AML_Projekt

January 13, 2025

1 Wind Turbine Predictive Maintenance

Classification of Error Detection of Wind Turbine from IIoT Data with the help of ML models!

2 Original Data

[11]:
```
# Importing important Python Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns



# for preprocessing
from sklearn.preprocessing import MinMaxScaler, RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, StratifiedKFold,␣
 ↪GridSearchCV
from sklearn.manifold import TSNE
from sklearn.tree import plot_tree
from imblearn.combine import SMOTEENN
from collections import Counter
from sklearn.model_selection import learning_curve
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant


#  for evaluation
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, classification_report, ConfusionMatrixDisplay,␣
 ↪make_scorer

# models
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
```

```
import xgboost as xgb



import warnings
warnings.filterwarnings('ignore', category=FutureWarning, module='sklearn')
```

2.1 Data Retrieval

[12]:
```
df = pd.read_csv('pred_maint_wind.csv')
df.head()
```

[12]:

	DateTime	Time	Error	WEC: ava. windspeed	WEC: max. windspeed	WEC: min. windspeed	WEC: ava. Rotation	WEC: max. Rotation	WEC: min. Rotation	WEC: ava. Power	…	Rectifier cabinet temp.	Yaw inverter cabinet temp.	Fan inverter cabinet temp.	Ambient temp.	Tower temp.	Control cabinet temp.	Transformer temp.
0	5/1/2014 0:00	1398920448	0	6.9	9.4	2.9	0.0	0.02	0.0	0	…	24	20	25	12	14	24	34
1	5/1/2014 0:09	1398920960	0	5.3	8.9	1.6	0.0	0.01	0.0	0	…	24	20	25	12	14	24	34
2	5/1/2014 0:20	1398921600	0	5.0	9.5	1.4	0.0	0.04	0.0	0	…	24	20	25	12	14	24	34
3	5/1/2014 0:30	1398922240	0	4.4	8.3	1.3	0.0	0.08	0.0	0	…	23	21	25	12	14	24	34
4	5/1/2014 0:39	1398922752	0	5.7	9.7	1.2	0.0	0.05	0.0	0	…	23	21	25	12	14	23	34

```
   RTU: ava. Setpoint 1  Inverter averages  Inverter std dev
0                  2501          25.272728          1.103713
1                  2501          25.272728          1.103713
2                  2501          25.272728          1.103713
3                  2501          25.272728          1.103713
4                  2501          25.272728          1.103713

[5 rows x 66 columns]
```

[13]:
```
df.info
```

[13]:
```
<bound method DataFrame.info of              DateTime        Time  Error  WEC:
ava. windspeed  \
0       5/1/2014 0:00  1398920448      0                 6.9
1       5/1/2014 0:09  1398920960      0                 5.3
2       5/1/2014 0:20  1398921600      0                 5.0
3       5/1/2014 0:30  1398922240      0                 4.4
4       5/1/2014 0:39  1398922752      0                 5.7
…                 …         …    …                  …
49022  4/8/2015 23:20  1428553216      0                 3.9
49023  4/8/2015 23:30  1428553856      0                 3.9
49024  4/8/2015 23:39  1428554368      0                 4.2
49025  4/8/2015 23:50  1428555008      0                 4.1
49026   4/9/2015 0:00  1428555648      0                 4.8

       WEC: max. windspeed  WEC: min. windspeed  WEC: ava. Rotation  \
0                      9.4                  2.9                0.00
1                      8.9                  1.6                0.00
2                      9.5                  1.4                0.00
3                      8.3                  1.3                0.00
4                      9.7                  1.2                0.00
…                      …                    …                  …
49022                  5.5                  2.2                6.75
49023                  5.6                  2.9                6.64
49024                  6.7                  2.6                7.18
49025                  6.6                  2.7                7.02
49026                  6.0                  3.3                8.39

       WEC: max. Rotation  WEC: min. Rotation  WEC: ava. Power  …  \
0                    0.02                0.00                0  …
1                    0.01                0.00                0  …
2                    0.04                0.00                0  …
3                    0.08                0.00                0  …
4                    0.05                0.00                0  …
…                    …                   …                  …  …
49022                7.40                6.01              147  …
49023                7.06                6.33              128  …
```

```
49024                8.83                6.22              163  …
49025                7.94                6.20              160  …
49026                9.48                7.14              284  …

       Rectifier cabinet temp.  Yaw inverter cabinet temp.  \
0                           24                          20
1                           24                          20
2                           24                          20
3                           23                          21
4                           23                          21
…                          …                           …
49022                       33                          23
49023                       34                          23
49024                       34                          23
49025                       33                          23
49026                       33                          22

       Fan inverter cabinet temp.  Ambient temp.  Tower temp.  \
0                              25             12           14
1                              25             12           14
2                              25             12           14
3                              25             12           14
4                              25             12           14
…                             …              …           …
49022                          28              9           17
49023                          28              9           17
49024                          28              9           18
49025                          28              9           17
49026                          28              9           17

       Control cabinet temp.  Transformer temp.  RTU: ava. Setpoint 1  \
0                         24                 34                  2501
1                         24                 34                  2501
2                         24                 34                  2501
3                         24                 34                  2501
4                         23                 34                  2501
…                        …                  …                  …
49022                     27                 35                  3050
49023                     27                 35                  3050
49024                     27                 34                  3050
49025                     27                 34                  3050
49026                     27                 34                  3050

       Inverter averages  Inverter std dev
0              25.272728          1.103713
1              25.272728          1.103713
2              25.272728          1.103713
```

```
3              25.272728          1.103713
4              25.272728          1.103713
…                    …                 …
49022          24.454546          3.474583
49023          24.454546          3.445683
49024          24.363636          3.413876
49025          24.000000          3.376389
49026          23.818182          3.250175

[49027 rows x 66 columns]>
```

[14]:
```
df.describe()
```

[14]:
```
                Time         Error  WEC: ava. windspeed  WEC: max. windspeed  \
count   4.902700e+04  49027.000000         49027.000000         49027.000000
mean    1.413762e+09      0.938748             6.874626             9.340286
std     8.559693e+06     14.442141             3.694776             5.157448
min     1.398920e+09      0.000000             0.000000             0.000000
25%     1.406352e+09      0.000000             4.200000             5.800000
50%     1.413706e+09      0.000000             6.500000             8.600000
75%     1.421179e+09      0.000000             8.900000            11.700000
max     1.428556e+09    246.000000            32.099998            51.099998

       WEC: min. windspeed  WEC: ava. Rotation  WEC: max. Rotation  \
count         49027.000000         49027.00000        49027.000000
mean             12.244133             8.67852            9.547354
std             223.186866             4.14345            4.482192
min               0.000000             0.00000            0.000000
25%               2.600000             6.33000            6.740000
50%               4.400000             8.97000           10.060000
75%               6.300000            11.92000           13.550000
max            6553.500000            14.73000           18.910000

       WEC: min. Rotation  WEC: ava. Power  WEC: max. Power  …  \
count        49027.000000     49027.000000     49027.000000  …
mean             8.515034       942.261244      1214.015400  …
std             22.394531      1008.930159      1168.858993  …
min              0.000000         0.000000         0.000000  …
25%              5.880000        87.000000       138.000000  …
50%              7.850000       536.000000       802.000000  …
75%             10.390000      1551.000000      2326.000000  …
max            655.349976      3071.000000      3216.000000  …

       Rectifier cabinet temp.  Yaw inverter cabinet temp.  \
count             49027.000000                49027.000000
mean                 30.335958                   24.320211
std                   5.623608                    4.918045
```

```
min                    0.000000                    0.000000
25%                   26.000000                   20.000000
50%                   30.000000                   25.000000
75%                   34.000000                   28.000000
max                   49.000000                   38.000000

       Fan inverter cabinet temp.  Ambient temp.   Tower temp.  \
count                49027.000000  49027.000000  49027.000000
mean                    28.802456     13.380219     23.116303
std                      5.185007      5.246230      6.360604
min                      0.000000      0.000000      0.000000
25%                     25.000000      9.000000     19.000000
50%                     29.000000     13.000000     24.000000
75%                     33.000000     17.000000     28.000000
max                     44.000000     35.000000     36.000000

       Control cabinet temp.  Transformer temp.  RTU: ava. Setpoint 1  \
count           49027.000000       49027.000000          49027.000000
mean               31.766537          43.992596           2988.628184
std                 6.381892          10.404843            172.074485
min                 0.000000         -19.000000              0.000000
25%                27.000000          37.000000           3050.000000
50%                33.000000          43.000000           3050.000000
75%                36.000000          48.000000           3050.000000
max                45.000000          71.000000           3050.000000

       Inverter averages  Inverter std dev
count       49027.000000     49027.000000
mean           27.828410         1.855781
std             5.595795         1.269928
min           -14.000000         0.000000
25%            24.363636         1.206045
50%            28.454546         1.566699
75%            31.818182         2.370270
max            42.545456        23.512859

[8 rows x 65 columns]
```

[15]:
```
df.dtypes
```

[15]:
```
DateTime                 object
Time                      int64
Error                     int64
WEC: ava. windspeed     float64
WEC: max. windspeed     float64
                          …
Control cabinet temp.     int64
```

```
Transformer temp.          int64
RTU: ava. Setpoint 1       int64
Inverter averages        float64
Inverter std dev         float64
Length: 66, dtype: object
```

[16]:
```
df.shape
```

[16]:
```
(49027, 66)
```

[17]:
```
df.columns
```

[17]:
```
Index(['DateTime', 'Time', 'Error', 'WEC: ava. windspeed',
       'WEC: max. windspeed', 'WEC: min. windspeed', 'WEC: ava. Rotation',
       'WEC: max. Rotation', 'WEC: min. Rotation', 'WEC: ava. Power',
       'WEC: max. Power', 'WEC: min. Power',
       'WEC: ava. Nacel position including cable twisting',
       'WEC: Operating Hours', 'WEC: Production kWh',
       'WEC: Production minutes', 'WEC: ava. reactive Power',
       'WEC: max. reactive Power', 'WEC: min. reactive Power',
       'WEC: ava. available P from wind',
       'WEC: ava. available P technical reasons',
       'WEC: ava. Available P force majeure reasons',
       'WEC: ava. Available P force external reasons',
       'WEC: ava. blade angle A', 'Sys 1 inverter 1 cabinet temp.',
       'Sys 1 inverter 2 cabinet temp.', 'Sys 1 inverter 3 cabinet temp.',
       'Sys 1 inverter 4 cabinet temp.', 'Sys 1 inverter 5 cabinet temp.',
       'Sys 1 inverter 6 cabinet temp.', 'Sys 1 inverter 7 cabinet temp.',
       'Sys 2 inverter 1 cabinet temp.', 'Sys 2 inverter 2 cabinet temp.',
       'Sys 2 inverter 3 cabinet temp.', 'Sys 2 inverter 4 cabinet temp.',
       'Sys 2 inverter 5 cabinet temp.', 'Sys 2 inverter 6 cabinet temp.',
       'Sys 2 inverter 7 cabinet temp.', 'Spinner temp.',
       'Front bearing temp.', 'Rear bearing temp.',
       'Pitch cabinet blade A temp.', 'Pitch cabinet blade B temp.',
       'Pitch cabinet blade C temp.', 'Blade A temp.', 'Blade B temp.',
       'Blade C temp.', 'Rotor temp. 1', 'Rotor temp. 2', 'Stator temp. 1',
       'Stator temp. 2', 'Nacelle ambient temp. 1', 'Nacelle ambient temp. 2',
       'Nacelle temp.', 'Nacelle cabinet temp.', 'Main carrier temp.',
       'Rectifier cabinet temp.', 'Yaw inverter cabinet temp.',
       'Fan inverter cabinet temp.', 'Ambient temp.', 'Tower temp.',
       'Control cabinet temp.', 'Transformer temp.', 'RTU: ava. Setpoint 1',
       'Inverter averages', 'Inverter std dev'],
      dtype='object')
```

2.2 Exploring and Cleaning

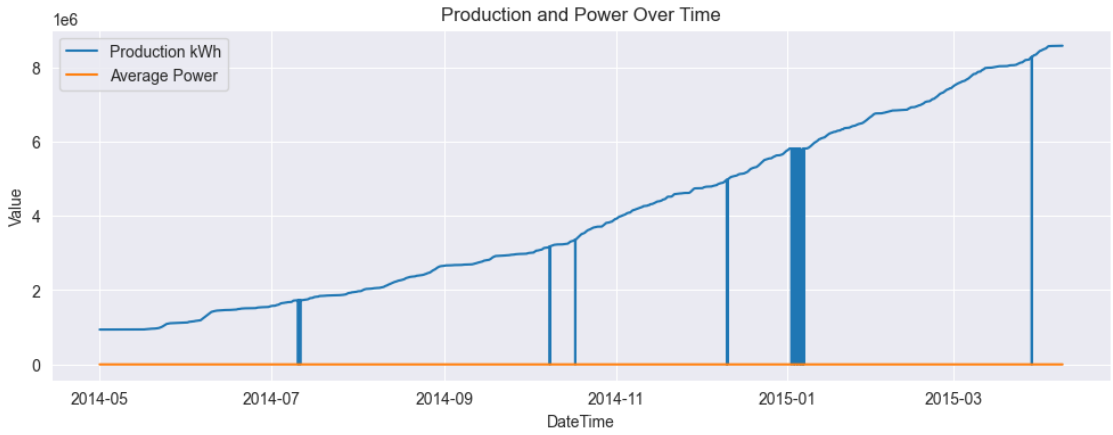
[18]:
```
# Check for missing values
print("Missing Values per Column:")
print(df.isna().sum())

# Option 1: Drop columns with significant missing values
df = df.dropna(axis=1, thresh=int(0.8 * len(df)))  # Keep columns with at least␣
 ↪80% non-null values

# Option 2: Impute missing values for numerical columns
for col in df.select_dtypes(include=['float64', 'int64']).columns:
    df[col] = df[col].fillna(df[col].median())  # Replace with median

# Option 3: Impute missing values for categorical columns
for col in df.select_dtypes(include=['object', 'category']).columns:
    df[col] = df[col].fillna(df[col].mode()[0])  # Replace with mode
```

```
Missing Values per Column:
DateTime                  0
Time                      0
Error                     0
WEC: ava. windspeed       0
WEC: max. windspeed       0
                         ..
Control cabinet temp.     0
Transformer temp.         0
RTU: ava. Setpoint 1      0
Inverter averages         0
Inverter std dev          0
Length: 66, dtype: int64
```

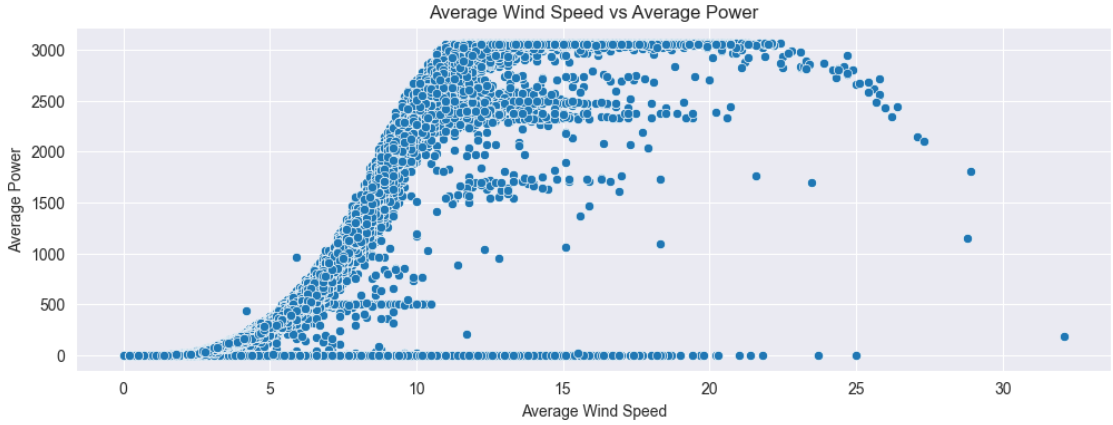
As we can see we have relatively cleaned dataset as we have not found any missing value.

[19]:
```
# Convert DateTime column to a datetime object if not already
df['DateTime'] = pd.to_datetime(df['DateTime'])
```

[20]:
```
# Plotting production and power over time
plt.figure(figsize=(12, 4))
plt.plot(df['DateTime'], df['WEC: Production kWh'], label='Production kWh')
plt.plot(df['DateTime'], df['WEC: ava. Power'], label='Average Power')
plt.xlabel('DateTime')
plt.ylabel('Value')
plt.title('Production and Power Over Time')
plt.legend()
plt.show()
```



[21]:
```
# Scatter plot for average wind speed and average rotation
plt.figure(figsize=(12, 4))
sns.scatterplot(data=df, x='WEC: ava. windspeed', y='WEC: ava. Power')
plt.title('Average Wind Speed vs Average Power')
plt.xlabel('Average Wind Speed')
plt.ylabel('Average Power')
plt.show()
```

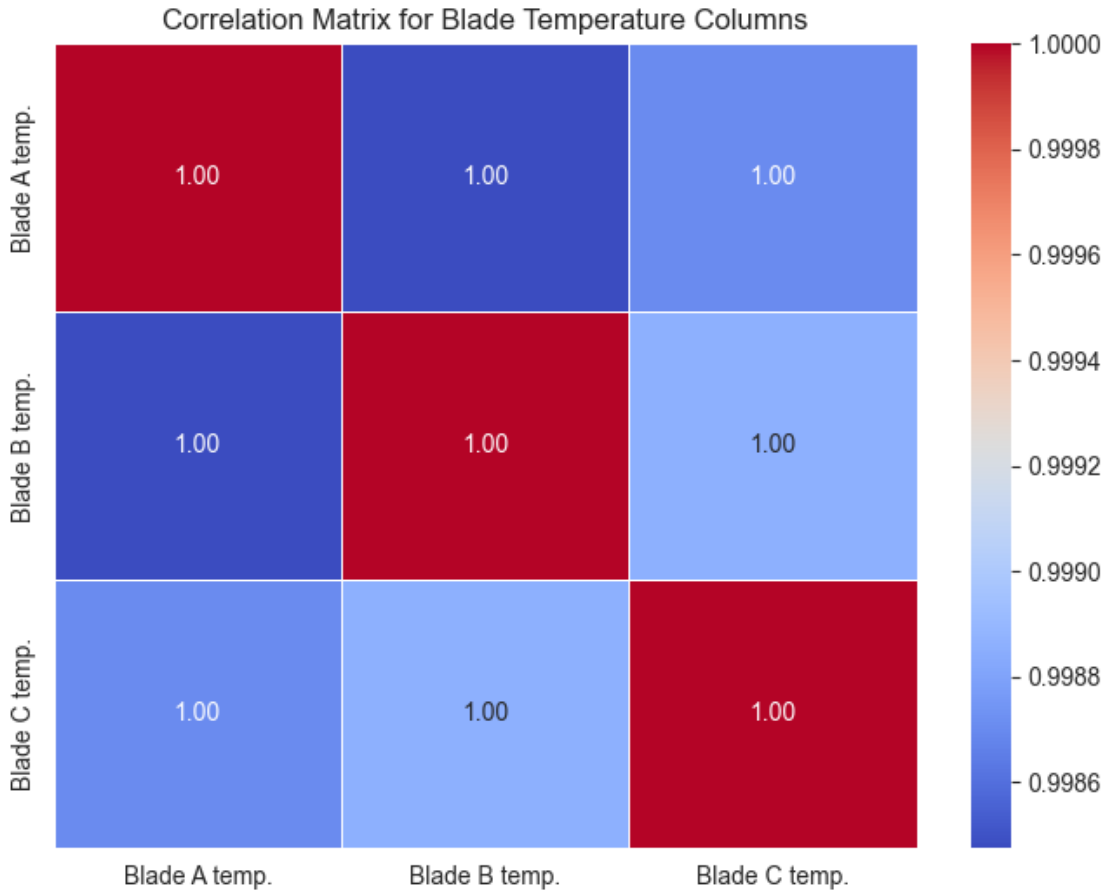


- The average power (`WEC: ava. Power`) remains constant because wind turbines operate within a fixed capacity range, determined by design and average wind conditions.
- Production (`WEC: Production kWh`) increases over time due to longer operational periods, reduced downtime(in total 6 instances), or improved turbine utilization, even when average power output stays stable.
- The turbine's rotation reaches its maximum at approximately **14.7**, likely due to design constraints and safety measures.
- Wind speed above **8.9** contributes less to rotation increases, with speeds beyond **8.9** having no further impact. In some cases, with wind speed over **23**, it leads to a decrease in rotations.

- This plateau indicates effective turbine control to optimize performance while preventing mechanical damage at high wind speeds.

[22]:
```
# Assuming 'df' is your dataset and it contains columns related to blades
# Select relevant blade columns
blade_columns = ['Blade A temp.', 'Blade B temp.', 'Blade C temp.']  # Update this list if there are more blade-related columns

# Select the subset of the DataFrame containing only these columns
df_blades = df[blade_columns]

# Calculate the correlation matrix for these blade-related columns
correlation_matrix_blades = df_blades.corr()

# Print the correlation matrix
print("Correlation matrix for Blade columns:")
print(correlation_matrix_blades)

# Optional: Visualize the correlation matrix using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix_blades, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title("Correlation Matrix for Blade Temperature Columns")
plt.show()
```

```
Correlation matrix for Blade columns:
               Blade A temp.  Blade B temp.  Blade C temp.
Blade A temp.       1.000000       0.998474       0.998702
Blade B temp.       0.998474       1.000000       0.998857
Blade C temp.       0.998702       0.998857       1.000000
```

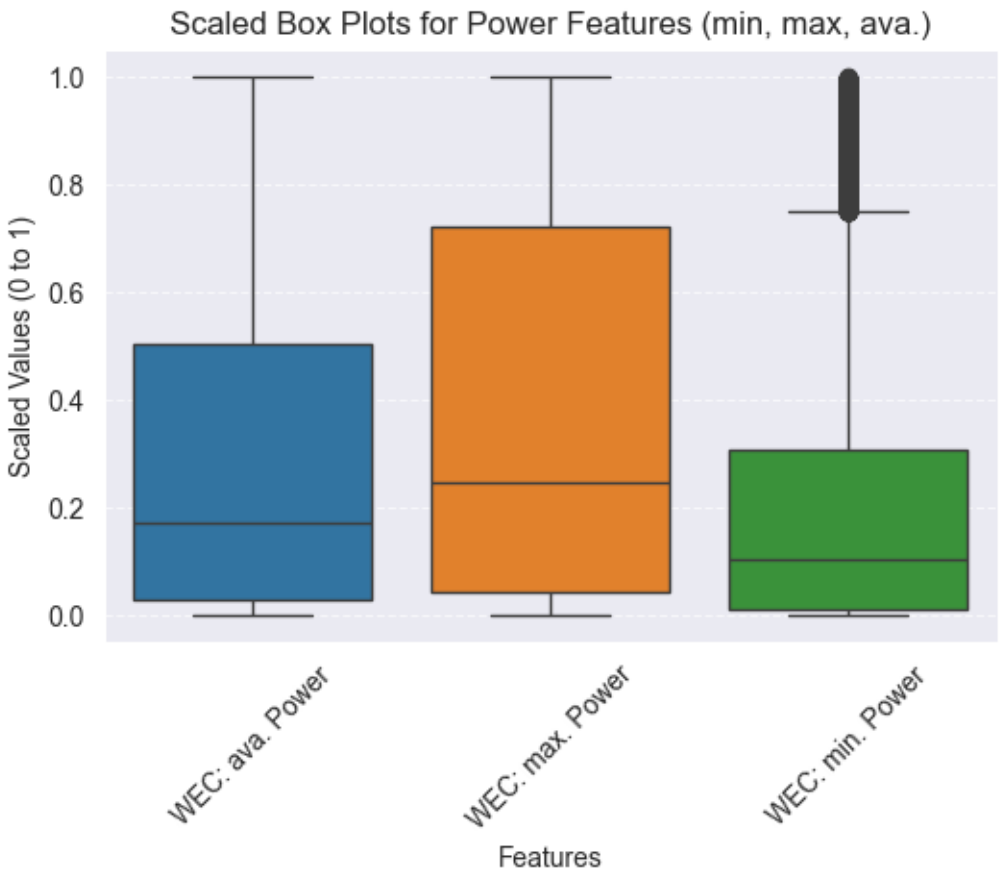


[23]:
```
# Function to normalize data and plot boxplots, including printing the boxplot␣
↪values
def plot_boxplots_with_scaling(df, feature_type, feature_keywords,␣
↪exclude_keywords=None):
    """
    Normalizes data and plots boxplots for min, max, and average columns for a␣
↪specific feature type.
    Prints boxplot statistics.

    :param df: The DataFrame containing the data
    :param feature_type: The type of feature (e.g., 'Power', 'Windspeed')
    :param feature_keywords: A list of keywords to filter the columns (e.g.,␣
↪['min', 'max', 'ava.'])
    :param exclude_keywords: List of keywords to exclude from the column␣
↪filtering (e.g., ['reactive']).
    """
    # Filter columns containing the feature keywords and the feature type
```

```
    relevant_columns = [col for col in df.columns if feature_type in col and␣
↪any(keyword in col for keyword in feature_keywords)]

    # Exclude columns with specific keywords, if provided
    if exclude_keywords:
        relevant_columns = [col for col in relevant_columns if not any(keyword␣
↪in col.lower() for keyword in exclude_keywords)]

    # Ensure there are columns to plot
    if relevant_columns:
        # Normalize the data using MinMaxScaler
        scaler = MinMaxScaler()
        scaled_data = scaler.fit_transform(df[relevant_columns])
        scaled_df = pd.DataFrame(scaled_data, columns=relevant_columns)

        # Compute descriptive statistics for boxplot values
        boxplot_stats = scaled_df.describe()

        # Print boxplot values
        print(f"\nBoxplot values for {feature_type} features ({', '.
↪join(feature_keywords)}):")
        print(boxplot_stats)

        # Plot the boxplot
        plt.figure(figsize=(6, 4))
        sns.boxplot(data=scaled_df)
        plt.xticks(rotation=45)
        plt.title(f'Scaled Box Plots for {feature_type} Features ({", ".
↪join(feature_keywords)})')
        plt.ylabel('Scaled Values (0 to 1)')
        plt.xlabel('Features')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        plt.show()
    else:
        print(f"No relevant columns found for {feature_type} with keywords␣
↪{feature_keywords}.")

# Define feature types and keywords to look for
feature_types = [
    {'feature_type': 'Power', 'exclude_keywords': ['reactive']},  # Only Power␣
↪(not reactive)
    {'feature_type': 'reactive Power', 'exclude_keywords': None},  # Reactive␣
↪Power
    {'feature_type': 'windspeed', 'exclude_keywords': None},  # Windspeed
    {'feature_type': 'Rotation', 'exclude_keywords': None},  # Rotation
]
```

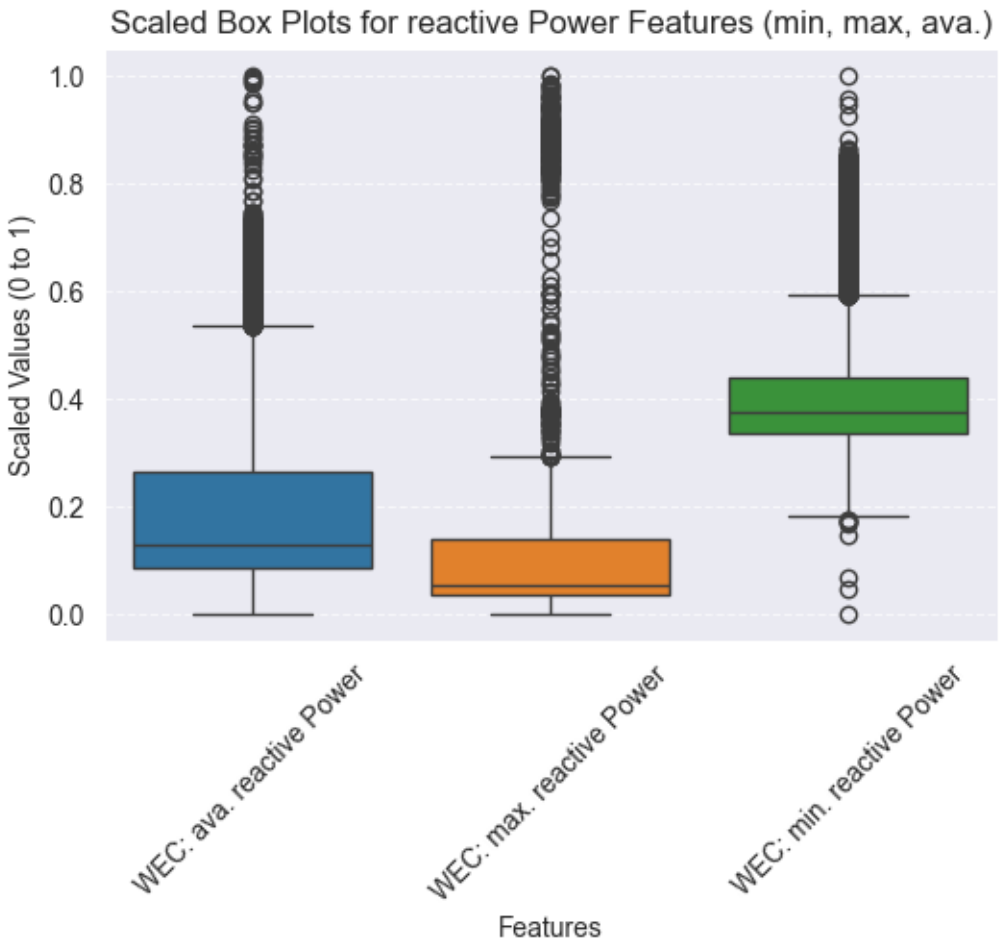
```
feature_keywords = ['min', 'max', 'ava.']

# Generate scaled box plots for each feature type
for feature in feature_types:
    plot_boxplots_with_scaling(
        df,
        feature_type=feature['feature_type'],
        feature_keywords=feature_keywords,
        exclude_keywords=feature.get('exclude_keywords')
    )
```

```
Boxplot values for Power features (min, max, ava.):
       WEC: ava. Power  WEC: max. Power  WEC: min. Power
count     49027.000000     49027.000000     49027.000000
mean          0.306826         0.377492         0.215213
std           0.328535         0.363451         0.270104
min           0.000000         0.000000         0.000000
25%           0.028330         0.042910         0.011889
50%           0.174536         0.249378         0.104029
75%           0.505047         0.723259         0.307794
max           1.000000         1.000000         1.000000
```



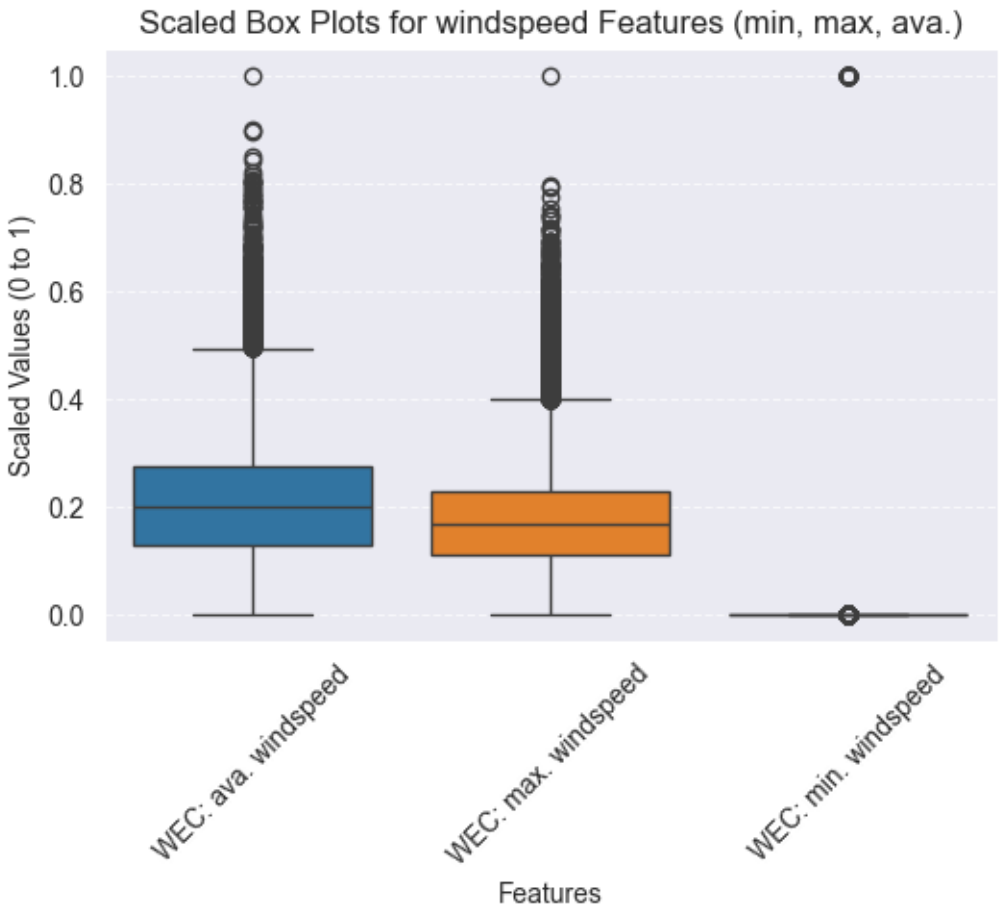
Boxplot values for reactive Power features (min, max, ava.):

	WEC: ava. reactive Power	WEC: max. reactive Power	WEC: min. reactive Power
count	49027.000000	49027.000000	49027.000000
mean	0.207139	0.094518	0.415801
std	0.194494	0.097377	0.127754
min	0.000000	0.000000	0.000000
25%	0.085450	0.036490	0.335125
50%	0.129330	0.054735	0.376344
75%	0.265589	0.139010	0.439068
max	1.000000	1.000000	1.000000



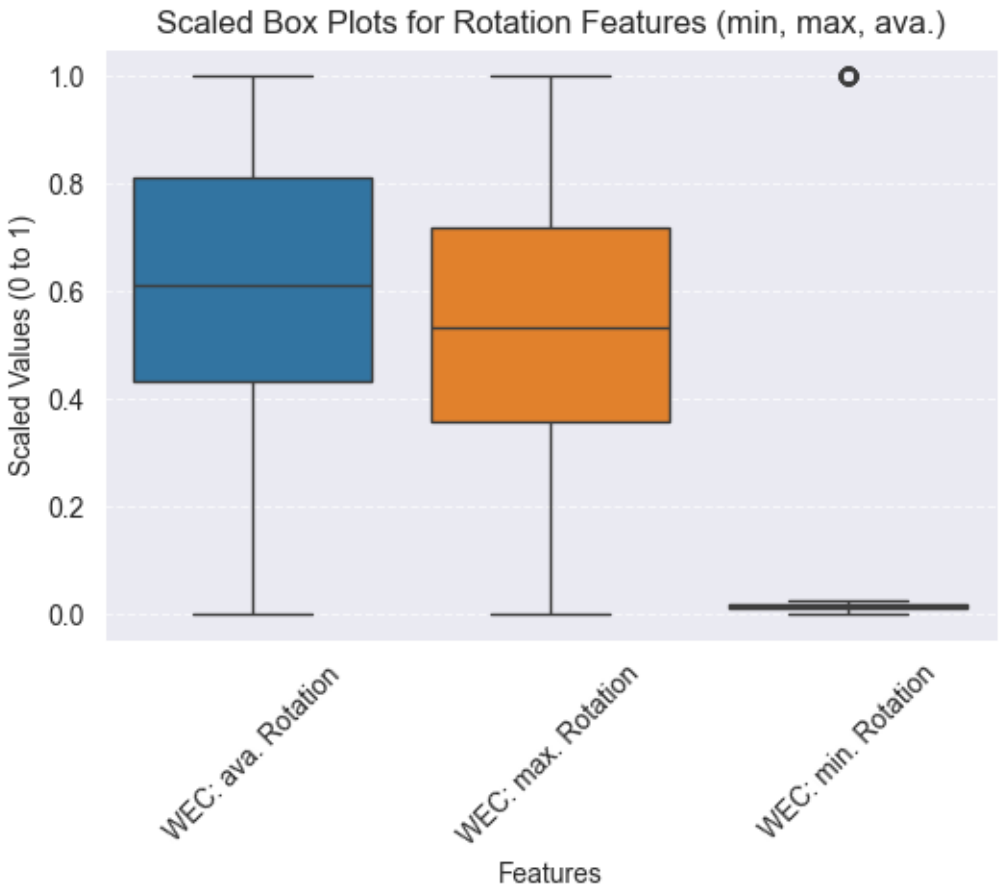
Boxplot values for windspeed features (min, max, ava.):

	WEC: ava. windspeed	WEC: max. windspeed	WEC: min. windspeed
count	49027.000000	49027.000000	49027.000000
mean	0.214163	0.182784	0.001868
std	0.115102	0.100929	0.034056
min	0.000000	0.000000	0.000000
25%	0.130841	0.113503	0.000397
50%	0.202492	0.168297	0.000671
75%	0.277259	0.228963	0.000961
max	1.000000	1.000000	1.000000



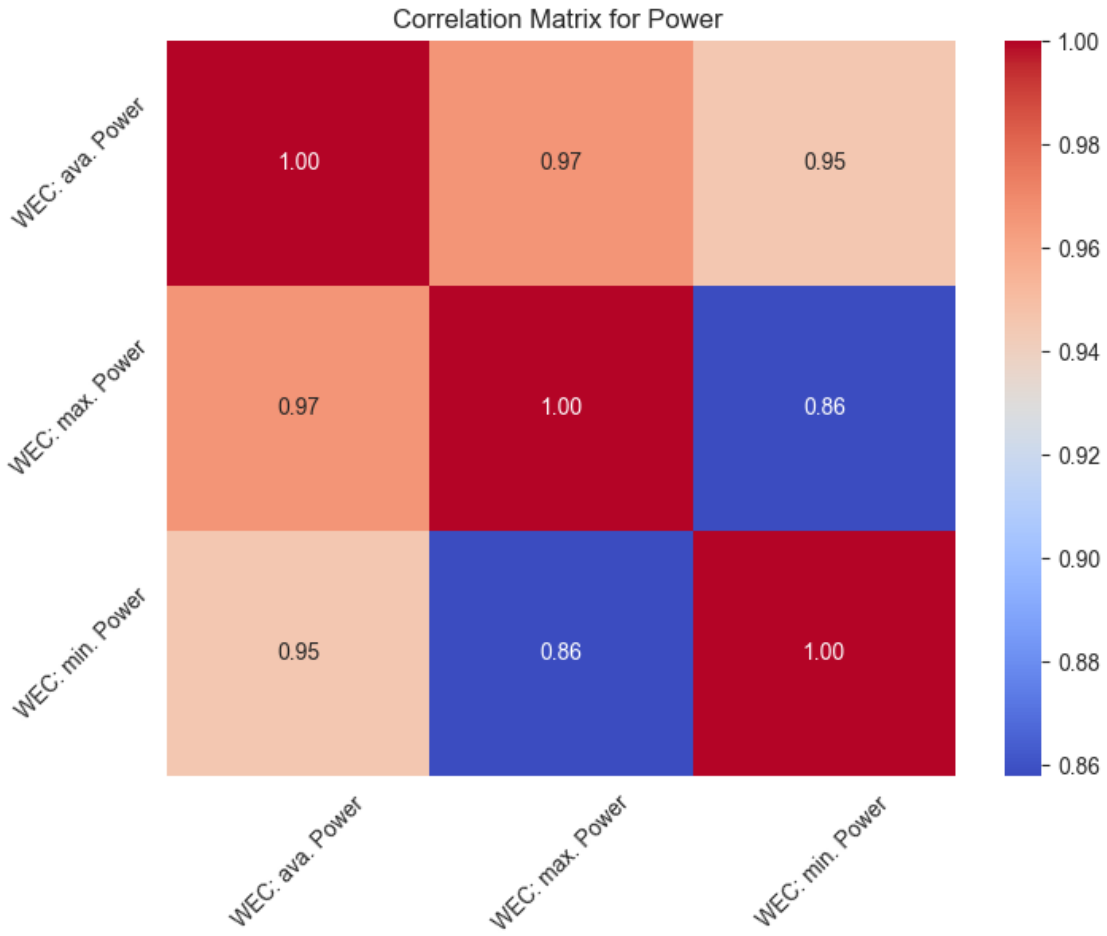
Boxplot values for Rotation features (min, max, ava.):

	WEC: ava. Rotation	WEC: max. Rotation	WEC: min. Rotation
count	49027.000000	49027.000000	49027.000000
mean	0.589173	0.504884	0.012993
std	0.281293	0.237028	0.034172
min	0.000000	0.000000	0.000000
25%	0.429735	0.356425	0.008972
50%	0.608961	0.531994	0.011978
75%	0.809233	0.716552	0.015854
max	1.000000	1.000000	1.000000



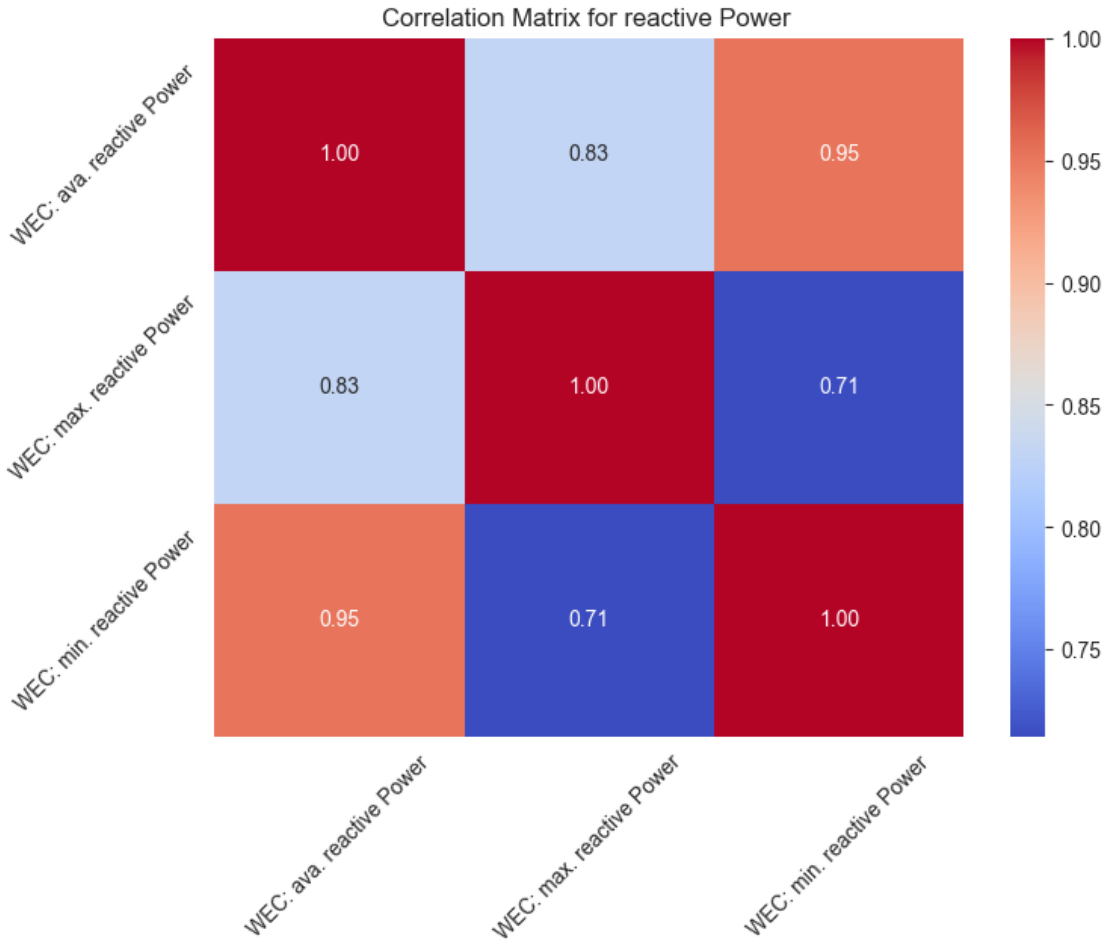
[24]:
```python
# Function to compute and plot correlation matrix for a specific feature type
def plot_correlation_matrix(df, feature_type, exclude_keywords=None):
    """
    Plots a correlation matrix for min, max, and avg columns of a given feature␣
 ↪type.

    :param df: DataFrame containing the data
    :param feature_type: The type of feature to filter (e.g., 'Power',␣
 ↪'Reactive Power')
    :param exclude_keywords: List of keywords to exclude from the column␣
 ↪filtering (e.g., ['reactive']).
    """
    # Filter columns containing the feature type and min, max, or average
    relevant_columns = [col for col in df.columns if feature_type in col and
                        ('min' in col.lower() or 'max' in col.lower() or 'ava.'␣
 ↪in col.lower())]

    # Exclude columns with specific keywords, if provided
```

```
    if exclude_keywords:
        relevant_columns = [col for col in relevant_columns if not any(keyword␣
↪in col.lower() for keyword in exclude_keywords)]

    if relevant_columns:
        # Calculate the correlation matrix
        correlation_matrix = df[relevant_columns].corr()

        # Plot the heatmap
        plt.figure(figsize=(8, 6))
        sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
        plt.title(f'Correlation Matrix for {feature_type} ')
        plt.xticks(rotation=45)
        plt.yticks(rotation=45)
        plt.show()

        # Print correlation matrix for inspection
        print(f"\nCorrelation Matrix for {feature_type} :")
        print(correlation_matrix)
    else:
        print(f"No relevant columns found for {feature_type}.")

# Generate correlation matrices
plot_correlation_matrix(df, feature_type='Power', exclude_keywords=['reactive'])
plot_correlation_matrix(df, feature_type='reactive Power')
plot_correlation_matrix(df, feature_type='windspeed')
plot_correlation_matrix(df, feature_type='Rotation')
```

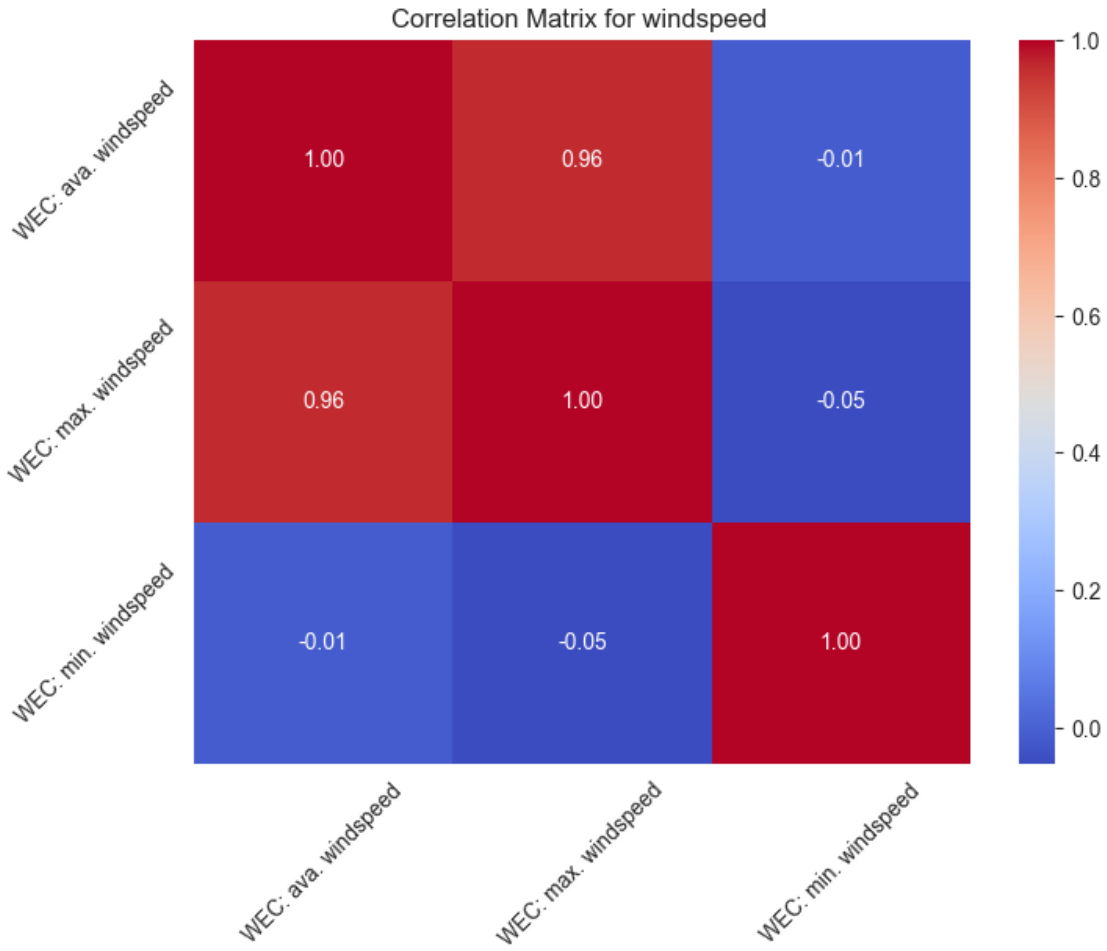


Correlation Matrix for Power :

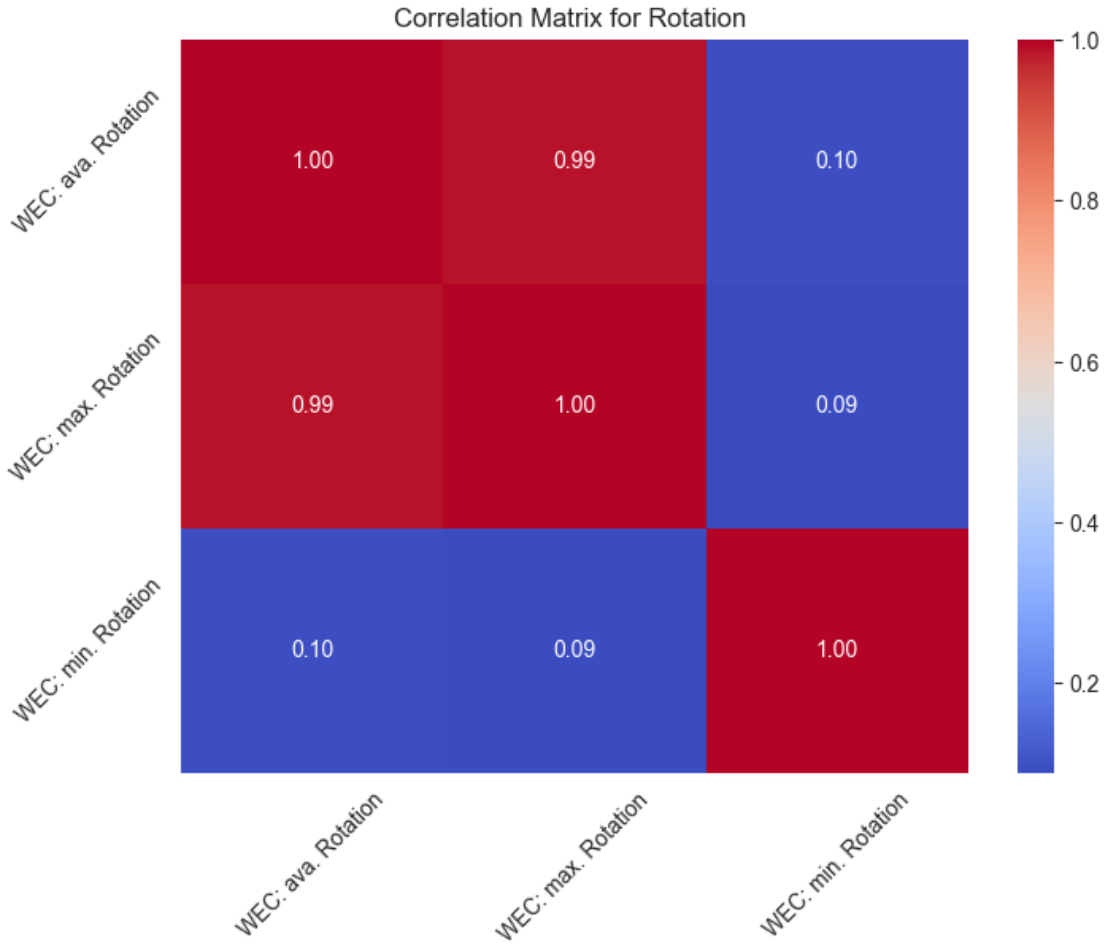
	WEC: ava. Power	WEC: max. Power	WEC: min. Power
WEC: ava. Power	1.000000	0.965711	0.945421
WEC: max. Power	0.965711	1.000000	0.857933
WEC: min. Power	0.945421	0.857933	1.000000




```
Correlation Matrix for reactive Power :
                          WEC: ava. reactive Power  WEC: max. reactive Power  \
WEC: ava. reactive Power                  1.000000                  0.831072
WEC: max. reactive Power                  0.831072                  1.000000
WEC: min. reactive Power                  0.952421                  0.714523

                          WEC: min. reactive Power
WEC: ava. reactive Power                  0.952421
WEC: max. reactive Power                  0.714523
WEC: min. reactive Power                  1.000000
```




```
Correlation Matrix for windspeed :
                     WEC: ava. windspeed  WEC: max. windspeed  \
WEC: ava. windspeed             1.000000             0.961435
WEC: max. windspeed             0.961435             1.000000
WEC: min. windspeed            -0.011400            -0.052131

                     WEC: min. windspeed
WEC: ava. windspeed            -0.011400
WEC: max. windspeed            -0.052131
WEC: min. windspeed             1.000000
```




```
Correlation Matrix for Rotation :
                     WEC: ava. Rotation  WEC: max. Rotation  WEC: min. Rotation
WEC: ava. Rotation             1.000000            0.988052            0.095114
WEC: max. Rotation             0.988052            1.000000            0.088235
WEC: min. Rotation             0.095114            0.088235            1.000000
```

After analyzing the description of the 12 columns and their correlation matrix, the following recommendations are made for feature selection to optimize the model:

Features to Retain:

1. **Windspeed**: Retain **`WEC: ava. windspeed`** because it captures the overall trend and is highly correlated with `WEC: max. windspeed` (correlation: 0.961), making it the most representative feature.
2. **Rotation**: Retain **`WEC: ava. Rotation`**, as it reflects the turbine's overall behavior and has a very high correlation with `WEC: max. Rotation` (correlation: 0.988).
3. **Power**: Retain **`WEC: ava. Power`**, which represents the general power output and is highly

correlated with both `WEC: max. Power` (correlation: 0.965) and `WEC: min. Power` (correlation: 0.945).
4. **Reactive Power**: Retain `WEC: ava. reactive Power`, as it captures the overall trends and has a high correlation with `WEC: min. reactive Power` (correlation: 0.952).

Features to Drop:

1. `WEC: min. windspeed`: Drop due to a poor correlation with other windspeed features and the presence of an unrealistic outlier (`6553.5`).
2. `WEC: min. Rotation`: Drop because of low correlations with average (0.095) and max rotation (0.088), and potential data quality issues.
3. `WEC: max. Power` and `WEC: min. Power`: Drop as they are highly redundant with `WEC: ava. Power`, adding unnecessary noise.
4. `WEC: max. reactive Power` and `WEC: min. reactive Power`: Drop to avoid redundancy and moderate correlations (e.g., max reactive power correlates 0.831 with average).

Key Reasons:

1. Simplifies the model by avoiding redundancy (e.g., removing features with high correlation to another retained feature).
2. Improves data quality by excluding features with outliers or poor correlations.
3. Focuses on features that provide the most representative information (e.g., average values).

Next Steps Before Modeling:

1. **Outlier Removal**: Address extreme values in the `min` columns for all categories.
2. **Feature Scaling**: Apply scaling (e.g., MinMaxScaler or StandardScaler) to normalize feature ranges.
3. **Dimensionality Reduction**: Proceed with only the retained features to reduce noise and improve model efficiency.

[25]:
```
# Function to plot boxplots and print descriptive statistics for inverter␣
↪temperatures
def plot_inverter_temps_with_stats(df, inverter_columns, system_name):
    """
    Plots boxplots and prints descriptive statistics for a given system's␣
↪inverter temperatures.

    :param df: The DataFrame containing the data
    :param inverter_columns: List of column names for the inverter temperatures
    :param system_name: Name of the system (e.g., 'Sys 1', 'Sys 2') for labeling
    """
    if not inverter_columns:
        print(f"No columns provided for {system_name}.")
        return

    # Compute descriptive statistics
    stats = df[inverter_columns].describe()
    print(f"\nBoxplot values for {system_name} Inverter Temperatures:")
```

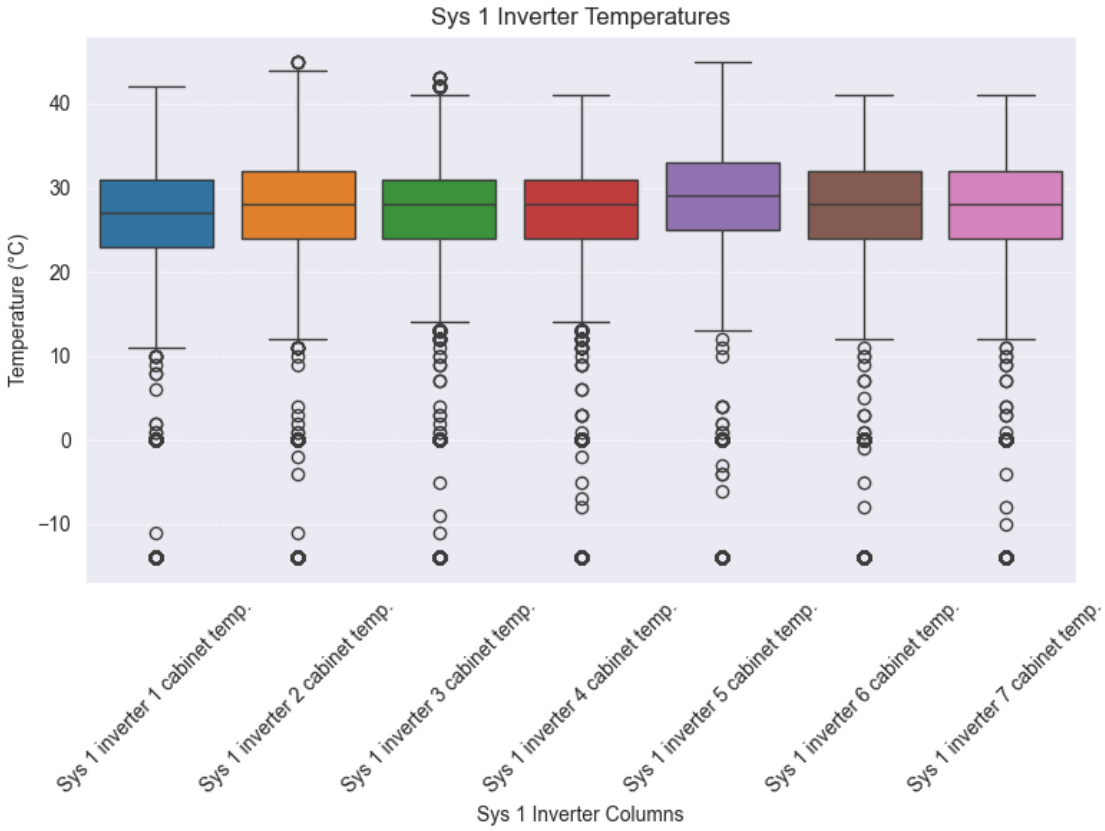
```
    print(stats)

    # Plot boxplot
    plt.figure(figsize=(8, 6))
    sns.boxplot(data=df[inverter_columns])
    plt.xticks(rotation=45)
    plt.title(f'{system_name} Inverter Temperatures')
    plt.ylabel('Temperature (°C)')
    plt.xlabel(f'{system_name} Inverter Columns')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

# Columns for inverter temperatures
sys1_inverter_temps = [
    'Sys 1 inverter 1 cabinet temp.', 'Sys 1 inverter 2 cabinet temp.',
    'Sys 1 inverter 3 cabinet temp.', 'Sys 1 inverter 4 cabinet temp.',
    'Sys 1 inverter 5 cabinet temp.', 'Sys 1 inverter 6 cabinet temp.',
    'Sys 1 inverter 7 cabinet temp.'
]
sys2_inverter_temps = [
    'Sys 2 inverter 1 cabinet temp.', 'Sys 2 inverter 2 cabinet temp.',
    'Sys 2 inverter 3 cabinet temp.', 'Sys 2 inverter 4 cabinet temp.',
    'Sys 2 inverter 5 cabinet temp.', 'Sys 2 inverter 6 cabinet temp.',
    'Sys 2 inverter 7 cabinet temp.'
]

# Plot and print statistics for Sys 1 and Sys 2 inverter temperatures
plot_inverter_temps_with_stats(df, sys1_inverter_temps, "Sys 1")
plot_inverter_temps_with_stats(df, sys2_inverter_temps, "Sys 2")
```

```
Boxplot values for Sys 1 Inverter Temperatures:
       Sys 1 inverter 1 cabinet temp.  Sys 1 inverter 2 cabinet temp.  \
count                    49027.000000                    49027.000000
mean                        26.633712                       27.703816
std                          6.361488                        6.220329
min                        -14.000000                      -14.000000
25%                         23.000000                       24.000000
50%                         27.000000                       28.000000
75%                         31.000000                       32.000000
max                         42.000000                       45.000000

       Sys 1 inverter 3 cabinet temp.  Sys 1 inverter 4 cabinet temp.  \
count                    49027.000000                    49027.000000
mean                        27.279560                       27.361393
std                          5.811341                        5.785126
```

min	-14.000000	-14.000000
25%	24.000000	24.000000
50%	28.000000	28.000000
75%	31.000000	31.000000
max	43.000000	41.000000

	Sys 1 inverter 5 cabinet temp.	Sys 1 inverter 6 cabinet temp. \
count	49027.00000	49027.000000
mean	28.65470	27.550472
std	6.22394	6.112337
min	-14.00000	-14.000000
25%	25.00000	24.000000
50%	29.00000	28.000000
75%	33.00000	32.000000
max	45.00000	41.000000

	Sys 1 inverter 7 cabinet temp.
count	49027.000000
mean	27.650968
std	5.774417
min	-14.000000
25%	24.000000
50%	28.000000
75%	32.000000
max	41.000000



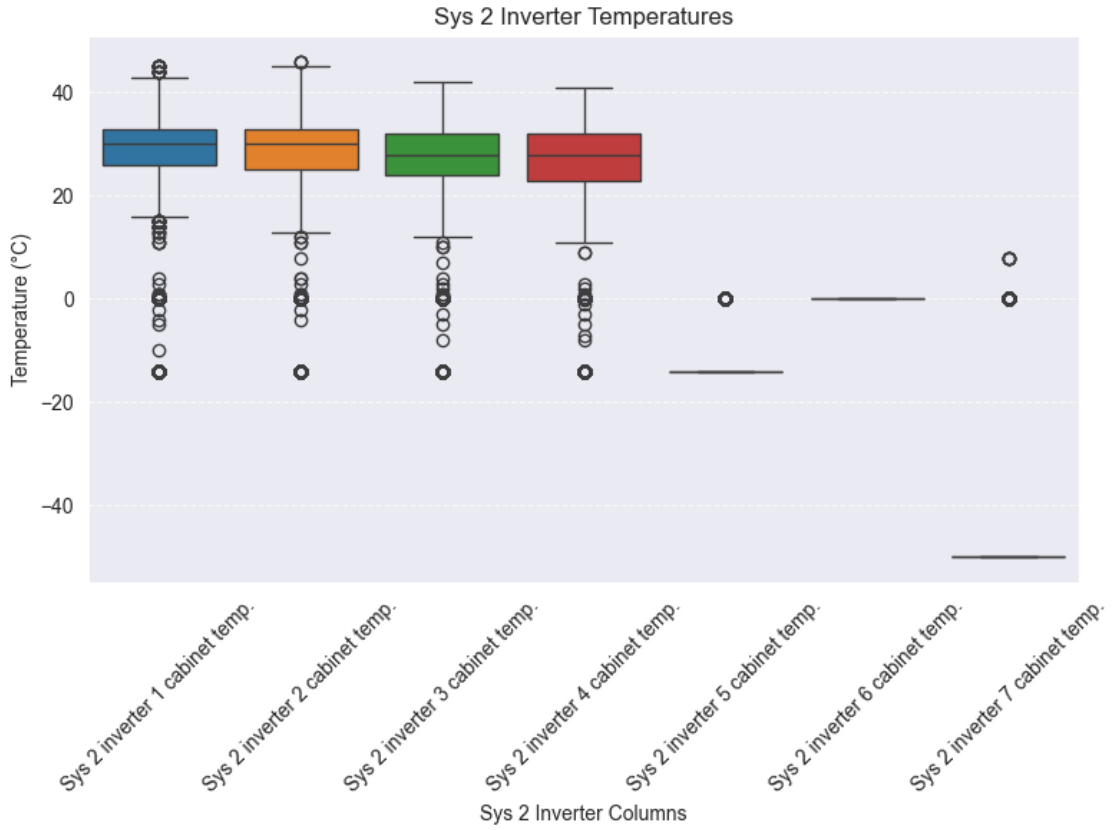
Boxplot values for Sys 2 Inverter Temperatures:

	Sys 2 inverter 1 cabinet temp.	Sys 2 inverter 2 cabinet temp.	\
count	49027.000000	49027.000000	
mean	29.134926	28.868032	
std	5.702693	5.697637	
min	-14.000000	-14.000000	
25%	26.000000	25.000000	
50%	30.000000	30.000000	
75%	33.000000	33.000000	
max	45.000000	46.000000	

	Sys 2 inverter 3 cabinet temp.	Sys 2 inverter 4 cabinet temp.	\
count	49027.000000	49027.000000	
mean	27.849593	27.425337	
std	5.832961	5.869853	
min	-14.000000	-14.000000	
25%	24.000000	23.000000	
50%	28.000000	28.000000	
75%	32.000000	32.000000	
max	42.000000	41.000000	

	Sys 2 inverter 5 cabinet temp.	Sys 2 inverter 6 cabinet temp.	\
count	49027.000000	49027.0	
mean	-13.952883	0.0	
std	0.810821	0.0	
min	-14.000000	0.0	
25%	-14.000000	0.0	
50%	-14.000000	0.0	
75%	-14.000000	0.0	
max	0.000000	0.0	

	Sys 2 inverter 7 cabinet temp.
count	49027.000000
mean	-49.825810
std	2.954089
min	-50.000000
25%	-50.000000
50%	-50.000000
75%	-50.000000
max	8.000000



These boxplots show that `Sys 1` inverter temperatures are consistent and realistic, with mean values around **27°C to 28°C**, standard deviations of ~**5.7°C to 6.3°C**, and maximum temperatures up to **45°C**, indicating normal operation.

In contrast, `Sys 2` displays anomalies, including extreme negative mean values (e.g., **-13.95°C**, **-49.83°C**) and minimum values of **-50°C**, likely due to faulty sensors or inactive inverters.

These irregularities in `Sys 2` require further investigation to ensure data reliability before analysis.

[26]:
```
# Plot a histogram of the 'Error' column
plt.figure(figsize=(6, 3))
df['Error'].plot(kind='hist', bins=30, edgecolor='black')
plt.title("Distribution of Error")
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.show()


# Filter out zeros
non_zero_errors = df[df['Error'] != 0]['Error']

# Plot distribution of non-zero Error values
plt.figure(figsize=(6, 3))
plt.hist(non_zero_errors, bins=50, color='salmon', edgecolor='black')
plt.title("Distribution of Non-Zero 'Error' Values")
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.show()

# Count the occurrences of each unique value in the 'Error' column
error_distribution = df['Error'].value_counts().sort_index()

# Print the distribution
print("Distribution of 'Error' values:")
print(error_distribution)
```


```
Distribution of 'Error' values:
Error
0      48727
10        35
20        18
30        13
40         9
50        18
```

```
60          9
70         10
80          9
90         14
100         1
246       164
Name: count, dtype: int64
```

2.2.1 Distribution Analysis of the `Error` Column

The `Error` column in the dataset exhibits a **highly imbalanced distribution**:

- The majority of instances (48,727) correspond to `Error = 0`, representing normal operation without any errors.
- Non-zero error values, which likely indicate fault or abnormal states, occur far less frequently:
  - The second most frequent error (`Error = 246`) has **164 instances**, which is significantly smaller compared to the normal operation.
  - Other errors (e.g., `10`, `20`, `30`, etc.) have frequencies ranging between **1** and **35**, with `Error = 100` occurring only **once**.

This imbalance suggests that the dataset is heavily dominated by normal operational states, making it challenging to predict fault states (non-zero errors) without proper handling.

2.2.2 Implications of Imbalance:

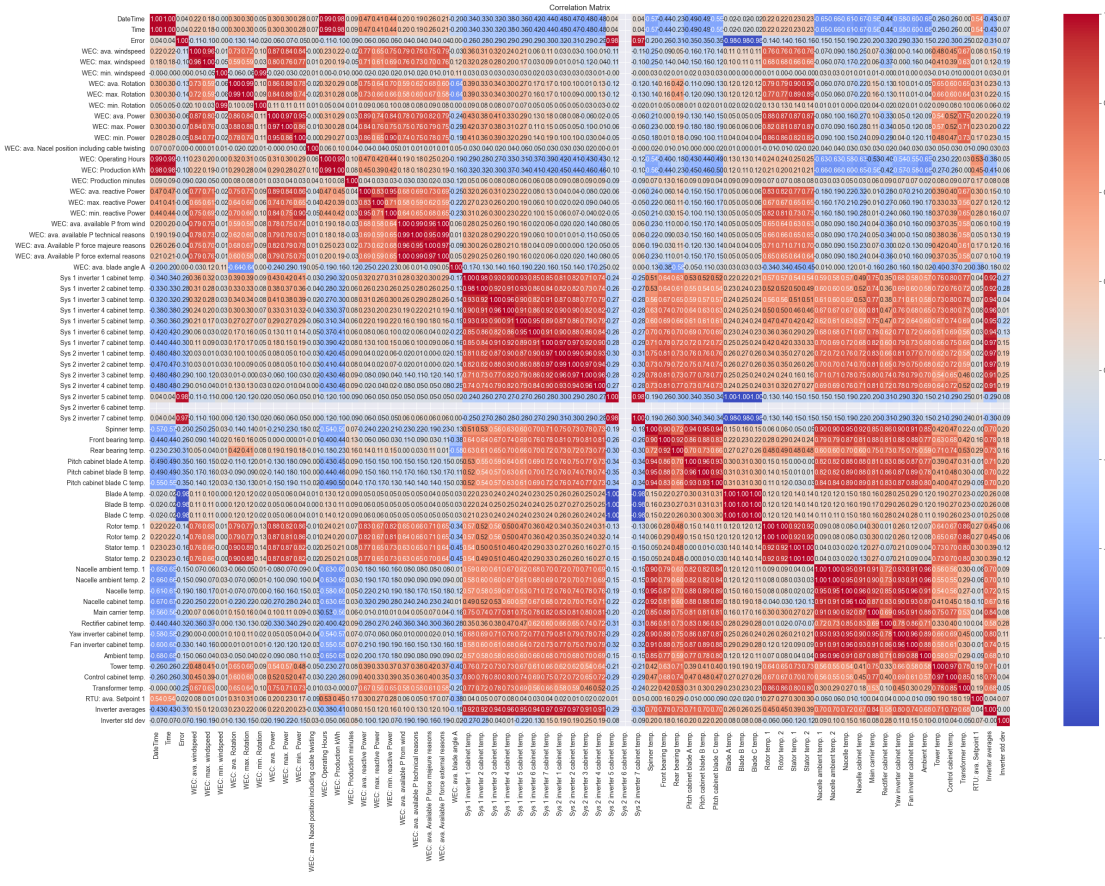
1. **Class Imbalance in Classification**:
   - If this column is used as a target for classification, the imbalance could lead to a model biased towards predicting the majority class (`Error = 0`), potentially overlooking fault states.
   - Special techniques such as **oversampling (e.g., SMOTE)**, **undersampling**, or using **class-weighted loss functions** would be required to handle this imbalance.
2. **Impact on Evaluation Metrics**:
   - Standard accuracy might be misleading in this case, as predicting only `Error = 0` would yield high accuracy due to the dominance of normal instances.
   - Metrics such as **Precision**, **Recall** and **F1-Score* should be prioritized to evaluate model performance.
3. **Rare Error Values**:
   - Errors with very low frequencies (e.g., `Error = 100` with only 1 instance) may not provide sufficient data for meaningful model training.
   - These rare error types could be grouped into an "Other Errors" class to simplify the classification problem.

2.2.3 Conclusion:

The extreme imbalance in the `Error` column poses challenges for classification and requires careful preprocessing and model selection to ensure the minority classes (fault states) are effectively identified.

2.3 Preparing & Transforming

[27]:
```
# Compute the correlation matrix
correlation_matrix = df.corr()

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(30, 20))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f",␣
 ↪cbar=True)
plt.title('Correlation Matrix')
plt.show()
```



[28]:
```
# Filter correlations related to the feature 'Error'
error_correlations_all = correlation_matrix['Error']

# Set the absolute threshold
threshold = 0.1

# Display all correlations with 'Error':
print("All correlations with 'Error':")
```

```
print(error_correlations_all)

# Identify columns with correlation below the threshold of 0.1
low_correlation_columns = error_correlations_all[abs(error_correlations_all) <␣
 ↪threshold]

# Display columns with correlation below the threshold
print(f"\nColumns with correlation below the absolute threshold of {threshold}␣
 ↪for 'Error':")
print(low_correlation_columns)

# Count columns with correlation below the threshold
low_correlation_count = len(low_correlation_columns)
print(f"\nNumber of columns with correlation below {threshold} for 'Error':␣
 ↪{low_correlation_count}")

# Identify columns with correlation above or equal to the threshold of 0.1
high_correlation_columns = error_correlations_all[abs(error_correlations_all)␣
 ↪>= threshold]

# Display columns with correlation above or equal to the threshold
print(f"\nColumns with correlation above or equal to the absolute threshold of␣
 ↪{threshold} for 'Error':")
print(high_correlation_columns)

# Count columns with correlation above the threshold
high_correlation_count = len(high_correlation_columns)
print(f"\nNumber of columns with correlation above or equal to {threshold} for␣
 ↪'Error': {high_correlation_count}")
```

```
All correlations with 'Error':
DateTime                 0.041173
Time                     0.041182
Error                    1.000000
WEC: ava. windspeed     -0.108580
WEC: max. windspeed     -0.102705
                            …
Control cabinet temp.   -0.298812
Transformer temp.       -0.251442
RTU: ava. Setpoint 1     0.019532
Inverter averages       -0.309423
Inverter std dev        -0.069573
Name: Error, Length: 66, dtype: float64

Columns with correlation below the absolute threshold of 0.1 for 'Error':
DateTime                                              0.041173
Time                                                  0.041182
```

```
WEC: min. windspeed                                  -0.003485
WEC: min. Rotation                                   -0.024497
WEC: ava. Power                                      -0.060350
WEC: max. Power                                      -0.067009
WEC: min. Power                                      -0.051487
WEC: ava. Nacel position including cable twisting     0.002400
WEC: Production kWh                                  -0.096689
WEC: Production minutes                              -0.088421
WEC: ava. reactive Power                             -0.062579
WEC: max. reactive Power                             -0.058301
WEC: min. reactive Power                             -0.058070
WEC: ava. available P from wind                      -0.043813
WEC: ava. available P technical reasons              -0.044759
WEC: ava. Available P force majeure reasons          -0.041619
WEC: ava. Available P force external reasons         -0.043766
WEC: ava. blade angle A                               0.001085
RTU: ava. Setpoint 1                                  0.019532
Inverter std dev                                     -0.069573
Name: Error, dtype: float64

Number of columns with correlation below 0.1 for 'Error': 20

Columns with correlation above or equal to the absolute threshold of 0.1 for
'Error':
Error                             1.000000
WEC: ava. windspeed              -0.108580
WEC: max. windspeed              -0.102705
WEC: ava. Rotation               -0.134699
WEC: max. Rotation               -0.136421
WEC: Operating Hours             -0.113564
Sys 1 inverter 1 cabinet temp.   -0.260673
Sys 1 inverter 2 cabinet temp.   -0.277267
Sys 1 inverter 3 cabinet temp.   -0.291097
Sys 1 inverter 4 cabinet temp.   -0.293301
Sys 1 inverter 5 cabinet temp.   -0.291151
Sys 1 inverter 6 cabinet temp.   -0.286002
Sys 1 inverter 7 cabinet temp.   -0.297080
Sys 2 inverter 1 cabinet temp.   -0.315480
Sys 2 inverter 2 cabinet temp.   -0.311920
Sys 2 inverter 3 cabinet temp.   -0.294460
Sys 2 inverter 4 cabinet temp.   -0.290495
Sys 2 inverter 5 cabinet temp.    0.982501
Sys 2 inverter 7 cabinet temp.    0.966030
Spinner temp.                    -0.196058
Front bearing temp.              -0.264613
Rear bearing temp.               -0.308946
Pitch cabinet blade A temp.      -0.345752
Pitch cabinet blade B temp.      -0.350812
```

```
Pitch cabinet blade C temp.      -0.349681
Blade A temp.                    -0.980186
Blade B temp.                    -0.980764
Blade C temp.                    -0.980137
Rotor temp. 1                    -0.141469
Rotor temp. 2                    -0.143229
Stator temp. 1                   -0.158704
Stator temp. 2                   -0.158978
Nacelle ambient temp. 1          -0.151876
Nacelle ambient temp. 2          -0.150921
Nacelle temp.                    -0.192121
Nacelle cabinet temp.            -0.218781
Main carrier temp.               -0.203210
Rectifier cabinet temp.          -0.322171
Yaw inverter cabinet temp.       -0.293122
Fan inverter cabinet temp.       -0.329463
Ambient temp.                    -0.146742
Tower temp.                      -0.218595
Control cabinet temp.            -0.298812
Transformer temp.                -0.251442
Inverter averages                -0.309423
Name: Error, dtype: float64

Number of columns with correlation above or equal to 0.1 for 'Error': 45
```

[29]:
```
# Assume df is your DataFrame and 'Error' is your target column
# Select only the numeric columns
X_numeric = df.select_dtypes(include=[np.number])

# Handle missing values - filling NaNs with 0 (you could use other methods␣
 ↪based on your context)
X_numeric.fillna(0, inplace=True)

# Add constant to the dataset (necessary for VIF calculation)
X_numeric_with_const = add_constant(X_numeric)

# Calculate VIF for each feature
vif_data = pd.DataFrame()
vif_data["Feature"] = X_numeric_with_const.columns
vif_data["VIF"] = [variance_inflation_factor(X_numeric_with_const.values, i)␣
 ↪for i in range(X_numeric_with_const.shape[1])]

# Set a VIF threshold (between 6 and 10) and filter based on it
vif_threshold_lower = 6
vif_threshold_upper = 10
filtered_vif = vif_data[(vif_data["VIF"] >= vif_threshold_lower) &␣
 ↪(vif_data["VIF"] <= vif_threshold_upper)]
```

```
# Now filter features that are correlated with 'Error' column
error_correlations = X_numeric.corrwith(df['Error']).abs()  # Absolute␣
 ↪correlations with 'Error'
filtered_features = filtered_vif[filtered_vif["Feature"].
 ↪isin(error_correlations[error_correlations > 0].index)]

# Print the filtered features
print(f"Filtered features with VIF between {vif_threshold_lower} and␣
 ↪{vif_threshold_upper} and correlated with 'Error':")
print(filtered_features)
```

```
D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\statsmodels\regression\linear_model.py:1782: RuntimeWarning: divide by
zero encountered in scalar divide
  return 1 - self.ssr/self.centered_tss
D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\statsmodels\regression\linear_model.py:1782: RuntimeWarning: invalid
value encountered in scalar divide
  return 1 - self.ssr/self.centered_tss
```

```
Filtered features with VIF between 6 and 10 and correlated with 'Error':
                    Feature       VIF
17  WEC: max. reactive Power  8.139816
65          Inverter std dev  8.203170
```

```
D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\numpy\lib\function_base.py:2897: RuntimeWarning: invalid value
encountered in divide
  c /= stddev[:, None]
D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\numpy\lib\function_base.py:2898: RuntimeWarning: invalid value
encountered in divide
  c /= stddev[None, :]
```

- **WEC: max. reactive Power** (VIF: 8.14) and **Inverter std dev** (VIF: 8.20) are the features with moderate to high multicollinearity (VIF between 6 and 10).
  - These features have a relatively high correlation with other predictor variables, which might indicate some redundancy in the information they provide.
  - The presence of high multicollinearity could distort the model's ability to accurately estimate the coefficients for these features, leading to unstable predictions.
- Consider **reviewing these features** further, especially if they are highly correlated with other features or each other.
- You might **drop** these features, **combine** them with others.

[30]:
```
# List of columns to drop manually based on their correlation with 'Error'
columns_to_drop = [ 'Time',
```

```
    'WEC: min. windspeed', 'WEC: max. windspeed',
    'WEC: min. Rotation', 'WEC: max. Rotation',
    'WEC: max. Power', 'WEC: min. Power',
    'WEC: min. reactive Power', 'WEC: max. reactive Power',
    'Sys 2 inverter 5 cabinet temp.',
    'Sys 2 inverter 6 cabinet temp.', 'Sys 2 inverter 7 cabinet temp.',
    'WEC: ava. Nacel position including cable twisting',
    'WEC: Production kWh',
    'WEC: Production minutes',
    'WEC: ava. available P from wind',
    'WEC: ava. available P technical reasons',
    'WEC: ava. Available P force majeure reasons',
    'WEC: ava. Available P force external reasons',
    'WEC: ava. blade angle A',
    'RTU: ava. Setpoint 1',
    'Inverter std dev',
    'Blade A Temp.','Blade B temp.', 'Blade C temp.'
]

# Drop specified columns from the dataset
df_filtered = df.drop(columns=columns_to_drop, errors='ignore')

# Print remaining columns and their count
remaining_columns = df_filtered.columns.tolist()
print(f"Remaining columns: {remaining_columns}")
print(f"Number of remaining columns: {len(remaining_columns)}")
```

```
Remaining columns: ['DateTime', 'Error', 'WEC: ava. windspeed', 'WEC: ava.
Rotation', 'WEC: ava. Power', 'WEC: Operating Hours', 'WEC: ava. reactive
Power', 'Sys 1 inverter 1 cabinet temp.', 'Sys 1 inverter 2 cabinet temp.', 'Sys
1 inverter 3 cabinet temp.', 'Sys 1 inverter 4 cabinet temp.', 'Sys 1 inverter 5
cabinet temp.', 'Sys 1 inverter 6 cabinet temp.', 'Sys 1 inverter 7 cabinet
temp.', 'Sys 2 inverter 1 cabinet temp.', 'Sys 2 inverter 2 cabinet temp.', 'Sys
2 inverter 3 cabinet temp.', 'Sys 2 inverter 4 cabinet temp.', 'Spinner temp.',
'Front bearing temp.', 'Rear bearing temp.', 'Pitch cabinet blade A temp.',
'Pitch cabinet blade B temp.', 'Pitch cabinet blade C temp.', 'Blade A temp.',
'Rotor temp. 1', 'Rotor temp. 2', 'Stator temp. 1', 'Stator temp. 2', 'Nacelle
ambient temp. 1', 'Nacelle ambient temp. 2', 'Nacelle temp.', 'Nacelle cabinet
temp.', 'Main carrier temp.', 'Rectifier cabinet temp.', 'Yaw inverter cabinet
temp.', 'Fan inverter cabinet temp.', 'Ambient temp.', 'Tower temp.', 'Control
cabinet temp.', 'Transformer temp.', 'Inverter averages']
Number of remaining columns: 42
```

The cols are dropped based upon either they had high correlation (more than abs 0.9) and very low correlation(below and equal to abs 0.1) as well as anomaly cols **Sys 2 Inverter 6 Cabinet Temp**. Later on in steps Preparation and Transformation we will add means features for cols like sys 1 and sys 2 temps

[31]:
```
# Extract meaningful time-related features
df_filtered['Month'] = df_filtered['DateTime'].dt.month
df_filtered['Week'] = df_filtered['DateTime'].dt.isocalendar().week
df_filtered['DayOfWeek'] = df_filtered['DateTime'].dt.dayofweek  # 0=Monday,␣
 ↪6=Sunday

# Add a seasonal feature with encoding
def assign_season(month):
    if month in [12, 1, 2]:
        return 0  # Winter
    elif month in [3, 4, 5]:
        return 1  # Spring
    elif month in [6, 7, 8]:
        return 2  # Summer
    elif month in [9, 10, 11]:
        return 3  # Autumn

df_filtered['Season'] = df_filtered['Month'].apply(assign_season)

# Display the first few rows to verify
print(df_filtered[['DateTime', 'Month', 'Week', 'DayOfWeek', 'Season']].head())
```

```
             DateTime  Month  Week  DayOfWeek  Season
0 2014-05-01 00:00:00      5    18          3       1
1 2014-05-01 00:09:00      5    18          3       1
2 2014-05-01 00:20:00      5    18          3       1
3 2014-05-01 00:30:00      5    18          3       1
4 2014-05-01 00:39:00      5    18          3       1
```

[32]:
```
# Aggregating system temperatures
df_filtered['Avg sys 1 inverter temp'] = df_filtered[['Sys 1 inverter 1 cabinet␣
 ↪temp.', 'Sys 1 inverter 2 cabinet temp.',
                                                      'Sys 1 inverter 3 cabinet temp.
 ↪', 'Sys 1 inverter 4 cabinet temp.',
                                                      'Sys 1 inverter 5 cabinet temp.
 ↪', 'Sys 1 inverter 6 cabinet temp.',
                                                      'Sys 1 inverter 7 cabinet temp.
 ↪']].mean(axis=1)

df_filtered['Avg sys 2 inverter temp'] = df_filtered[['Sys 2 inverter 1 cabinet␣
 ↪temp.', 'Sys 2 inverter 2 cabinet temp.',
                                                          'Sys 2 inverter 3 cabinet␣
 ↪temp.', 'Sys 2 inverter 4 cabinet temp.']].mean(axis=1)
# Calculate the mean for Rotor temps
df_filtered['Rotor temp mean'] = df_filtered[['Rotor temp. 1', 'Rotor temp.␣
 ↪2']].mean(axis=1)
```

```
# Calculate the mean for Stator temps
df_filtered['Stator temp mean'] = df_filtered[['Stator temp. 1', 'Stator temp.␣
 ↪2']].mean(axis=1)

# Calculate the mean for Nacelle ambient temps
df_filtered['Nacelle ambient temp mean'] = df_filtered[['Nacelle ambient temp.␣
 ↪1', 'Nacelle ambient temp. 2']].mean(axis=1)
```

[33]:
```
# Define the condition for binary classification
df_filtered['Error Binary'] = df_filtered['Error'].apply(lambda x: 0 if x == 0␣
 ↪else 1)

# Check the distribution of the new binary classes
binary_distribution = df_filtered['Error Binary'].value_counts()

# Print the distribution
print("Distribution of 'Error Binary':")
print(binary_distribution)

# Calculate the total number of "Error Occurred" entries
error_total = binary_distribution[1]  # Count of rows where Error_Class_Binary␣
 ↪= 1
no_error_total = binary_distribution[0]  # Count of rows where␣
 ↪Error_Class_Binary = 0

print(f"\nSummary:")
print(f"Total No Error instances: {no_error_total}")
print(f"Total Error Occurred instances: {error_total}")
```

```
Distribution of 'Error Binary':
Error Binary
0    48727
1      300
Name: count, dtype: int64

Summary:
Total No Error instances: 48727
Total Error Occurred instances: 300
```

[34]:
```
columns_to_process = ['WEC: ava. windspeed', 'WEC: ava. Power', 'WEC: ava.␣
 ↪Rotation', 'WEC: ava. reactive Power']


# Winsorize outliers (cap at 1st and 99th percentiles)
for col in columns_to_process:
    lower_bound = np.percentile(df_filtered[col], 1)
    upper_bound = np.percentile(df_filtered[col], 99)
```

```
    df_filtered[col] = np.clip(df_filtered[col], lower_bound, upper_bound)

# Apply Min-Max Scaling
scaler = MinMaxScaler()
df_filtered[columns_to_process] = scaler.
 ↪fit_transform(df_filtered[columns_to_process])

# Display the scaled DataFrame
df_filtered.describe()
```

[34]:

	DateTime	Error	WEC: ava. windspeed	\
count	49027	49027.000000	49027.000000	
mean	2014-10-19 18:19:49.915760640	0.938748	0.371083	
min	2014-05-01 00:00:00	0.000000	0.000000	
25%	2014-07-26 00:14:30	0.000000	0.218391	
50%	2014-10-19 03:00:00	0.000000	0.350575	
75%	2015-01-13 13:54:00	0.000000	0.488506	
max	2015-04-09 00:00:00	246.000000	1.000000	
std	NaN	14.442141	0.207993	

	WEC: ava. Rotation	WEC: ava. Power	WEC: Operating Hours	\
count	49027.000000	49027.000000	49027.000000	
mean	0.590772	0.307024	4214.576866	
min	0.000000	0.000000	0.000000	
25%	0.430905	0.028348	2360.000000	
50%	0.610620	0.174650	4179.000000	
75%	0.811436	0.505376	6008.000000	
max	1.000000	1.000000	7884.000000	
std	0.282051	0.328747	2113.523332	

	WEC: ava. reactive Power	Sys 1 inverter 1 cabinet temp.	\
count	49027.000000	49027.000000	
mean	0.277163	26.633712	
min	0.000000	-14.000000	
25%	0.100671	23.000000	
50%	0.164430	27.000000	
75%	0.362416	31.000000	
max	1.000000	42.000000	
std	0.281466	6.361488	

	Sys 1 inverter 2 cabinet temp.	Sys 1 inverter 3 cabinet temp.	…	\
count	49027.000000	49027.000000	…	
mean	27.703816	27.279560	…	
min	-14.000000	-14.000000	…	
25%	24.000000	24.000000	…	
50%	28.000000	28.000000	…	
75%	32.000000	31.000000	…	

```
max                          45.000000                          43.000000  …
std                           6.220329                           5.811341  …

              Month       Week     DayOfWeek        Season  \
count  49027.000000    49027.0  49027.000000  49027.000000
mean       6.682644  27.086381      2.993208      1.531727
min        1.000000        1.0      0.000000      0.000000
25%        3.000000       13.0      1.000000      0.000000
50%        7.000000       28.0      3.000000      2.000000
75%       10.000000       40.0      5.000000      3.000000
max       12.000000       52.0      6.000000      3.000000
std        3.498049  15.253814      1.998024      1.141495

       Avg sys 1 inverter temp  Avg sys 2 inverter temp  Rotor temp mean  \
count             49027.000000             49027.000000     49027.000000
mean                 27.547803                28.319472        52.699237
min                 -14.000000               -14.000000         0.000000
25%                  24.142857                24.750000        42.000000
50%                  28.142857                28.750000        48.000000
75%                  31.571429                32.500000        56.000000
max                  42.428571                42.750000       125.500000
std                   5.781560                 5.677413        22.826007

       Stator temp mean  Nacelle ambient temp mean  Error Binary
count      49027.000000               49027.000000  49027.000000
mean          60.558468                  12.472403      0.006119
min            0.000000                   0.000000      0.000000
25%           45.000000                   9.000000      0.000000
50%           61.500000                  12.500000      0.000000
75%           73.500000                  16.000000      0.000000
max          118.000000                  28.000000      1.000000
std           23.642834                   4.744836      0.077986

[8 rows x 52 columns]
```

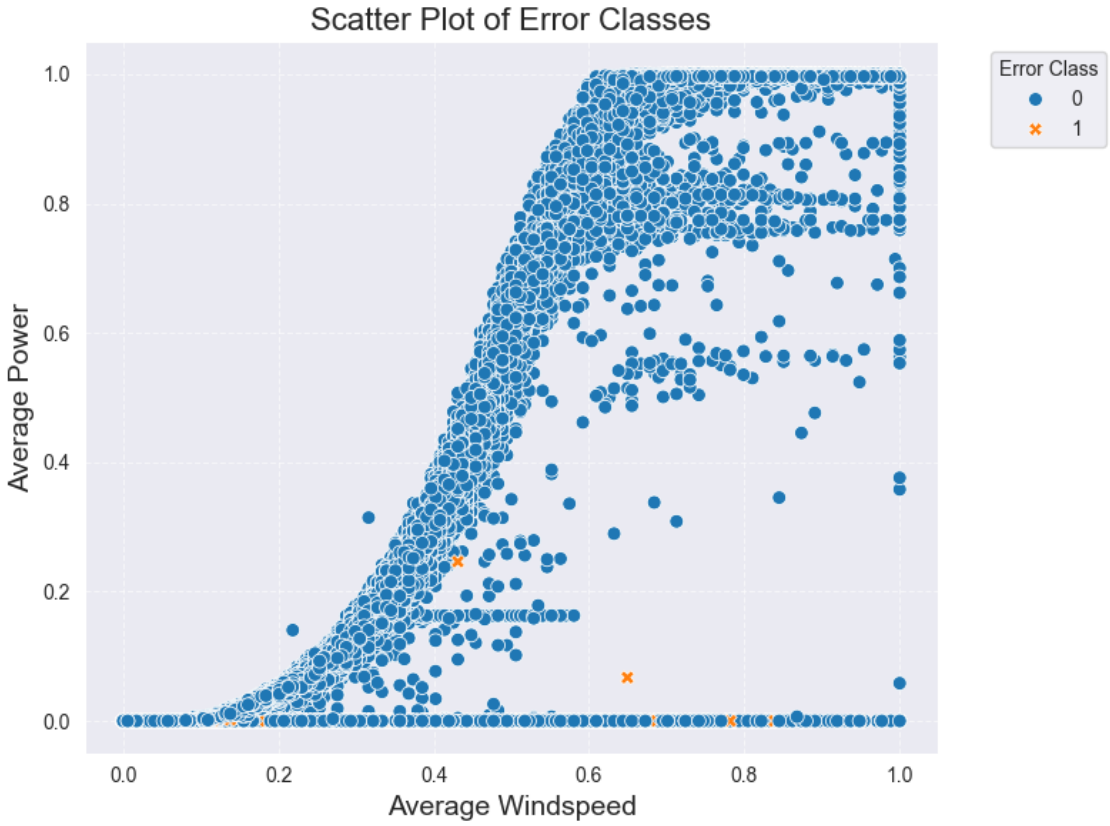
[35]:
```
# Drop columns that will not help in prediction or are redundant
cols_to_drop = ['Error', 'DateTime', 'Sys 1 inverter 1 cabinet temp.', 'Sys 1␣
 ↪inverter 2 cabinet temp.',
                                                 'Sys 1 inverter 3 cabinet temp.
 ↪', 'Sys 1 inverter 4 cabinet temp.',
                                                 'Sys 1 inverter 5 cabinet temp.
 ↪', 'Sys 1 inverter 6 cabinet temp.',
                                                 'Sys 1 inverter 7 cabinet temp.
 ↪', 'Sys 2 inverter 1 cabinet temp.', 'Sys 2 inverter 2 cabinet temp.',␣
 ↪"Rotor temp. 1",
    "Rotor temp. 2",
    "Stator temp. 1",
```

```
    "Stator temp. 2",
    "Nacelle ambient temp. 1",
    "Nacelle ambient temp. 2",
                                                    'Sys 2 inverter 3 cabinet␣
 ↪temp.', 'Sys 2 inverter 4 cabinet temp.']  # 'Error' could be dropped if you␣
 ↪are creating separate classification tasks.
df_filtered = df_filtered.drop(columns=cols_to_drop, errors='ignore')

# 8. **Final Dataset Inspection**
print(f"Shape of the dataset after feature engineering: {df_filtered.shape}")
print("Columns after feature engineering:")
print(df_filtered.columns.tolist())
```

```
Shape of the dataset after feature engineering: (49027, 33)
Columns after feature engineering:
['WEC: ava. windspeed', 'WEC: ava. Rotation', 'WEC: ava. Power', 'WEC: Operating
Hours', 'WEC: ava. reactive Power', 'Spinner temp.', 'Front bearing temp.',
'Rear bearing temp.', 'Pitch cabinet blade A temp.', 'Pitch cabinet blade B
temp.', 'Pitch cabinet blade C temp.', 'Blade A temp.', 'Nacelle temp.',
'Nacelle cabinet temp.', 'Main carrier temp.', 'Rectifier cabinet temp.', 'Yaw
inverter cabinet temp.', 'Fan inverter cabinet temp.', 'Ambient temp.', 'Tower
temp.', 'Control cabinet temp.', 'Transformer temp.', 'Inverter averages',
'Month', 'Week', 'DayOfWeek', 'Season', 'Avg sys 1 inverter temp', 'Avg sys 2
inverter temp', 'Rotor temp mean', 'Stator temp mean', 'Nacelle ambient temp
mean', 'Error Binary']
```

[36]:
```
# Scatter plot with different colors for each Error_Class
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=df_filtered,
    x='WEC: ava. windspeed',  # Replace with an appropriate feature for the␣
 ↪x-axis
    y='WEC: ava. Power',      # Replace with an appropriate feature for the␣
 ↪y-axis
    hue='Error Binary',        # Class-based coloring
    palette='tab10',          # Use a color palette with enough distinct colors
    style='Error Binary',      # Different markers for each class
    s=50                      # Marker size
)

# Add labels and title
plt.title('Scatter Plot of Error Classes', fontsize=16)
plt.xlabel('Average Windspeed', fontsize=14)
plt.ylabel('Average Power', fontsize=14)
plt.legend(title='Error Class', bbox_to_anchor=(1.05, 1), loc='upper left')  #␣
 ↪Legend outside the plot
plt.grid(True, linestyle='--', alpha=0.6)
```

```
plt.tight_layout()
plt.show()
```



2.4 Model Development and Training

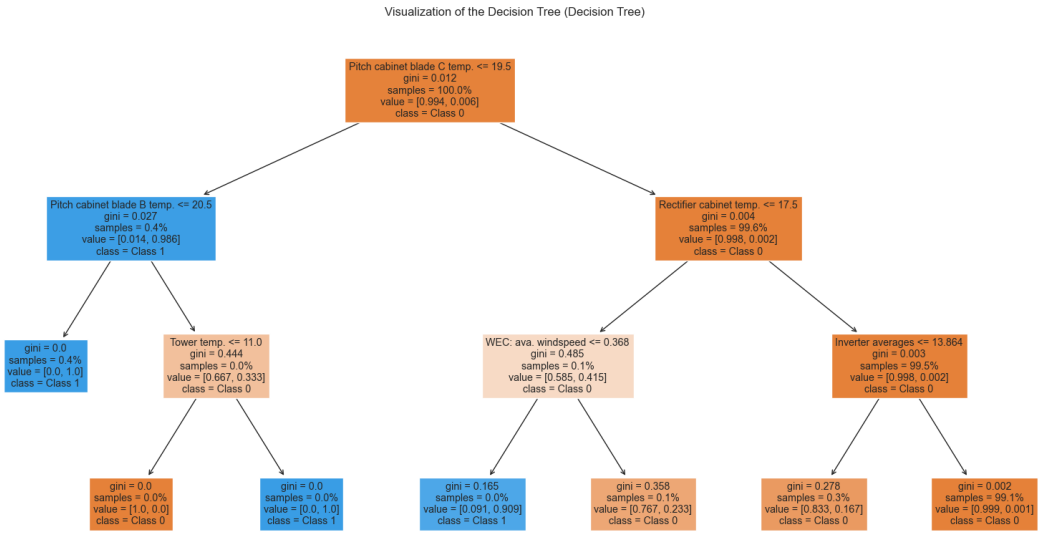
[37]:
```
# 1. Split dataset into features (X) and target (y)
X = df_filtered.drop(columns=['Error Binary'])
y = df_filtered['Error Binary']
```

[38]:
```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=42, stratify=y)
```

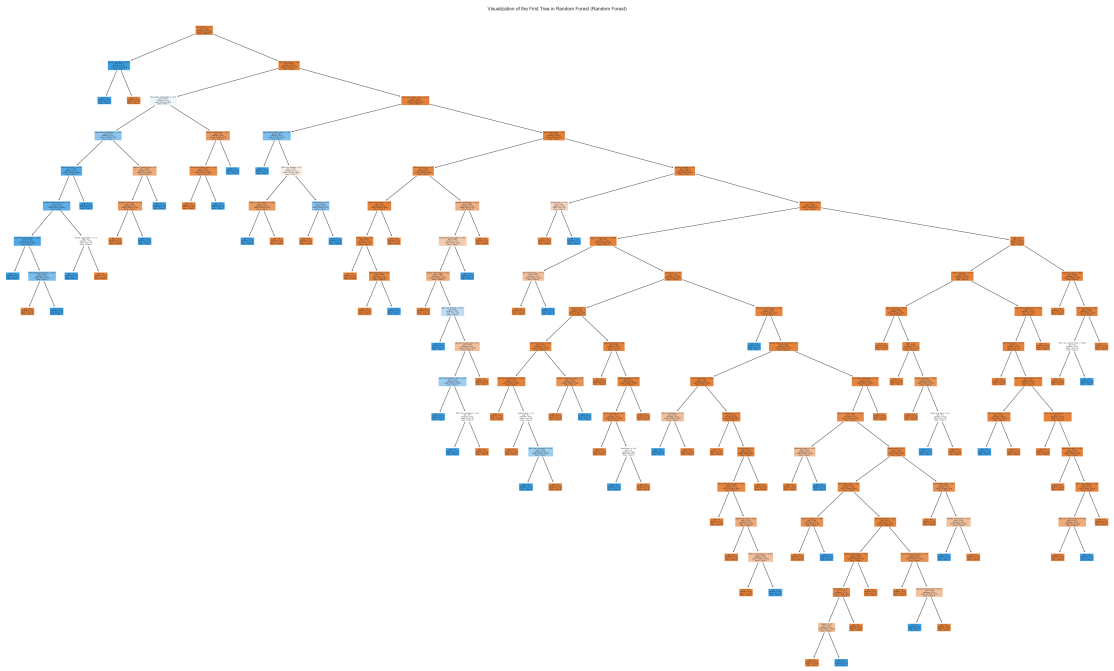
[39]:
```
# Define a dictionary of models to test
models = {
    'Decision Tree': DecisionTreeClassifier(max_depth=3, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=50, random_state=42),
    'XGBoost': XGBClassifier(random_state=42, eval_metric='mlogloss')  #␣
 ↪Configure XGBoost
}
```

[40]:
```
# Training Section (Fit)
for model_name, model in models.items():
    print(f"\nTraining {model_name}...")

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

    # If the model is a Decision Tree, plot the tree structure
    if isinstance(model, DecisionTreeClassifier):
        plt.figure(figsize=(20, 10))
        plot_tree(model, filled=True, feature_names=X_train.columns.tolist(),␣
 ↪class_names=['Class 0', 'Class 1'], proportion=True)
        plt.title(f'Visualization of the Decision Tree ({model_name})')
        plt.show()

    # If the model is a Random Forest, visualize the first tree of the forest
    if isinstance(model, RandomForestClassifier):
        # Visualize the first tree of the Random Forest
        plt.figure(figsize=(50, 30))
        plot_tree(model.estimators_[0], filled=True, feature_names=X_train.
 ↪columns.tolist(), class_names=['Class 0', 'Class 1'], proportion=True)
        plt.title(f'Visualization of the First Tree in Random Forest␣
 ↪({model_name})')
        plt.show()

    # If the model is XGBoost, visualize the first tree
    if isinstance(model, XGBClassifier):
        # Optionally, plot the first tree of the XGBoost model
        plt.figure(figsize=(20, 10))
        xgb.plot_tree(model, num_trees=0)
        plt.title(f'Visualization of the First Tree in XGBoost ({model_name})')
        plt.show()
```

```
Training Decision Tree…
```



Training Random Forest…



Training XGBoost…

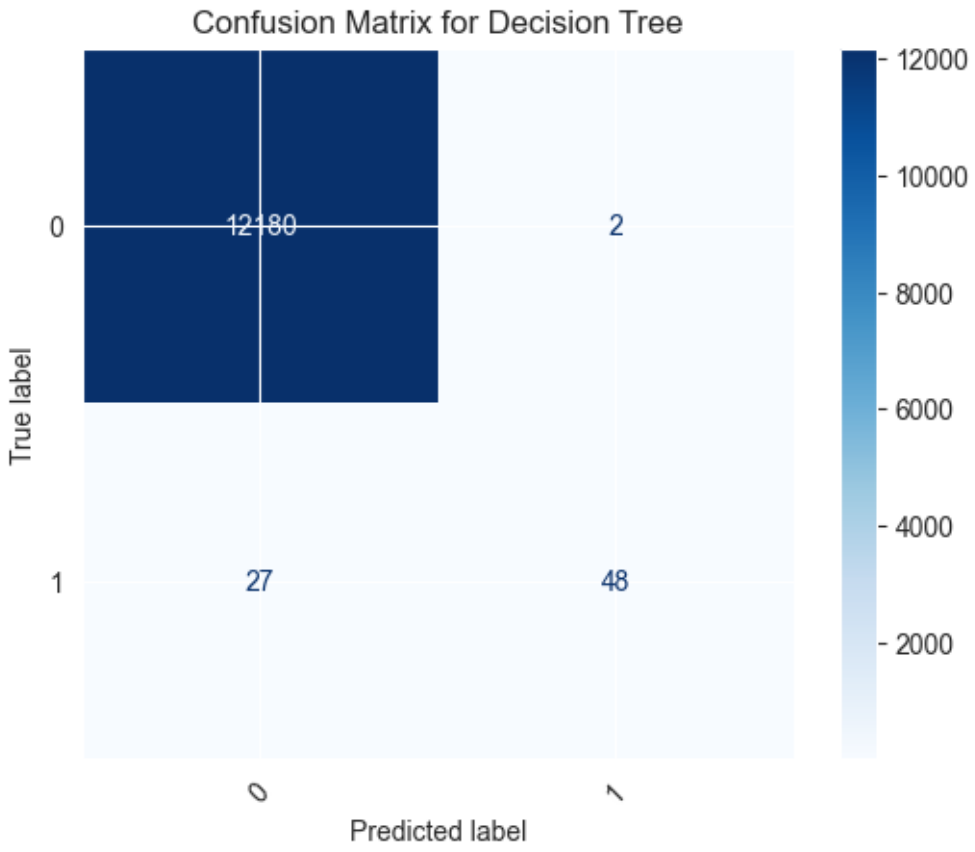
<Figure size 2000x1000 with 0 Axes>

2.5 Model Validation and Evaluation

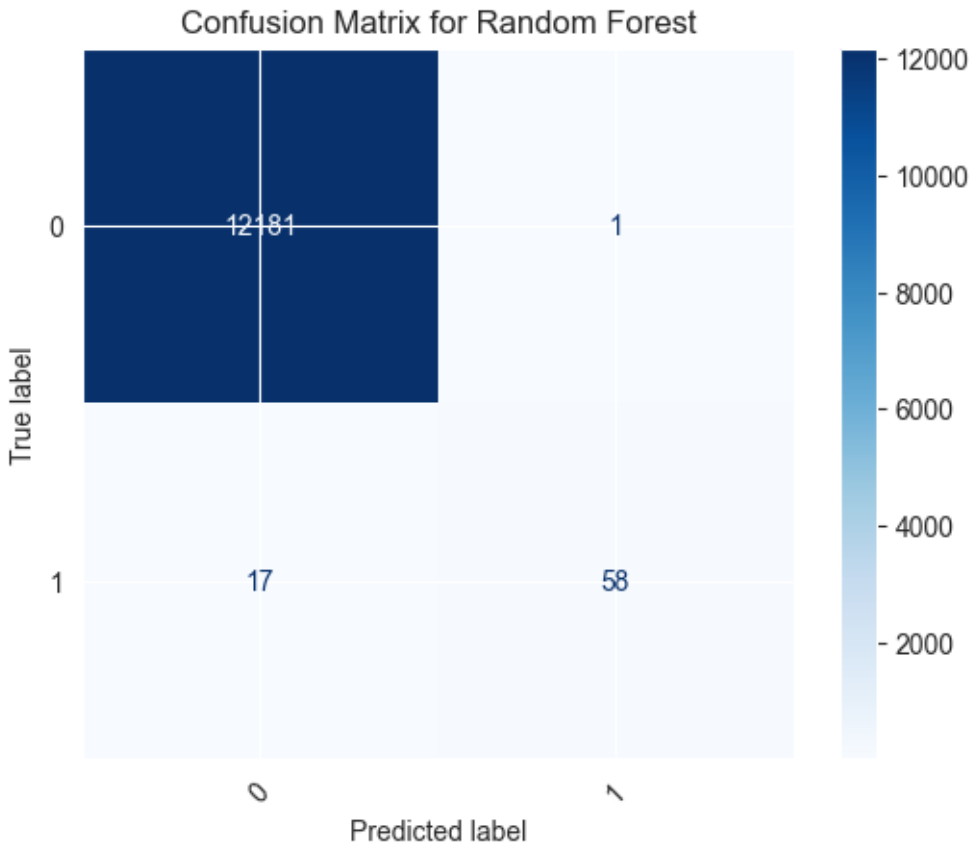
[41]:
```
# Evaluation Section
evaluation_results = {}

for model_name, model in models.items():
    print(f"\nEvaluating {model_name}...")

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

    # Compute evaluation metrics
    accuracy = accuracy_score(y_test, y_pred)
    evaluation_results[model_name] = {'Accuracy': accuracy}

    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Print results
    print(f"Accuracy for {model_name}: {accuracy:.4f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    print("\nConfusion Matrix:")
    print(cm)

    # Plot confusion matrix
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
↪classes_)
    disp.plot(cmap=plt.cm.Blues, xticks_rotation=45)
    plt.title(f"Confusion Matrix for {model_name}")
```

```
    plt.show()

# Summary of Results
print("\n--- Evaluation Results ---")
for model_name, metrics in evaluation_results.items():
    print(f"{model_name} - Accuracy: {metrics['Accuracy']:.4f}")
```

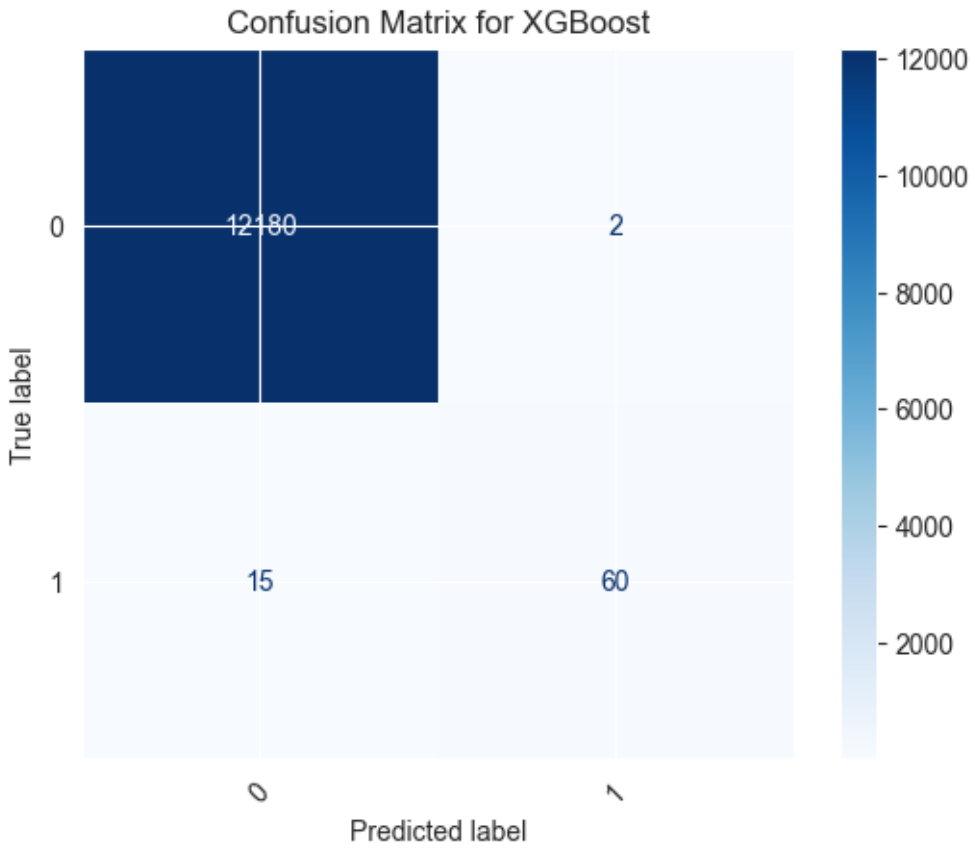
```
Evaluating Decision Tree…
Accuracy for Decision Tree: 0.9976

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.96      0.64      0.77        75

    accuracy                           1.00     12257
   macro avg       0.98      0.82      0.88     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12180     2]
 [   27    48]]
```




```
Evaluating Random Forest…
Accuracy for Random Forest: 0.9985

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.98      0.77      0.87        75

    accuracy                           1.00     12257
   macro avg       0.99      0.89      0.93     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12181     1]
 [   17    58]]
```



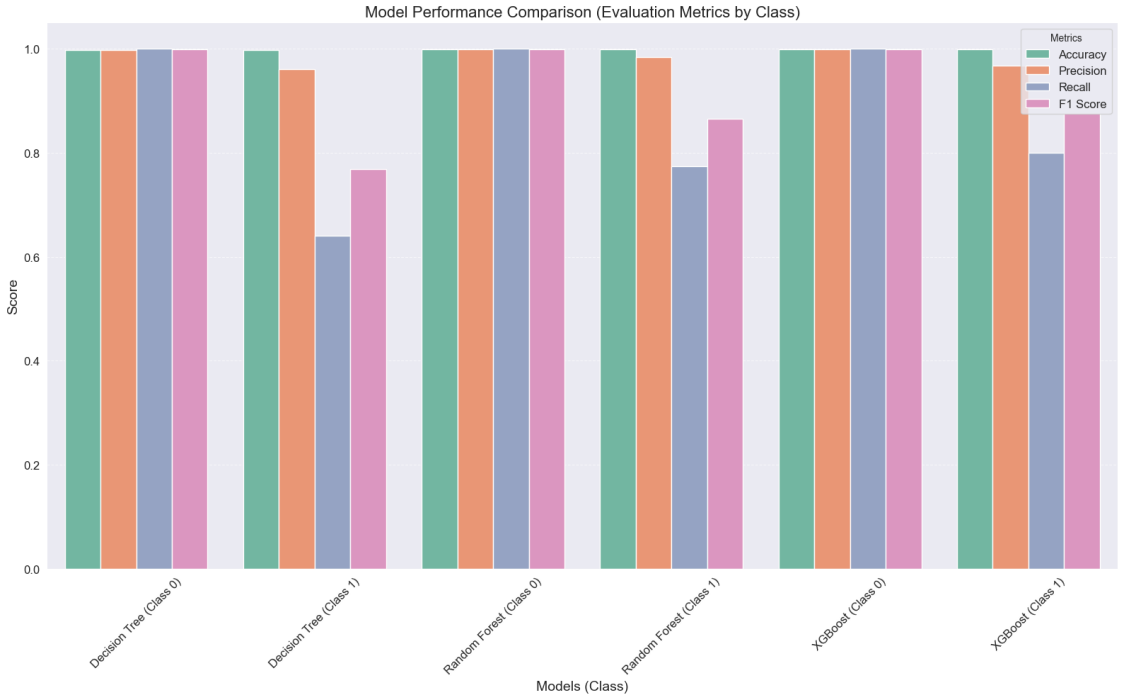

```
Evaluating XGBoost…
Accuracy for XGBoost: 0.9986

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.97      0.80      0.88        75

    accuracy                           1.00     12257
   macro avg       0.98      0.90      0.94     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12180     2]
 [   15    60]]
```




```
--- Evaluation Results ---
Decision Tree - Accuracy: 0.9976
Random Forest - Accuracy: 0.9985
XGBoost - Accuracy: 0.9986
```

[42]:
```python
# Initialize results list to store evaluation metrics for each model
results = []

# Evaluate each model and calculate metrics for both classes
for model_name, model in models.items():
    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics for Class 0
    accuracy = accuracy_score(y_test, y_pred)
    precision_0 = precision_score(y_test, y_pred, pos_label=0)
    recall_0 = recall_score(y_test, y_pred, pos_label=0)
    f1_0 = f1_score(y_test, y_pred, pos_label=0)
```

```
    results.append({'Model': f"{model_name} (Class 0)", 'Metric': 'Accuracy',␣
 ↪'Value': accuracy})
    results.append({'Model': f"{model_name} (Class 0)", 'Metric': 'Precision',␣
 ↪'Value': precision_0})
    results.append({'Model': f"{model_name} (Class 0)", 'Metric': 'Recall',␣
 ↪'Value': recall_0})
    results.append({'Model': f"{model_name} (Class 0)", 'Metric': 'F1 Score',␣
 ↪'Value': f1_0})

    # Calculate metrics for Class 1
    precision_1 = precision_score(y_test, y_pred, pos_label=1)
    recall_1 = recall_score(y_test, y_pred, pos_label=1)
    f1_1 = f1_score(y_test, y_pred, pos_label=1)

    results.append({'Model': f"{model_name} (Class 1)", 'Metric': 'Accuracy',␣
 ↪'Value': accuracy})  # Accuracy is the same
    results.append({'Model': f"{model_name} (Class 1)", 'Metric': 'Precision',␣
 ↪'Value': precision_1})
    results.append({'Model': f"{model_name} (Class 1)", 'Metric': 'Recall',␣
 ↪'Value': recall_1})
    results.append({'Model': f"{model_name} (Class 1)", 'Metric': 'F1 Score',␣
 ↪'Value': f1_1})

# Convert results to DataFrame
comparison_df = pd.DataFrame(results)

# Plot model performance comparison (all evaluation metrics for both classes)
plt.figure(figsize=(16, 10))
sns.barplot(data=comparison_df, x="Model", y="Value", hue="Metric",␣
 ↪palette="Set2", dodge=True)
plt.title("Model Performance Comparison (Evaluation Metrics by Class)",␣
 ↪fontsize=16)
plt.ylabel("Score", fontsize=14)
plt.xlabel("Models (Class)", fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.legend(title="Metrics", loc="upper right", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Print the metrics table for both classes
print("\nDetailed Metrics for Each Model and Class:")
print(comparison_df)
```



Detailed Metrics for Each Model and Class:

	Model	Metric	Value
0	Decision Tree (Class 0)	Accuracy	0.997634
1	Decision Tree (Class 0)	Precision	0.997788
2	Decision Tree (Class 0)	Recall	0.999836
3	Decision Tree (Class 0)	F1 Score	0.998811
4	Decision Tree (Class 1)	Accuracy	0.997634
5	Decision Tree (Class 1)	Precision	0.960000
6	Decision Tree (Class 1)	Recall	0.640000
7	Decision Tree (Class 1)	F1 Score	0.768000
8	Random Forest (Class 0)	Accuracy	0.998531
9	Random Forest (Class 0)	Precision	0.998606
10	Random Forest (Class 0)	Recall	0.999918
11	Random Forest (Class 0)	F1 Score	0.999262
12	Random Forest (Class 1)	Accuracy	0.998531
13	Random Forest (Class 1)	Precision	0.983051
14	Random Forest (Class 1)	Recall	0.773333
15	Random Forest (Class 1)	F1 Score	0.865672
16	XGBoost (Class 0)	Accuracy	0.998613
17	XGBoost (Class 0)	Precision	0.998770
18	XGBoost (Class 0)	Recall	0.999836
19	XGBoost (Class 0)	F1 Score	0.999303
20	XGBoost (Class 1)	Accuracy	0.998613
21	XGBoost (Class 1)	Precision	0.967742

```
22        XGBoost (Class 1)     Recall  0.800000
23        XGBoost (Class 1)   F1 Score  0.875912
```

[43]:
```python
# Ensure the models dictionary has only unique models
unique_models = {model_name: model for model_name, model in models.items()}

# Iterate through unique models for evaluation
for model_name, model in unique_models.items():
    print(f"\nEvaluating {model_name}...")

    # Training performance
    y_train_pred = model.predict(X_train)  # Use the specific model object to␣
 ↪predict
    train_accuracy = accuracy_score(y_train, y_train_pred)

    # Test performance
    y_test_pred = model.predict(X_test)  # Use the specific model object to␣
 ↪predict
    test_accuracy = accuracy_score(y_test, y_test_pred)

    print(f"Training Accuracy for {model_name}: {train_accuracy:.4f}")
    print(f"Test Accuracy for {model_name}: {test_accuracy:.4f}")
```

```
Evaluating Decision Tree…
Training Accuracy for Decision Tree: 0.9981
Test Accuracy for Decision Tree: 0.9976

Evaluating Random Forest…
Training Accuracy for Random Forest: 1.0000
Test Accuracy for Random Forest: 0.9985

Evaluating XGBoost…
Training Accuracy for XGBoost: 1.0000
Test Accuracy for XGBoost: 0.9986
```

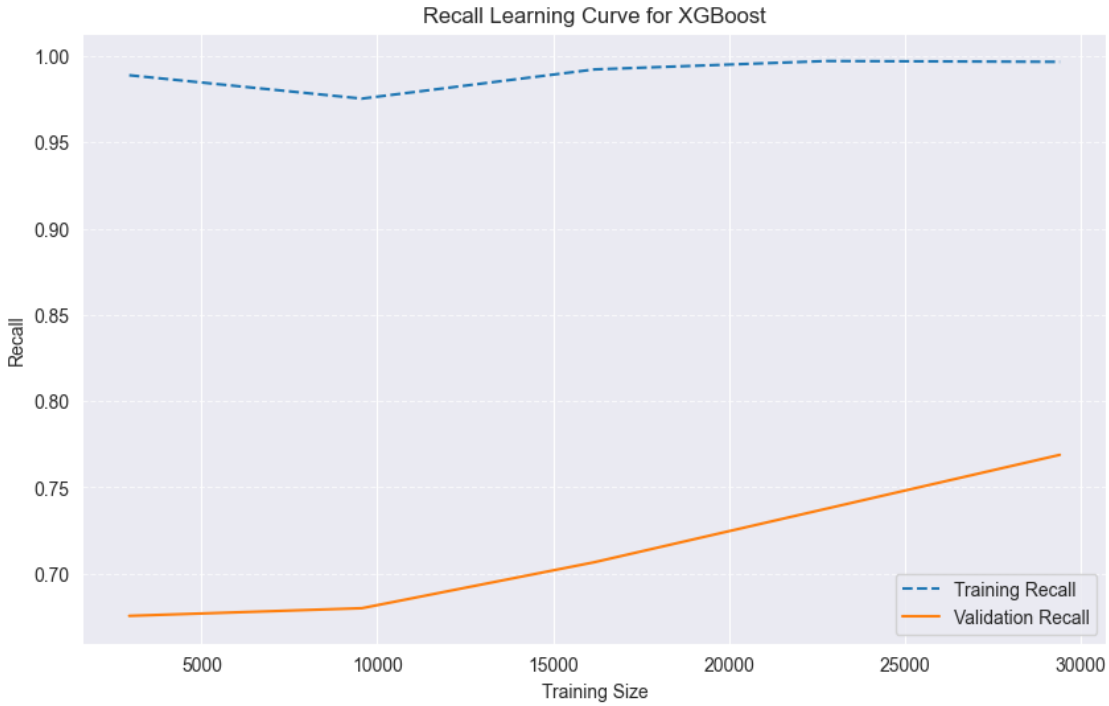
[44]:
```python
# Iterate through each model to plot learning curves for Recall and F1-Score
for model_name, model in models.items():
    print(f"\nPlotting Learning Curve for {model_name}...")

    # Generate learning curve for Recall
    train_sizes_recall, train_scores_recall, validation_scores_recall =␣
 ↪learning_curve(
        model, X_train, y_train, cv=5, scoring=make_scorer(recall_score,␣
 ↪pos_label=1)
    )
```

```
    # Plot Recall learning curve
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes_recall, train_scores_recall.mean(axis=1),␣
↪label='Training Recall', linestyle='--')
    plt.plot(train_sizes_recall, validation_scores_recall.mean(axis=1),␣
↪label='Validation Recall')
    plt.xlabel('Training Size')
    plt.ylabel('Recall')
    plt.title(f'Recall Learning Curve for {model_name}')
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()
```

```
Plotting Learning Curve for Decision Tree…
```


```
Plotting Learning Curve for Random Forest…
```


```
Plotting Learning Curve for XGBoost…
```



[45]:
```
# Set up Stratified K-Fold cross-validation
stratified_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Iterate through each model and compute Stratified K-Fold cross-validation␣
 ↪scores
for model_name, model in models.items():
    print(f"\nPerforming Stratified Cross-Validation for {model_name}...")

    # Perform cross-validation
    cross_val_scores = cross_val_score(model, X_train, y_train,␣
 ↪cv=stratified_kfold, scoring='recall')

    # Print the results for the current model
    print(f"Stratified Cross-Validation Scores for {model_name}:␣
 ↪{cross_val_scores}")
    print(f"Mean Stratified Cross-Validation Score for {model_name}:␣
 ↪{cross_val_scores.mean():.4f}")
```

```
Performing Stratified Cross-Validation for Decision Tree…
Stratified Cross-Validation Scores for Decision Tree: [0.73333333 0.68888889
0.73333333 0.57777778 0.71111111]
Mean Stratified Cross-Validation Score for Decision Tree: 0.6889

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.82222222 0.68888889
0.75555556 0.64444444 0.71111111]
Mean Stratified Cross-Validation Score for Random Forest: 0.7244

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.86666667 0.71111111
0.82222222 0.64444444 0.77777778]
Mean Stratified Cross-Validation Score for XGBoost: 0.7644
```

2.6 Hyperparameter Tuning

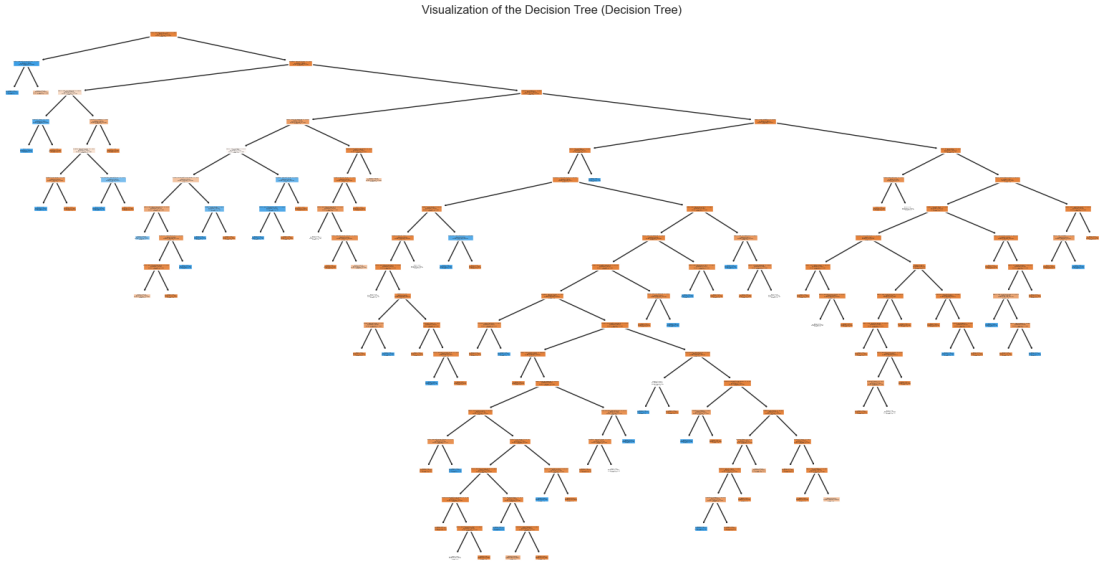
[46]:
```
# Define hyperparameters for each model
param_grid = {
    'Decision Tree': {
        'max_depth': [5, 10, 15, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },
    'Random Forest': {
        'n_estimators': [100, 150, 200],
        'max_depth': [10, 20, 30, 50],
```

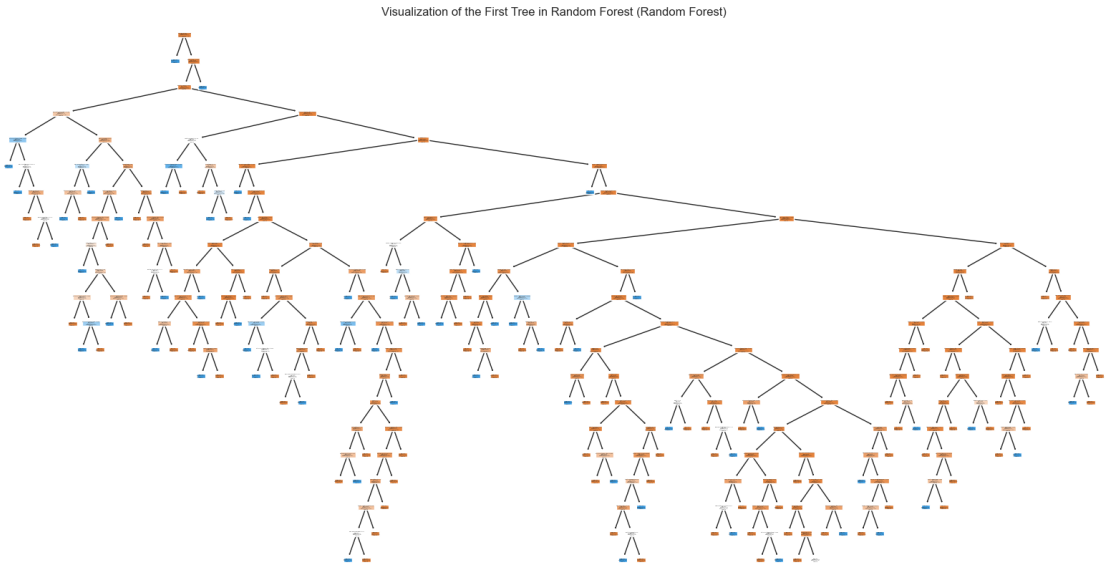
```
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'bootstrap': [True, False]
    },
    'XGBoost': {
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [3, 6, 9, 12],
        'learning_rate': [0.01, 0.1, 0.3],
        'subsample': [0.7, 1.0],
        'colsample_bytree': [0.7, 1.0],
    },
}

# Define the models
models = {
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'XGBoost': XGBClassifier(random_state=42),
}

# Initialize a dictionary to store the best model for each
best_models = {}

# Perform hyperparameter tuning using GridSearchCV for each model
for model_name, model in models.items():
    print(f"Performing GridSearchCV for {model_name}...")

    # Initialize GridSearchCV for each model
    grid_search = GridSearchCV(
    estimator=model,
    param_grid=param_grid[model_name],
    cv=5,
    scoring=make_scorer(recall_score, pos_label=1),  # Use the custom scorer
    verbose=1,
    n_jobs=-1
)

    # Fit GridSearchCV
    grid_search.fit(X_train, y_train)

    # Store the best model for each model name
    best_models[model_name] = grid_search.best_estimator_

    print(f"Best parameters for {model_name}: {grid_search.best_params_}")
    print(f"Best score for {model_name}: {grid_search.best_score_}\n")

# You can now use the best models for evaluation or predictions
```

```
Performing GridSearchCV for Decision Tree…
Fitting 5 folds for each of 45 candidates, totalling 225 fits
Best parameters for Decision Tree: {'max_depth': 20, 'min_samples_leaf': 1,
'min_samples_split': 5}
Best score for Decision Tree: 0.768888888888889

Performing GridSearchCV for Random Forest…
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
Best parameters for Random Forest: {'bootstrap': False, 'max_depth': 20,
'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best score for Random Forest: 0.7555555555555556

Performing GridSearchCV for XGBoost…
Fitting 5 folds for each of 192 candidates, totalling 960 fits
Best parameters for XGBoost: {'colsample_bytree': 0.7, 'learning_rate': 0.1,
'max_depth': 6, 'n_estimators': 200, 'subsample': 0.7}
Best score for XGBoost: 0.7911111111111111
```

[47]:
```python
# Function to plot the Decision Tree, Random Forest, and XGBoost trees
def plot_best_models(best_models, X_train, model_names):
    for model_name in model_names:
        model = best_models[model_name]

        # Decision Tree: Plot the tree structure
        if isinstance(model, DecisionTreeClassifier):
            plt.figure(figsize=(20, 10))
            plot_tree(model, filled=True, feature_names=X_train.columns.
 ↪tolist(), class_names=model.classes_.astype(str), proportion=True)
            plt.title(f'Visualization of the Decision Tree ({model_name})')
            plt.show()

        # Random Forest: Plot one of the trees from the forest
        elif isinstance(model, RandomForestClassifier):
            tree_to_plot = model.estimators_[0]  # Get the first tree in the␣
 ↪forest
            plt.figure(figsize=(20, 10))
            plot_tree(tree_to_plot, filled=True, feature_names=X_train.columns.
 ↪tolist(), class_names=model.classes_.astype(str), proportion=True)
            plt.title(f'Visualization of the First Tree in Random Forest␣
 ↪({model_name})')
            plt.show()

        # XGBoost: Plot one of the trees from XGBoost
        elif isinstance(model, XGBClassifier):
            plt.figure(figsize=(20, 10))
            xgb.plot_tree(model, num_trees=0)  # Plot the first tree
```

```
            plt.title(f'Visualization of the First Tree in XGBoost␣
 ↪({model_name})')
            plt.show()

# Assuming best_models contains the best fitted models after hyperparameter␣
 ↪tuning
# and X_train is the training dataset
# List of model names used
model_names = ['Decision Tree', 'Random Forest', 'XGBoost']

# Call the function to plot the models
plot_best_models(best_models, X_train, model_names)
```





<Figure size 2000x1000 with 0 Axes>

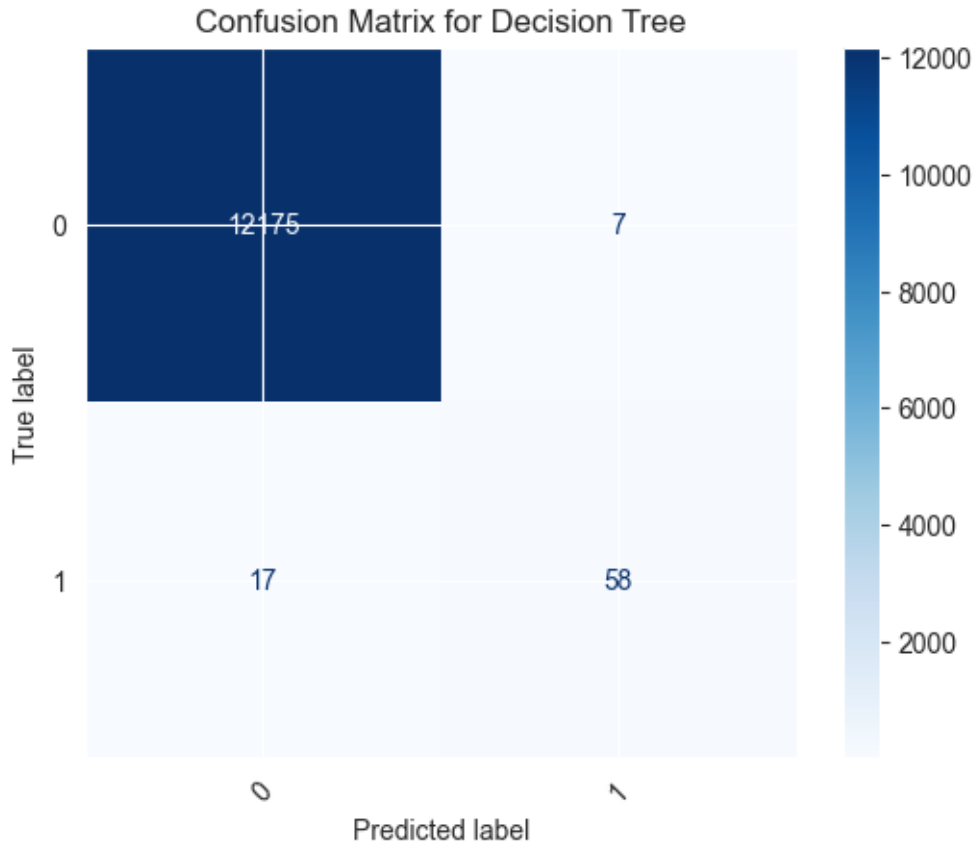
2.7 Evaluating and Validating after Tuning

[48]:
```
# Function to evaluate each model and display classification reports, confusion␣
↪matrices, and a summary
def evaluate_models(best_models, X_test, y_test):
    # Initialize a dictionary to store evaluation results
    evaluation_results = {}

    for model_name, model in best_models.items():
        print(f"\nEvaluating {model_name}...")
```

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

        # Compute evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
        recall = recall_score(y_test, y_pred, average='weighted')
        f1 = f1_score(y_test, y_pred, average='weighted')

        # Store metrics in results dictionary
        evaluation_results[model_name] = {
            'Accuracy': accuracy,
            'Precision': precision,
            'Recall': recall,
            'F1-Score': f1
        }

        # Print Accuracy
        print(f"Accuracy for {model_name}: {accuracy:.4f}")

        # Print classification report
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred))

        # Compute and display confusion matrix
        cm = confusion_matrix(y_test, y_pred)
        print("\nConfusion Matrix:")
        print(cm)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
 ↪classes_)
        disp.plot(cmap=plt.cm.Blues, xticks_rotation=45)
        plt.title(f"Confusion Matrix for {model_name}")
        plt.show()

# Assuming best_models contains the best-fitted models after hyperparameter␣
 ↪tuning
# and X_test, y_test are your test dataset
evaluate_models(best_models, X_test, y_test)
```

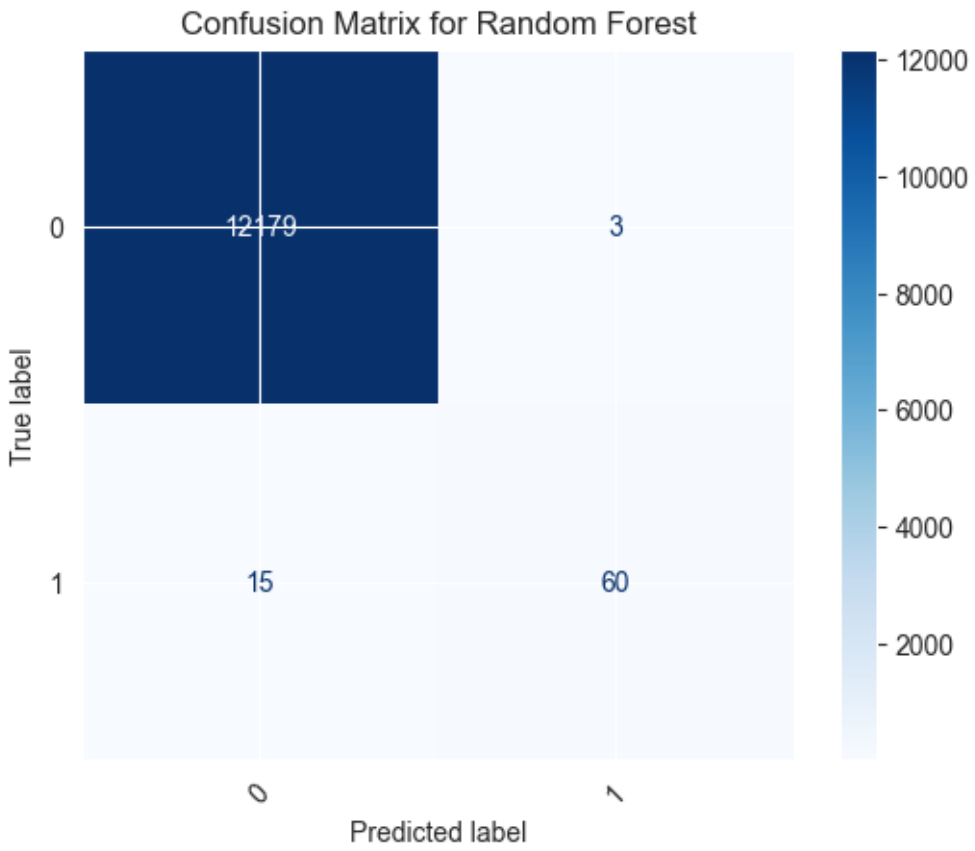
```
Evaluating Decision Tree…
Accuracy for Decision Tree: 0.9980

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
```

```
           1       0.89      0.77      0.83        75

    accuracy                           1.00     12257
   macro avg       0.95      0.89      0.91     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12175     7]
 [   17    58]]
```



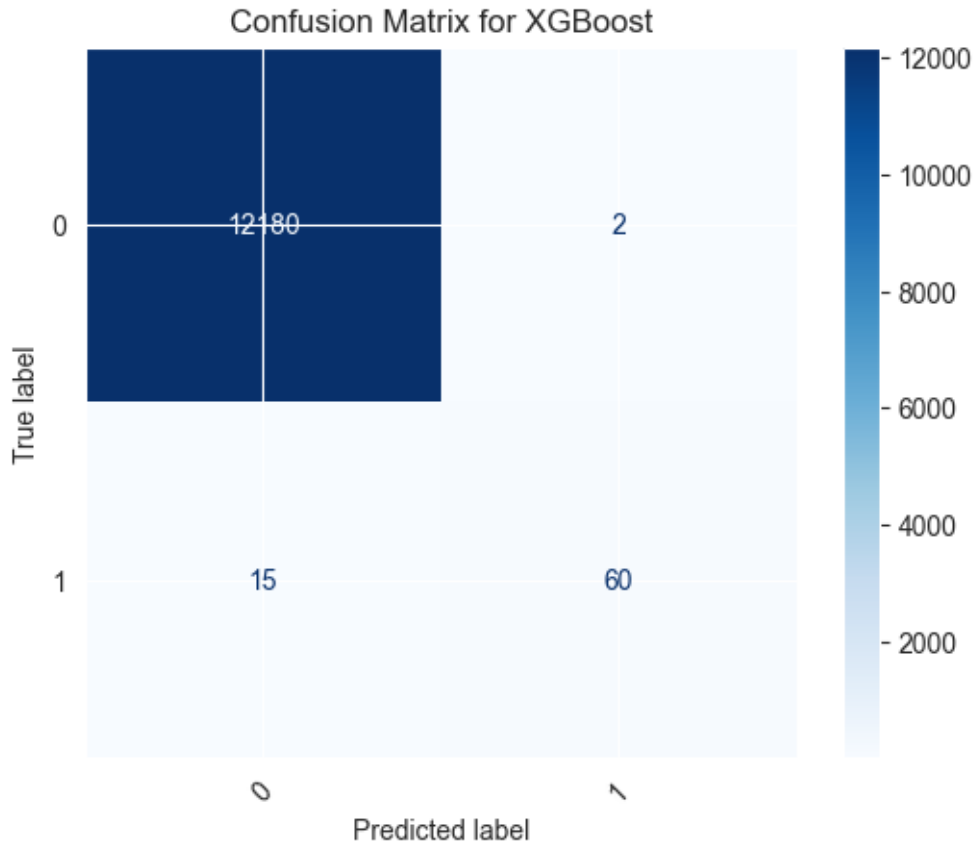

```
Evaluating Random Forest…
Accuracy for Random Forest: 0.9985

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.95      0.80      0.87        75
```

```
    accuracy                           1.00     12257
   macro avg       0.98      0.90      0.93     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12179     3]
 [   15    60]]
```



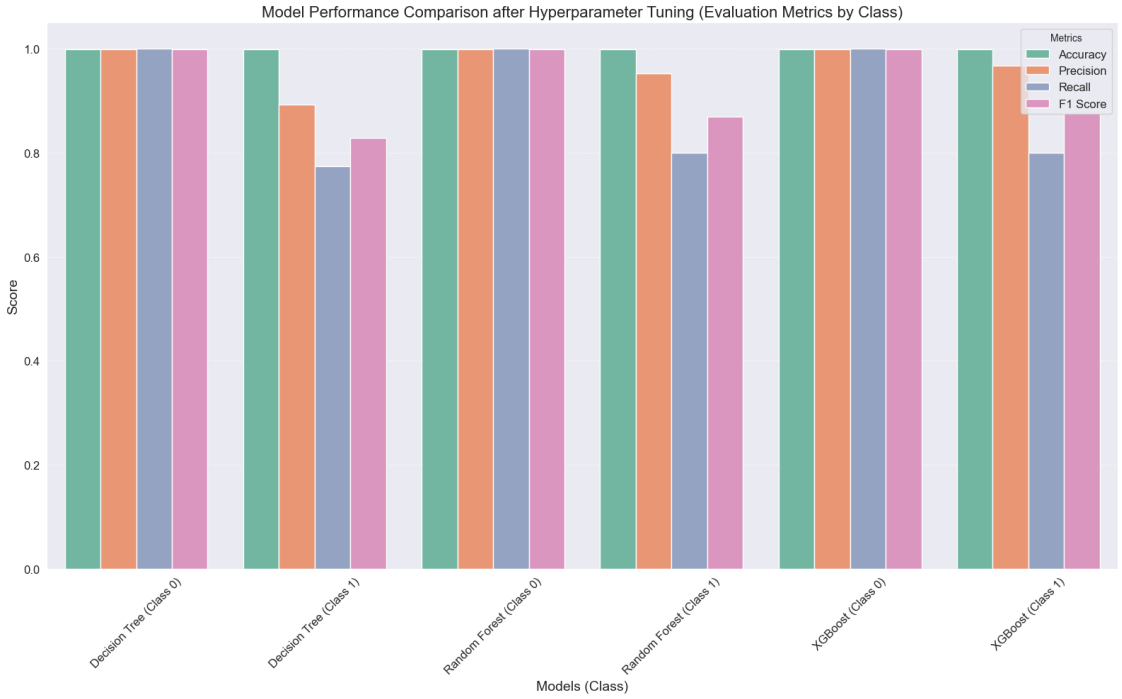

```
Evaluating XGBoost…
Accuracy for XGBoost: 0.9986

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.97      0.80      0.88        75

    accuracy                           1.00     12257
   macro avg       0.98      0.90      0.94     12257
```

```
weighted avg       1.00      1.00      1.00     12257

Confusion Matrix:
[[12180     2]
 [   15    60]]
```



[49]:
```python
# Initialize results list to store evaluation metrics for each model
results_2 = []

# Evaluate each model and calculate metrics for both classes
for model_name, model in best_models.items():
    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics for Class 0
    accuracy = accuracy_score(y_test, y_pred)
    precision_0 = precision_score(y_test, y_pred, pos_label=0)
    recall_0 = recall_score(y_test, y_pred, pos_label=0)
    f1_0 = f1_score(y_test, y_pred, pos_label=0)
```

```
    results_2.append({'Model': f"{model_name} (Class 0)", 'Metric': 'Accuracy',␣
 ↪'Value': accuracy})
    results_2.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
 ↪'Precision', 'Value': precision_0})
    results_2.append({'Model': f"{model_name} (Class 0)", 'Metric': 'Recall',␣
 ↪'Value': recall_0})
    results_2.append({'Model': f"{model_name} (Class 0)", 'Metric': 'F1 Score',␣
 ↪'Value': f1_0})

    # Calculate metrics for Class 1
    precision_1 = precision_score(y_test, y_pred, pos_label=1)
    recall_1 = recall_score(y_test, y_pred, pos_label=1)
    f1_1 = f1_score(y_test, y_pred, pos_label=1)

    results_2.append({'Model': f"{model_name} (Class 1)", 'Metric': 'Accuracy',␣
 ↪'Value': accuracy})  # Accuracy is the same
    results_2.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Precision', 'Value': precision_1})
    results_2.append({'Model': f"{model_name} (Class 1)", 'Metric': 'Recall',␣
 ↪'Value': recall_1})
    results_2.append({'Model': f"{model_name} (Class 1)", 'Metric': 'F1 Score',␣
 ↪'Value': f1_1})

# Convert results to DataFrame
comparison_df = pd.DataFrame(results_2)

# Plot model performance comparison (all evaluation metrics for both classes)
plt.figure(figsize=(16, 10))
sns.barplot(data=comparison_df, x="Model", y="Value", hue="Metric",␣
 ↪palette="Set2", dodge=True)
plt.title("Model Performance Comparison after Hyperparameter Tuning (Evaluation␣
 ↪Metrics by Class)", fontsize=16)
plt.ylabel("Score", fontsize=14)
plt.xlabel("Models (Class)", fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.legend(title="Metrics", loc="upper right", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Print the metrics table for both classes
print("\nDetailed Metrics for Each Model and Class after Hyperparameter Tuning:
 ↪")
print(comparison_df)
```



Detailed Metrics for Each Model and Class after Hyperparameter Tuning:

	Model	Metric	Value
0	Decision Tree (Class 0)	Accuracy	0.998042
1	Decision Tree (Class 0)	Precision	0.998606
2	Decision Tree (Class 0)	Recall	0.999425
3	Decision Tree (Class 0)	F1 Score	0.999015
4	Decision Tree (Class 1)	Accuracy	0.998042
5	Decision Tree (Class 1)	Precision	0.892308
6	Decision Tree (Class 1)	Recall	0.773333
7	Decision Tree (Class 1)	F1 Score	0.828571
8	Random Forest (Class 0)	Accuracy	0.998531
9	Random Forest (Class 0)	Precision	0.998770
10	Random Forest (Class 0)	Recall	0.999754
11	Random Forest (Class 0)	F1 Score	0.999262
12	Random Forest (Class 1)	Accuracy	0.998531
13	Random Forest (Class 1)	Precision	0.952381
14	Random Forest (Class 1)	Recall	0.800000
15	Random Forest (Class 1)	F1 Score	0.869565
16	XGBoost (Class 0)	Accuracy	0.998613
17	XGBoost (Class 0)	Precision	0.998770
18	XGBoost (Class 0)	Recall	0.999836
19	XGBoost (Class 0)	F1 Score	0.999303
20	XGBoost (Class 1)	Accuracy	0.998613
21	XGBoost (Class 1)	Precision	0.967742

```
22        XGBoost (Class 1)     Recall  0.800000
23        XGBoost (Class 1)   F1 Score  0.875912
```

[50]:
```python
# Iterate through unique best models for evaluation
for model_name, best_model in best_models.items():
    print(f"\nEvaluating {model_name}...")

    # Training performance for Recall (Class 1)
    y_train_pred = best_model.predict(X_train)
    train_recall = recall_score(y_train, y_train_pred, pos_label=1)

    # Test performance for Recall (Class 1)
    y_test_pred = best_model.predict(X_test)
    test_recall = recall_score(y_test, y_test_pred, pos_label=1)

    print(f"Training Recall for {model_name} after Hyperparameter Tuning:␣
 ↪{train_recall:.4f}")
    print(f"Test Recall for {model_name} after Hyperparameter Tuning:␣
 ↪{test_recall:.4f}")
```

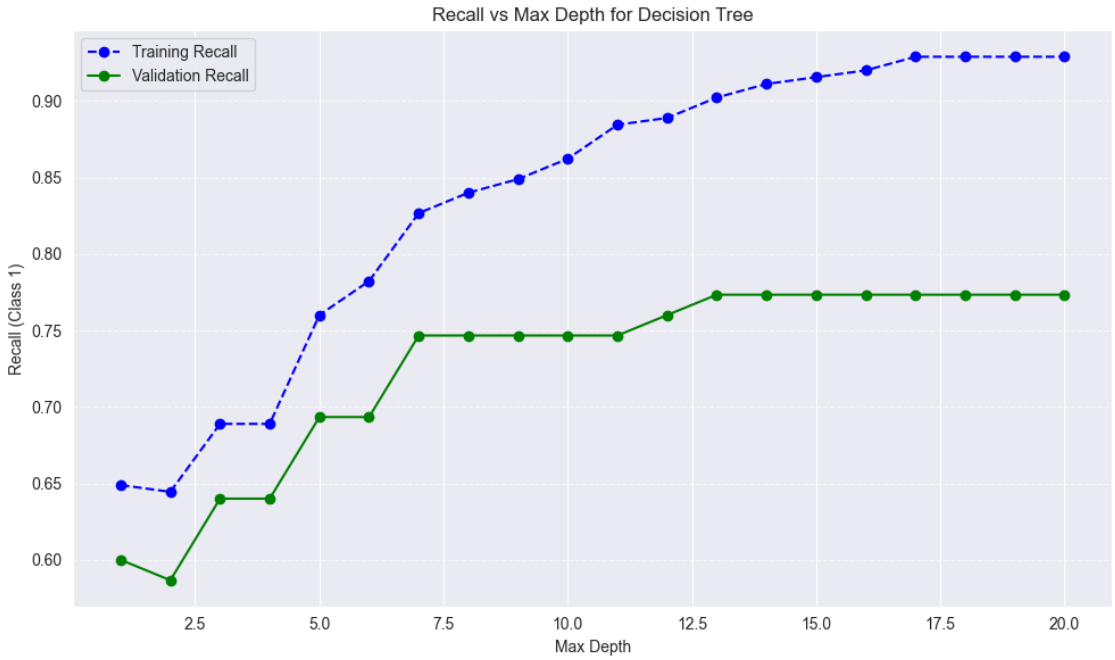
```
Evaluating Decision Tree…
Training Recall for Decision Tree after Hyperparameter Tuning: 0.9289
Test Recall for Decision Tree after Hyperparameter Tuning: 0.7733

Evaluating Random Forest…
Training Recall for Random Forest after Hyperparameter Tuning: 1.0000
Test Recall for Random Forest after Hyperparameter Tuning: 0.8000

Evaluating XGBoost…
Training Recall for XGBoost after Hyperparameter Tuning: 0.9689
Test Recall for XGBoost after Hyperparameter Tuning: 0.8000
```

[51]:
```python
# Function to plot Recall vs max_depth for Decision Tree and Random Forest
def plot_max_depth_vs_recall(models, X_train, y_train, X_test, y_test):
    for model_name, best_model in models.items():
        if isinstance(best_model, (DecisionTreeClassifier,␣
 ↪RandomForestClassifier)):
            print(f"\nGenerating Recall Curve for {model_name}...")

            # Define max_depth range
            max_depth_range = range(1, 21)
            training_recalls = []
            validation_recalls = []

            for max_depth in max_depth_range:
                # Update model with current max_depth and fit
```

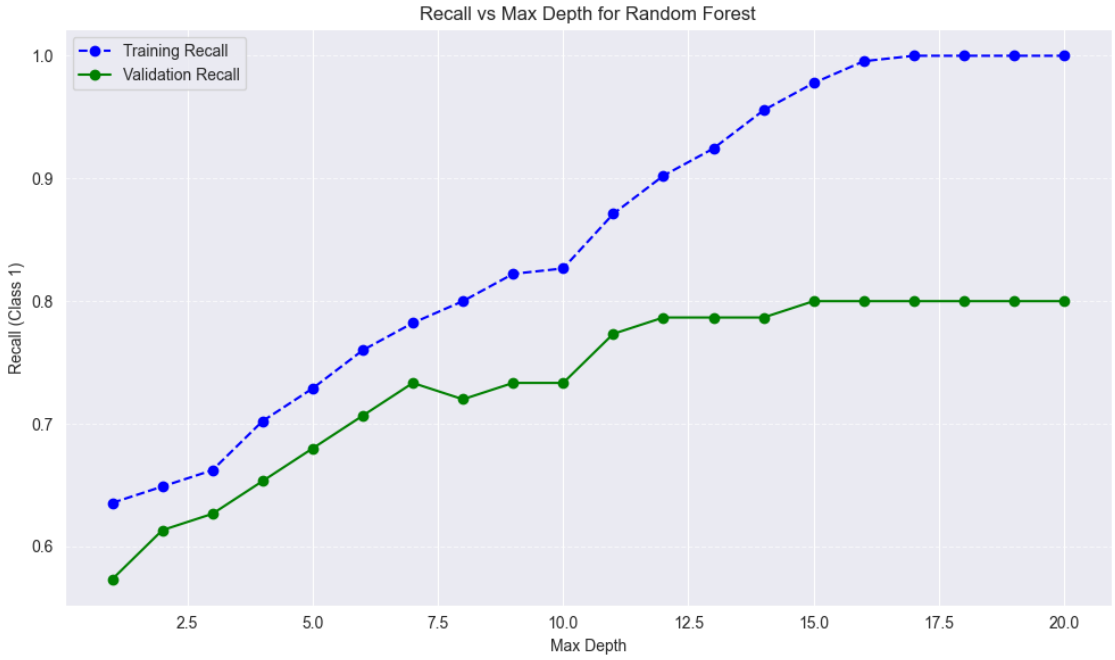
```
                model = best_model.set_params(max_depth=max_depth)
                model.fit(X_train, y_train)

                # Calculate Recall for Class 1
                train_recall = recall_score(y_train, model.predict(X_train),␣
 ↪pos_label=1)
                val_recall = recall_score(y_test, model.predict(X_test),␣
 ↪pos_label=1)

                training_recalls.append(train_recall)
                validation_recalls.append(val_recall)

            # Plot Recall vs max_depth
            plt.figure(figsize=(10, 6))
            plt.plot(max_depth_range, training_recalls, label='Training␣
 ↪Recall', linestyle='--', marker='o', color='blue')
            plt.plot(max_depth_range, validation_recalls, label='Validation␣
 ↪Recall', marker='o', color='green')
            plt.xlabel('Max Depth')
            plt.ylabel('Recall (Class 1)')
            plt.title(f'Recall vs Max Depth for {model_name}')
            plt.legend(loc='best')
            plt.grid(axis='y', linestyle='--', alpha=0.7)
            plt.tight_layout()
            plt.show()

# Function to plot Recall vs n_estimators for XGBoost
def plot_n_estimators_vs_recall(models, X_train, y_train, X_test, y_test):
    for model_name, best_model in models.items():
        if isinstance(best_model, XGBClassifier):
            print(f"\nGenerating Recall Curve for {model_name}...")

            # Define n_estimators range
            n_estimators_range = [50, 100, 150, 200, 250]
            training_recalls = []
            validation_recalls = []

            for n_estimators in n_estimators_range:
                # Update model with current n_estimators and fit
                model = best_model.set_params(n_estimators=n_estimators)
                model.fit(X_train, y_train)

                # Calculate Recall for Class 1
                train_recall = recall_score(y_train, model.predict(X_train),␣
 ↪pos_label=1)
                val_recall = recall_score(y_test, model.predict(X_test),␣
 ↪pos_label=1)
```

```
            training_recalls.append(train_recall)
            validation_recalls.append(val_recall)

        # Plot Recall vs n_estimators
        plt.figure(figsize=(10, 6))
        plt.plot(n_estimators_range, training_recalls, label='Training␣
↪Recall', linestyle='--', marker='o', color='blue')
        plt.plot(n_estimators_range, validation_recalls, label='Validation␣
↪Recall', marker='o', color='green')
        plt.xlabel('Number of Estimators')
        plt.ylabel('Recall (Class 1)')
        plt.title(f'Recall vs Number of Estimators for {model_name}')
        plt.legend(loc='best')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.show()

# Call the functions
plot_max_depth_vs_recall(best_models, X_train, y_train, X_test, y_test)
plot_n_estimators_vs_recall(best_models, X_train, y_train, X_test, y_test)
```

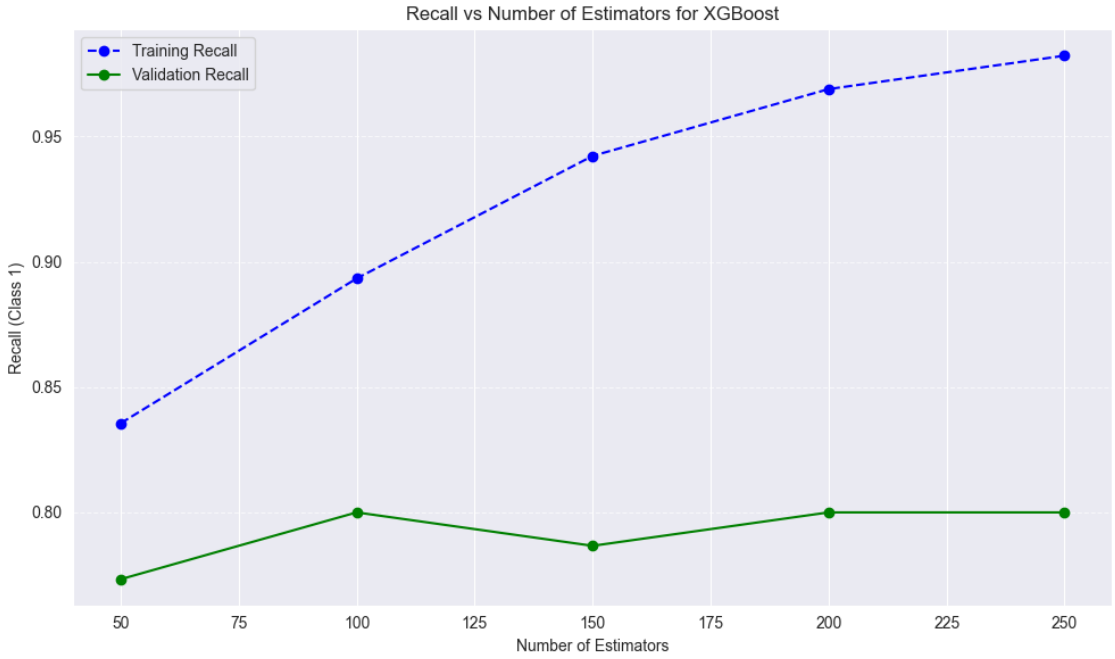
```
Generating Recall Curve for Decision Tree…
```



```
Generating Recall Curve for Random Forest…
```




```
Generating Recall Curve for XGBoost…
```



[52]:
```
# Set up Stratified K-Fold cross-validation
stratified_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Cross-Validation performance after hyperparameter tuning for each model
for model_name, best_model in best_models.items():
    print(f"\nPerforming Stratified Cross-Validation for {model_name}...")

    # Compute stratified cross-validation scores
    cross_val_scores = cross_val_score(best_model, X_train, y_train,␣
 ↪cv=stratified_kfold, scoring='recall')

    # Print the results
    print(f"Stratified Cross-Validation Scores for {model_name}:␣
 ↪{cross_val_scores}")
    print(f"Mean Stratified Cross-Validation Score for {model_name}:␣
 ↪{cross_val_scores.mean():.4f}")
```

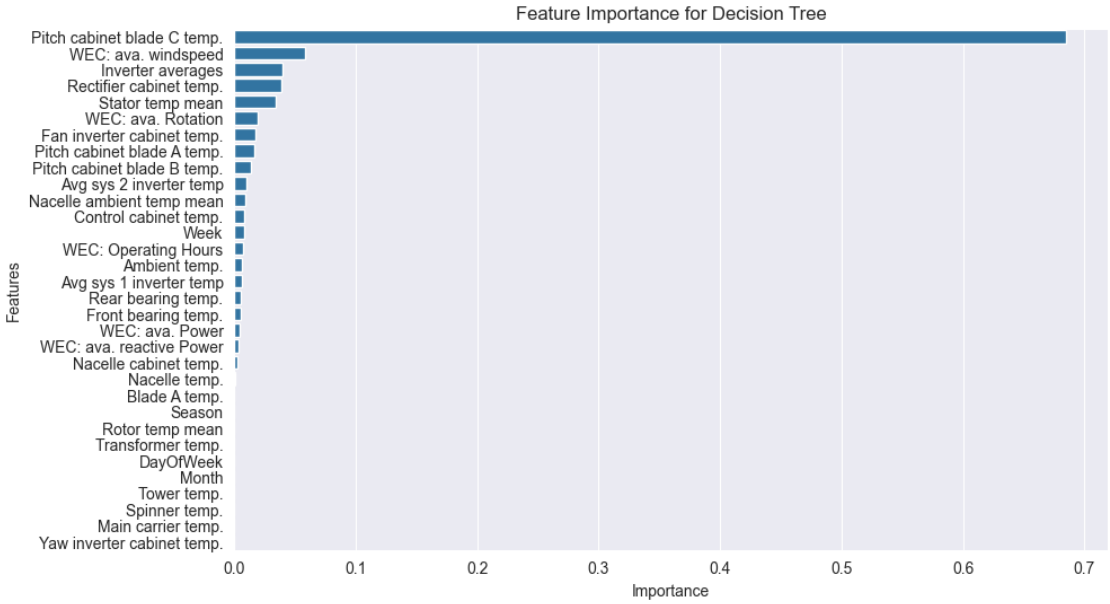
```
Performing Stratified Cross-Validation for Decision Tree…
Stratified Cross-Validation Scores for Decision Tree: [0.84444444 0.73333333
0.77777778 0.73333333 0.68888889]
Mean Stratified Cross-Validation Score for Decision Tree: 0.7556

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.8        0.68888889
0.77777778 0.62222222 0.73333333]
Mean Stratified Cross-Validation Score for Random Forest: 0.7244

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.86666667 0.71111111 0.8
0.68888889 0.77777778]
Mean Stratified Cross-Validation Score for XGBoost: 0.7689
```

2.8 Feature Importance

[53]:
```
# Function to plot and print feature importance for each model
def plot_and_print_feature_importance(models, X_train, model_names):
    for model_name in model_names:
        model = models[model_name]

        # Decision Tree and Random Forest
        if isinstance(model, (DecisionTreeClassifier, RandomForestClassifier)):
            feature_importance = model.feature_importances_

            # Create a DataFrame for feature importance
            importance_df = pd.DataFrame({
                'Feature': X_train.columns,
```

```
                'Importance': feature_importance
            }).sort_values(by='Importance', ascending=False)

            # Print feature importance values
            print(f"\nFeature Importance for {model_name}:")
            print(importance_df)

            # Plotting
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df)
            plt.title(f'Feature Importance for {model_name}')
            plt.xlabel('Importance')
            plt.ylabel('Features')
            plt.show()

        # XGBoost Model
        elif isinstance(model, XGBClassifier):
            importance = model.feature_importances_
            feature_names = X_train.columns

            # Create a DataFrame for feature importance
            importance_df = pd.DataFrame({
                'Feature': feature_names,
                'Importance': importance
            }).sort_values(by='Importance', ascending=False)

            # Print feature importance values
            print(f"\nFeature Importance for {model_name}:")
            print(importance_df)

            # Plotting
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df)
            plt.title(f'Feature Importance for {model_name}')
            plt.xlabel('Importance')
            plt.ylabel('Features')
            plt.show()

# Assuming best_models contains the best fitted models after hyperparameter␣
 ↪tuning
# and X_train is your training dataset
model_names = ['Decision Tree', 'Random Forest', 'XGBoost']

# Call the function to plot and print feature importance for each model
plot_and_print_feature_importance(best_models, X_train, model_names)
```

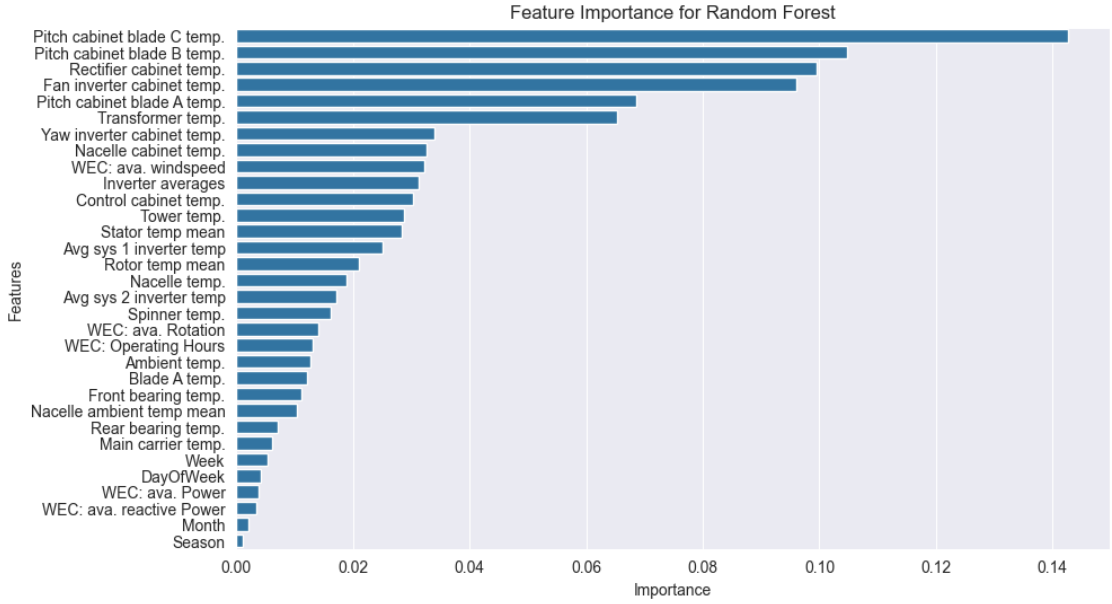
```
Feature Importance for Decision Tree:
                        Feature  Importance
10  Pitch cabinet blade C temp.    0.685273
0           WEC: ava. windspeed    0.058212
22            Inverter averages    0.039906
15      Rectifier cabinet temp.    0.038510
30              Stator temp mean    0.034549
1            WEC: ava. Rotation    0.019413
17   Fan inverter cabinet temp.    0.017641
8   Pitch cabinet blade A temp.    0.016397
9   Pitch cabinet blade B temp.    0.013486
28      Avg sys 2 inverter temp    0.009638
31    Nacelle ambient temp mean    0.008794
20        Control cabinet temp.    0.008463
24                         Week    0.007730
3          WEC: Operating Hours    0.007244
18                Ambient temp.    0.006212
27      Avg sys 1 inverter temp    0.006126
7             Rear bearing temp.    0.005776
6            Front bearing temp.    0.004952
2                WEC: ava. Power    0.004098
4      WEC: ava. reactive Power    0.003867
13        Nacelle cabinet temp.    0.002833
12                Nacelle temp.    0.000881
11                Blade A temp.    0.000000
26                       Season    0.000000
29              Rotor temp mean    0.000000
21            Transformer temp.    0.000000
25                    DayOfWeek    0.000000
23                        Month    0.000000
19                  Tower temp.    0.000000
5                 Spinner temp.    0.000000
14           Main carrier temp.    0.000000
16   Yaw inverter cabinet temp.    0.000000
```



Feature Importance for Random Forest:

	Feature	Importance
10	Pitch cabinet blade C temp.	0.142712
9	Pitch cabinet blade B temp.	0.104770
15	Rectifier cabinet temp.	0.099558
17	Fan inverter cabinet temp.	0.096101
8	Pitch cabinet blade A temp.	0.068587
21	Transformer temp.	0.065269
16	Yaw inverter cabinet temp.	0.034055
13	Nacelle cabinet temp.	0.032551
0	WEC: ava. windspeed	0.032144
22	Inverter averages	0.031265
20	Control cabinet temp.	0.030390
19	Tower temp.	0.028838
30	Stator temp mean	0.028451
27	Avg sys 1 inverter temp	0.025118
29	Rotor temp mean	0.021002
12	Nacelle temp.	0.018946
28	Avg sys 2 inverter temp	0.017190
5	Spinner temp.	0.016165
1	WEC: ava. Rotation	0.013973
3	WEC: Operating Hours	0.013164
18	Ambient temp.	0.012720
11	Blade A temp.	0.012141
6	Front bearing temp.	0.011204
31	Nacelle ambient temp mean	0.010293

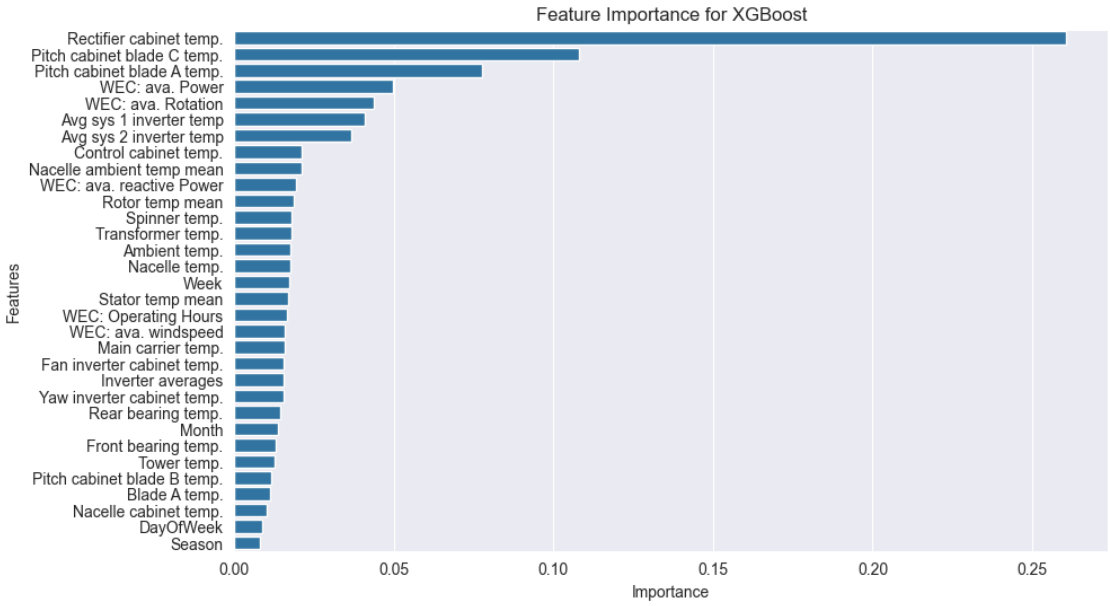
7	Rear bearing temp.	0.007110
14	Main carrier temp.	0.006041
24	Week	0.005395
25	DayOfWeek	0.004215
2	WEC: ava. Power	0.003888
4	WEC: ava. reactive Power	0.003417
23	Month	0.002147
26	Season	0.001182



Feature Importance for XGBoost:

	Feature	Importance
15	Rectifier cabinet temp.	0.260726
10	Pitch cabinet blade C temp.	0.107980
8	Pitch cabinet blade A temp.	0.077732
2	WEC: ava. Power	0.049793
1	WEC: ava. Rotation	0.043701
27	Avg sys 1 inverter temp	0.040911
28	Avg sys 2 inverter temp	0.036757
20	Control cabinet temp.	0.021118
31	Nacelle ambient temp mean	0.020922
4	WEC: ava. reactive Power	0.019499
29	Rotor temp mean	0.018655
5	Spinner temp.	0.017792
21	Transformer temp.	0.017780
18	Ambient temp.	0.017547
12	Nacelle temp.	0.017487
24	Week	0.017317

```
30            Stator temp mean    0.016717
3          WEC: Operating Hours    0.016676
0           WEC: ava. windspeed    0.015759
14           Main carrier temp.    0.015753
17   Fan inverter cabinet temp.    0.015559
22            Inverter averages    0.015430
16   Yaw inverter cabinet temp.    0.015302
7            Rear bearing temp.    0.014270
23                        Month    0.013549
6          Front bearing temp.    0.012970
19                  Tower temp.    0.012529
9   Pitch cabinet blade B temp.    0.011493
11                Blade A temp.    0.011225
13       Nacelle cabinet temp.    0.010070
25                    DayOfWeek    0.008843
26                       Season    0.008138
```



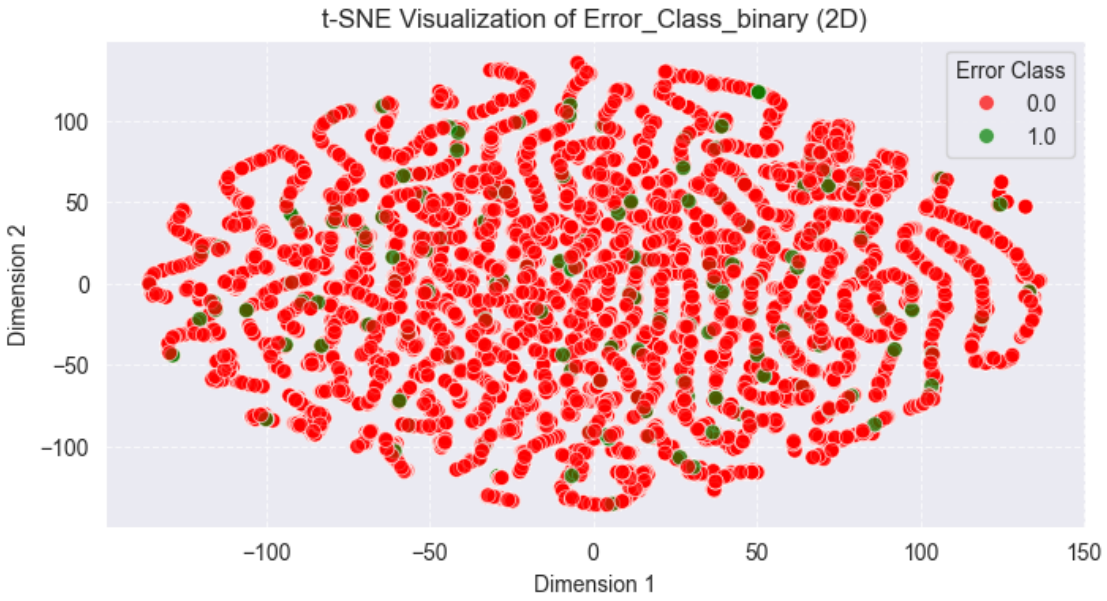
[54]:
```python
def get_important_features(best_models, X_train, threshold=0.01):
    # Dictionary to hold importance scores
    feature_importance_scores = pd.DataFrame(0, index=X_train.columns,␣
 ↪columns=['Decision Tree', 'Random Forest', 'XGBoost'])

    for model_name, model in best_models.items():
        # Decision Tree and Random Forest
        if isinstance(model, (DecisionTreeClassifier, RandomForestClassifier)):
            importance = model.feature_importances_
            feature_importance_scores[model_name] = importance
```

```
        # XGBoost
        elif isinstance(model, XGBClassifier):
            importance = model.feature_importances_
            feature_importance_scores[model_name] = importance

    # Average importance scores across models
    feature_importance_scores['Average'] = feature_importance_scores.
↪mean(axis=1)

    # Select features above the threshold
    selected_features =␣
↪feature_importance_scores[feature_importance_scores['Average'] > threshold].
↪index.tolist()

    print(f"Selected Features:\n{selected_features}")
    return selected_features

# Get important features
selected_features = get_important_features(best_models, X_train)
```

```
Selected Features:
['WEC: ava. windspeed', 'WEC: ava. Rotation', 'WEC: ava. Power', 'WEC: Operating
Hours', 'Spinner temp.', 'Pitch cabinet blade A temp.', 'Pitch cabinet blade B
temp.', 'Pitch cabinet blade C temp.', 'Nacelle temp.', 'Nacelle cabinet temp.',
'Rectifier cabinet temp.', 'Yaw inverter cabinet temp.', 'Fan inverter cabinet
temp.', 'Ambient temp.', 'Tower temp.', 'Control cabinet temp.', 'Transformer
temp.', 'Inverter averages', 'Week', 'Avg sys 1 inverter temp', 'Avg sys 2
inverter temp', 'Rotor temp mean', 'Stator temp mean', 'Nacelle ambient temp
mean']
```

[55]:
```
# Perform t-SNE transformation
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_train)

# Convert to DataFrame for easier plotting
tsne_df = pd.DataFrame(data=X_tsne, columns=['Dimension 1', 'Dimension 2'])
tsne_df['Error binary'] = y_train  # Add labels to the DataFrame

# Plot t-SNE scatter plot
plt.figure(figsize=(8, 4))
sns.scatterplot(
    data=tsne_df,
    x='Dimension 1',
    y='Dimension 2',
    hue='Error binary',
    palette=['red', 'green'],  # Customize colors for binary classes
```

```
    alpha=0.7,
    s=50
)
plt.title("t-SNE Visualization of Error_Class_binary (2D)")
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.legend(title='Error Class', loc='best')
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



3 Resampling Data with SMOTE-ENN

[56]:
```
# Filter training data for selected features
X_train = X_train[selected_features]

# Filter test data for the same selected features
X_test = X_test[selected_features]

print(X_train.shape)
print(X_test.shape)
```

```
(36770, 24)
(12257, 24)
```

[57]:
```
# Show class distribution before applying resampling
print("Class distribution before resampling:")
```

```
print(Counter(y_train))
```

```
Class distribution before resampling:
Counter({0: 36545, 1: 225})
```

[58]:
```
X_train = X_train.astype(np.float32)  # or another consistent type
y_train = y_train.astype(np.int32)  # ensure binary/int labels are int type
```

[59]:
```
# ---  SMOTE + ENN (SMOTEENN) ---
smote_enn = SMOTEENN(random_state=42)
X_resampled_smote_enn, y_resampled_smote_enn = smote_enn.fit_resample(X_train,␣
 ↪y_train)
print("\nClass distribution after SMOTE + ENN resampling:")
print(Counter(y_resampled_smote_enn))
```

```
Class distribution after SMOTE + ENN resampling:
Counter({0: 36412, 1: 36344})
```

[60]:
```
# Scatter plot with different colors for each Error_Class_Binary after␣
 ↪resampling
plt.figure(figsize=(8, 6))

# Ensure consistent column names and assign target column to the DataFrame
sns.scatterplot(
    data=pd.DataFrame(X_resampled_smote_enn, columns=X_train.columns).
 ↪assign(Error_Binary=y_resampled_smote_enn),
    x='WEC: ava. windspeed',  # Replace with an appropriate feature for the␣
 ↪x-axis
    y='WEC: ava. Power',      # Replace with an appropriate feature for the␣
 ↪y-axis
    hue='Error_Binary',        # Class-based coloring
    palette='tab10',          # Use a color palette with enough distinct colors
    style='Error_Binary',      # Different markers for each class
    s=50                      # Marker size
)

# Add labels and title
plt.title('Scatter Plot of Error Classes After Resampling', fontsize=16)
plt.xlabel('Average Windspeed', fontsize=14)
plt.ylabel('Average Power', fontsize=14)
plt.legend(title='Error Class', bbox_to_anchor=(1.05, 1), loc='upper left')  #␣
 ↪Legend outside the plot
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

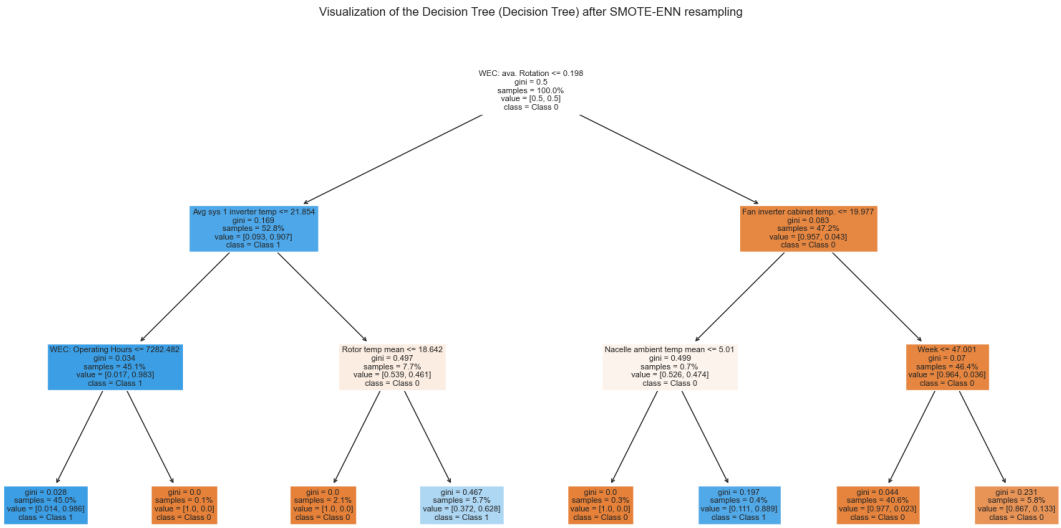
3.1 Model Development and Training with Resampled Data

[61]:
```
# Define the models for training
rsmpl_models = {
    'Decision Tree': DecisionTreeClassifier(max_depth=3, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=50, random_state=42),
    'XGBoost': XGBClassifier(random_state=42, eval_metric='mlogloss')  #␣
 ↪Configure XGBoost
}
```

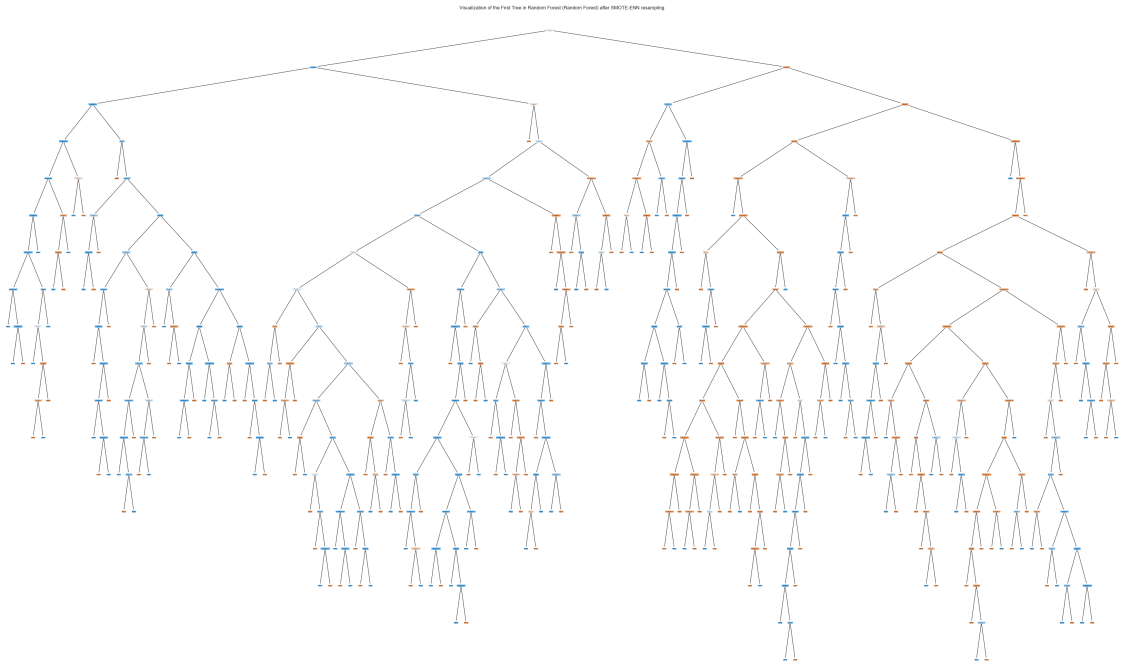
[62]:
```
# --- Training Section (Fit) with rsmpl_models ---
for model_name, model in rsmpl_models.items():
    print(f"\nTraining {model_name}...")

    # Train the model on the resampled data
    model.fit(X_resampled_smote_enn, y_resampled_smote_enn)

    # If the model is a Decision Tree, plot the tree structure
    if isinstance(model, DecisionTreeClassifier):
        plt.figure(figsize=(20, 10))
```

```
        plot_tree(model, filled=True, feature_names=X_resampled_smote_enn.
↪columns.tolist(),
                  class_names=['Class 0', 'Class 1'], proportion=True)
        plt.title(f'Visualization of the Decision Tree ({model_name}) after␣
↪SMOTE-ENN resampling')
        plt.show()

    # If the model is a Random Forest, visualize the first tree of the forest
    if isinstance(model, RandomForestClassifier):
        # Visualize the first tree of the Random Forest
        plt.figure(figsize=(50, 30))
        plot_tree(model.estimators_[0], filled=True,␣
↪feature_names=X_resampled_smote_enn.columns.tolist(),
                  class_names=['Class 0', 'Class 1'], proportion=True)
        plt.title(f'Visualization of the First Tree in Random Forest␣
↪({model_name}) after SMOTE-ENN resampling')
        plt.show()

    # If the model is XGBoost, visualize the first tree
    if isinstance(model, XGBClassifier):
        # Optionally, plot the first tree of the XGBoost model
        plt.figure(figsize=(20, 20))
        xgb.plot_tree(model, num_trees=0)  # Plot the first tree
        plt.title(f'Visualization of the First Tree in XGBoost ({model_name})␣
↪after SMOTE-ENN resampling')
        plt.show()
```

```
Training Decision Tree…
```



```
Training Random Forest…
```



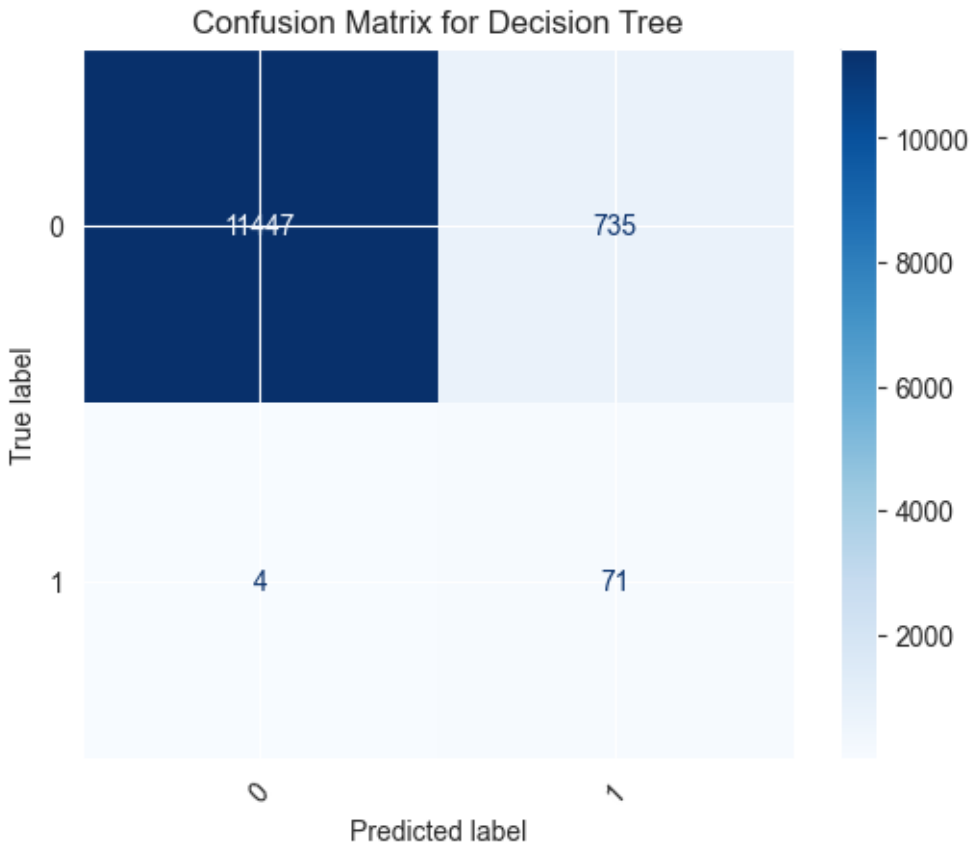
```
Training XGBoost…
<Figure size 2000x2000 with 0 Axes>
```

3.2 Evaluation and Validation after resampling

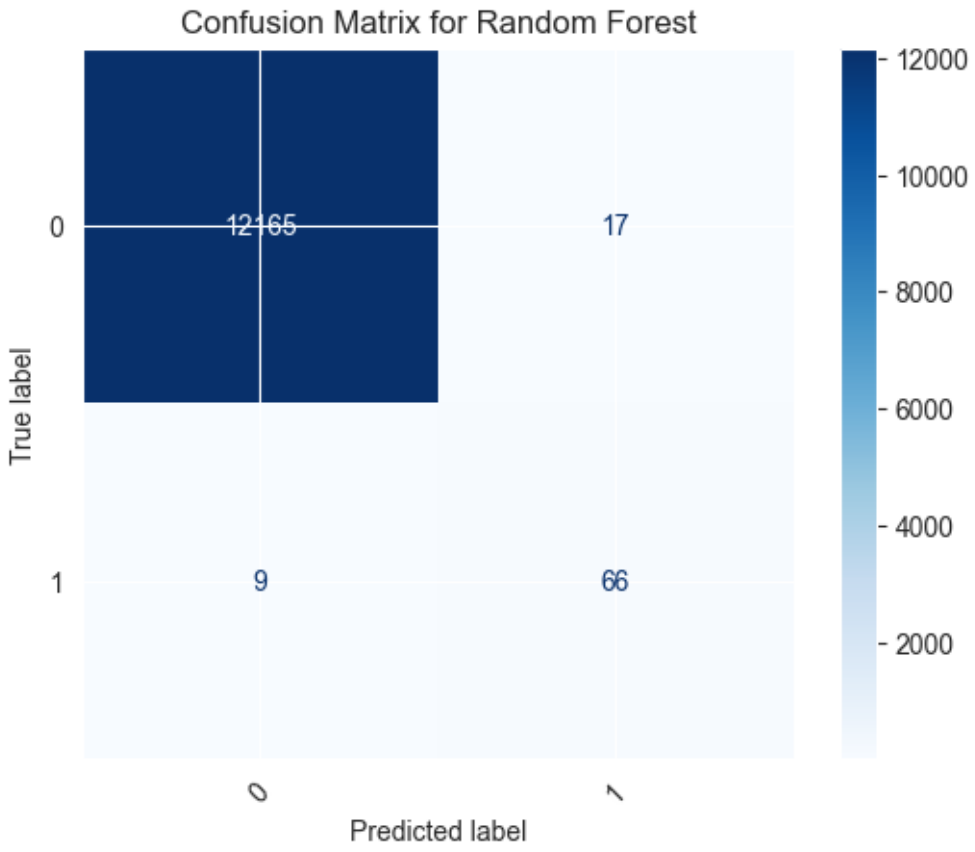
[63]:
```python
# Evaluate each trained model (after resampling)
for model_name, model in rsmpl_models.items():
    print(f"\nEvaluating {model_name}...")

    # Make predictions on the test set
    y_pred = model.predict(X_test)
```

```
    # Compute recall for Class 1
    recall = recall_score(y_test, y_pred, pos_label=1)
    print(f"{model_name} - Test Recall (Class 1): {recall:.4f}")

    # Print other evaluation metrics
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    print("\nConfusion Matrix:")
    print(cm)

    # Plot confusion matrix
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
↪classes_)
    disp.plot(cmap=plt.cm.Blues, xticks_rotation=45)
    plt.title(f"Confusion Matrix for {model_name}")
    plt.show()
```

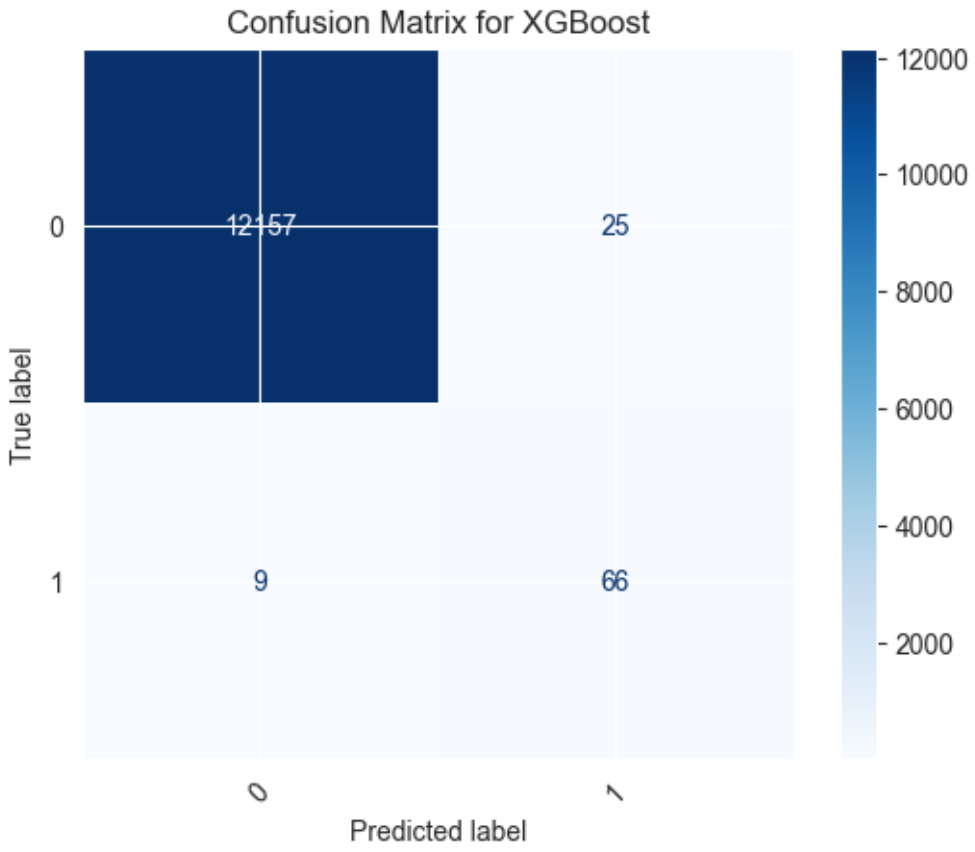
```
Evaluating Decision Tree…
Decision Tree - Test Recall (Class 1): 0.9467

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.94      0.97     12182
           1       0.09      0.95      0.16        75

    accuracy                           0.94     12257
   macro avg       0.54      0.94      0.56     12257
weighted avg       0.99      0.94      0.96     12257


Confusion Matrix:
[[11447   735]
 [    4    71]]
```




```
Evaluating Random Forest…
Random Forest - Test Recall (Class 1): 0.8800

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.80      0.88      0.84        75

    accuracy                           1.00     12257
   macro avg       0.90      0.94      0.92     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12165    17]
 [    9    66]]
```

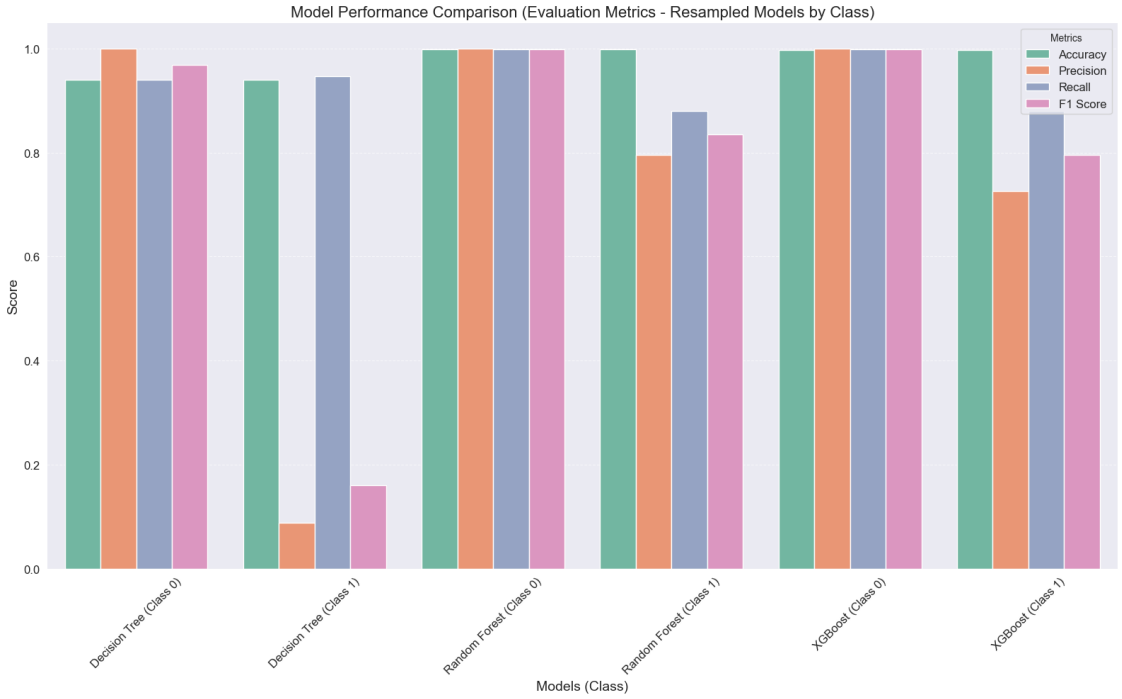



```
Evaluating XGBoost…
XGBoost - Test Recall (Class 1): 0.8800

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.73      0.88      0.80        75

    accuracy                           1.00     12257
   macro avg       0.86      0.94      0.90     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12157    25]
 [    9    66]]
```



[64]:
```
# Initialize results list to store evaluation metrics for each resampled model
results_rsmpl_1 = []

# Evaluate each resampled model and calculate metrics for both classes
for model_name, model in rsmpl_models.items():
    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics for Class 0
    accuracy = accuracy_score(y_test, y_pred)
    precision_0 = precision_score(y_test, y_pred, pos_label=0)
    recall_0 = recall_score(y_test, y_pred, pos_label=0)
    f1_0 = f1_score(y_test, y_pred, pos_label=0)

    results_rsmpl_1.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
↪'Accuracy', 'Value': accuracy})
    results_rsmpl_1.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
↪'Precision', 'Value': precision_0})
    results_rsmpl_1.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
↪'Recall', 'Value': recall_0})
```

```
    results_rsmpl_1.append({'Model': f"{model_name} (Class 0)", 'Metric': 'F1␣
 ↪Score', 'Value': f1_0})

    # Calculate metrics for Class 1
    precision_1 = precision_score(y_test, y_pred, pos_label=1)
    recall_1 = recall_score(y_test, y_pred, pos_label=1)
    f1_1 = f1_score(y_test, y_pred, pos_label=1)

    results_rsmpl_1.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Accuracy', 'Value': accuracy})  # Accuracy is the same
    results_rsmpl_1.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Precision', 'Value': precision_1})
    results_rsmpl_1.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Recall', 'Value': recall_1})
    results_rsmpl_1.append({'Model': f"{model_name} (Class 1)", 'Metric': 'F1␣
 ↪Score', 'Value': f1_1})

# Convert results to DataFrame
comparison_df_rsmpl = pd.DataFrame(results_rsmpl_1)

# Plot model performance comparison (all evaluation metrics for resampled␣
 ↪models by class)
plt.figure(figsize=(16, 10))
sns.barplot(data=comparison_df_rsmpl, x="Model", y="Value", hue="Metric",␣
 ↪palette="Set2", dodge=True)
plt.title("Model Performance Comparison (Evaluation Metrics - Resampled Models␣
 ↪by Class)", fontsize=16)
plt.ylabel("Score", fontsize=14)
plt.xlabel("Models (Class)", fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.legend(title="Metrics", loc="upper right", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Print the metrics table for both classes
print("\nDetailed Metrics for Each Resampled Model and Class:")
print(comparison_df_rsmpl)
```



Detailed Metrics for Each Resampled Model and Class:

	Model	Metric	Value
0	Decision Tree (Class 0)	Accuracy	0.939708
1	Decision Tree (Class 0)	Precision	0.999651
2	Decision Tree (Class 0)	Recall	0.939665
3	Decision Tree (Class 0)	F1 Score	0.968730
4	Decision Tree (Class 1)	Accuracy	0.939708
5	Decision Tree (Class 1)	Precision	0.088089
6	Decision Tree (Class 1)	Recall	0.946667
7	Decision Tree (Class 1)	F1 Score	0.161180
8	Random Forest (Class 0)	Accuracy	0.997879
9	Random Forest (Class 0)	Precision	0.999261
10	Random Forest (Class 0)	Recall	0.998604
11	Random Forest (Class 0)	F1 Score	0.998933
12	Random Forest (Class 1)	Accuracy	0.997879
13	Random Forest (Class 1)	Precision	0.795181
14	Random Forest (Class 1)	Recall	0.880000
15	Random Forest (Class 1)	F1 Score	0.835443
16	XGBoost (Class 0)	Accuracy	0.997226
17	XGBoost (Class 0)	Precision	0.999260
18	XGBoost (Class 0)	Recall	0.997948
19	XGBoost (Class 0)	F1 Score	0.998604
20	XGBoost (Class 1)	Accuracy	0.997226
21	XGBoost (Class 1)	Precision	0.725275

```
22        XGBoost (Class 1)     Recall  0.880000
23        XGBoost (Class 1)   F1 Score  0.795181
```

[65]:
```
# Ensure the rsmpl_models dictionary has only unique models
unique_rsmpl_models = {model_name: model for model_name, model in rsmpl_models.
 ↪items()}

# Iterate through unique resampled models for evaluation
for model_name, model in unique_rsmpl_models.items():
    print(f"\nEvaluating {model_name}...")

    # Training performance
    y_train_pred = model.predict(X_train)  # Predict using the model on␣
 ↪training data
    train_recall = recall_score(y_train, y_train_pred, pos_label=1)  # Compute␣
 ↪training recall for Class 1

    # Test performance
    y_test_pred = model.predict(X_test)  # Predict using the model on test data
    test_recall = recall_score(y_test, y_test_pred, pos_label=1)  # Compute␣
 ↪test recall for Class 1

    # Print training and test recall
    print(f"Training Recall for {model_name} (Class 1): {train_recall:.4f}")
    print(f"Test Recall for {model_name} (Class 1): {test_recall:.4f}")
```

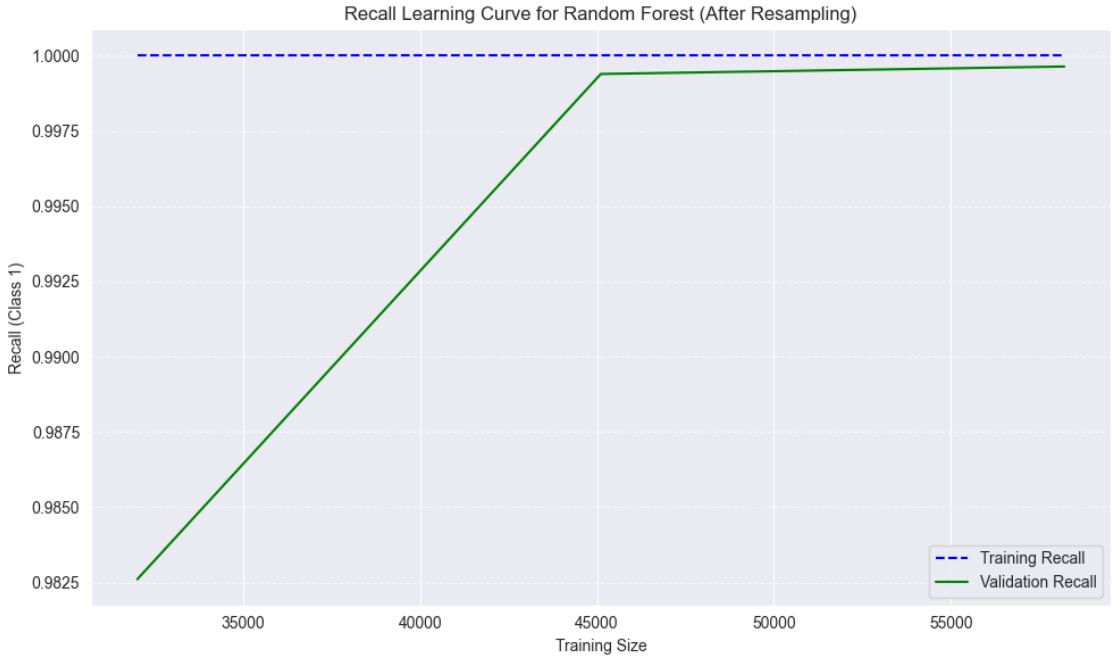
```
Evaluating Decision Tree…
Training Recall for Decision Tree (Class 1): 0.9644
Test Recall for Decision Tree (Class 1): 0.9467

Evaluating Random Forest…
Training Recall for Random Forest (Class 1): 0.9778
Test Recall for Random Forest (Class 1): 0.8800

Evaluating XGBoost…
Training Recall for XGBoost (Class 1): 0.9822
Test Recall for XGBoost (Class 1): 0.8800
```

[66]:
```
# Iterate through resampled models to plot learning curves for Recall
for model_name, model in rsmpl_models.items():
    print(f"\nPlotting Learning Curve for {model_name}...")

    # Generate learning curve for Recall (Class 1)
    train_sizes_recall, train_scores_recall, validation_scores_recall =␣
 ↪learning_curve(
```

```
        model, X_resampled_smote_enn, y_resampled_smote_enn, cv=5,␣
↪scoring=make_scorer(recall_score, pos_label=1), n_jobs=-1
    )

    # Plot Recall learning curve
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes_recall, train_scores_recall.mean(axis=1),␣
↪label='Training Recall', linestyle='--', color='blue')
    plt.plot(train_sizes_recall, validation_scores_recall.mean(axis=1),␣
↪label='Validation Recall', color='green')
    plt.xlabel('Training Size')
    plt.ylabel('Recall (Class 1)')
    plt.title(f'Recall Learning Curve for {model_name} (After Resampling)')
    plt.legend(loc='best')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()
```

```
Plotting Learning Curve for Decision Tree…
```


```
Plotting Learning Curve for Random Forest…
```




```
Plotting Learning Curve for XGBoost…
```

[67]:
```
# Set up Stratified K-Fold cross-validation
stratified_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Iterate through each resampled model and compute Stratified K-Fold␣
 ↪cross-validation scores
for model_name, model in rsmpl_models.items():
    print(f"\nPerforming Stratified Cross-Validation for {model_name}...")

    # Perform cross-validation
    cross_val_scores = cross_val_score(model, X_train, y_train,␣
 ↪cv=stratified_kfold, scoring='recall')

    # Print the results for the current model
    print(f"Stratified Cross-Validation Scores for {model_name}:␣
 ↪{cross_val_scores}")
    print(f"Mean Stratified Cross-Validation Score for {model_name}:␣
 ↪{cross_val_scores.mean():.4f}")
```

```
Performing Stratified Cross-Validation for Decision Tree…
Stratified Cross-Validation Scores for Decision Tree: [0.73333333 0.68888889
0.75555556 0.55555556 0.71111111]
Mean Stratified Cross-Validation Score for Decision Tree: 0.6889

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.8        0.68888889
0.75555556 0.57777778 0.71111111]
Mean Stratified Cross-Validation Score for Random Forest: 0.7067

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.84444444 0.71111111 0.8
0.64444444 0.77777778]
Mean Stratified Cross-Validation Score for XGBoost: 0.7556
```

3.3 Hyperparameter Tuning after Resampling data

[68]:
```
# Define hyperparameters for each model
param_grid_rsmpl = {
    'Decision Tree': {
        'max_depth': [5, 10, 15, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },
    'Random Forest': {
        'n_estimators': [100, 150, 200],
        'max_depth': [10, 20, 30, 50],
        'min_samples_split': [2, 5, 10],
```

```
        'min_samples_leaf': [1, 2, 4],
        'bootstrap': [True, False]
    },
    'XGBoost': {
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [3, 6, 9, 12],
        'learning_rate': [0.01, 0.1, 0.3],
        'subsample': [0.7, 1.0],
        'colsample_bytree': [0.7, 1.0],
    },
}

# Define the models
models = {
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'XGBoost': XGBClassifier(random_state=42),
}

# Initialize a dictionary to store the best model for each
rsmpl_best_models = {}

# Perform hyperparameter tuning using GridSearchCV for each model
for model_name, model in models.items():
    print(f"Performing GridSearchCV for {model_name}...")

    # Initialize GridSearchCV for each model
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid_rsmpl[model_name],
        cv=5,
        scoring=make_scorer(f1_score, pos_label=1),  # F1 score specifically␣
 ↪for Class 1
        verbose=1,
        n_jobs=-1
    )

    # Fit GridSearchCV on the resampled data (use the previously resampled data)
    grid_search.fit(X_resampled_smote_enn, y_resampled_smote_enn)

    # Store the best model for each model name
    rsmpl_best_models[model_name] = grid_search.best_estimator_

    print(f"Best parameters for {model_name}: {grid_search.best_params_}")
    print(f"Best F1 score for {model_name}: {grid_search.best_score_}\n")
```

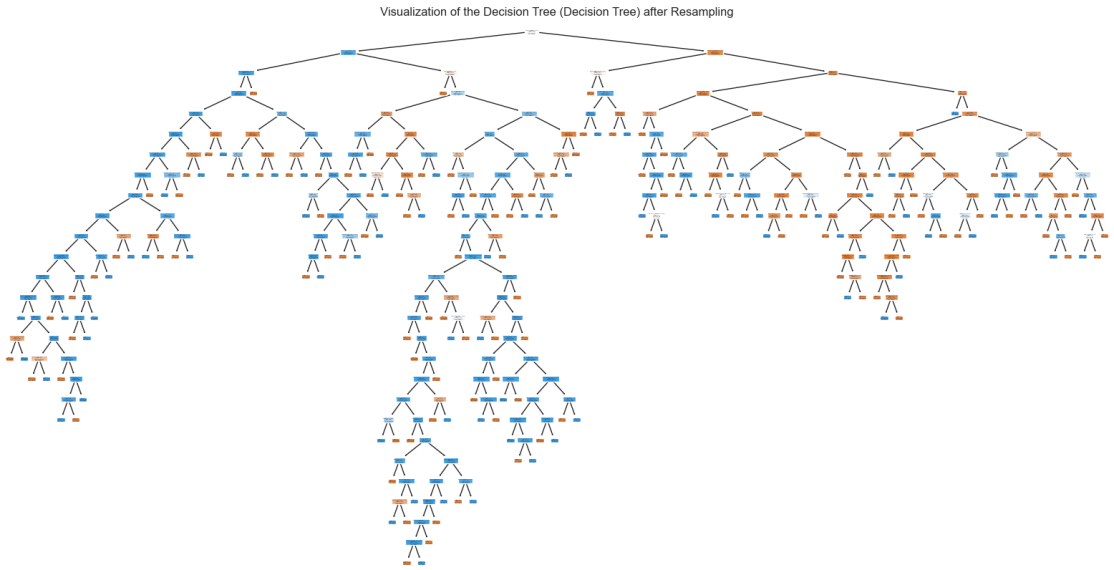
```
# The best models are now stored in the `rsmpl_best_models` dictionary for␣
 ↪further evaluation or predictions
```

```
Performing GridSearchCV for Decision Tree…
Fitting 5 folds for each of 45 candidates, totalling 225 fits
Best parameters for Decision Tree: {'max_depth': 30, 'min_samples_leaf': 1,
'min_samples_split': 2}
Best F1 score for Decision Tree: 0.9983908590934657

Performing GridSearchCV for Random Forest…
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
Best parameters for Random Forest: {'bootstrap': False, 'max_depth': 30,
'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 150}
Best F1 score for Random Forest: 0.9997660458627291

Performing GridSearchCV for XGBoost…
Fitting 5 folds for each of 192 candidates, totalling 960 fits
Best parameters for XGBoost: {'colsample_bytree': 1.0, 'learning_rate': 0.3,
'max_depth': 6, 'n_estimators': 200, 'subsample': 1.0}
Best F1 score for XGBoost: 0.9997386680722509
```

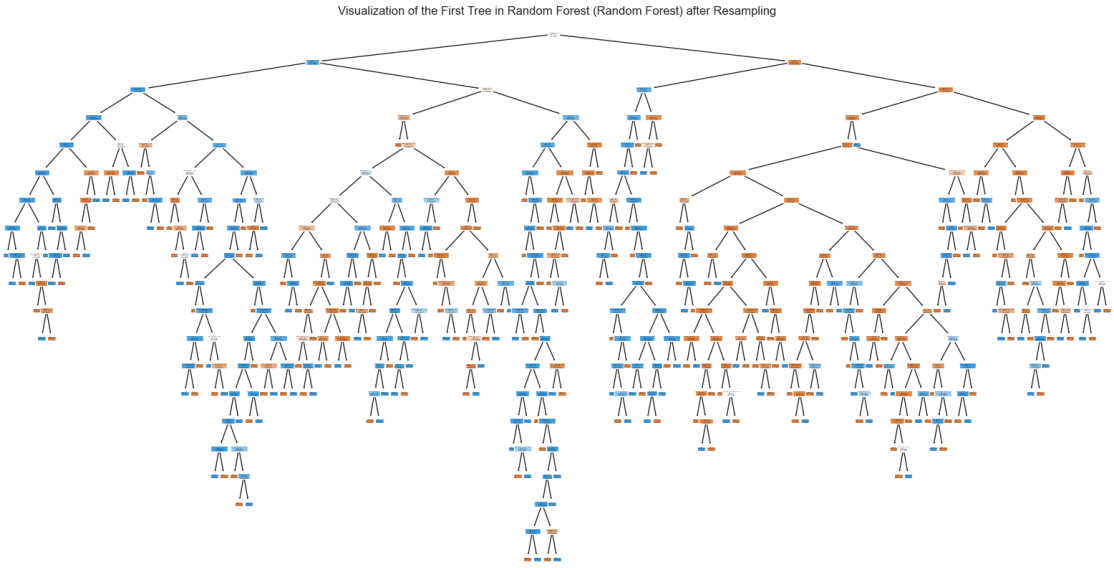
[69]:
```
# Function to plot the best fitted models (Decision Tree, Random Forest,␣
 ↪XGBoost) after hyperparameter tuning and resampling
def plot_best_models_after_resampling(rsmpl_best_models, feature_names,␣
 ↪class_names):
    for model_name, model in rsmpl_best_models.items():
        print(f"\nVisualizing {model_name}...")

        # Decision Tree: Plot the tree structure
        if isinstance(model, DecisionTreeClassifier):
            plt.figure(figsize=(20, 10))
            plot_tree(
                model,
                filled=True,
                feature_names=feature_names,
                class_names=class_names,
                proportion=True
            )
            plt.title(f'Visualization of the Decision Tree ({model_name}) after␣
 ↪Resampling')
            plt.show()

        # Random Forest: Plot one of the trees from the forest
        elif isinstance(model, RandomForestClassifier):
            tree_to_plot = model.estimators_[0]  # Get the first tree in the␣
 ↪forest
```

```
            plt.figure(figsize=(20, 10))
            plot_tree(
                tree_to_plot,
                filled=True,
                feature_names=feature_names,
                class_names=class_names,
                proportion=True
            )
            plt.title(f'Visualization of the First Tree in Random Forest␣
 ↪({model_name}) after Resampling')
            plt.show()

        # XGBoost: Plot one of the trees from XGBoost
        elif isinstance(model, XGBClassifier):
            plt.figure(figsize=(20, 10))
            xgb.plot_tree(model, num_trees=0, fmap='', ax=None)  # Plot the␣
 ↪first tree
            plt.title(f'Visualization of the First Tree in XGBoost␣
 ↪({model_name}) after Resampling')
            plt.show()

# Assuming rsmpl_best_models contains the best fitted models after␣
 ↪hyperparameter tuning and resampling
# and the feature and class names are provided
feature_names = X_resampled_smote_enn.columns.tolist()  # Ensure this matches␣
 ↪your dataset
class_names = ["Class 0", "Class 1"]  # Replace with actual class names if␣
 ↪available

# Call the function to plot the models
plot_best_models_after_resampling(rsmpl_best_models, feature_names, class_names)
```

```
Visualizing Decision Tree…
```




```
Visualizing Random Forest…
```




```
Visualizing XGBoost…
<Figure size 2000x1000 with 0 Axes>
```

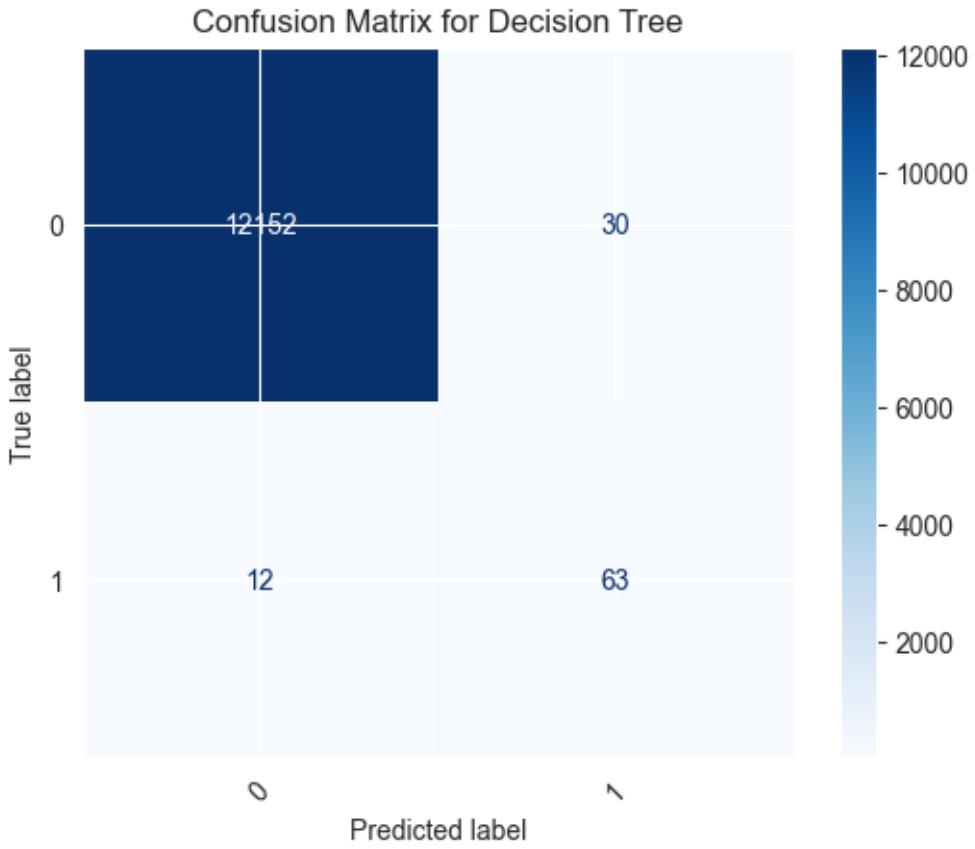
3.4 Evaluating and Validating model after Resampling & Hyperparameter Tuning

[70]:
```
# Initialize results list to store evaluation metrics for each resampled and␣
 ↪hyperparameter-tuned model
results_rsmpl_2 = []

# Evaluate each resampled model after hyperparameter tuning
for model_name, model in rsmpl_best_models.items():
    print(f"\nEvaluating {model_name} after Hyperparameter Tuning and␣
 ↪Resampling...")

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

    # Compute metrics for Class 0
    accuracy = accuracy_score(y_test, y_pred)
    precision_0 = precision_score(y_test, y_pred, pos_label=0)
    recall_0 = recall_score(y_test, y_pred, pos_label=0)
    f1_0 = f1_score(y_test, y_pred, pos_label=0)

    results_rsmpl_2.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
 ↪'Accuracy', 'Value': accuracy})
    results_rsmpl_2.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
 ↪'Precision', 'Value': precision_0})
    results_rsmpl_2.append({'Model': f"{model_name} (Class 0)", 'Metric':␣
 ↪'Recall', 'Value': recall_0})
    results_rsmpl_2.append({'Model': f"{model_name} (Class 0)", 'Metric': 'F1␣
 ↪Score', 'Value': f1_0})

    # Compute metrics for Class 1
    precision_1 = precision_score(y_test, y_pred, pos_label=1)
    recall_1 = recall_score(y_test, y_pred, pos_label=1)
    f1_1 = f1_score(y_test, y_pred, pos_label=1)

    results_rsmpl_2.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Accuracy', 'Value': accuracy})  # Same accuracy
```

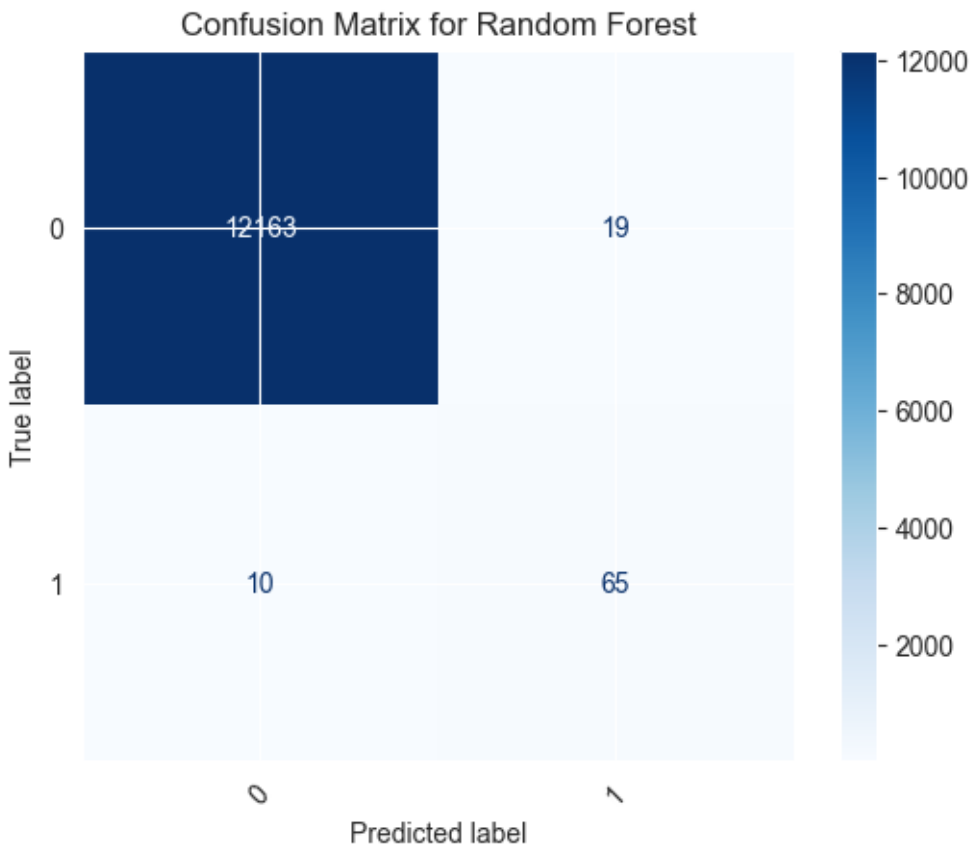
```
    results_rsmpl_2.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Precision', 'Value': precision_1})
    results_rsmpl_2.append({'Model': f"{model_name} (Class 1)", 'Metric':␣
 ↪'Recall', 'Value': recall_1})
    results_rsmpl_2.append({'Model': f"{model_name} (Class 1)", 'Metric': 'F1␣
 ↪Score', 'Value': f1_1})

    # Print Classification Report
    print(f"\nClassification Report for {model_name}:")
    print(classification_report(y_test, y_pred))

    # Save the confusion matrix plot for separate generation
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
 ↪classes_)
    disp.plot(cmap=plt.cm.Blues, xticks_rotation=45)
    plt.title(f"Confusion Matrix for {model_name}")
    plt.show()

# Convert results to DataFrame
comparison_df_rsmpl = pd.DataFrame(results_rsmpl_2)

# Plot model performance comparison (all evaluation metrics for resampled␣
 ↪models by class)
plt.figure(figsize=(16, 10))
sns.barplot(data=comparison_df_rsmpl, x="Model", y="Value", hue="Metric",␣
 ↪palette="Set2", dodge=True)
plt.title("Model Performance Comparison (Evaluation Metrics - Resampled & Tuned␣
 ↪Models by Class)", fontsize=16)
plt.ylabel("Score", fontsize=14)
plt.xlabel("Models (Class)", fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.legend(title="Metrics", loc="upper right", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Print the metrics table for both classes
print("\nDetailed Metrics for Each Resampled & Tuned Model and Class:")
print(comparison_df_rsmpl)
```

```
Evaluating Decision Tree after Hyperparameter Tuning and Resampling…

Classification Report for Decision Tree:
              precision    recall  f1-score   support
```

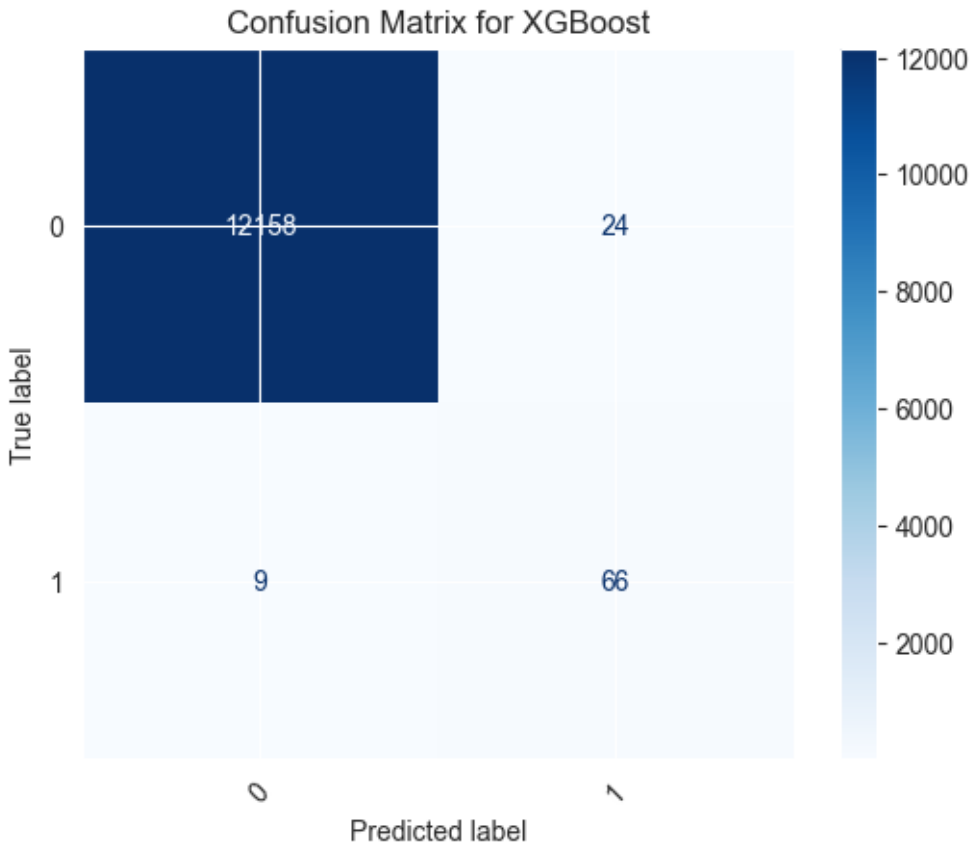
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.68      0.84      0.75        75

    accuracy                           1.00     12257
   macro avg       0.84      0.92      0.87     12257
weighted avg       1.00      1.00      1.00     12257
```

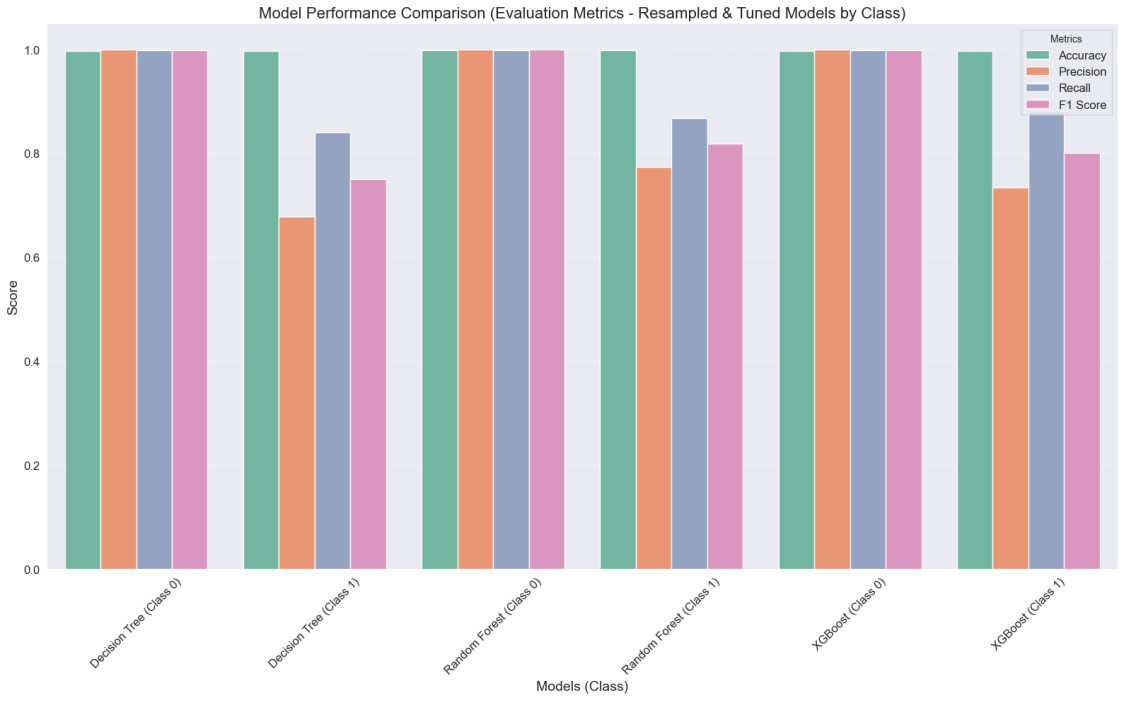



```
Evaluating Random Forest after Hyperparameter Tuning and Resampling…

Classification Report for Random Forest:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.77      0.87      0.82        75

    accuracy                           1.00     12257
   macro avg       0.89      0.93      0.91     12257
weighted avg       1.00      1.00      1.00     12257
```




```
Evaluating XGBoost after Hyperparameter Tuning and Resampling…

Classification Report for XGBoost:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.73      0.88      0.80        75

    accuracy                           1.00     12257
   macro avg       0.87      0.94      0.90     12257
weighted avg       1.00      1.00      1.00     12257
```


Confusion Matrix for XGBoost
12158
24
9
66
0
1
True label
0
1
Predicted label
12000
10000
8000
6000
4000
2000


Model Performance Comparison (Evaluation Metrics - Resampled & Tuned Models by Class)
Metrics
Accuracy
Precision
Recall
F1 Score
Score
1.0
0.8
0.6
0.4
0.2
0.0
Decision Tree (Class 0)
Decision Tree (Class 1)
Random Forest (Class 0)
Random Forest (Class 1)
XGBoost (Class 0)
XGBoost (Class 1)
Models (Class)

```
Detailed Metrics for Each Resampled & Tuned Model and Class:
                      Model     Metric     Value
0   Decision Tree (Class 0)   Accuracy  0.996573
1   Decision Tree (Class 0)  Precision  0.999013
2   Decision Tree (Class 0)     Recall  0.997537
3   Decision Tree (Class 0)   F1 Score  0.998275
4   Decision Tree (Class 1)   Accuracy  0.996573
5   Decision Tree (Class 1)  Precision  0.677419
6   Decision Tree (Class 1)     Recall  0.840000
7   Decision Tree (Class 1)   F1 Score  0.750000
8   Random Forest (Class 0)   Accuracy  0.997634
9   Random Forest (Class 0)  Precision  0.999179
10  Random Forest (Class 0)     Recall  0.998440
11  Random Forest (Class 0)   F1 Score  0.998809
12  Random Forest (Class 1)   Accuracy  0.997634
13  Random Forest (Class 1)  Precision  0.773810
14  Random Forest (Class 1)     Recall  0.866667
15  Random Forest (Class 1)   F1 Score  0.817610
16        XGBoost (Class 0)   Accuracy  0.997308
17        XGBoost (Class 0)  Precision  0.999260
18        XGBoost (Class 0)     Recall  0.998030
19        XGBoost (Class 0)   F1 Score  0.998645
20        XGBoost (Class 1)   Accuracy  0.997308
21        XGBoost (Class 1)  Precision  0.733333
22        XGBoost (Class 1)     Recall  0.880000
23        XGBoost (Class 1)   F1 Score  0.800000
```

[71]:
```python
# Iterate through unique resampled models after hyperparameter tuning for␣
 ↪evaluation
for model_name, model in rsmpl_best_models.items():
    print(f"\nEvaluating {model_name} after Hyperparameter Tuning and␣
 ↪Resampling...")

    # Predict on the training set
    y_train_pred = model.predict(X_train)  # Predict using the model on␣
 ↪training data
    train_f1 = f1_score(y_train, y_train_pred, pos_label=1)  # Compute training␣
 ↪F1-Score for Class 1

    # Predict on the test set
    y_test_pred = model.predict(X_test)  # Predict using the model on test data
    test_f1 = f1_score(y_test, y_test_pred, pos_label=1)  # Compute test␣
 ↪F1-Score for Class 1
```

```
    # Print training and test F1-Score for the current model
    print(f"Training F1-Score for {model_name} (Class 1): {train_f1:.4f}")
    print(f"Test F1-Score for {model_name} (Class 1): {test_f1:.4f}")
```

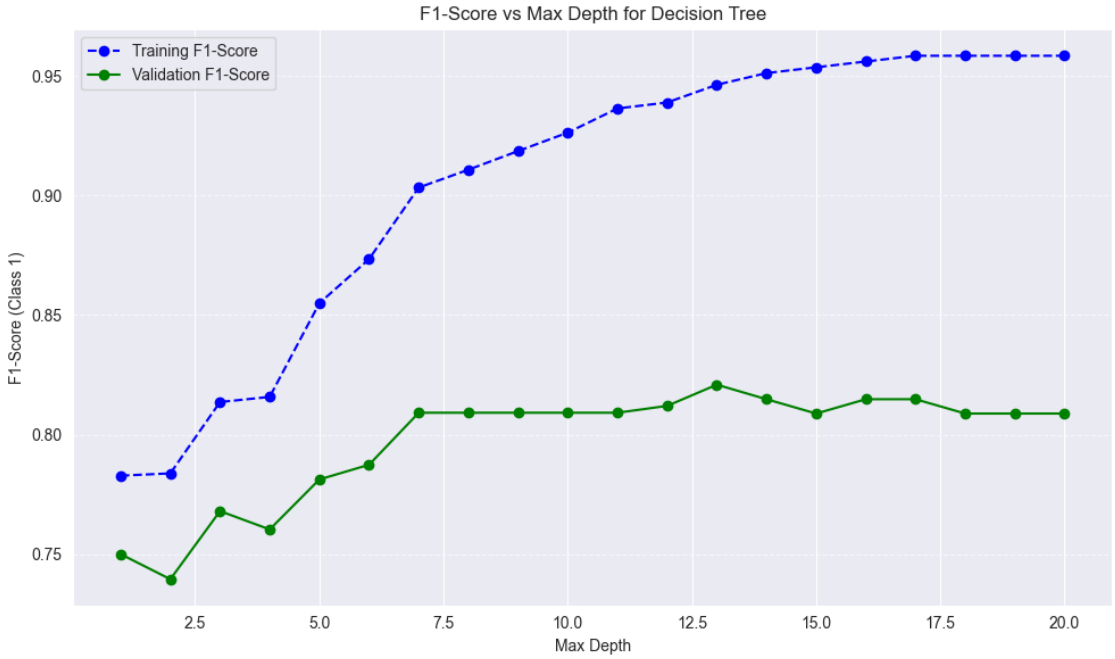
```
Evaluating Decision Tree after Hyperparameter Tuning and Resampling…
Training F1-Score for Decision Tree (Class 1): 0.8588
Test F1-Score for Decision Tree (Class 1): 0.7500

Evaluating Random Forest after Hyperparameter Tuning and Resampling…
Training F1-Score for Random Forest (Class 1): 0.8494
Test F1-Score for Random Forest (Class 1): 0.8176

Evaluating XGBoost after Hyperparameter Tuning and Resampling…
Training F1-Score for XGBoost (Class 1): 0.8435
Test F1-Score for XGBoost (Class 1): 0.8000
```

[72]:
```
# Function to plot F1-Score vs max_depth for Decision Tree and Random Forest
def plot_max_depth_vs_f1(models, X_train, y_train, X_test, y_test):
    for model_name, best_model in models.items():
        if isinstance(best_model, (DecisionTreeClassifier,␣
↪RandomForestClassifier)):
            print(f"\nGenerating F1-Score Curve for {model_name}...")

            # Define max_depth range
            max_depth_range = range(1, 21)
            training_f1_scores = []
            validation_f1_scores = []

            for max_depth in max_depth_range:
                # Update model with current max_depth and fit
                model = best_model.set_params(max_depth=max_depth)
                model.fit(X_train, y_train)

                # Calculate F1-Score for Class 1
                train_f1 = f1_score(y_train, model.predict(X_train),␣
↪pos_label=1)
                val_f1 = f1_score(y_test, model.predict(X_test), pos_label=1)

                training_f1_scores.append(train_f1)
                validation_f1_scores.append(val_f1)

            # Plot F1-Score vs max_depth
            plt.figure(figsize=(10, 6))
            plt.plot(max_depth_range, training_f1_scores, label='Training␣
↪F1-Score', linestyle='--', marker='o', color='blue')
```

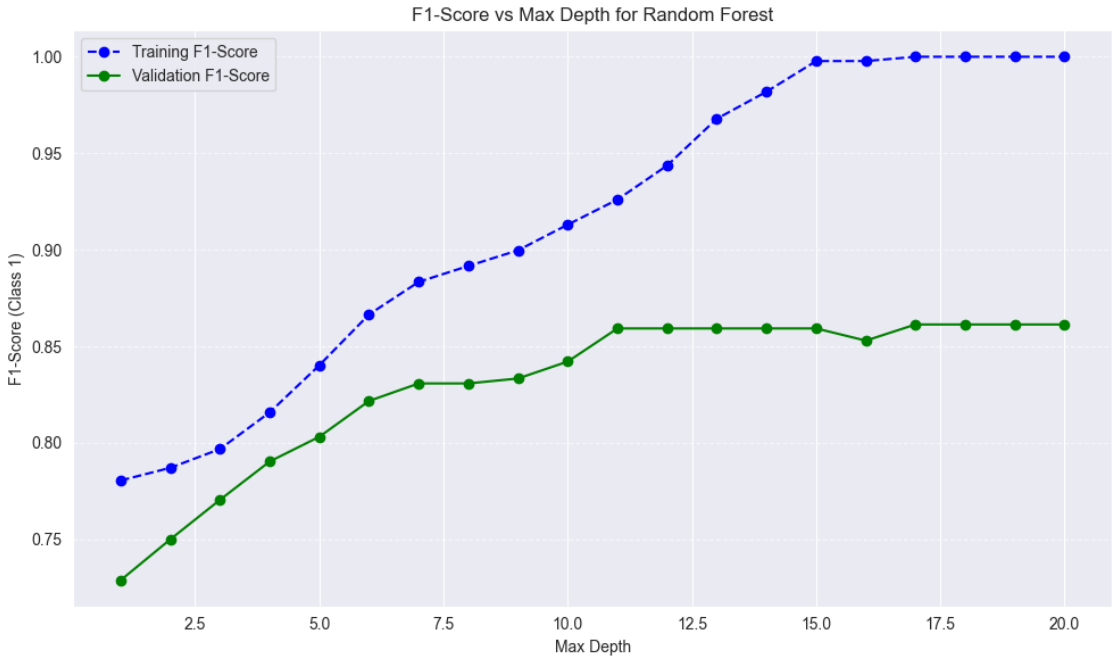
```
            plt.plot(max_depth_range, validation_f1_scores, label='Validation␣
 ↪F1-Score', marker='o', color='green')
            plt.xlabel('Max Depth')
            plt.ylabel('F1-Score (Class 1)')
            plt.title(f'F1-Score vs Max Depth for {model_name}')
            plt.legend(loc='best')
            plt.grid(axis='y', linestyle='--', alpha=0.7)
            plt.tight_layout()
            plt.show()

# Function to plot F1-Score vs n_estimators for XGBoost
def plot_n_estimators_vs_f1(models, X_train, y_train, X_test, y_test):
    for model_name, best_model in models.items():
        if isinstance(best_model, XGBClassifier):
            print(f"\nGenerating F1-Score Curve for {model_name}...")

            # Define n_estimators range
            n_estimators_range = [50, 100, 150, 200, 250]
            training_f1_scores = []
            validation_f1_scores = []

            for n_estimators in n_estimators_range:
                # Update model with current n_estimators and fit
                model = best_model.set_params(n_estimators=n_estimators)
                model.fit(X_train, y_train)

                # Calculate F1-Score for Class 1
                train_f1 = f1_score(y_train, model.predict(X_train),␣
 ↪pos_label=1)
                val_f1 = f1_score(y_test, model.predict(X_test), pos_label=1)

                training_f1_scores.append(train_f1)
                validation_f1_scores.append(val_f1)

            # Plot F1-Score vs n_estimators
            plt.figure(figsize=(10, 6))
            plt.plot(n_estimators_range, training_f1_scores, label='Training␣
 ↪F1-Score', linestyle='--', marker='o', color='blue')
            plt.plot(n_estimators_range, validation_f1_scores,␣
 ↪label='Validation F1-Score', marker='o', color='green')
            plt.xlabel('Number of Estimators')
            plt.ylabel('F1-Score (Class 1)')
            plt.title(f'F1-Score vs Number of Estimators for {model_name}')
            plt.legend(loc='best')
            plt.grid(axis='y', linestyle='--', alpha=0.7)
            plt.tight_layout()
            plt.show()
```

```
# Call the functions
plot_max_depth_vs_f1(best_models, X_train, y_train, X_test, y_test)
plot_n_estimators_vs_f1(best_models, X_train, y_train, X_test, y_test)
```

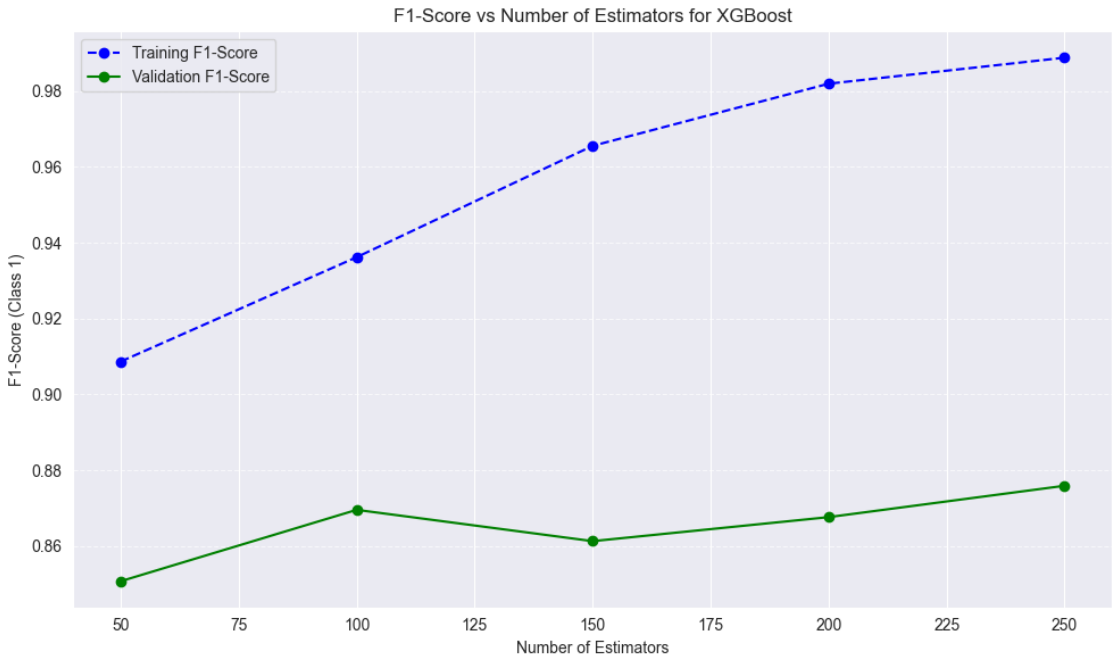
```
Generating F1-Score Curve for Decision Tree…
```




```
Generating F1-Score Curve for Random Forest…
```




```
Generating F1-Score Curve for XGBoost…
```



[73]:
```
# Set up Stratified K-Fold cross-validation after resampling and hyperparameter␣
 ↪tuning
stratified_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Iterate through each resampled model and compute Stratified K-Fold␣
 ↪cross-validation scores
for model_name, model in rsmpl_best_models.items():
    print(f"\nPerforming Stratified Cross-Validation for {model_name} after␣
 ↪Resampling and Hyperparameter Tuning...")

    # Perform cross-validation using F1-Score for Class 1
    cross_val_scores = cross_val_score(
        model, X_train, y_train,
        cv=stratified_kfold,
        scoring=make_scorer(f1_score, pos_label=1)
    )

    # Print the results for the current model
    print(f"Stratified Cross-Validation F1-Scores for {model_name}:␣
 ↪{cross_val_scores}")
    print(f"Mean Stratified Cross-Validation F1-Score for {model_name}:␣
 ↪{cross_val_scores.mean():.4f}")
```

```
Performing Stratified Cross-Validation for Decision Tree after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation F1-Scores for Decision Tree: [0.7755102  0.70212766
0.76086957 0.71428571 0.76744186]
Mean Stratified Cross-Validation F1-Score for Decision Tree: 0.7440

Performing Stratified Cross-Validation for Random Forest after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation F1-Scores for Random Forest: [0.8        0.775
0.85365854 0.6835443  0.81395349]
Mean Stratified Cross-Validation F1-Score for Random Forest: 0.7852

Performing Stratified Cross-Validation for XGBoost after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation F1-Scores for XGBoost: [0.85714286 0.81012658
0.82758621 0.71604938 0.83333333]
Mean Stratified Cross-Validation F1-Score for XGBoost: 0.8088
```

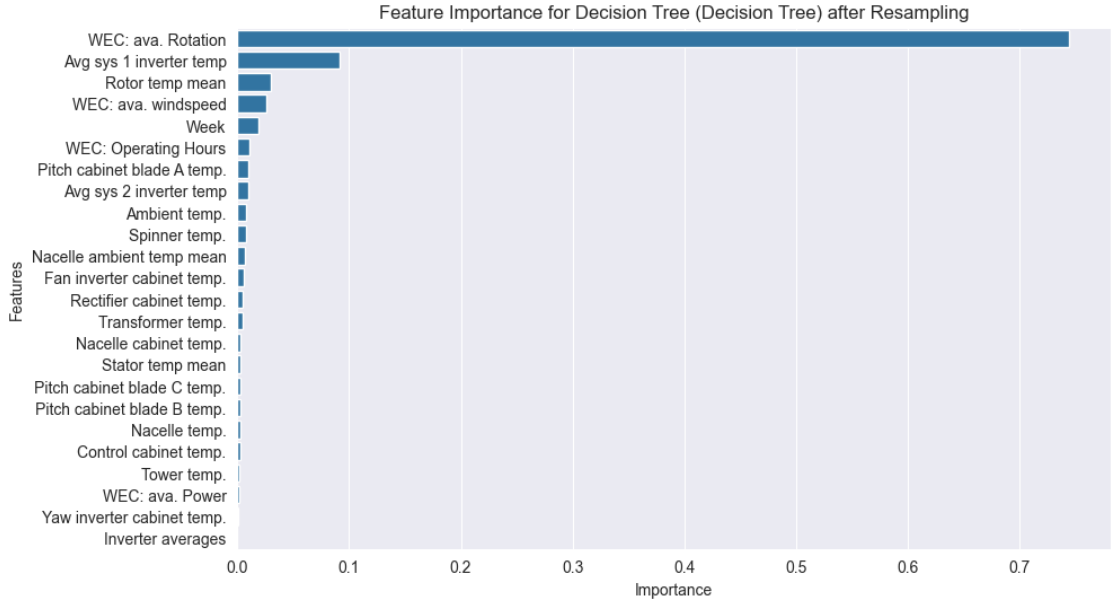
3.5 Feature Importance after Resampling & Hyperparameter Tuning

[74]:
```
# Function to plot and print feature importance for resampled models
def plot_and_print_feature_importance_after_resampling(rsmpl_best_models,␣
↪X_resampled_smote_enn):
    for model_name, model in rsmpl_best_models.items():
        print(f"\nFeature Importance for {model_name} after Resampling:")

        # Decision Tree: Print and plot feature importance
        if isinstance(model, DecisionTreeClassifier):
            feature_importance = model.feature_importances_

            # Create a DataFrame for feature importance
            importance_df = pd.DataFrame({
                'Feature': X_resampled_smote_enn.columns,
                'Importance': feature_importance
            }).sort_values(by='Importance', ascending=False)

            # Print feature importance
            print(importance_df)

            # Plot feature importance
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df)
            plt.title(f'Feature Importance for {model_name} (Decision Tree)␣
↪after Resampling')
            plt.xlabel('Importance')
            plt.ylabel('Features')
            plt.show()

        # Random Forest: Print and plot feature importance
        elif isinstance(model, RandomForestClassifier):
            feature_importance = model.feature_importances_

            # Create a DataFrame for feature importance
            importance_df = pd.DataFrame({
                'Feature': X_resampled_smote_enn.columns,
                'Importance': feature_importance
            }).sort_values(by='Importance', ascending=False)

            # Print feature importance
            print(importance_df)

            # Plot feature importance
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df)
```

```
            plt.title(f'Feature Importance for {model_name} (Random Forest)␣
 ↪after Resampling')
            plt.xlabel('Importance')
            plt.ylabel('Features')
            plt.show()

        # XGBoost: Print and plot feature importance
        elif isinstance(model, XGBClassifier):
            feature_importance = model.feature_importances_
            feature_names = X_resampled_smote_enn.columns

            # Create a DataFrame for feature importance
            importance_df = pd.DataFrame({
                'Feature': feature_names,
                'Importance': feature_importance
            }).sort_values(by='Importance', ascending=False)

            # Print feature importance
            print(importance_df)

            # Plot feature importance
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df)
            plt.title(f'Feature Importance for {model_name} (XGBoost) after␣
 ↪Resampling')
            plt.xlabel('Importance')
            plt.ylabel('Features')
            plt.show()

# Assuming rsmpl_best_models contains the best fitted models after␣
 ↪hyperparameter tuning and resampling
# and X_resampled_smote_enn is the resampled training dataset
plot_and_print_feature_importance_after_resampling(rsmpl_best_models,␣
 ↪X_resampled_smote_enn)
```

```
Feature Importance for Decision Tree after Resampling:
                        Feature  Importance
1            WEC: ava. Rotation    0.744529
19      Avg sys 1 inverter temp    0.091076
21              Rotor temp mean    0.030069
0           WEC: ava. windspeed    0.025643
18                         Week    0.019322
3          WEC: Operating Hours    0.010803
5   Pitch cabinet blade A temp.    0.009958
20      Avg sys 2 inverter temp    0.009635
13                Ambient temp.    0.007846
```

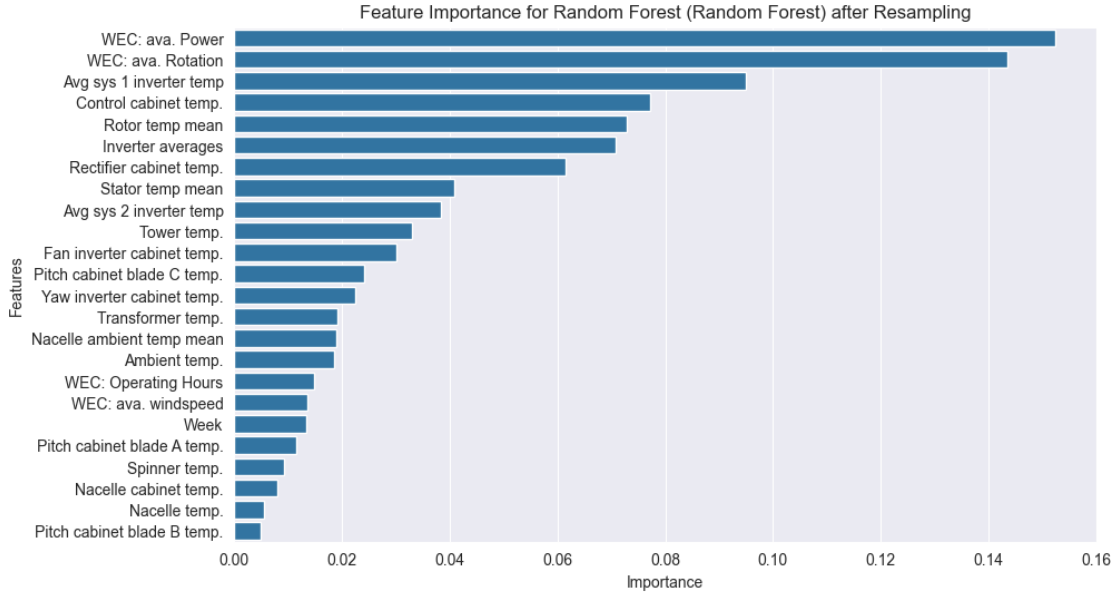
4	Spinner temp.	0.007519
23	Nacelle ambient temp mean	0.006951
12	Fan inverter cabinet temp.	0.006184
10	Rectifier cabinet temp.	0.005165
16	Transformer temp.	0.004650
9	Nacelle cabinet temp.	0.003199
22	Stator temp mean	0.002947
7	Pitch cabinet blade C temp.	0.002861
6	Pitch cabinet blade B temp.	0.002533
8	Nacelle temp.	0.002436
15	Control cabinet temp.	0.002369
14	Tower temp.	0.002013
2	WEC: ava. Power	0.001342
11	Yaw inverter cabinet temp.	0.000908
17	Inverter averages	0.000041



Feature Importance for Random Forest after Resampling:

	Feature	Importance
2	WEC: ava. Power	0.152461
1	WEC: ava. Rotation	0.143521
19	Avg sys 1 inverter temp	0.094870
15	Control cabinet temp.	0.077278
21	Rotor temp mean	0.072841
17	Inverter averages	0.070721
10	Rectifier cabinet temp.	0.061568
22	Stator temp mean	0.040762
20	Avg sys 2 inverter temp	0.038367

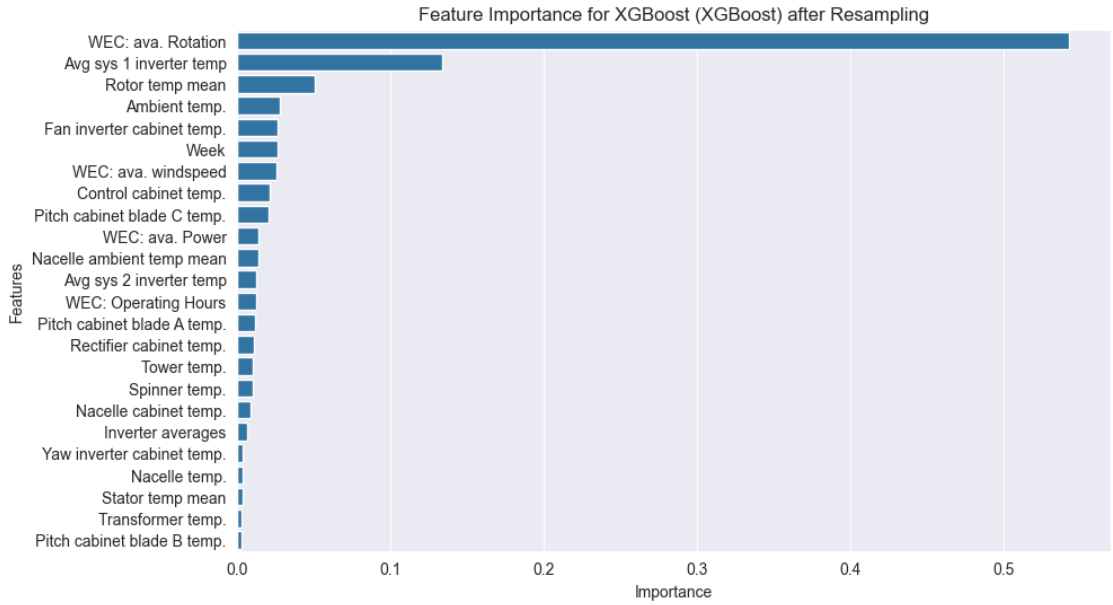
14	Tower temp.	0.033045
12	Fan inverter cabinet temp.	0.030143
7	Pitch cabinet blade C temp.	0.024077
11	Yaw inverter cabinet temp.	0.022545
16	Transformer temp.	0.019111
23	Nacelle ambient temp mean	0.018949
13	Ambient temp.	0.018588
3	WEC: Operating Hours	0.014837
0	WEC: ava. windspeed	0.013608
18	Week	0.013316
5	Pitch cabinet blade A temp.	0.011581
4	Spinner temp.	0.009316
9	Nacelle cabinet temp.	0.007939
8	Nacelle temp.	0.005619
6	Pitch cabinet blade B temp.	0.004934



Feature Importance for XGBoost after Resampling:

	Feature	Importance
1	WEC: ava. Rotation	0.542969
19	Avg sys 1 inverter temp	0.133812
21	Rotor temp mean	0.050806
13	Ambient temp.	0.027516
12	Fan inverter cabinet temp.	0.026172
18	Week	0.026114
0	WEC: ava. windspeed	0.025553
15	Control cabinet temp.	0.021345
7	Pitch cabinet blade C temp.	0.020691

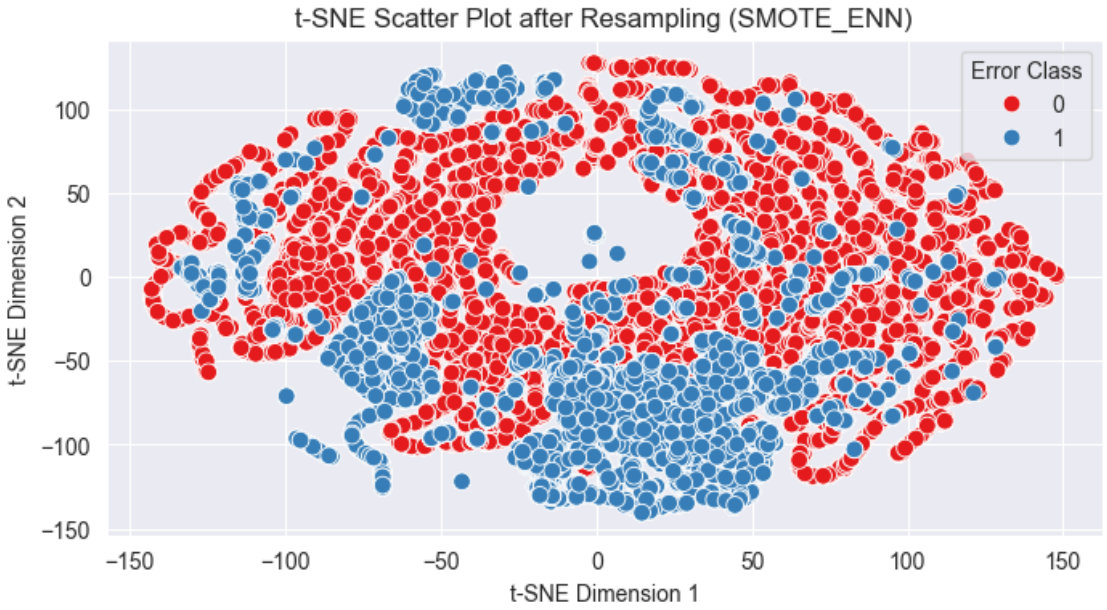
```
2                WEC: ava. Power    0.013954
23    Nacelle ambient temp mean     0.013733
20      Avg sys 2 inverter temp     0.012662
3          WEC: Operating Hours     0.011998
5   Pitch cabinet blade A temp.     0.011201
10      Rectifier cabinet temp.     0.010592
14                 Tower temp.      0.010204
4                Spinner temp.      0.009739
9         Nacelle cabinet temp.     0.008923
17            Inverter averages     0.006514
11   Yaw inverter cabinet temp.     0.003585
8                 Nacelle temp.     0.003311
22             Stator temp mean     0.003278
16            Transformer temp.     0.002851
6   Pitch cabinet blade B temp.     0.002478
```



[ ]:

[75]:
```
# --- t-SNE Visualization ---
# Applying t-SNE to reduce dimensionality to 2D for visualization
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_resampled_smote_enn)

# Create a DataFrame for visualization
tsne_df = pd.DataFrame(X_tsne, columns=['t-SNE Dimension 1', 't-SNE Dimension␣
↪2'])
tsne_df['Error_Class'] = y_resampled_smote_enn
```

```
# Plot the t-SNE scatter plot
plt.figure(figsize=(8, 4))
sns.scatterplot(x='t-SNE Dimension 1', y='t-SNE Dimension 2',␣
 ↪hue='Error_Class', palette='Set1', data=tsne_df, s=60)
plt.title('t-SNE Scatter Plot after Resampling (SMOTE_ENN)')
plt.legend(title='Error Class', loc='upper right')
plt.show()
```



[ ]:

[75]: