


Feature Importance - Tuned XGBoost
TOTAL_YIELD
TOTAL_YIELD_LAG1
DAILY_YIELD_LAG2
DAILY_YIELD
DAILY_YIELD_LAG1
AMBIENT_TEMPERATURE
0.0
0.1
0.2
0.3
0.4
Relative Importance