AML_Projekt

December 13, 2024

1 Wind Turbine Analytics

Data Analytics and Classification Model for Failure Detection of Wind Turbine from IIoT Data

2 Original Data

[113]:
```
# Importing important Python Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE


# for preprocessing
from sklearn.preprocessing import OneHotEncoder, StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, StratifiedKFold,␣
 ↪GridSearchCV
from xgboost import plot_importance
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,␣
 ↪roc_curve, roc_auc_score, auc

# 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

[114]:
```
df = pd.read_csv('pred_maint_wind.csv')
df.head()
```

[114]:

	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]
```

[115]:
```
df.info
```

[115]:
```
<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]>
```

[116]:
```
df.shape
```

[116]:
```
(49027, 66)
```

[117]:
```
df.describe()
```

[117]:
```
                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]
```

[118]:
```
df.dtypes
```

[118]:
```
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
```

2.2 Exploring and Cleaning

[119]:
```
df.columns
```

[119]:
```
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')
```

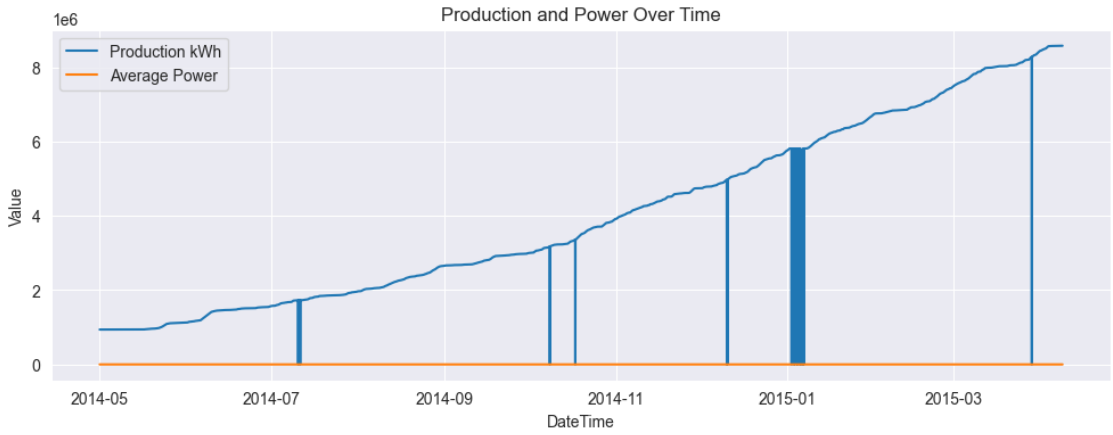
[120]:
```
# Check for missing values
print("Missing Values per Column:")
print(df.isna().sum())

# 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
```

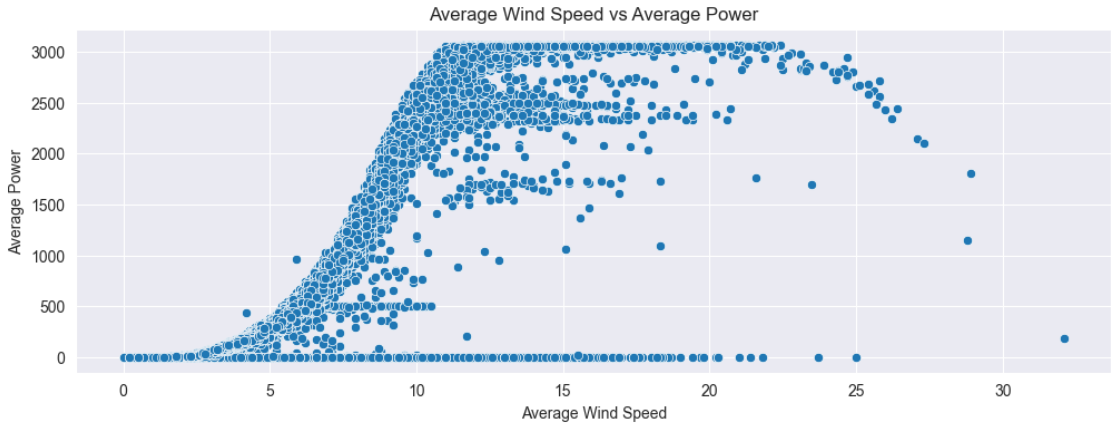
As we can see we have relatively cleaned dataset as we have not found any missing value.

[121]:
```
# Convert DateTime column to a datetime object if not already
df['DateTime'] = pd.to_datetime(df['DateTime'])
```

[122]:
```
# 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()
```



[123]:
```
# 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()
```

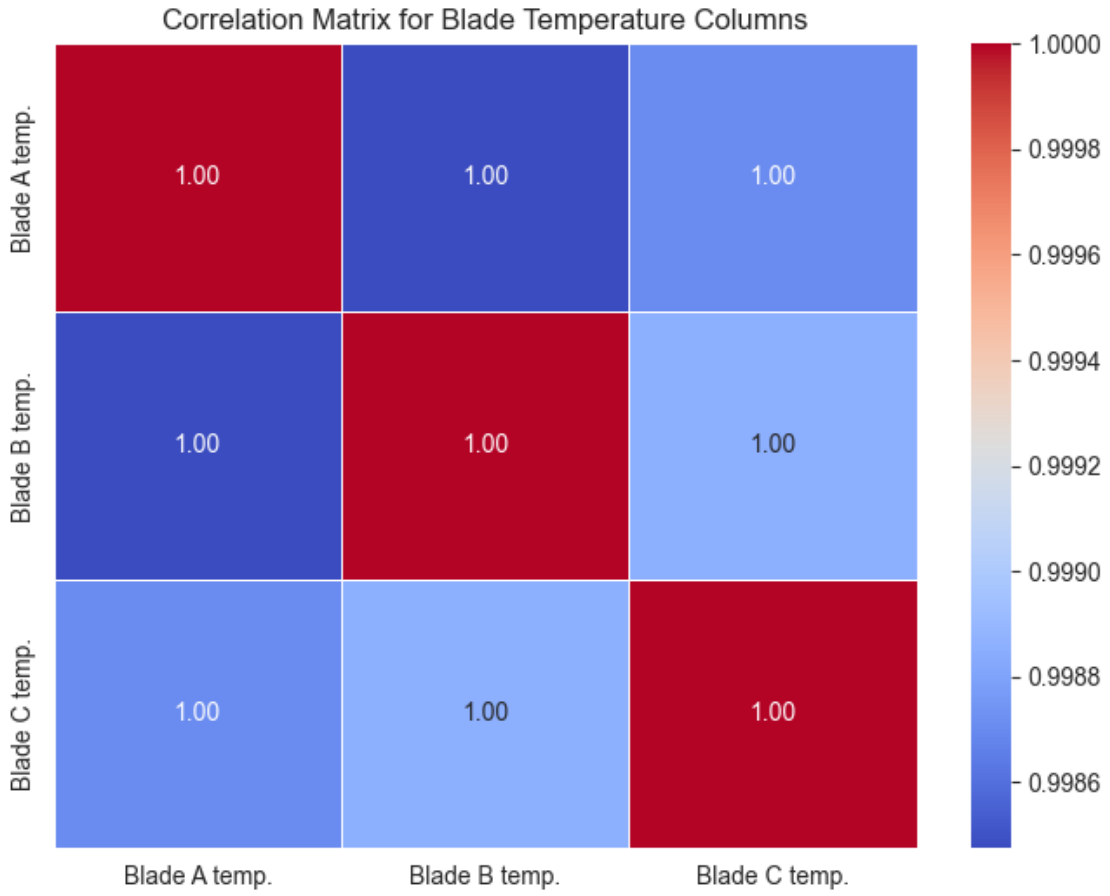


- The average power (`WEC: ava. Power`) remains constant because wind turbines operate within a fixed capacity range, determined by design and average wind conditions.
- Production (`WEC: Production kWh`) increases over time due to longer operational periods, reduced downtime(in total 6 instances), or 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.

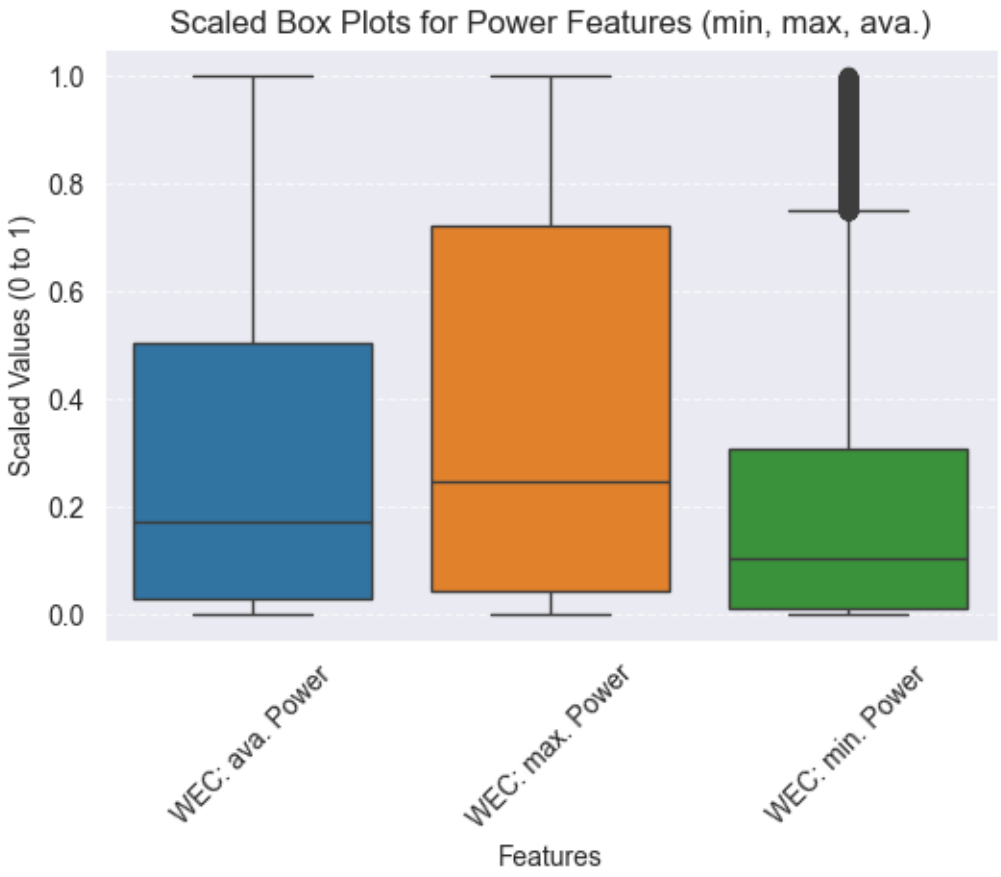
[124]:
```
# Assuming 'df' is your dataset and it contains columns related to blades
# Select relevant blade columns
blade_columns = ['Blade A temp.', 'Blade B temp.', 'Blade C temp.']  # Update this list if there are more blade-related columns

# 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()
```

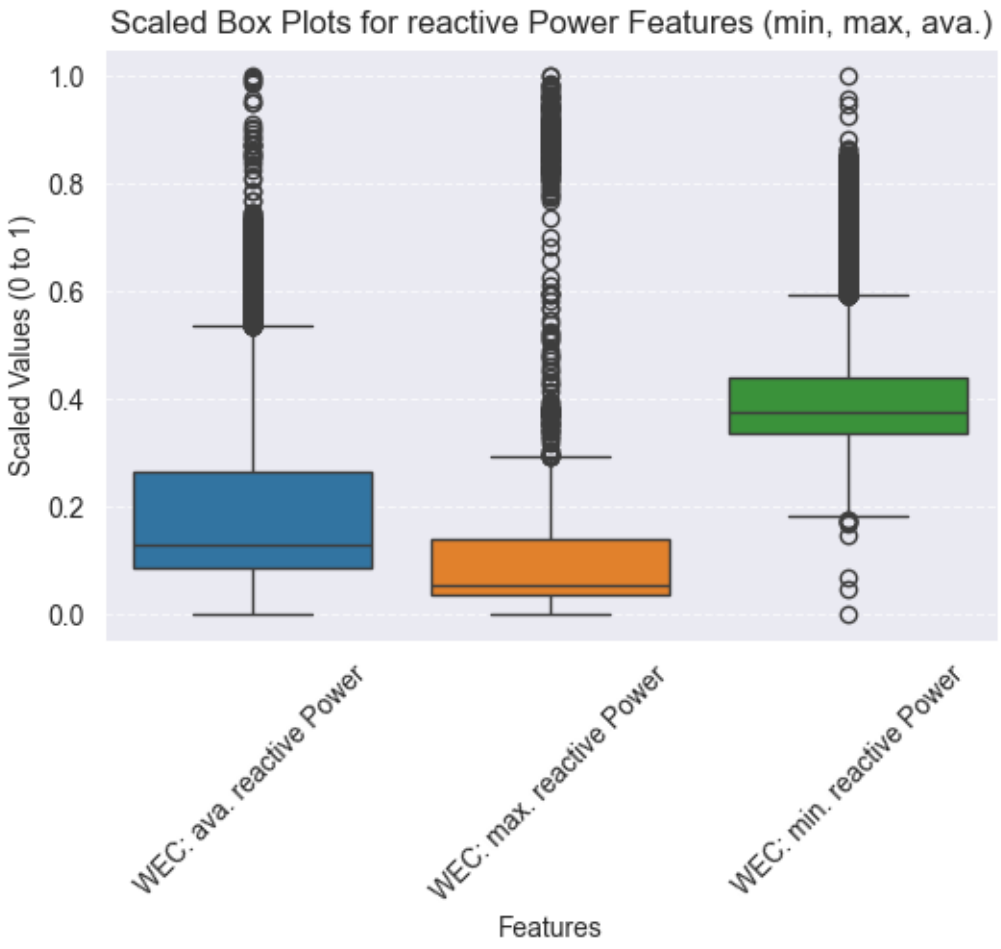
```
Correlation matrix for Blade columns:
               Blade A temp.  Blade B temp.  Blade C temp.
Blade A temp.       1.000000       0.998474       0.998702
Blade B temp.       0.998474       1.000000       0.998857
Blade C temp.       0.998702       0.998857       1.000000
```

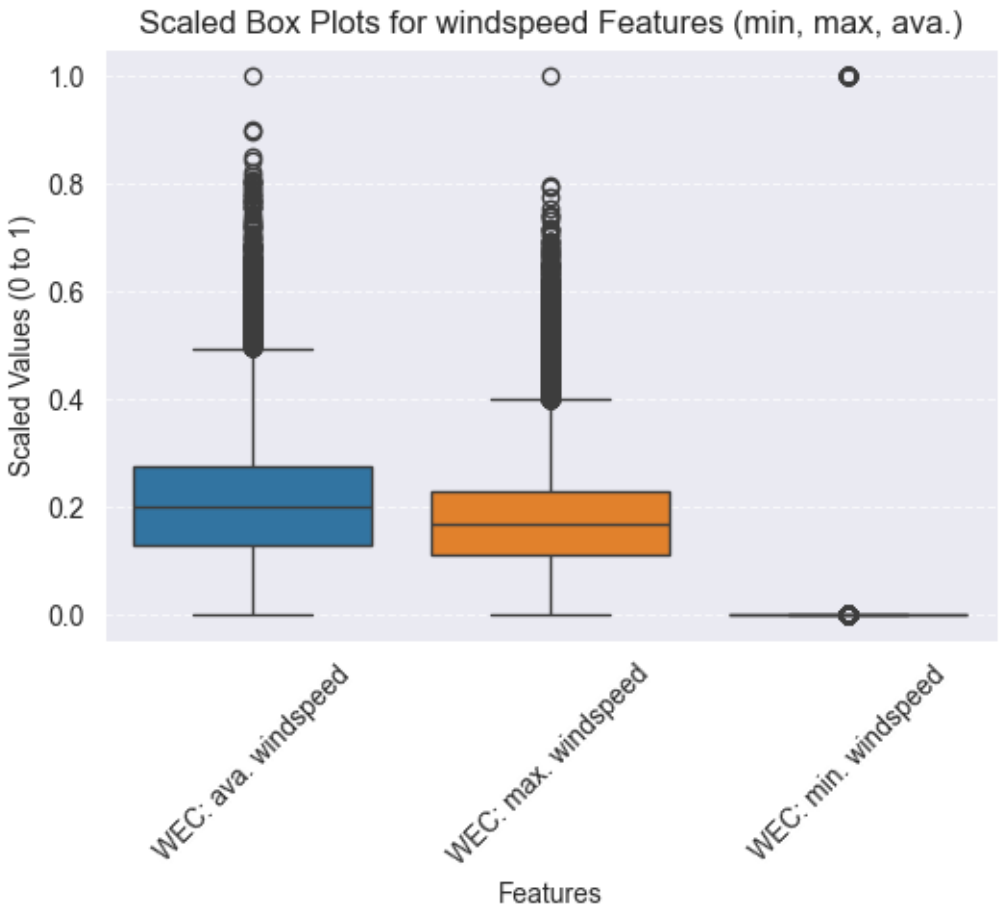


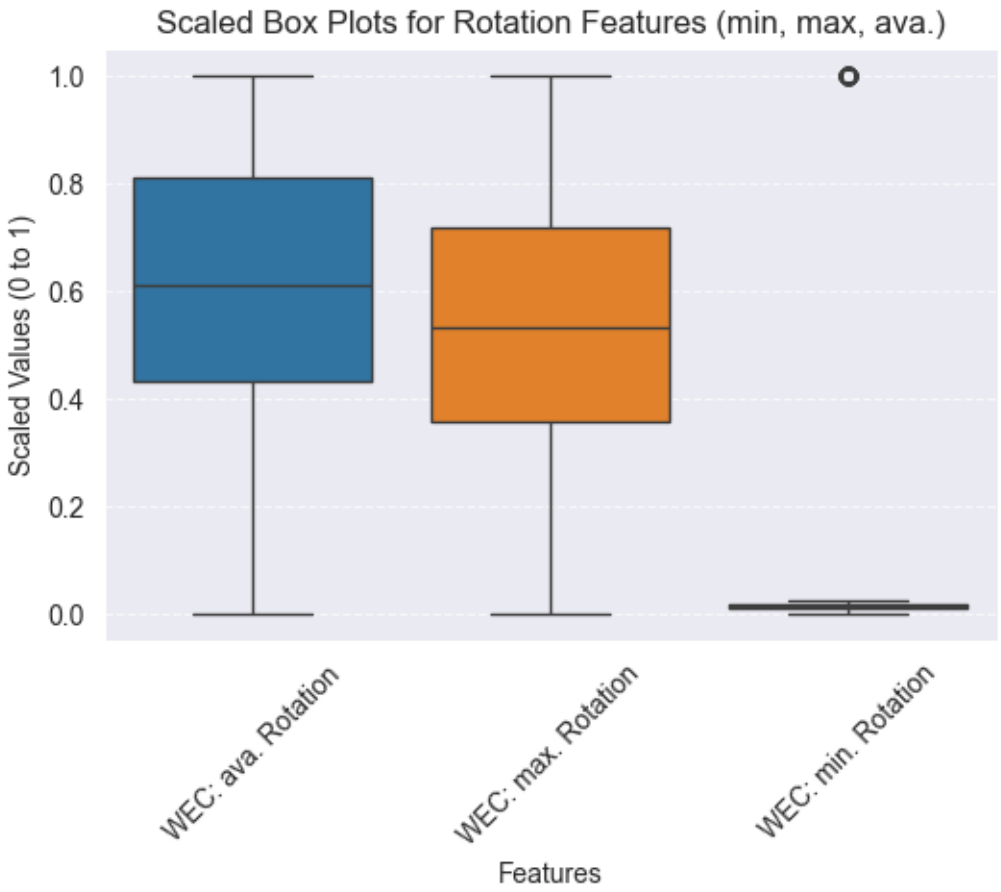
[125]:
```
# Function to normalize data and plot boxplots
def plot_boxplots_with_scaling(df, feature_type, feature_keywords,␣
↪exclude_keywords=None):
    """
    Normalizes data and plots boxplots for min, max, and average columns for a␣
↪specific feature type.

    :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)

        # 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')
    )
```


Scaled Box Plots for Power Features (min, max, ava.)
Scaled Values (0 to 1)
1.0
0.8
0.6
0.4
0.2
0.0
WEC: ava. Power
WEC: max. Power
WEC: min. Power
Features

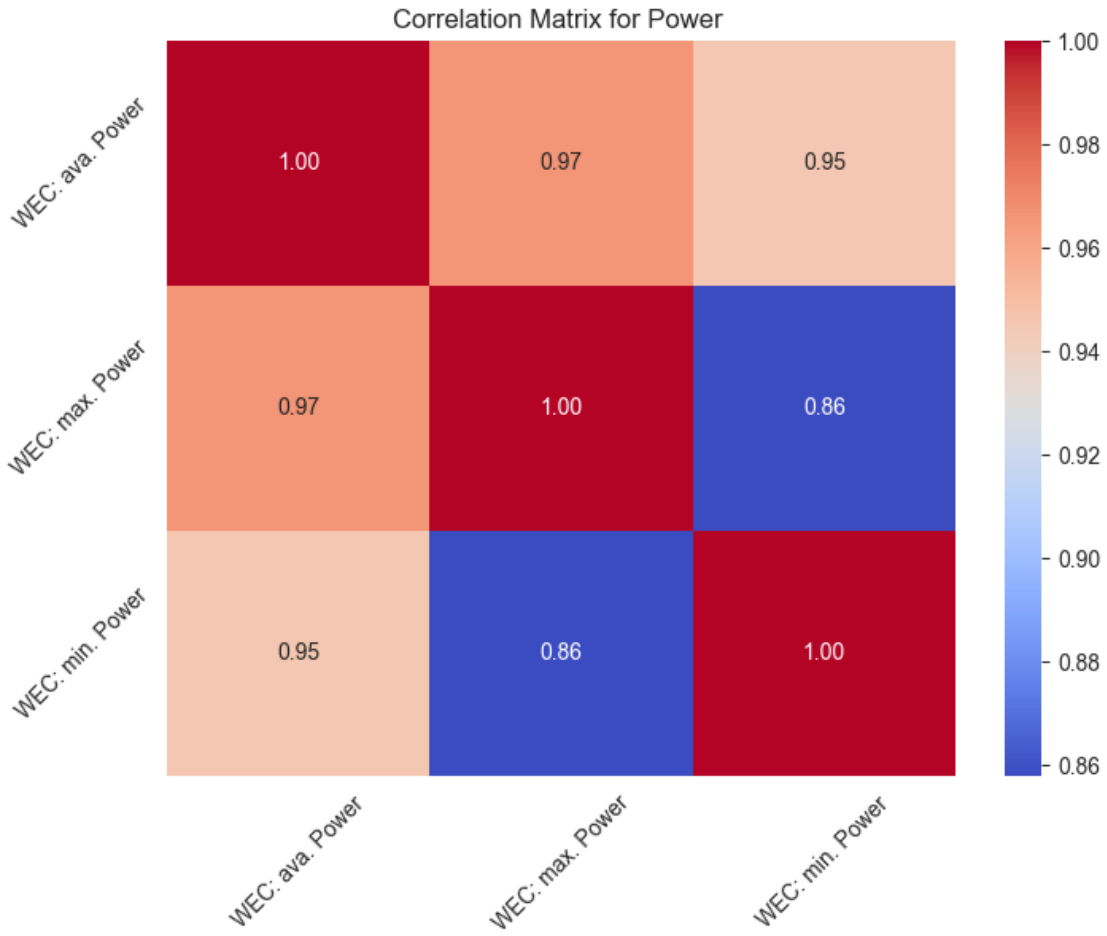

Scaled Box Plots for reactive Power Features (min, max, ava.)
Scaled Values (0 to 1)
1.0
0.8
0.6
0.4
0.2
0.0
WEC: ava. reactive Power
WEC: max. reactive Power
WEC: min. reactive Power
Features


Scaled Box Plots for windspeed Features (min, max, ava.)
Scaled Values (0 to 1)
1.0
0.8
0.6
0.4
0.2
0.0
WEC: ava. windspeed
WEC: max. windspeed
WEC: min. windspeed
Features



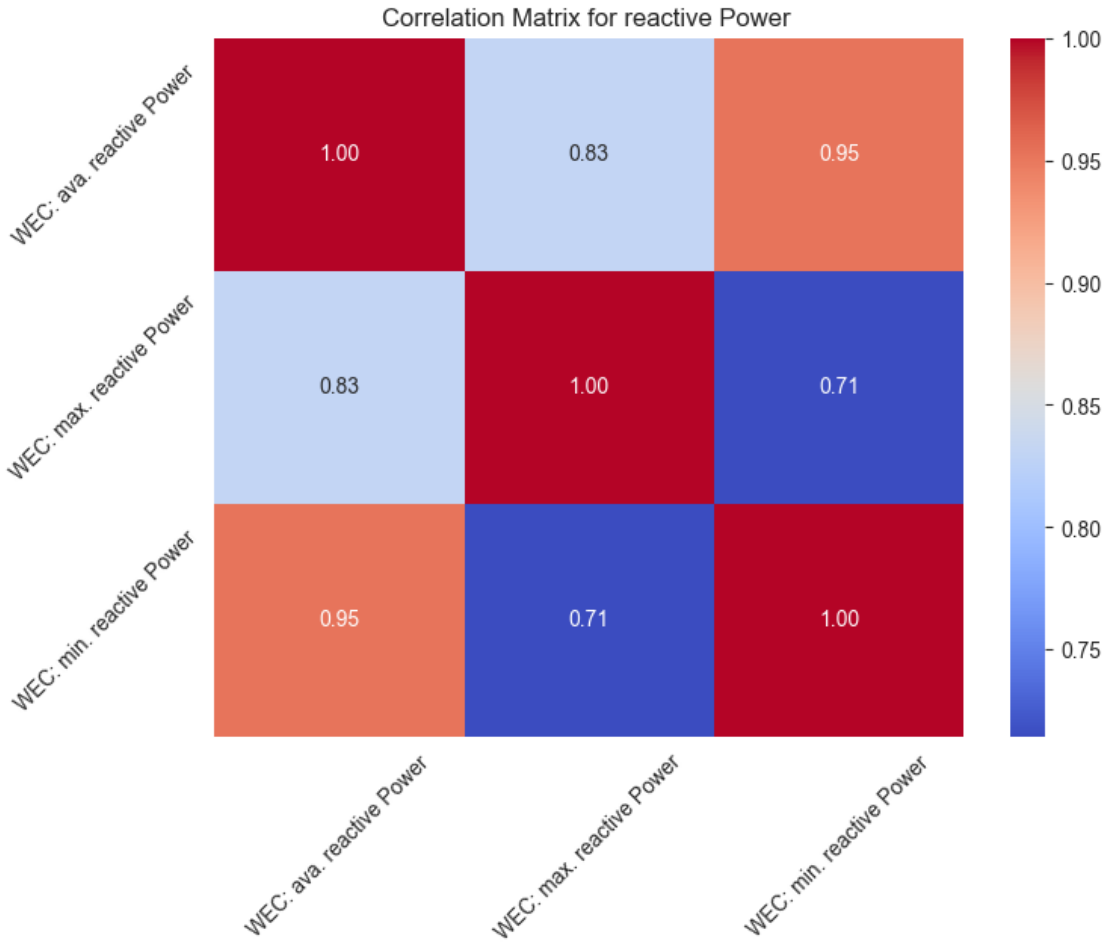
[126]:
```
# Function to compute and plot correlation matrix for a specific feature type
def plot_correlation_matrix(df, feature_type, exclude_keywords=None):
    """
    Plots a correlation matrix for min, max, and avg columns of a given feature␣
 ↪type.

    :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')
```

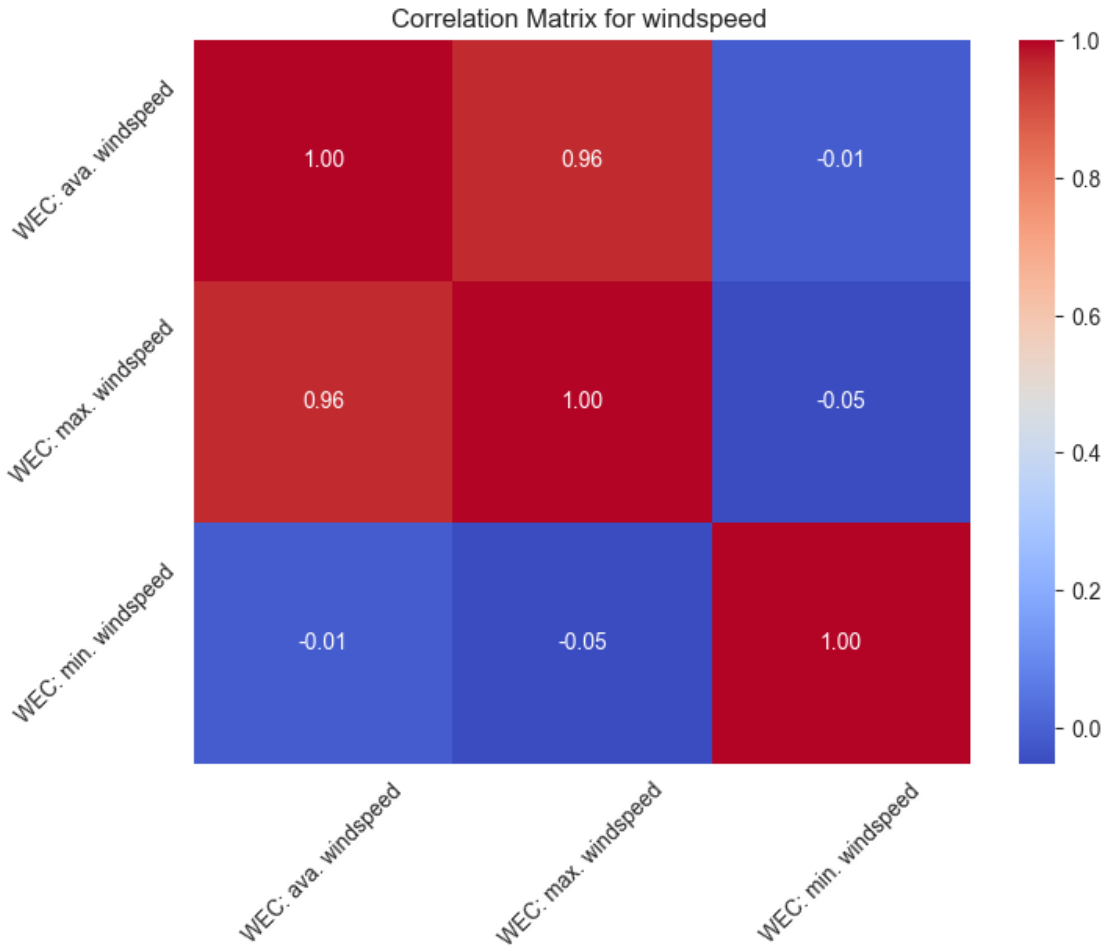


Correlation Matrix for Power :

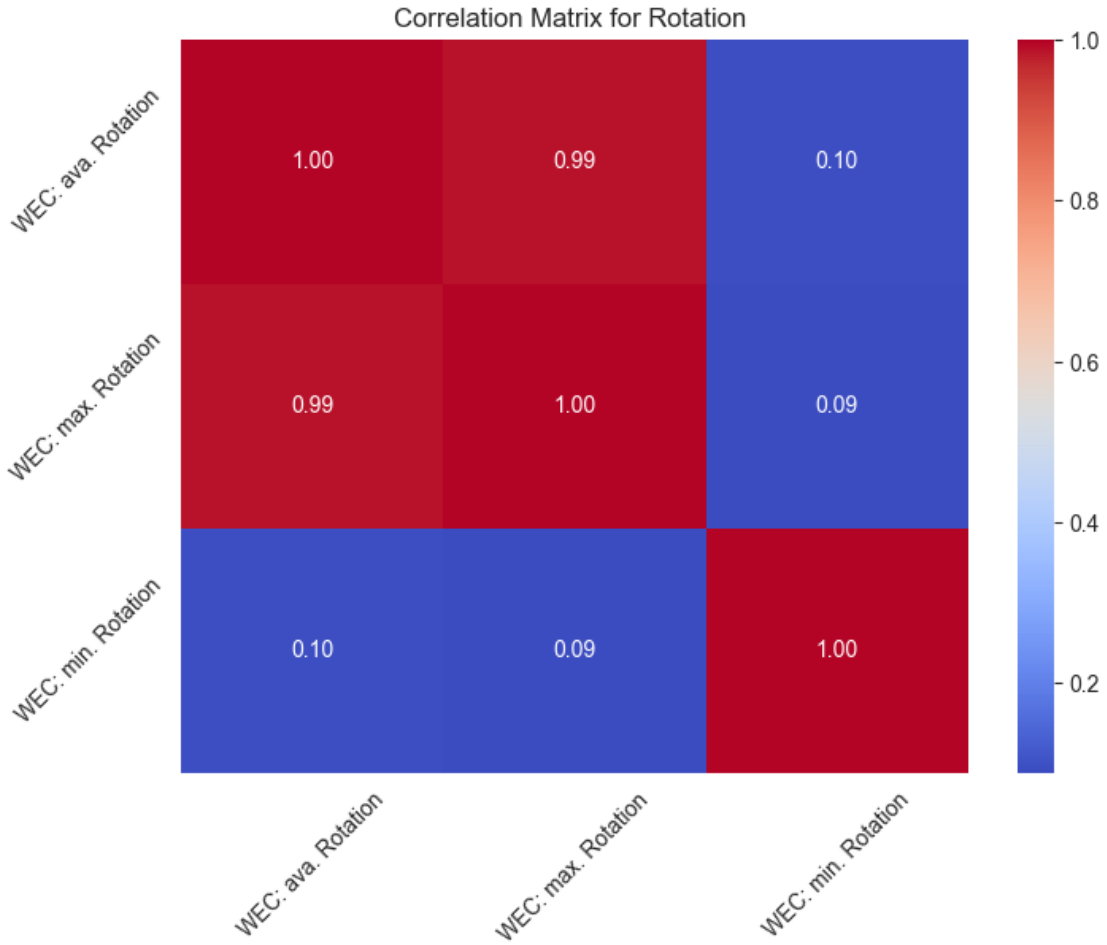
	WEC: ava. Power	WEC: max. Power	WEC: min. Power
WEC: ava. Power	1.000000	0.965711	0.945421
WEC: max. Power	0.965711	1.000000	0.857933
WEC: min. Power	0.945421	0.857933	1.000000




```
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:

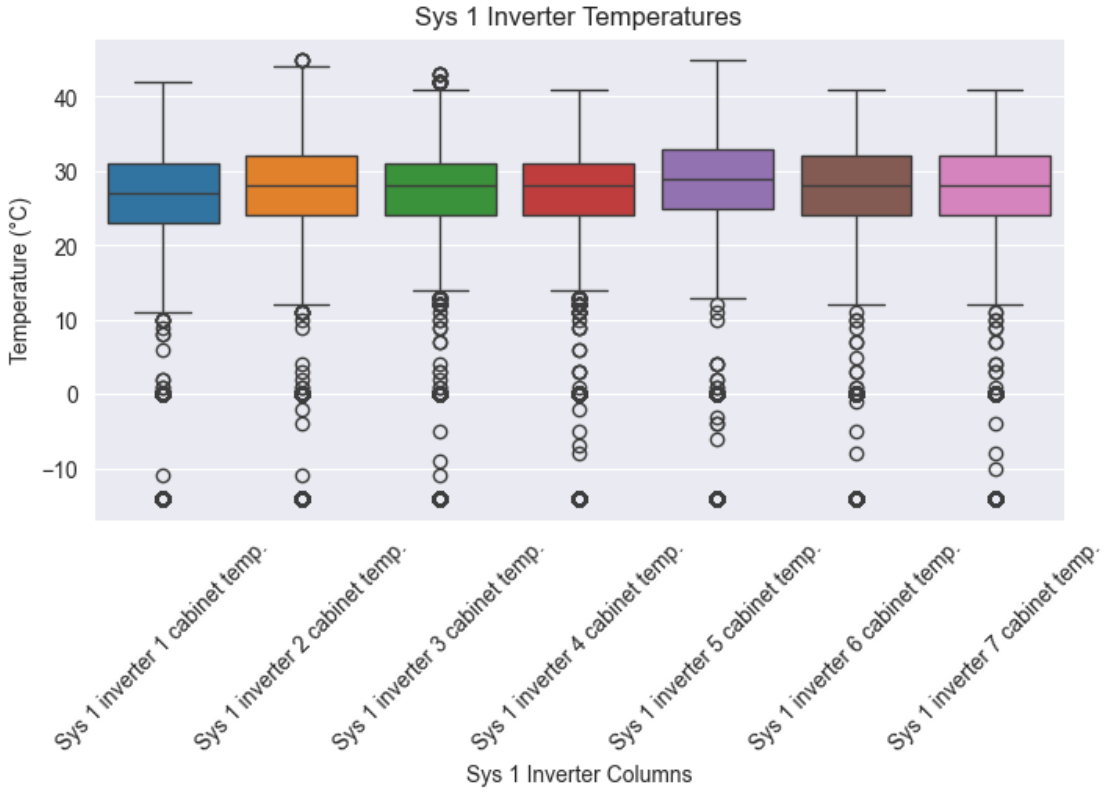
1. Simplifies the model by avoiding redundancy (e.g., removing features with high correlation to another retained feature).
2. Improves data quality by excluding features with outliers or poor correlations.
3. Focuses on features that provide the most representative information (e.g., average values).

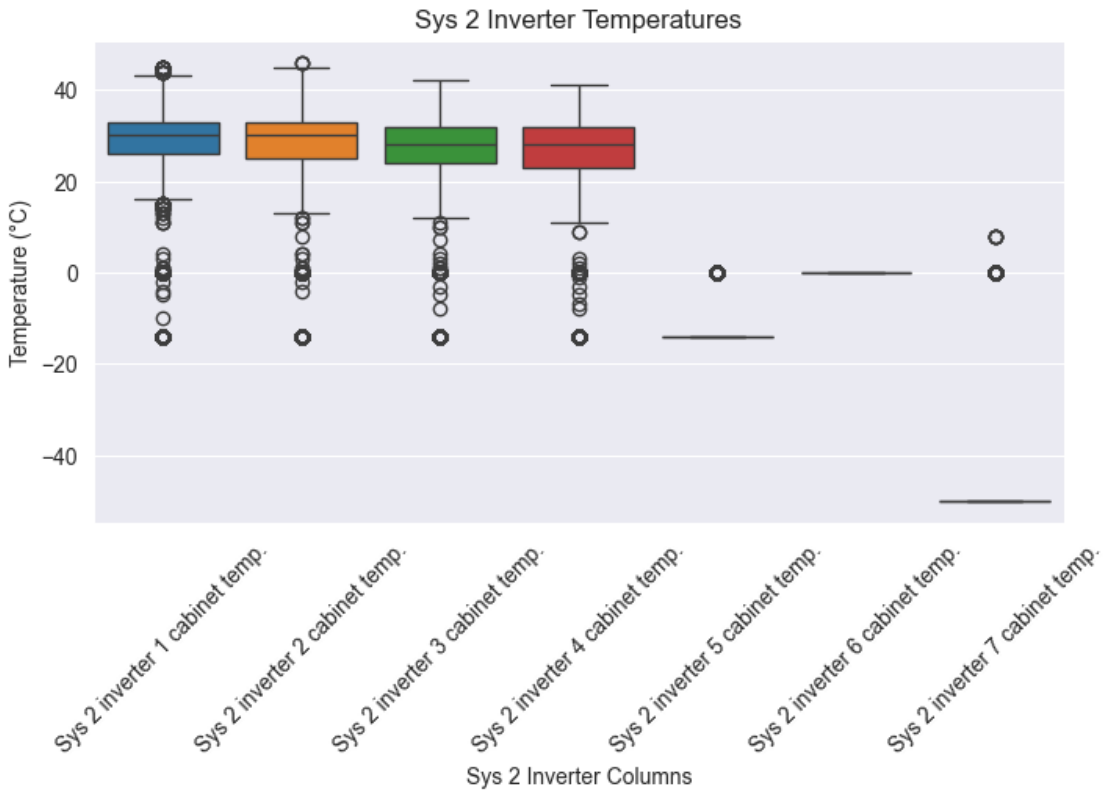
Next Steps Before Modeling:

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.

[127]:
```
# Plotting inverter temperatures together
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.']

# Plotting Sys 1 Inverter Temperatures
plt.figure(figsize=(8, 4))
sns.boxplot(data=df[sys1_inverter_temps])
```

```
plt.xticks(rotation=45)
plt.title('Sys 1 Inverter Temperatures')
plt.ylabel('Temperature (°C)')
plt.xlabel('Sys 1 Inverter Columns')
plt.show()

# Plotting Sys 2 Inverter Temperatures
plt.figure(figsize=(8, 4))
sns.boxplot(data=df[sys2_inverter_temps])
plt.xticks(rotation=45)
plt.title('Sys 2 Inverter Temperatures')
plt.ylabel('Temperature (°C)')
plt.xlabel('Sys 2 Inverter Columns')
plt.show()
```





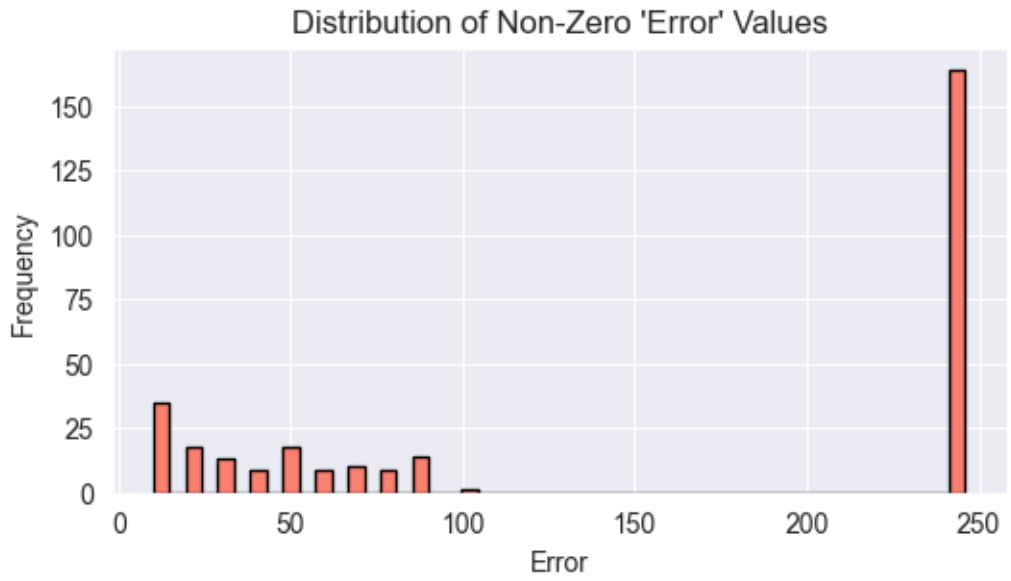
These boxplots show that `Sys 1` inverter temperatures are consistent and realistic, with mean values around **27°C to 28°C**, standard deviations of ~**5.7°C to 6.3°C**, and maximum temperatures up to **45°C**, indicating normal operation.

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.

[128]:
```
# 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:

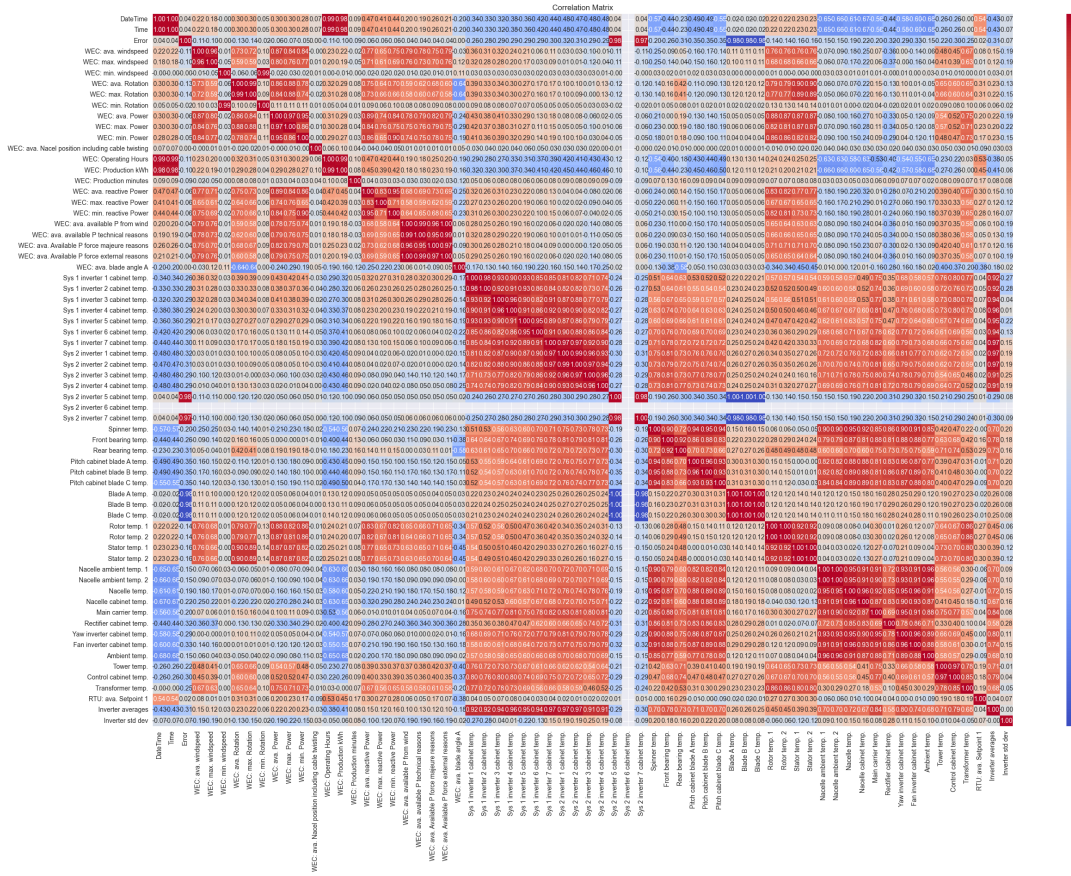
1. **Class Imbalance in Classification**:
   - If this column is used as a target for classification, the imbalance could lead to a model biased towards predicting the majority class (`Error = 0`), potentially overlooking fault states.
   - Special techniques such as **oversampling (e.g., SMOTE)**, **undersampling**, 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**, **F1-Score**, and **ROC-AUC** 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

[129]:
```
# 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()
```



[130]:
```
# 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
```

[131]:
```python
# 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_impl.py:3045: 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_impl.py:3046: 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.

[132]:
```
# List of columns to drop (e.g., ID columns or irrelevant features)
columns_to_drop = [
    'Time', 'WEC: min. windspeed',
    'WEC: max. windspeed', 'WEC: min. Rotation', 'WEC: max. Rotation', 'WEC:␣
 ↪min. Power',
    'WEC: max. 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.', 'Blade␣
 ↪B temp.', 'Blade C temp.',
    "WEC: Operating Hours",
    "Spinner temp.",
    "Front bearing temp.",
    "Rear bearing temp.",
    "Pitch cabinet blade B temp.",
```

```
    "Pitch cabinet blade C temp.",
    "Blade B temp.",
    "Blade C 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"
]

# Drop irrelevant or redundant columns
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: ava. Nacel position including cable
twisting', 'WEC: Production kWh', 'WEC: Production minutes', 'WEC: ava. 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.', 'Pitch cabinet blade A temp.', 'Blade A temp.',
'Rotor temp. 1', 'Rotor temp. 2', 'Stator temp. 1', 'Stator temp. 2', 'Nacelle
ambient temp. 1', 'Nacelle ambient temp. 2', 'RTU: ava. Setpoint 1', 'Inverter
std dev']
Number of remaining columns: 35
```

These cols are dropped based upon either they had high correlation or they have high mulitcollinearity. Later on in step Prepraring and Transforming we will add means features for cols like sys 1 and sys 2 temps

[133]:
```
# 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
```

[134]:
```
# Aggregating system temperatures
df_filtered['avg_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_sys2_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)
```

[135]:
```
# Define the condition for binary classification
df_filtered['Error_Class_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_Class_Binary'].value_counts()

# Print the distribution
print("Distribution of 'Error_Class_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_Class_Binary':
Error_Class_Binary
0    48727
1      300
Name: count, dtype: int64

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

[136]:
```
# Scale numerical features like power and windspeed, power, rotation and␣
 ↪reactiv power using MinMaxScaler
scaler = MinMaxScaler()
columns_to_scale = ['WEC: ava. windspeed', 'WEC: ava. Power', 'WEC: ava.␣
 ↪Rotation', 'WEC: ava. reactive Power']
df_filtered[columns_to_scale] = scaler.
 ↪fit_transform(df_filtered[columns_to_scale])
```

[137]:
```
# 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, 26)
Columns after feature engineering:
['WEC: ava. windspeed', 'WEC: ava. Rotation', 'WEC: ava. Power', 'WEC: ava.
Nacel position including cable twisting', 'WEC: Production kWh', 'WEC:
Production minutes', 'WEC: ava. 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', 'Pitch cabinet blade A temp.', 'Blade A temp.', 'RTU: ava.
Setpoint 1', 'Inverter std dev', 'Month', 'Week', 'DayOfWeek', 'Season',
'avg_inverter_temp', 'avg_sys2_inverter_temp', 'Rotor temp mean', 'Stator temp
mean', 'Nacelle ambient temp mean', 'Error_Class_Binary']
```

[138]:
```
# 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_Class_Binary',        # Class-based coloring
    palette='tab10',          # Use a color palette with enough distinct colors
```

```
    style='Error_Class_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

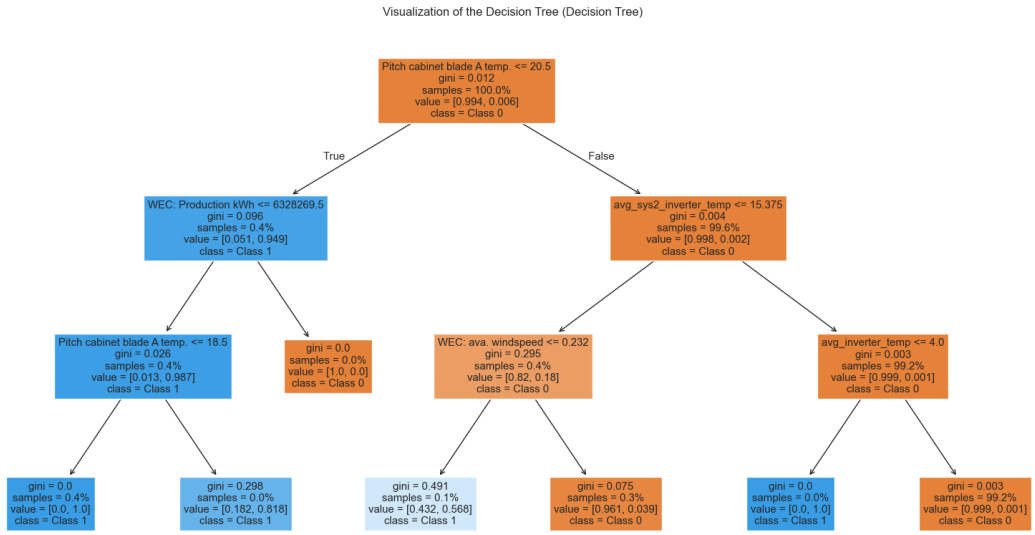
[139]:
```
# 1. Split dataset into features (X) and target (y)
X = df_filtered.drop(columns=['Error_Class_Binary'])
y = df_filtered['Error_Class_Binary']
```

[140]:
```
# 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)
```

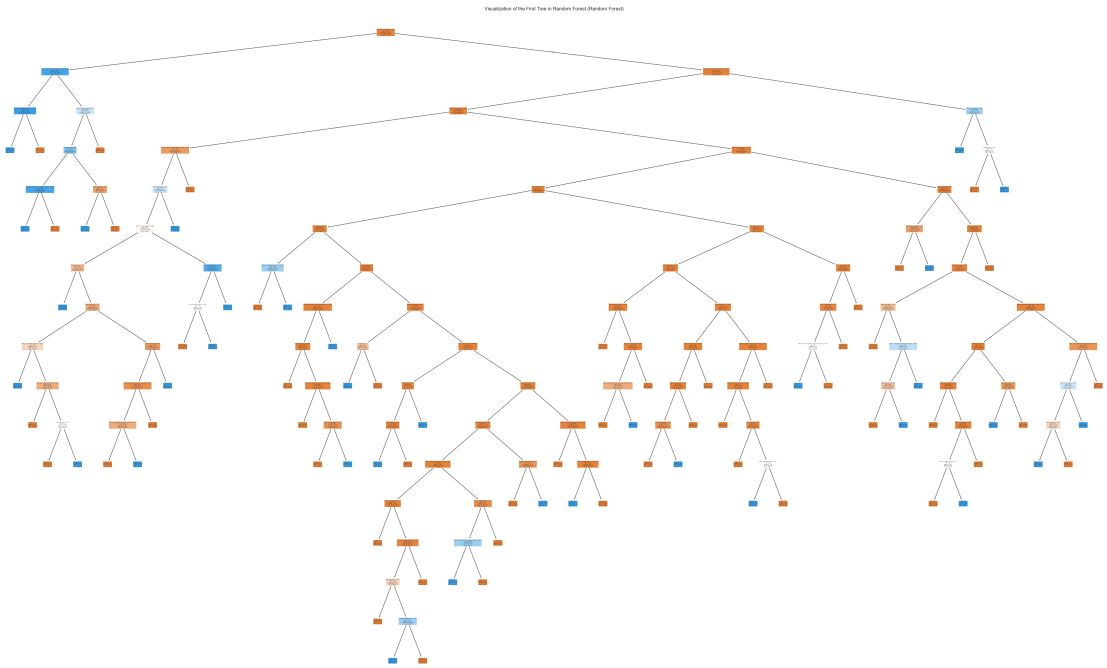
[141]:
```
# Define a dictionary of models to test
models = {
    'Decision Tree': DecisionTreeClassifier(max_depth=3, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=50, random_state=42),
    'XGBoost': XGBClassifier(random_state=42, eval_metric='mlogloss')  #␣
↪Configure XGBoost
}
```

[142]:
```
# 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…

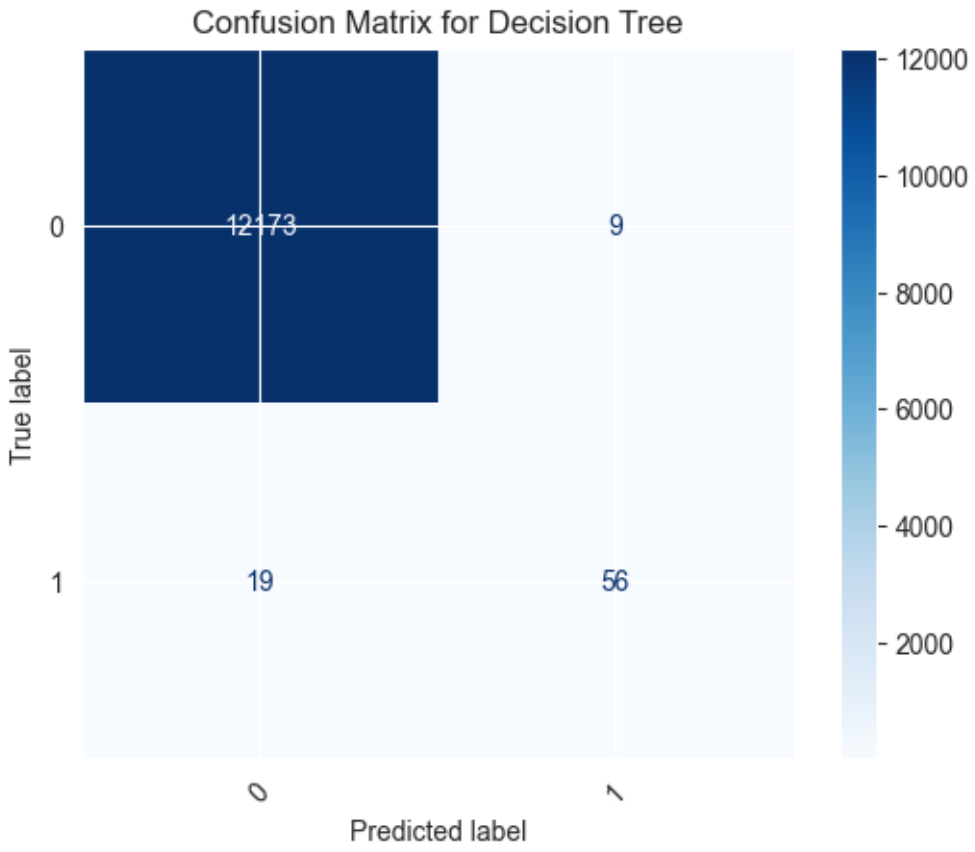
<Figure size 2000x1000 with 0 Axes>

2.5 Model Validation and Evaluation

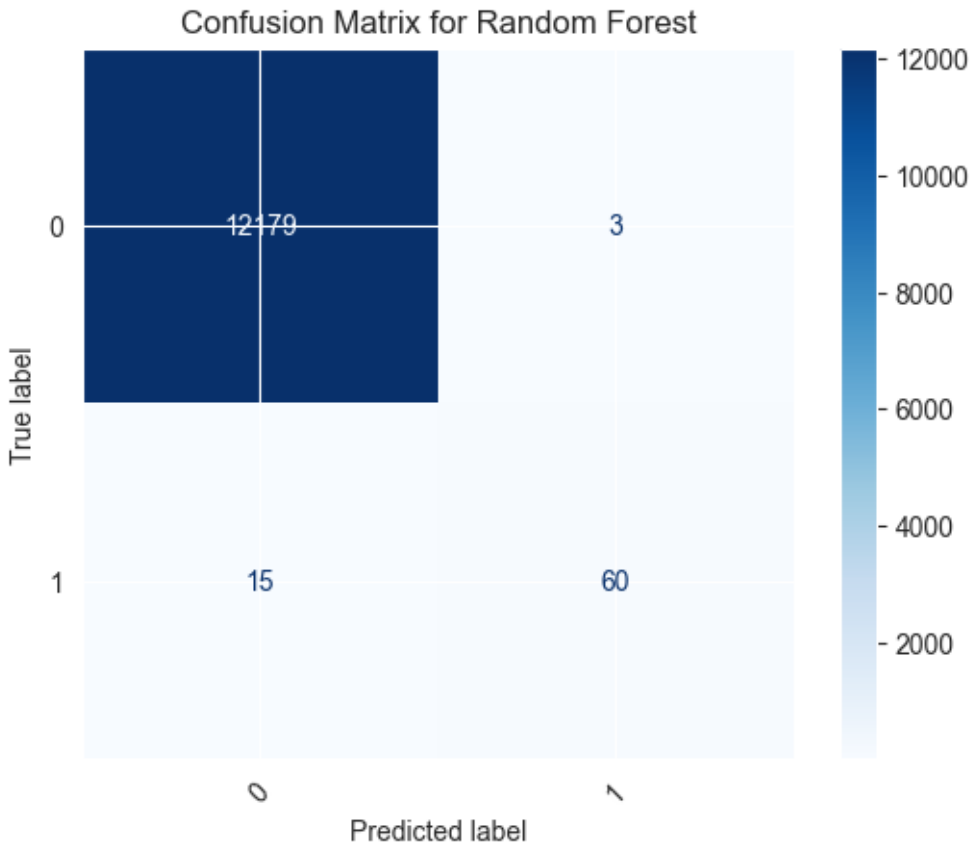
[143]:
```
# Evaluation Section
evaluation_results = {}

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}")
```

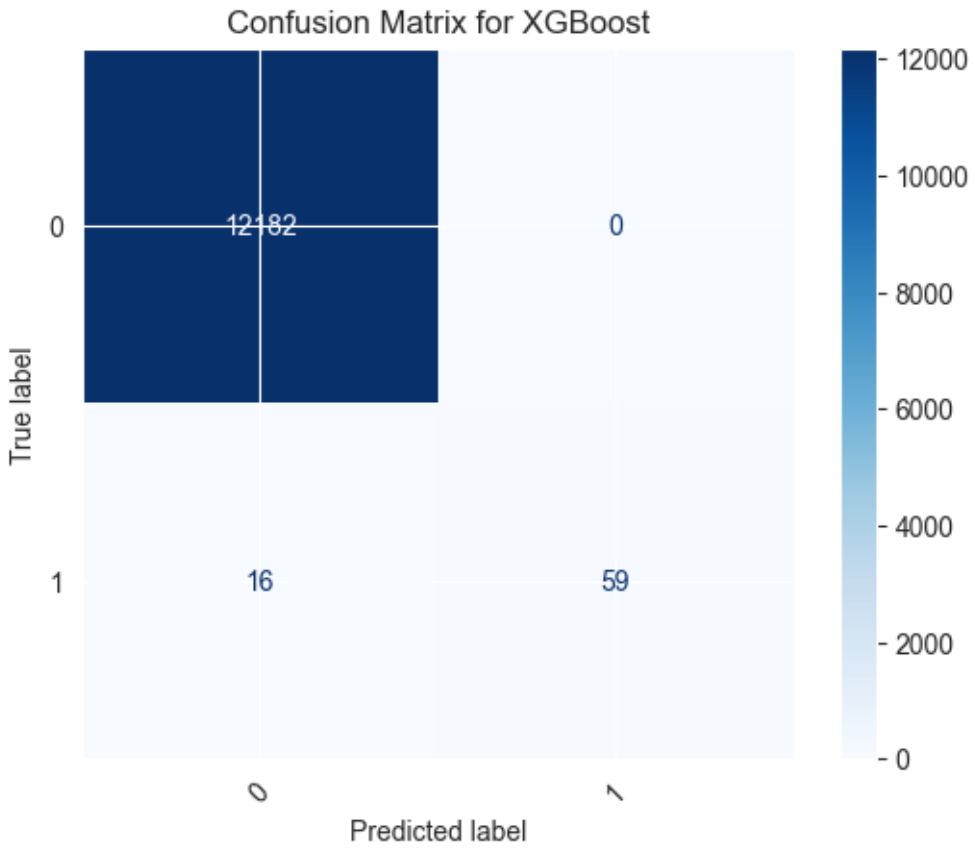
```
Evaluating Decision Tree…
Accuracy for Decision Tree: 0.9977

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.86      0.75      0.80        75

    accuracy                           1.00     12257
   macro avg       0.93      0.87      0.90     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12173     9]
 [   19    56]]
```




```
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]]
```



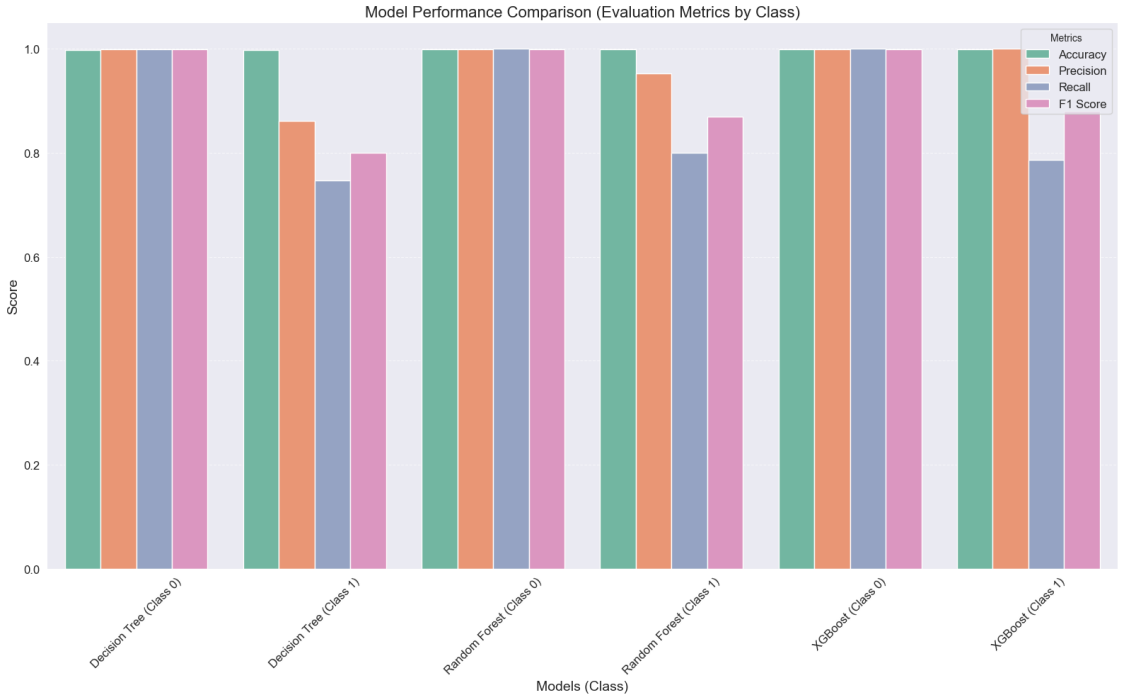

```
Evaluating XGBoost…
Accuracy for XGBoost: 0.9987

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       1.00      0.79      0.88        75

    accuracy                           1.00     12257
   macro avg       1.00      0.89      0.94     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12182     0]
 [   16    59]]
```




```
--- Evaluation Results ---
Decision Tree - Accuracy: 0.9977
Random Forest - Accuracy: 0.9985
XGBoost - Accuracy: 0.9987
```

[144]:
```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	0.997716
1	Decision Tree (Class 0)	Precision	0.998442
2	Decision Tree (Class 0)	Recall	0.999261
3	Decision Tree (Class 0)	F1 Score	0.998851
4	Decision Tree (Class 1)	Accuracy	0.997716
5	Decision Tree (Class 1)	Precision	0.861538
6	Decision Tree (Class 1)	Recall	0.746667
7	Decision Tree (Class 1)	F1 Score	0.800000
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	0.998695
17	XGBoost (Class 0)	Precision	0.998688
18	XGBoost (Class 0)	Recall	1.000000
19	XGBoost (Class 0)	F1 Score	0.999344
20	XGBoost (Class 1)	Accuracy	0.998695
21	XGBoost (Class 1)	Precision	1.000000

```
22        XGBoost (Class 1)     Recall  0.786667
23        XGBoost (Class 1)   F1 Score  0.880597
```

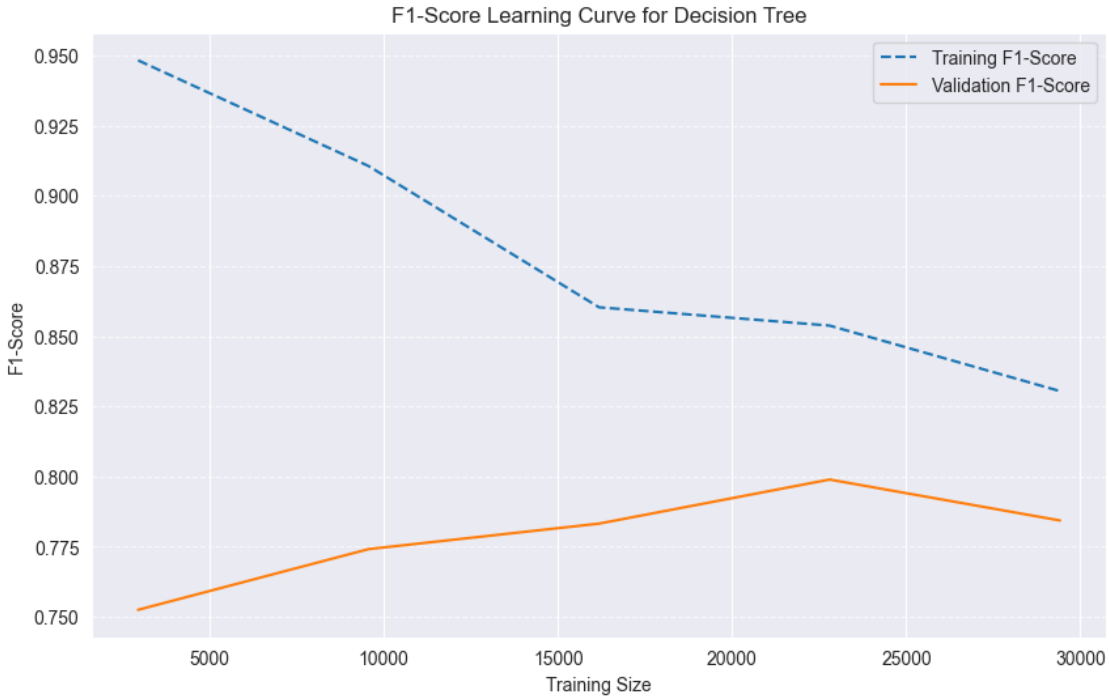
[145]:
```
# Iterate through models for evaluation
for model_name, model in models.items():
    print(f"\nEvaluating {model_name}...")

    # 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 : 0.9977

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 : 0.9987
```

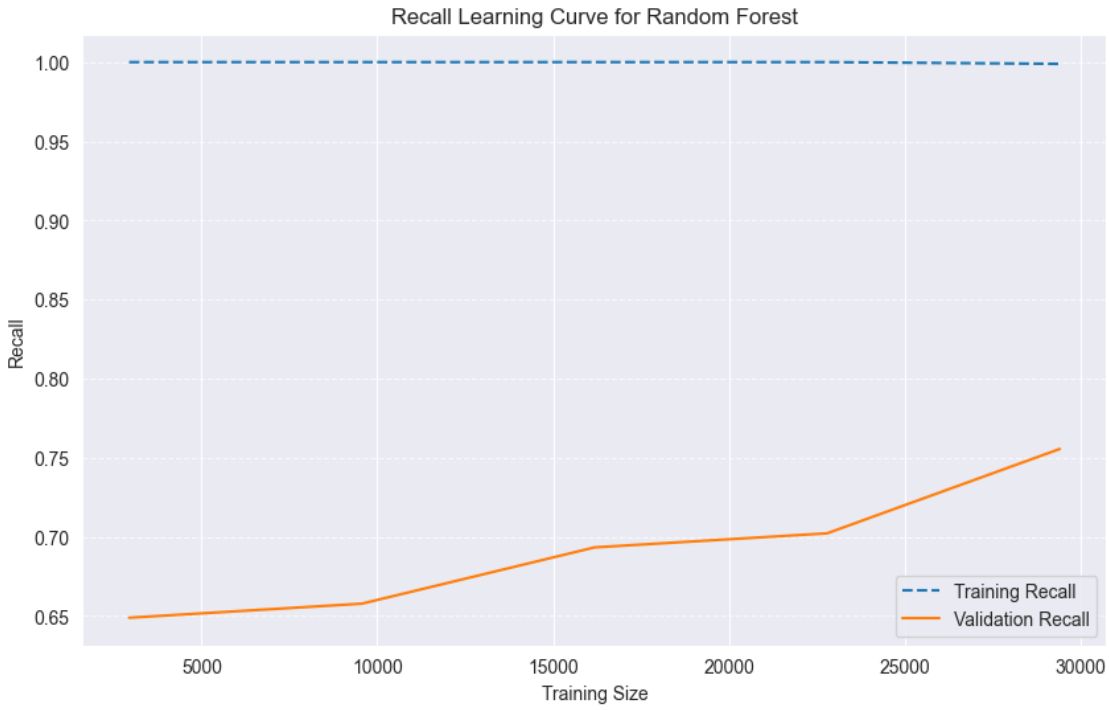
[146]:
```
# Iterate through each model to plot learning curves for Recall and F1-Score
for model_name, model in models.items():
    print(f"\nPlotting Learning Curve for {model_name}...")

    # 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)
    )

    # Generate learning curve for F1-Score
    train_sizes_f1, train_scores_f1, validation_scores_f1 = learning_curve(
```

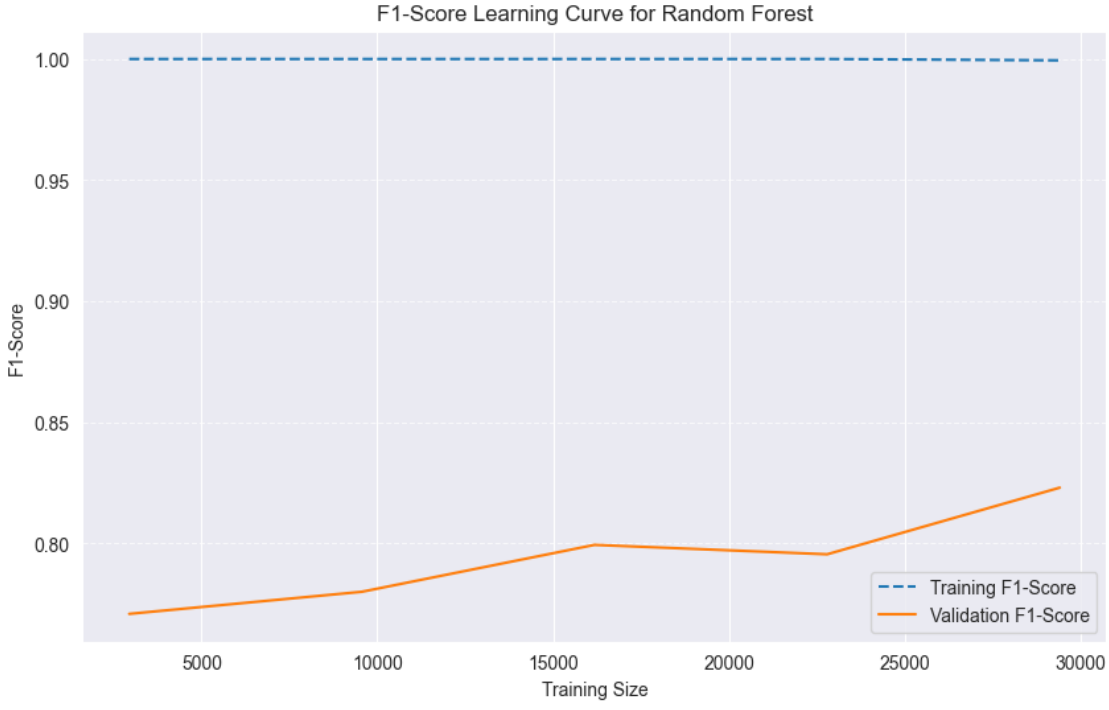
```
        model, X_train, y_train, cv=5, scoring=make_scorer(f1_score,␣
↪pos_label=1)
    )

    # 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()

    # Plot F1-Score learning curve
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes_f1, train_scores_f1.mean(axis=1), label='Training␣
↪F1-Score', linestyle='--')
    plt.plot(train_sizes_f1, validation_scores_f1.mean(axis=1),␣
↪label='Validation F1-Score')
    plt.xlabel('Training Size')
    plt.ylabel('F1-Score')
    plt.title(f'F1-Score Learning Curve for {model_name}')
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()
```

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

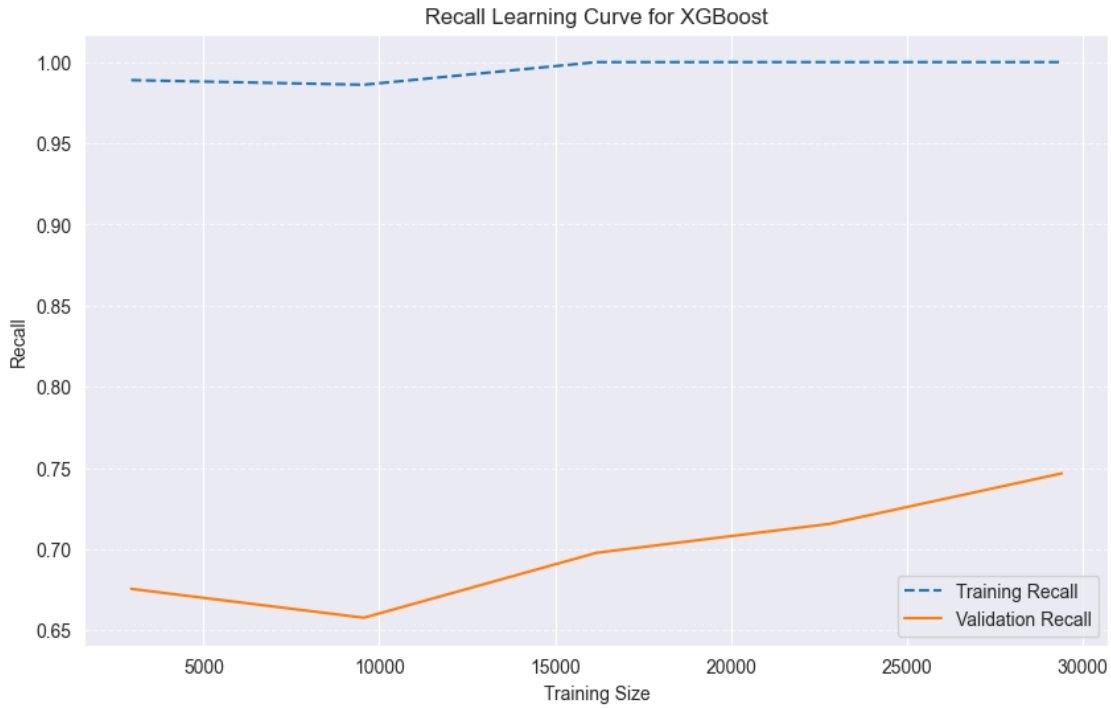


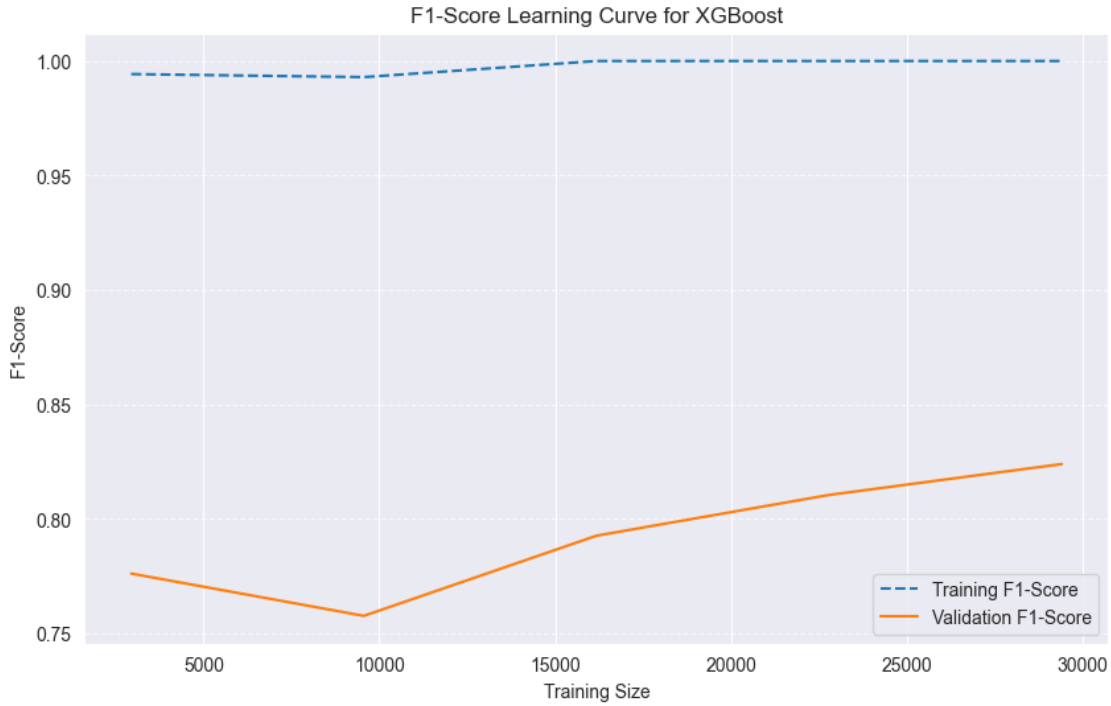

```
Plotting Learning Curve for Random Forest…
```


Recall Learning Curve for Random Forest
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Recall
5000
10000
15000
20000
25000
30000
Training Size
Training Recall
Validation Recall


F1-Score Learning Curve for Random Forest
1.00
0.95
0.90
0.85
0.80
F1-Score
5000
10000
15000
20000
25000
30000
Training Size
Training F1-Score
Validation F1-Score

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





[147]:
```
# 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='accuracy')

    # 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: [0.99796029 0.99809627
0.99782431 0.99700843 0.99782431]
Mean Stratified Cross-Validation Score for Decision Tree: 0.9977

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.99809627 0.99796029
0.99823225 0.99700843 0.99796029]
Mean Stratified Cross-Validation Score for Random Forest: 0.9979

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.99823225 0.99768833
0.99823225 0.99687245 0.99796029]
Mean Stratified Cross-Validation Score for XGBoost: 0.9978
```

2.6 Hyperparameter Tuning

[148]:
```
# 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],
```

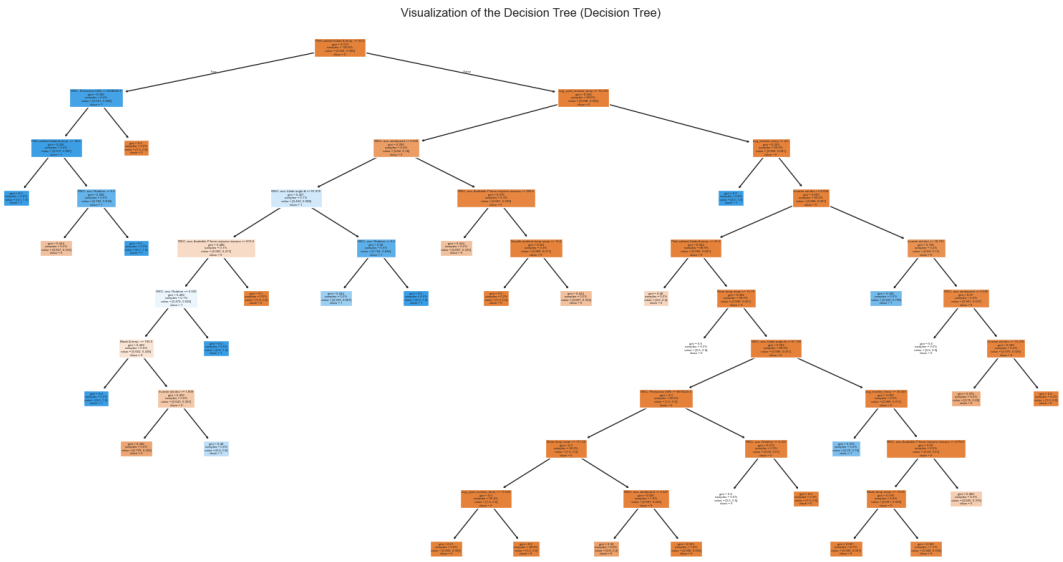
```
        'min_samples_leaf': [1, 2, 4],
        'bootstrap': [True, False]
    },
    'XGBoost': {
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [3, 6, 9, 12],
        'learning_rate': [0.01, 0.1, 0.3],
        'subsample': [0.7, 1.0],
        'colsample_bytree': [0.7, 1.0],
 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

# 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='accuracy', 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
```

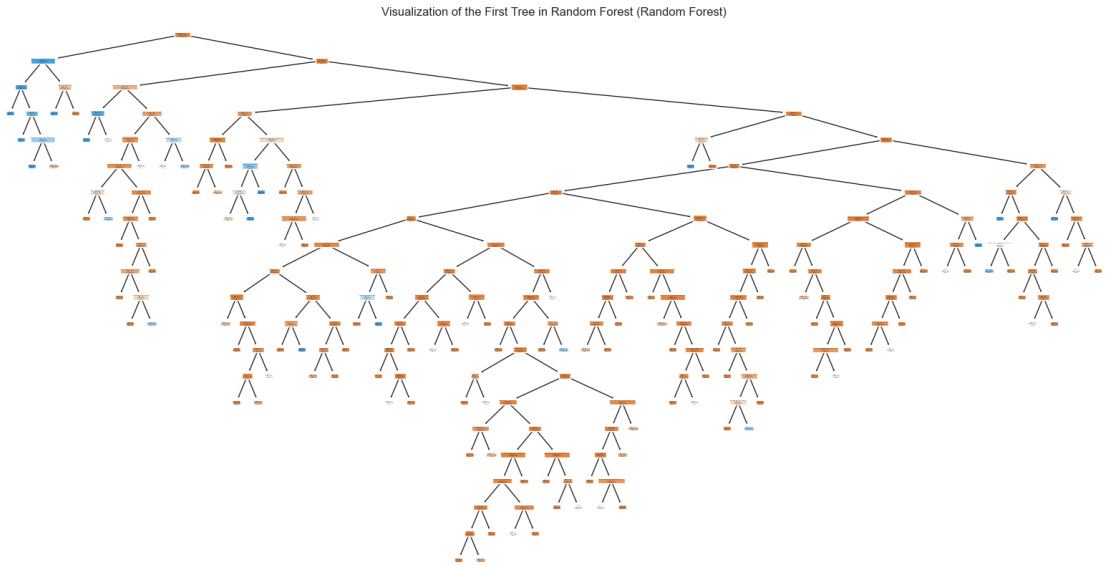
```
Performing GridSearchCV for Decision Tree…
Fitting 5 folds for each of 45 candidates, totalling 225 fits
Best parameters for Decision Tree: {'max_depth': 10, 'min_samples_leaf': 2,
'min_samples_split': 10}
Best score for Decision Tree: 0.9978243132988849
```

```
Performing GridSearchCV for Random Forest…
Fitting 5 folds for each of 216 candidates, totalling 1080 fits

D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\numpy\ma\core.py:2891: RuntimeWarning: invalid value encountered in
cast
  _data = np.array(data, dtype=dtype, copy=copy,

Best parameters for Random Forest: {'bootstrap': False, 'max_depth': 30,
'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 150}
Best score for Random Forest: 0.99820505847158

Performing GridSearchCV for XGBoost…
Fitting 5 folds for each of 192 candidates, totalling 960 fits

D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\numpy\ma\core.py:2891: RuntimeWarning: invalid value encountered in
cast
  _data = np.array(data, dtype=dtype, copy=copy,

Best parameters for XGBoost: {'colsample_bytree': 0.7, 'learning_rate': 0.3,
'max_depth': 6, 'n_estimators': 150, 'subsample': 1.0}
Best score for XGBoost: 0.9983954310579277
```

[149]:
```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)
```





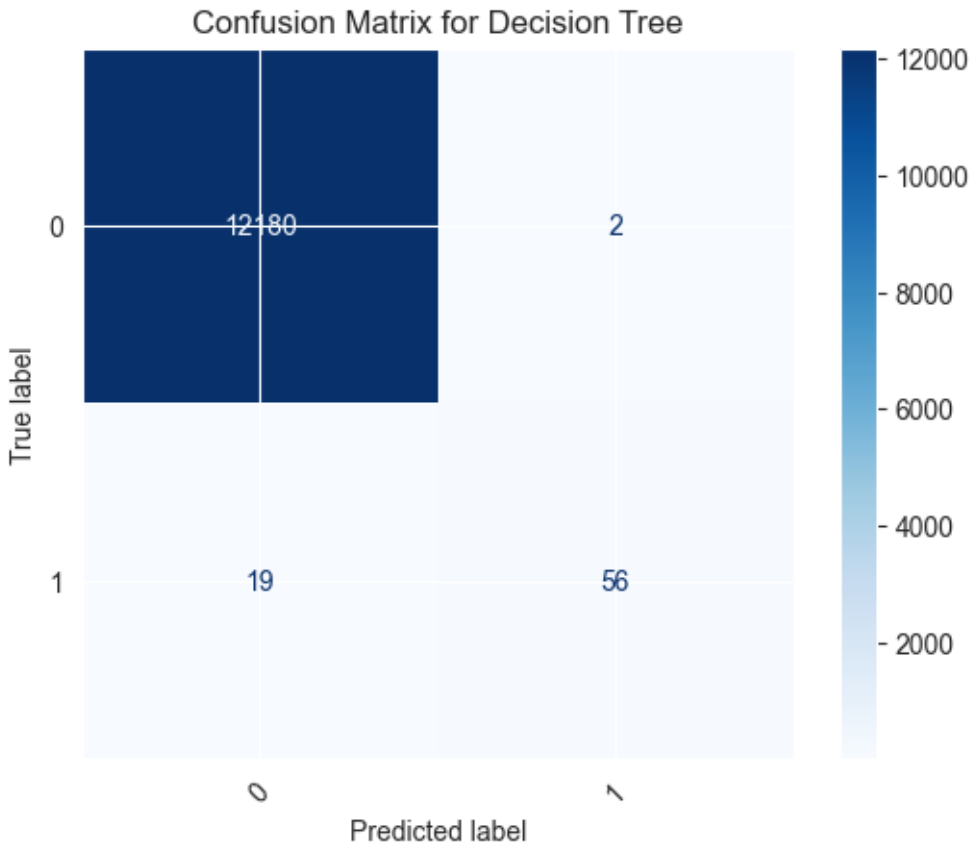
<Figure size 2000x1000 with 0 Axes>

2.7 Evaluating and Validating after Tuning

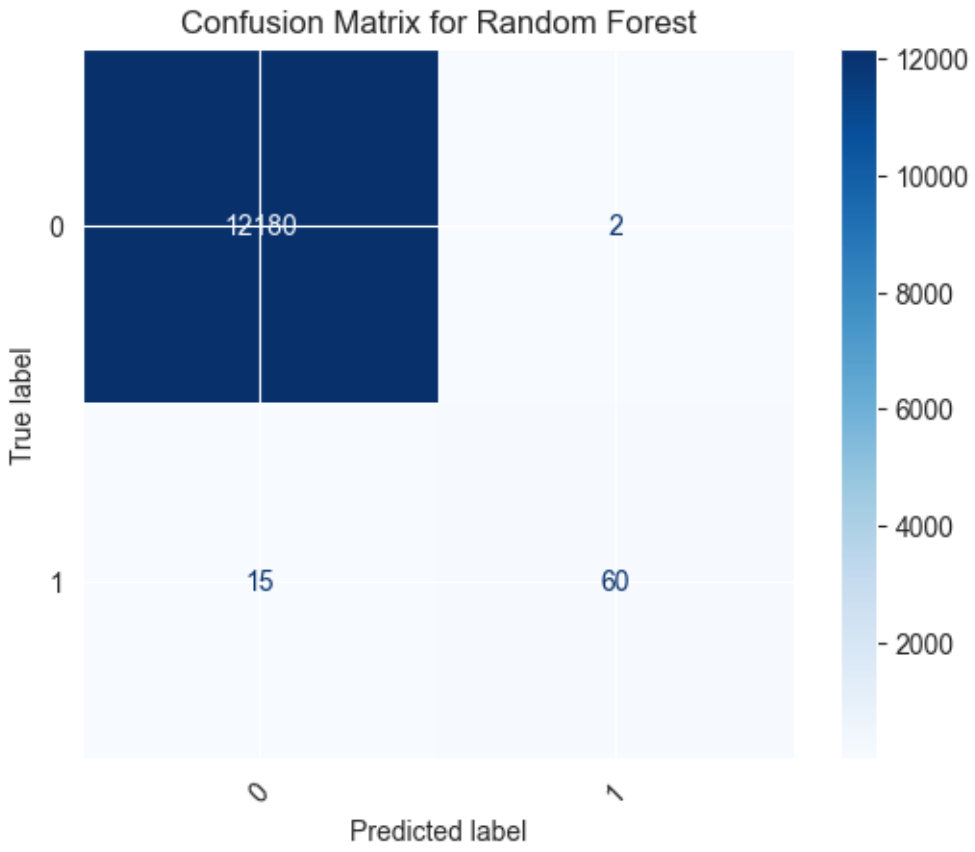
[150]:
```
# Function to evaluate each model and plot classification report and confusion␣
↪matrix
def evaluate_models(best_models, X_test, y_test, model_names):
    for model_name in model_names:
        model = best_models[model_name]

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

        # Print Classification Report
        print(f"\nClassification Report for {model_name}:")
```

```
        print(classification_report(y_test, y_pred))

        # Plot Confusion Matrix
        cm = confusion_matrix(y_test, y_pred)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
 ↪classes_)

        # Plot the confusion matrix
        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
model_names = ['Decision Tree', 'Random Forest', 'XGBoost']

# Call the function to generate reports and confusion matrices
evaluate_models(best_models, X_test, y_test, model_names)
```

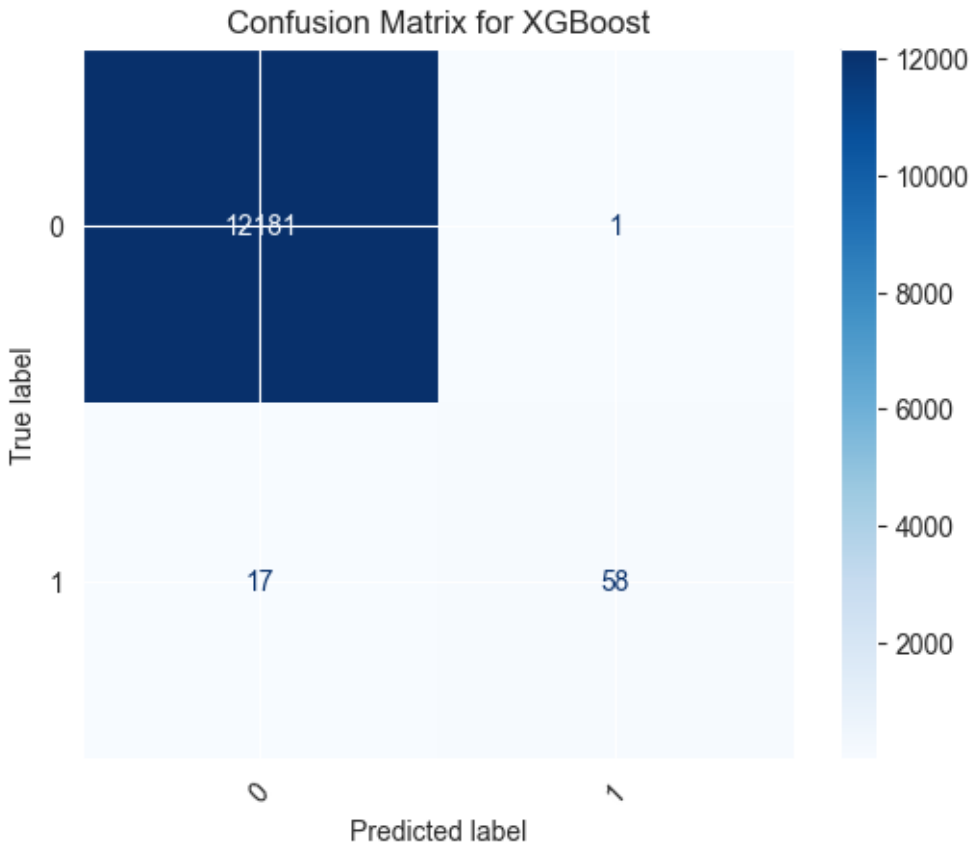
```
Classification Report for Decision Tree:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.97      0.75      0.84        75

    accuracy                           1.00     12257
   macro avg       0.98      0.87      0.92     12257
weighted avg       1.00      1.00      1.00     12257
```




```
Classification Report for Random Forest:
              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
```

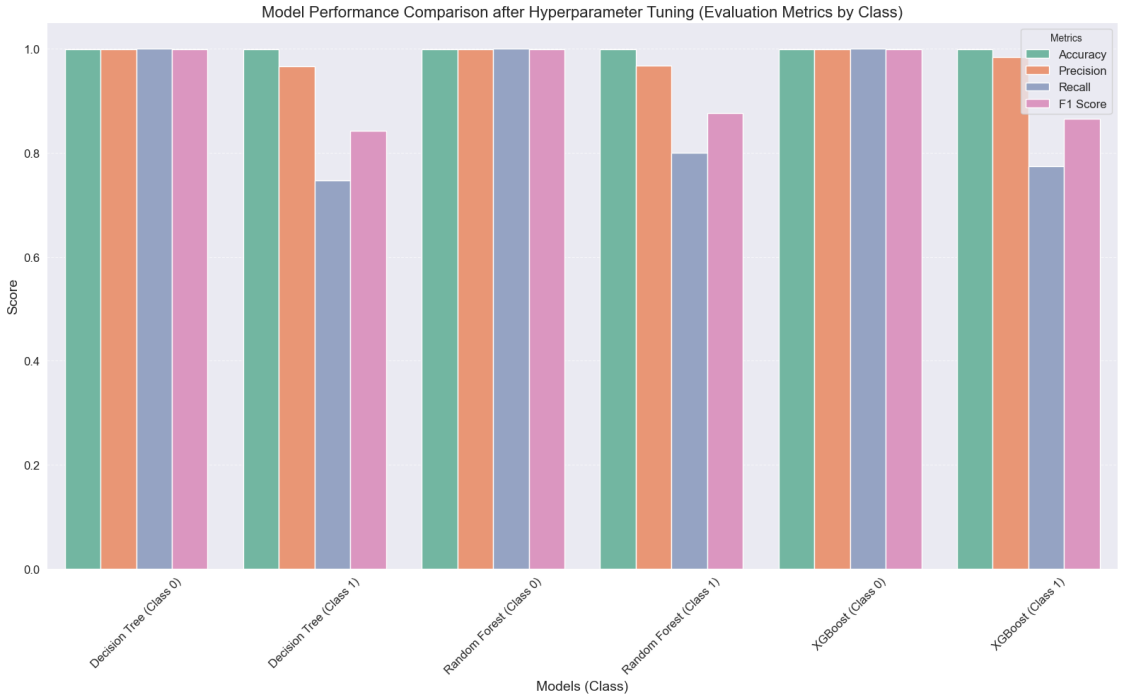



```
Classification Report for XGBoost:
              precision    recall  f1-score   support

           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
```



[151]:
```
# 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	0.998287
1	Decision Tree (Class 0)	Precision	0.998442
2	Decision Tree (Class 0)	Recall	0.999836
3	Decision Tree (Class 0)	F1 Score	0.999139
4	Decision Tree (Class 1)	Accuracy	0.998287
5	Decision Tree (Class 1)	Precision	0.965517
6	Decision Tree (Class 1)	Recall	0.746667
7	Decision Tree (Class 1)	F1 Score	0.842105
8	Random Forest (Class 0)	Accuracy	0.998613
9	Random Forest (Class 0)	Precision	0.998770
10	Random Forest (Class 0)	Recall	0.999836
11	Random Forest (Class 0)	F1 Score	0.999303
12	Random Forest (Class 1)	Accuracy	0.998613
13	Random Forest (Class 1)	Precision	0.967742
14	Random Forest (Class 1)	Recall	0.800000
15	Random Forest (Class 1)	F1 Score	0.875912
16	XGBoost (Class 0)	Accuracy	0.998531
17	XGBoost (Class 0)	Precision	0.998606
18	XGBoost (Class 0)	Recall	0.999918
19	XGBoost (Class 0)	F1 Score	0.999262
20	XGBoost (Class 1)	Accuracy	0.998531
21	XGBoost (Class 1)	Precision	0.983051

```
22        XGBoost (Class 1)     Recall  0.773333
23        XGBoost (Class 1)   F1 Score  0.865672
```

[152]:
```python
# Iterate through each best model for evaluation
for model_name, best_model in best_models.items():
    print(f"\nEvaluating {model_name}...")

    # Training performance
    y_train_pred = best_model.predict(X_train)
    train_accuracy = accuracy_score(y_train, y_train_pred)

    # Test performance
    y_test_pred = best_model.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    print(f"Training Accuracy for {model_name} after Hyperparameter Tuning:␣
↪{train_accuracy:.4f}")
    print(f"Test Accuracy for {model_name} after Hyperparameter Tuning:␣
↪{test_accuracy:.4f}")
```

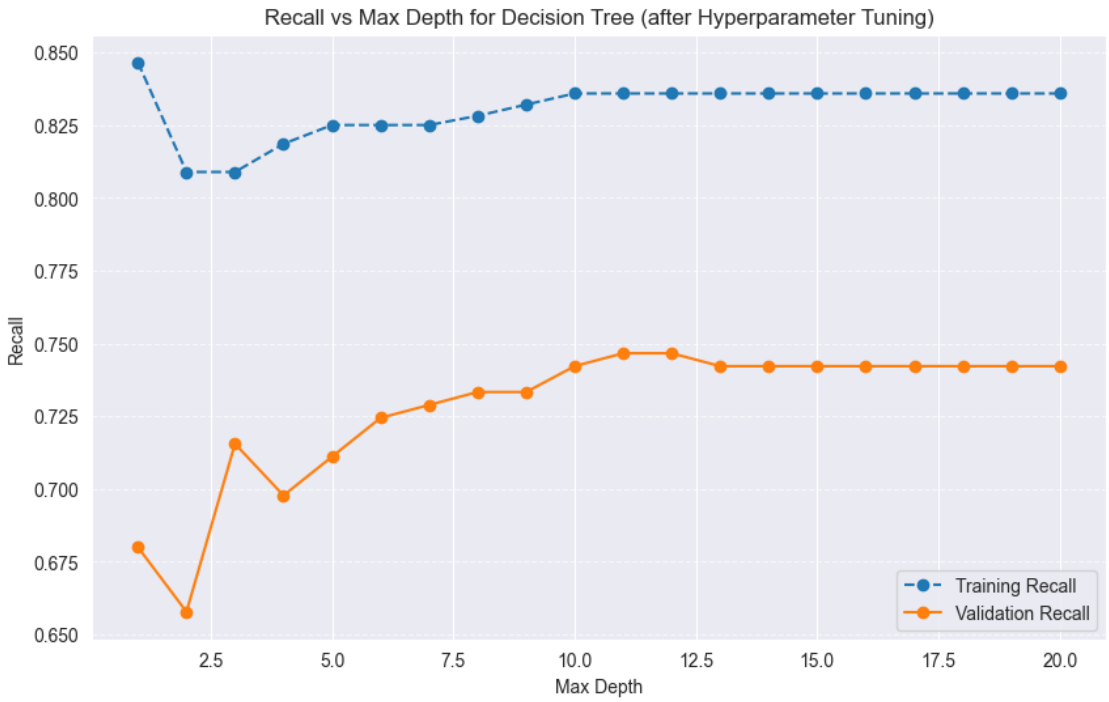
```
Evaluating Decision Tree…
Training Accuracy for Decision Tree after Hyperparameter Tuning: 0.9986
Test Accuracy for Decision Tree after Hyperparameter Tuning: 0.9983

Evaluating Random Forest…
Training Accuracy for Random Forest after Hyperparameter Tuning: 0.9996
Test Accuracy for Random Forest after Hyperparameter Tuning: 0.9986

Evaluating XGBoost…
Training Accuracy for XGBoost after Hyperparameter Tuning: 1.0000
Test Accuracy for XGBoost after Hyperparameter Tuning: 0.9985
```

[153]:
```python
# Import necessary libraries
from sklearn.metrics import make_scorer, recall_score, f1_score
import numpy as np

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

            # Define a range of max_depth values
            max_depth_range = range(1, 21)
```

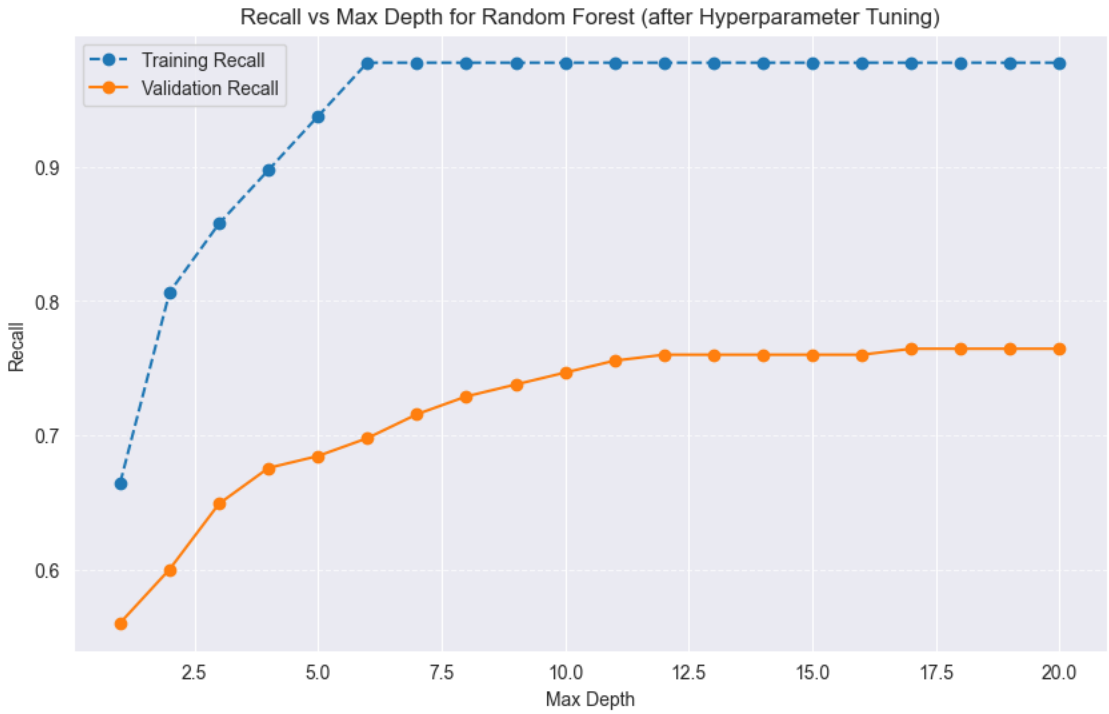
```
            # Initialize lists to store recall scores
            training_recalls = []
            validation_recalls = []

            for max_depth in max_depth_range:
                # Update max_depth for the model
                model = best_model
                model.set_params(max_depth=max_depth)

                # Compute learning curve
                train_sizes, train_scores, validation_scores = learning_curve(
                    model, X_train, y_train, cv=5,␣
 ↪scoring=make_scorer(recall_score, pos_label=1)
                )

                # Store mean recall scores
                training_recalls.append(train_scores.mean(axis=1).max())
                validation_recalls.append(validation_scores.mean(axis=1).max())

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


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

            # Define a range of n_estimators
            n_estimators_range = [50, 100, 150, 200, 250]

            # Initialize lists to store F1-scores
            training_f1_scores = []
            validation_f1_scores = []
```

```
            for n_estimators in n_estimators_range:
                # Update n_estimators for the model
                model = best_model
                model.set_params(n_estimators=n_estimators)

                # Compute learning curve
                train_sizes, train_scores, validation_scores = learning_curve(
                    model, X_train, y_train, cv=5,␣
↪scoring=make_scorer(f1_score, pos_label=1)
                )

                # Store mean F1 scores
                training_f1_scores.append(train_scores.mean(axis=1).max())
                validation_f1_scores.append(validation_scores.mean(axis=1).
↪max())

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


# Call the functions
plot_max_depth_vs_recall(best_models, X_train, y_train)
plot_n_estimators_vs_f1(best_models, X_train, y_train)
```

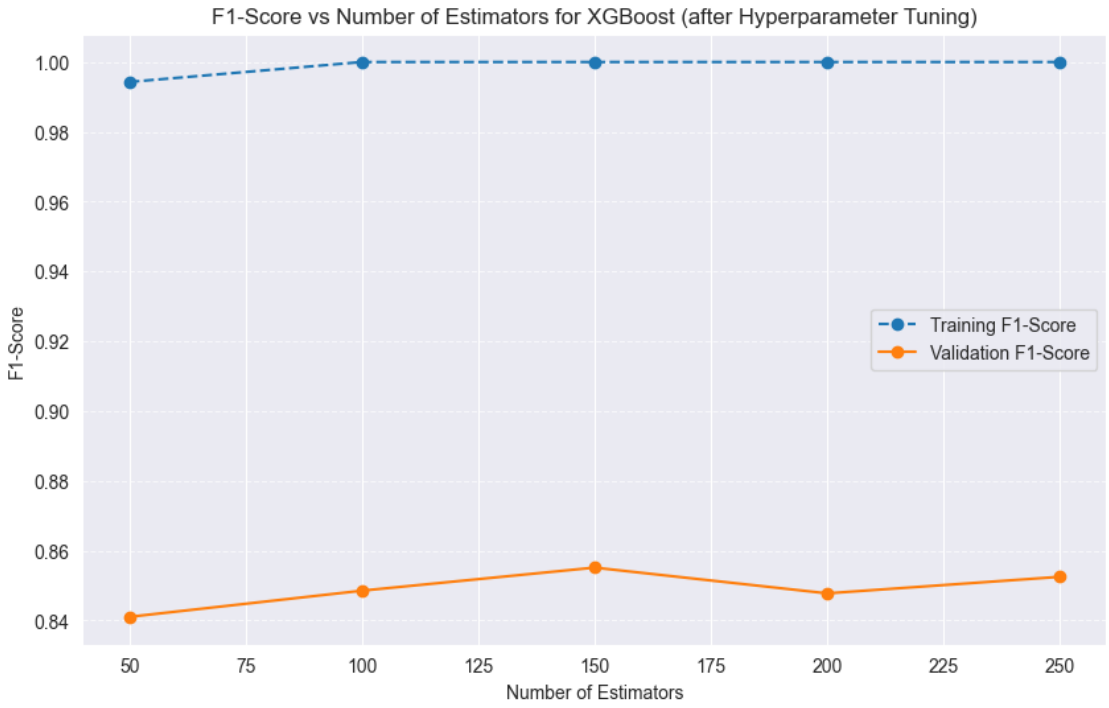
```
Generating max_depth vs Recall Curve for Decision Tree…
```




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



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



[154]:
```
# 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='accuracy')

    # 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: [0.99836823 0.99768833
```

```
0.99782431 0.99660049 0.99714441]
Mean Stratified Cross-Validation Score for Decision Tree: 0.9975

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.99850422 0.99809627
0.99823225 0.99700843 0.99809627]
Mean Stratified Cross-Validation Score for Random Forest: 0.9980

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.99891216 0.99741637
0.99809627 0.99700843 0.99768833]
Mean Stratified Cross-Validation Score for XGBoost: 0.9978
```

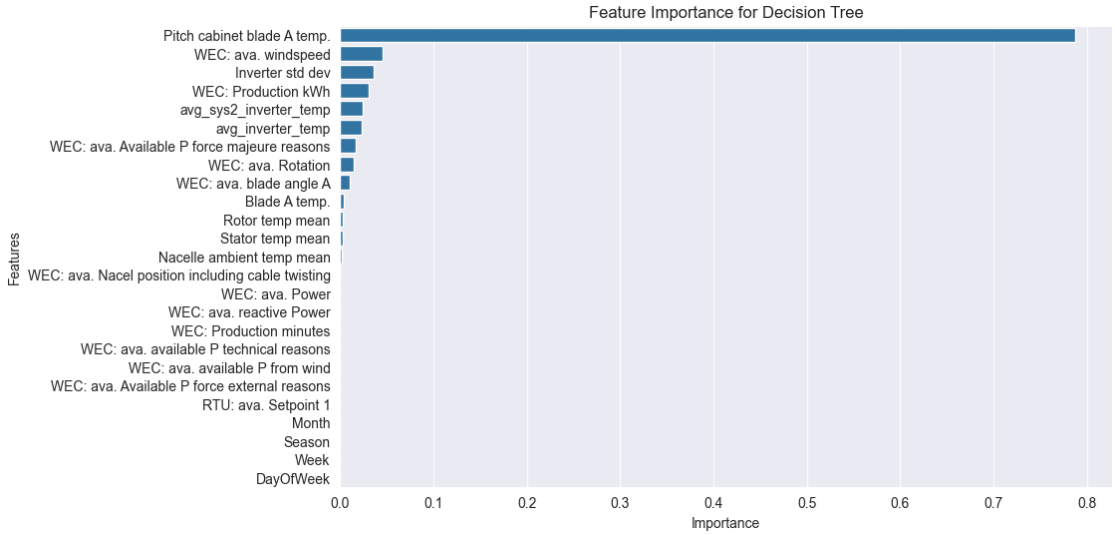
2.8 Feature Importance

[155]:
```python
# 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
12                          Pitch cabinet blade A temp.    0.787735
0                                  WEC: ava. windspeed    0.045472
15                                    Inverter std dev    0.035643
4                                  WEC: Production kWh    0.031166
21                               avg_sys2_inverter_temp    0.024443
20                                    avg_inverter_temp    0.023141
9          WEC: ava. Available P force majeure reasons    0.016711
1                                    WEC: ava. Rotation    0.014492
11                               WEC: ava. blade angle A    0.010131
13                                         Blade A temp.    0.004003
22                                    Rotor temp mean    0.002786
23                                    Stator temp mean    0.002491
24                             Nacelle ambient temp mean    0.001786
3    WEC: ava. Nacel position including cable twisting    0.000000
2                                       WEC: ava. Power    0.000000
6                                WEC: ava. reactive Power    0.000000
5                                 WEC: Production minutes    0.000000
8                WEC: ava. available P technical reasons    0.000000
```

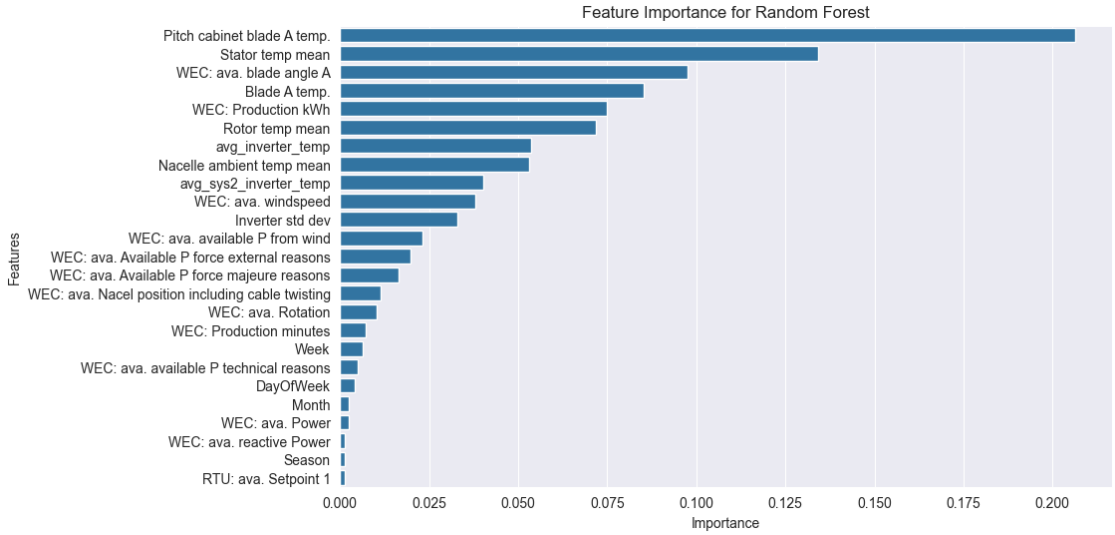
7	WEC: ava. available P from wind	0.000000
10	WEC: ava. Available P force external reasons	0.000000
14	RTU: ava. Setpoint 1	0.000000
16	Month	0.000000
19	Season	0.000000
17	Week	0.000000
18	DayOfWeek	0.000000



Feature Importance for Random Forest:

	Feature	Importance
12	Pitch cabinet blade A temp.	0.206451
23	Stator temp mean	0.134127
11	WEC: ava. blade angle A	0.097561
13	Blade A temp.	0.085140
4	WEC: Production kWh	0.074924
22	Rotor temp mean	0.071870
20	avg_inverter_temp	0.053516
24	Nacelle ambient temp mean	0.053193
21	avg_sys2_inverter_temp	0.040257
0	WEC: ava. windspeed	0.037835
15	Inverter std dev	0.032825
7	WEC: ava. available P from wind	0.023209
10	WEC: ava. Available P force external reasons	0.019754
9	WEC: ava. Available P force majeure reasons	0.016332
3	WEC: ava. Nacel position including cable twisting	0.011325
1	WEC: ava. Rotation	0.010166
5	WEC: Production minutes	0.007156
17	Week	0.006473
8	WEC: ava. available P technical reasons	0.004869

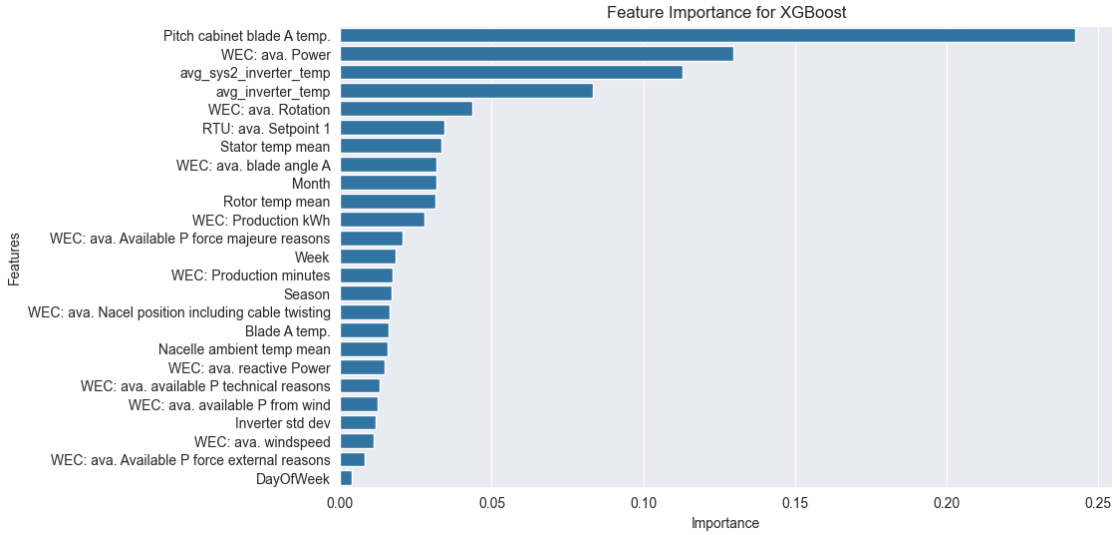
18	DayOfWeek	0.004225
16	Month	0.002556
2	WEC: ava. Power	0.002394
6	WEC: ava. reactive Power	0.001357
19	Season	0.001276
14	RTU: ava. Setpoint 1	0.001209



Feature Importance for XGBoost:

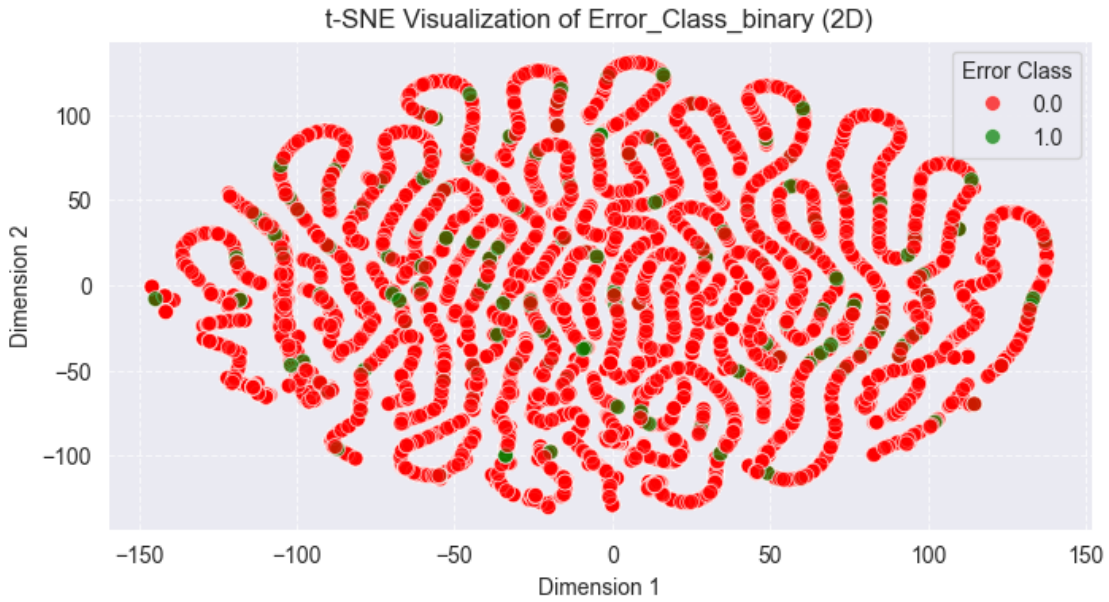
	Feature	Importance
12	Pitch cabinet blade A temp.	0.242544
2	WEC: ava. Power	0.129787
21	avg_sys2_inverter_temp	0.112977
20	avg_inverter_temp	0.083524
1	WEC: ava. Rotation	0.043774
14	RTU: ava. Setpoint 1	0.034395
23	Stator temp mean	0.033357
11	WEC: ava. blade angle A	0.031895
16	Month	0.031741
22	Rotor temp mean	0.031554
4	WEC: Production kWh	0.027975
9	WEC: ava. Available P force majeure reasons	0.020731
17	Week	0.018475
5	WEC: Production minutes	0.017358
19	Season	0.016949
3	WEC: ava. Nacel position including cable twisting	0.016346
13	Blade A temp.	0.015928
24	Nacelle ambient temp mean	0.015721
6	WEC: ava. reactive Power	0.014778
8	WEC: ava. available P technical reasons	0.013010

```
7                    WEC: ava. available P from wind    0.012532
15                                  Inverter std dev    0.011656
0                                 WEC: ava. windspeed    0.010988
10       WEC: ava. Available P force external reasons    0.008038
18                                          DayOfWeek    0.003968
```



[156]:
```python
# 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_Class_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_Class_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

[157]:
```
# Show class distribution before applying resampling
print("Class distribution before resampling:")
print(Counter(y_train))
```

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

[158]:
```
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
```

[159]:
```
# ---  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: 35384, 1: 34418})
```

[160]:
```
# Scatter plot with different colors for each Error_Class_Binary after␣
 ↪resampling
plt.figure(figsize=(8, 6))

sns.scatterplot(
    data=pd.DataFrame(X_resampled_smote_enn, columns=X_train.columns).
 ↪assign(Error_Class_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_Class_Binary',         # Class-based coloring
    palette='tab10',          # Use a color palette with enough distinct colors
    style='Error_Class_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()
```

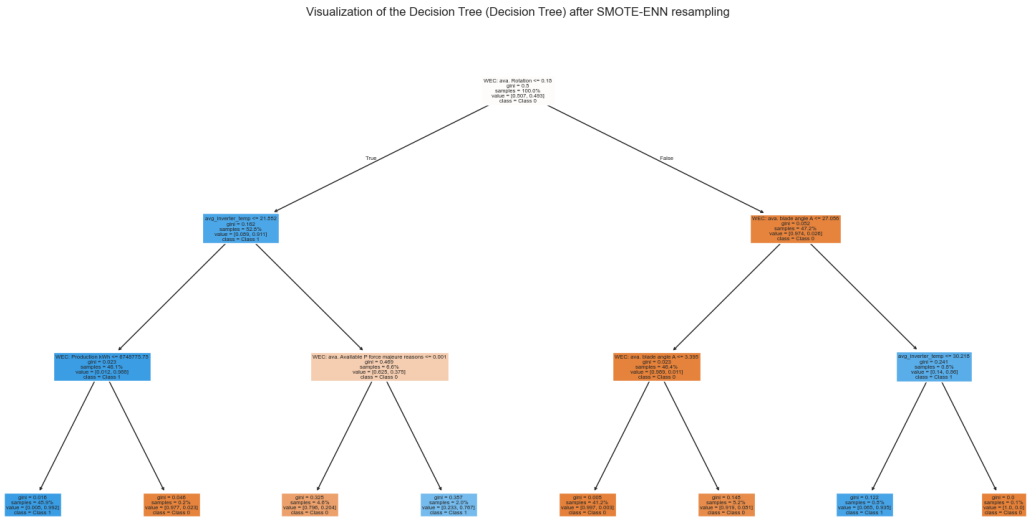
3.1 Model Development and Training with Resampled Data

[161]:
```
# 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
}
```

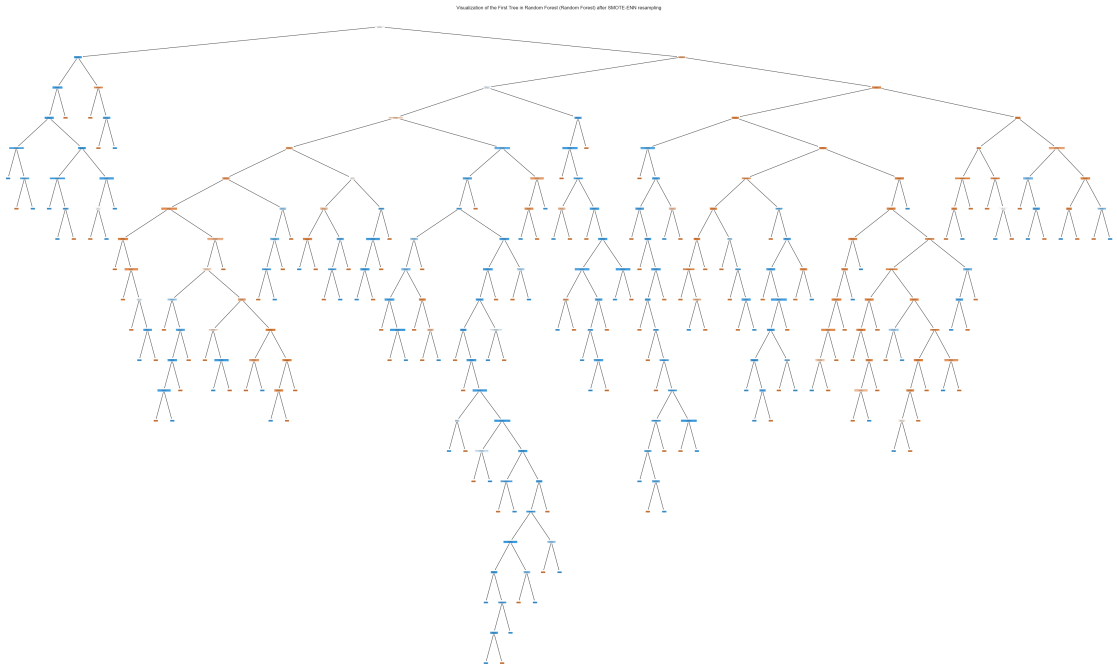
[162]:
```
# --- Training Section (Fit) with rsmpl_models ---
for model_name, model in rsmpl_models.items():
    print(f"\nTraining {model_name}...")

    # 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…
```



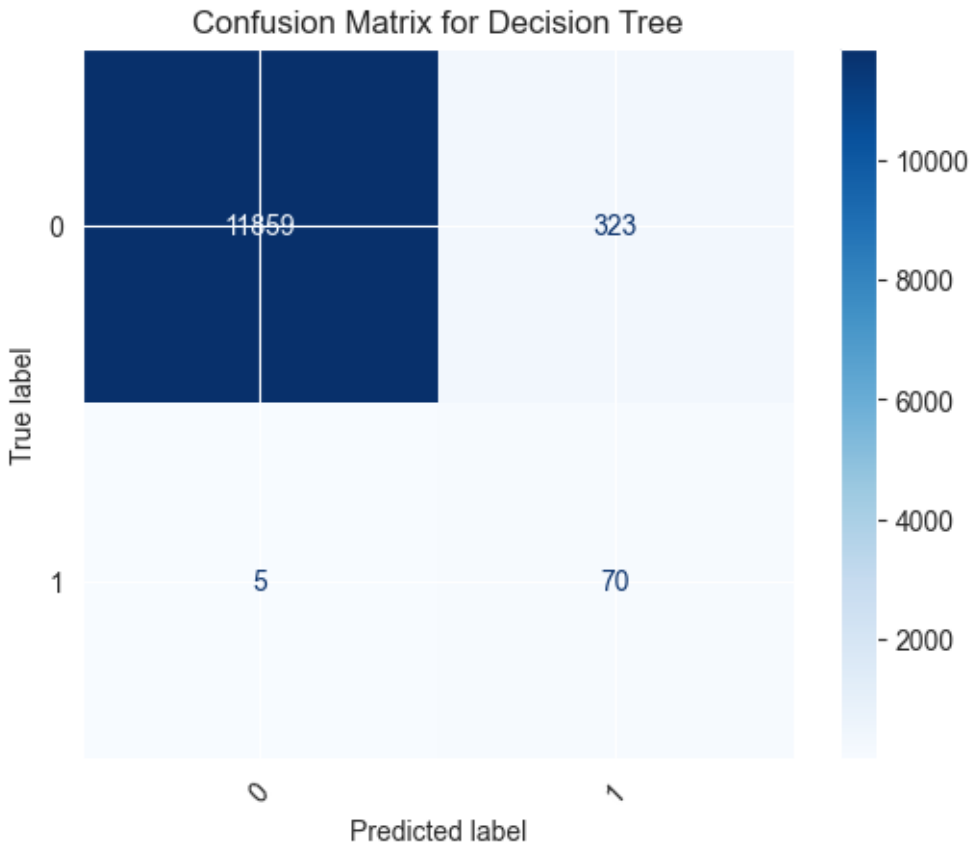
```
Training XGBoost…
<Figure size 2000x2000 with 0 Axes>
```

3.2 Evaluation and Validation after resampling

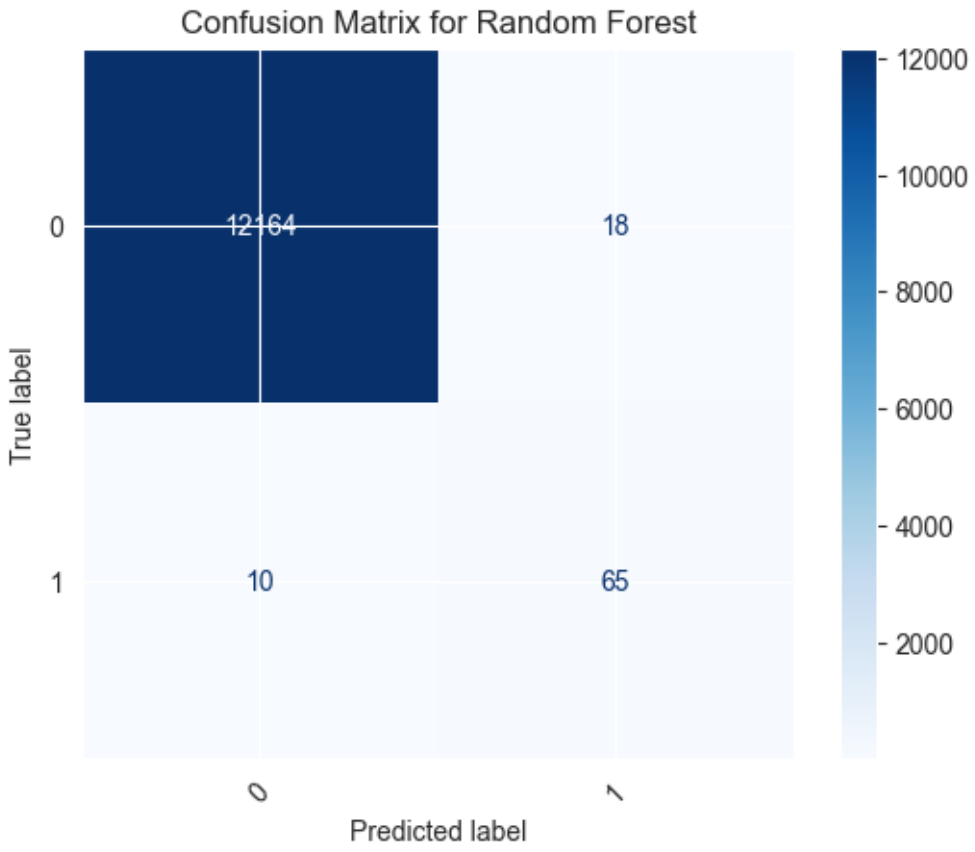
[163]:
```python
# Evaluate each trained model (after resampling)
for model_name, model in rsmpl_models.items():
    print(f"\nEvaluating {model_name}...")

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

```
    # Compute accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"{model_name} - Test Accuracy: {accuracy:.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()
```

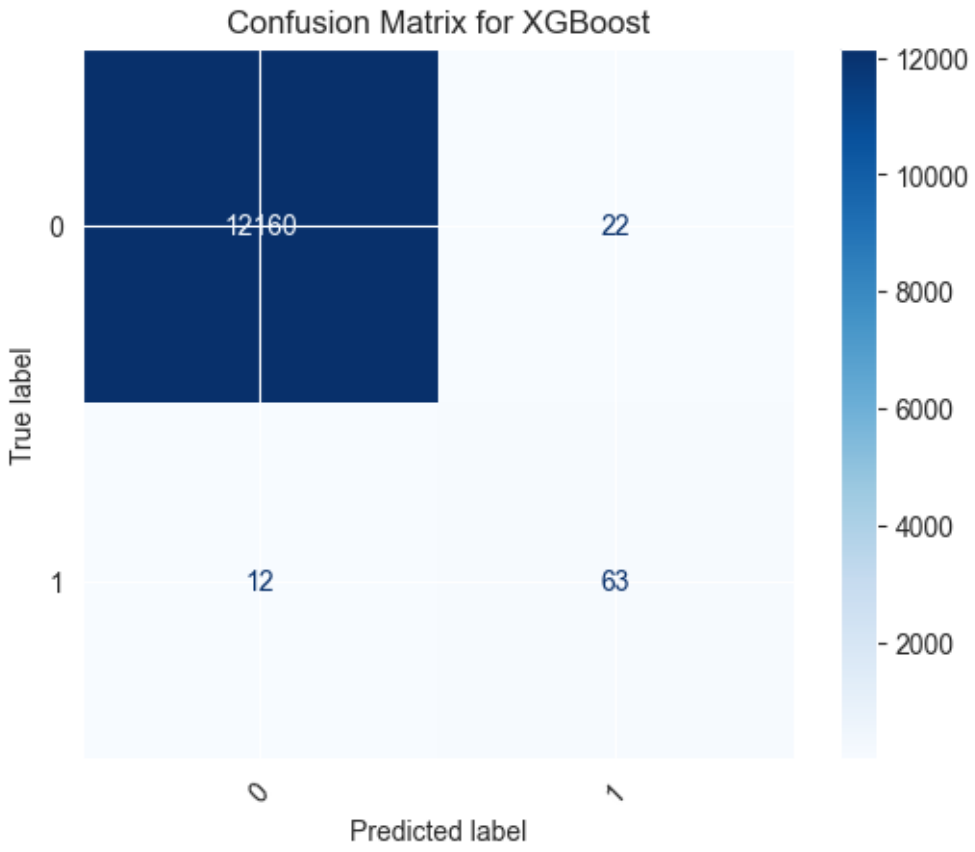
```
Evaluating Decision Tree…
Decision Tree - Test Accuracy: 0.9732

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     12182
           1       0.18      0.93      0.30        75

    accuracy                           0.97     12257
   macro avg       0.59      0.95      0.64     12257
weighted avg       0.99      0.97      0.98     12257


Confusion Matrix:
[[11859   323]
 [    5    70]]
```




```
Evaluating Random Forest…
Random Forest - Test Accuracy: 0.9977

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.78      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


Confusion Matrix:
[[12164    18]
 [   10    65]]
```

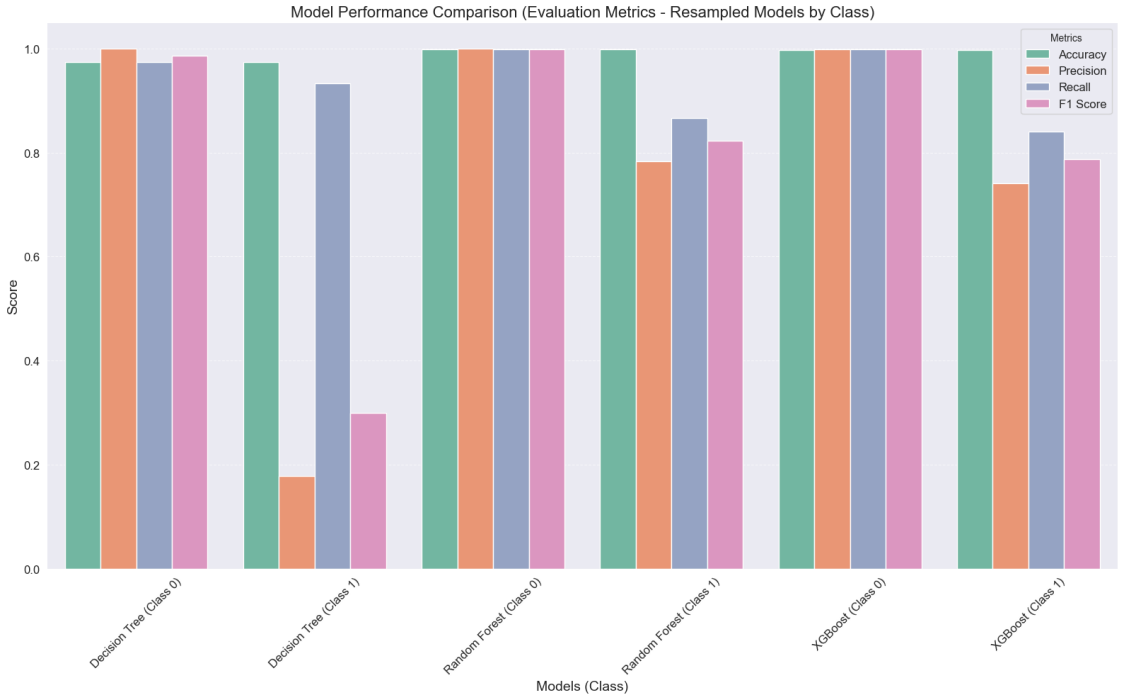



```
Evaluating XGBoost…
XGBoost - Test Accuracy: 0.9972

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     12182
           1       0.74      0.84      0.79        75

    accuracy                           1.00     12257
   macro avg       0.87      0.92      0.89     12257
weighted avg       1.00      1.00      1.00     12257


Confusion Matrix:
[[12160    22]
 [   12    63]]
```



[164]:
```
# 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	0.973240
1	Decision Tree (Class 0)	Precision	0.999579
2	Decision Tree (Class 0)	Recall	0.973485
3	Decision Tree (Class 0)	F1 Score	0.986359
4	Decision Tree (Class 1)	Accuracy	0.973240
5	Decision Tree (Class 1)	Precision	0.178117
6	Decision Tree (Class 1)	Recall	0.933333
7	Decision Tree (Class 1)	F1 Score	0.299145
8	Random Forest (Class 0)	Accuracy	0.997716
9	Random Forest (Class 0)	Precision	0.999179
10	Random Forest (Class 0)	Recall	0.998522
11	Random Forest (Class 0)	F1 Score	0.998850
12	Random Forest (Class 1)	Accuracy	0.997716
13	Random Forest (Class 1)	Precision	0.783133
14	Random Forest (Class 1)	Recall	0.866667
15	Random Forest (Class 1)	F1 Score	0.822785
16	XGBoost (Class 0)	Accuracy	0.997226
17	XGBoost (Class 0)	Precision	0.999014
18	XGBoost (Class 0)	Recall	0.998194
19	XGBoost (Class 0)	F1 Score	0.998604
20	XGBoost (Class 1)	Accuracy	0.997226
21	XGBoost (Class 1)	Precision	0.741176

```
22        XGBoost (Class 1)     Recall  0.840000
23        XGBoost (Class 1)   F1 Score  0.787500
```

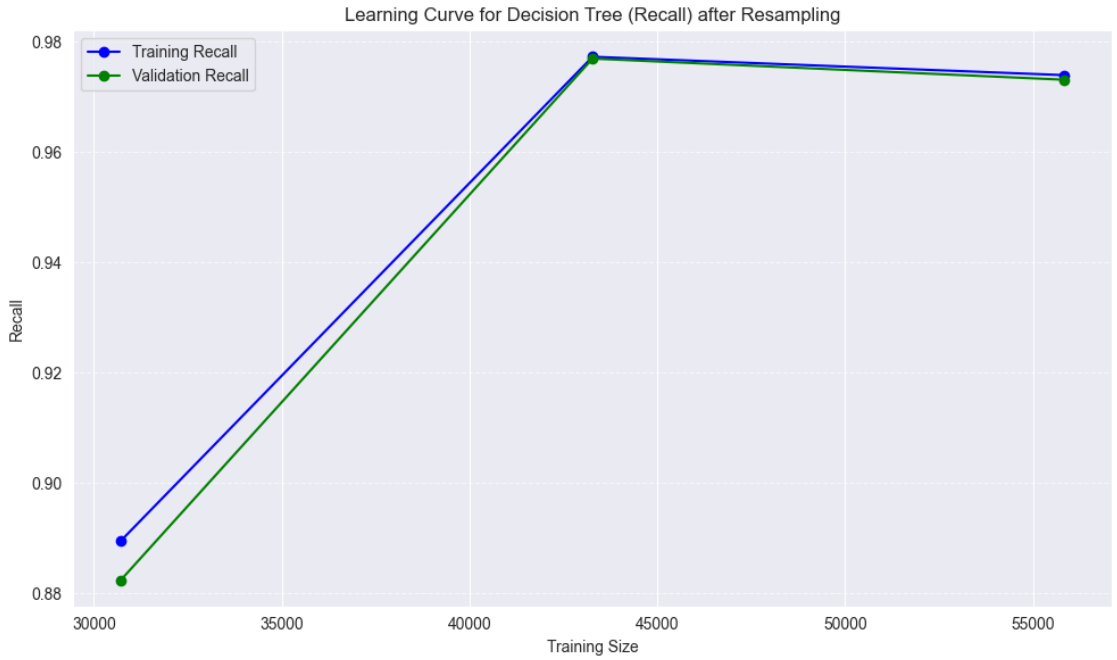
[165]:
```python
# Iterate through resampled models for evaluation
for model_name, model in rsmpl_models.items():
    print(f"\nEvaluating {model_name}...")

    # Training performance
    y_train_pred = model.predict(X_train)  # Predict using the model on␣
 ↪training data
    train_accuracy = accuracy_score(y_train, y_train_pred)  # Compute training␣
 ↪accuracy

    # Test performance
    y_test_pred = model.predict(X_test)  # Predict using the model on test data
    test_accuracy = accuracy_score(y_test, y_test_pred)  # Compute test accuracy

    # Print training and test accuracy
    print(f"Training Accuracy for {model_name}: {train_accuracy:.4f}")
    print(f"Test Accuracy for {model_name}: {test_accuracy:.4f}")
```

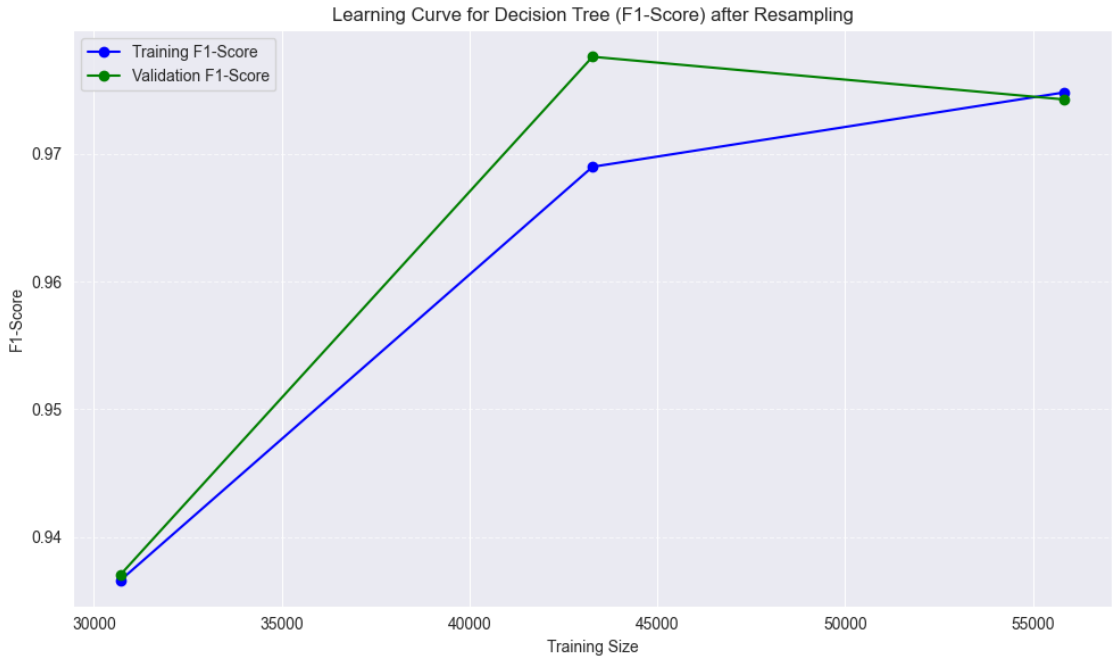
```
Evaluating Decision Tree…
Training Accuracy for Decision Tree: 0.9741
Test Accuracy for Decision Tree: 0.9732

Evaluating Random Forest…
Training Accuracy for Random Forest: 0.9979
Test Accuracy for Random Forest: 0.9977

Evaluating XGBoost…
Training Accuracy for XGBoost: 0.9976
Test Accuracy for XGBoost: 0.9972
```

[166]:
```python
# Iterate through resampled models to plot learning curves
for model_name, model in rsmpl_models.items():
    print(f"\nPlotting Learning Curve for {model_name}...")

    # Generate learning curve for Recall
    train_sizes, train_scores_recall, validation_scores_recall = learning_curve(
        model, X_resampled_smote_enn, y_resampled_smote_enn, cv=5,␣
 ↪scoring='recall', n_jobs=-1
    )

    # Plotting the learning curve for Recall
    plt.figure(figsize=(10, 6))
```

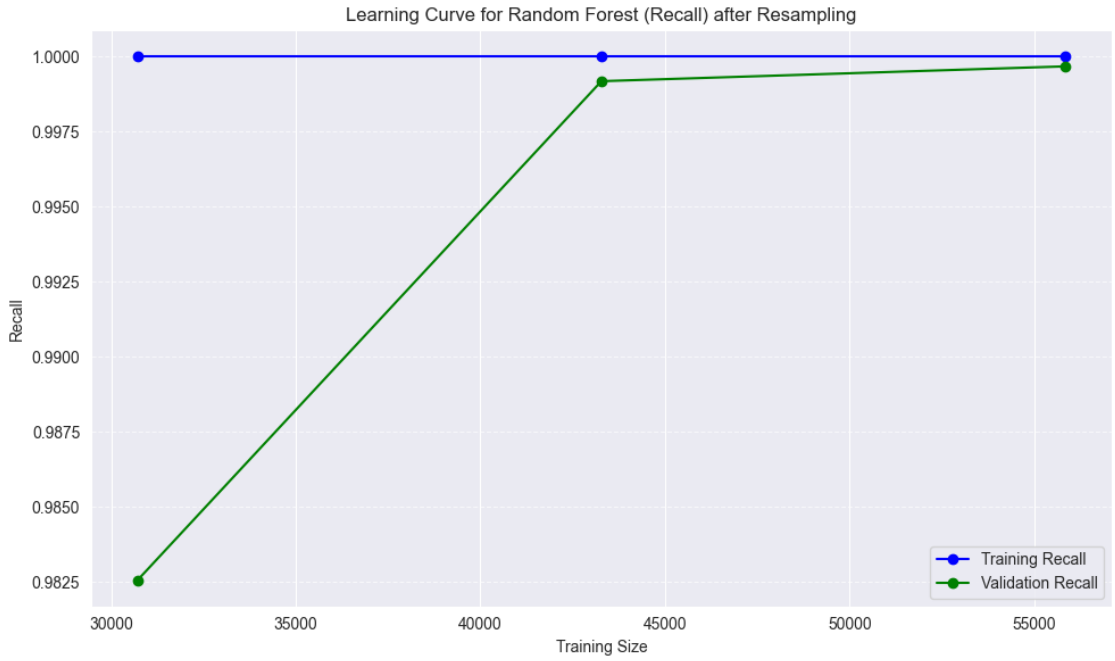
```
    plt.plot(train_sizes, train_scores_recall.mean(axis=1), label='Training␣
↪Recall', color='blue', marker='o')
    plt.plot(train_sizes, validation_scores_recall.mean(axis=1),␣
↪label='Validation Recall', color='green', marker='o')
    plt.xlabel('Training Size')
    plt.ylabel('Recall')
    plt.title(f'Learning Curve for {model_name} (Recall) after Resampling')
    plt.legend(loc='best')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

    # Generate learning curve for F1-Score
    train_sizes, train_scores_f1, validation_scores_f1 = learning_curve(
        model, X_resampled_smote_enn, y_resampled_smote_enn, cv=5,␣
↪scoring='f1', n_jobs=-1
    )

    # Plotting the learning curve for F1-Score
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_f1.mean(axis=1), label='Training␣
↪F1-Score', color='blue', marker='o')
    plt.plot(train_sizes, validation_scores_f1.mean(axis=1), label='Validation␣
↪F1-Score', color='green', marker='o')
    plt.xlabel('Training Size')
    plt.ylabel('F1-Score')
    plt.title(f'Learning Curve for {model_name} (F1-Score) after Resampling')
    plt.legend(loc='best')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()
```

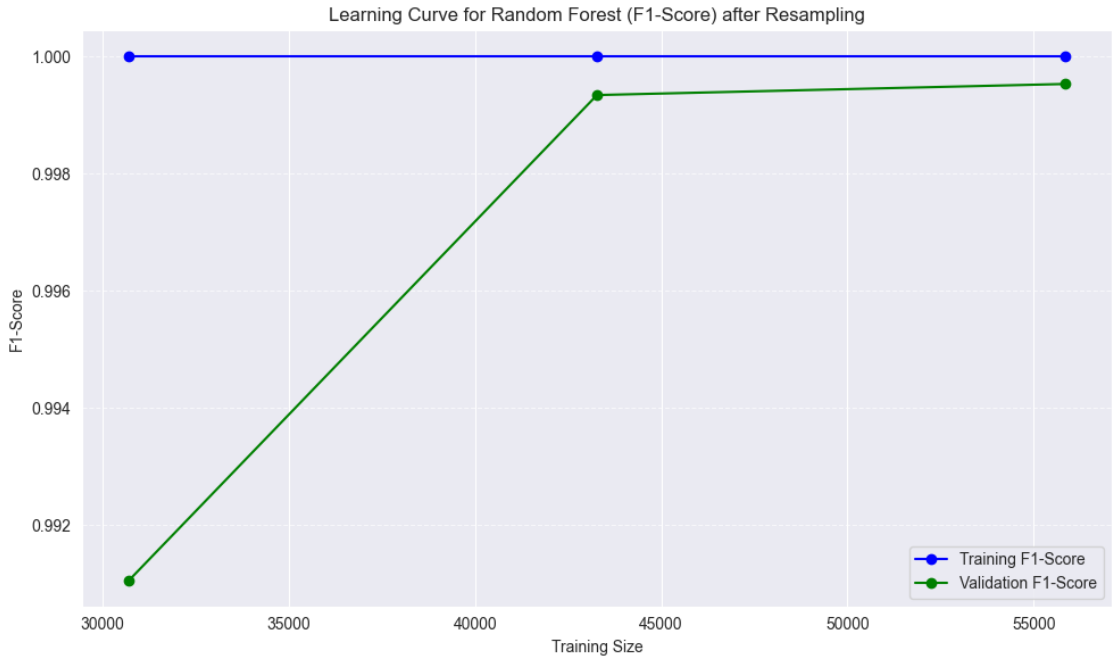
```
Plotting Learning Curve for Decision Tree…
```



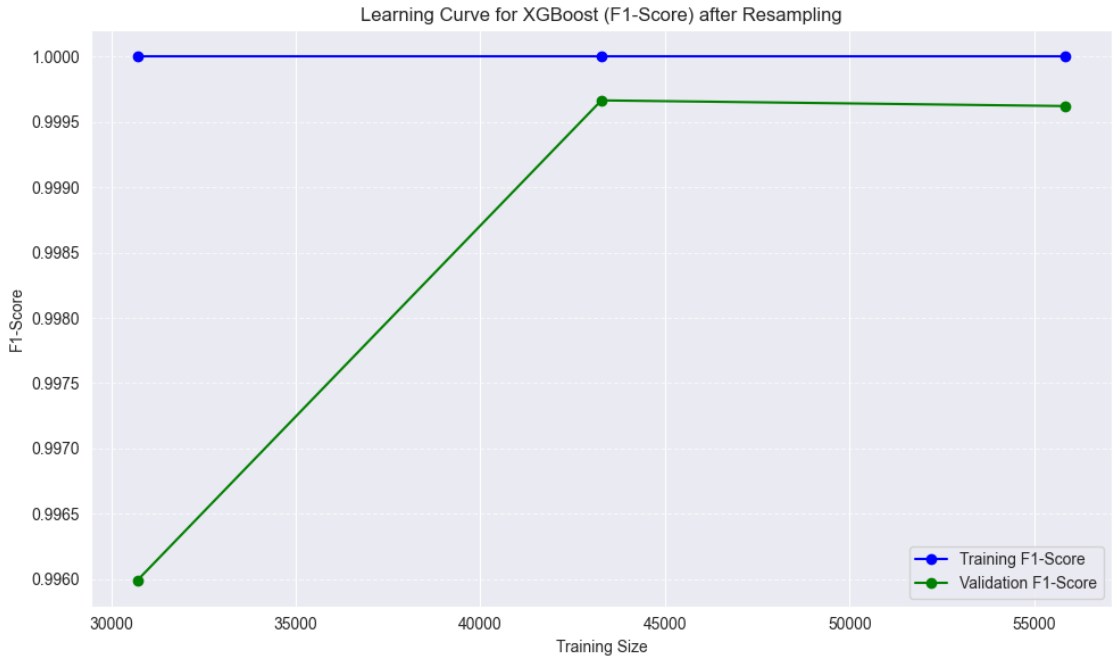



```
Plotting Learning Curve for Random Forest…
```






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



[167]:
```
# 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='accuracy')

    # 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: [0.99796029 0.99809627
0.99782431 0.99700843 0.99782431]
Mean Stratified Cross-Validation Score for Decision Tree: 0.9977

Performing Stratified Cross-Validation for Random Forest…
Stratified Cross-Validation Scores for Random Forest: [0.99809627 0.99796029
0.99823225 0.99700843 0.99796029]
Mean Stratified Cross-Validation Score for Random Forest: 0.9979

Performing Stratified Cross-Validation for XGBoost…
Stratified Cross-Validation Scores for XGBoost: [0.99823225 0.99768833
0.99823225 0.99687245 0.99796029]
Mean Stratified Cross-Validation Score for XGBoost: 0.9978
```

3.3 Hyperparameter Tuning after Resampling data

[168]:
```
# 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]
    },
```

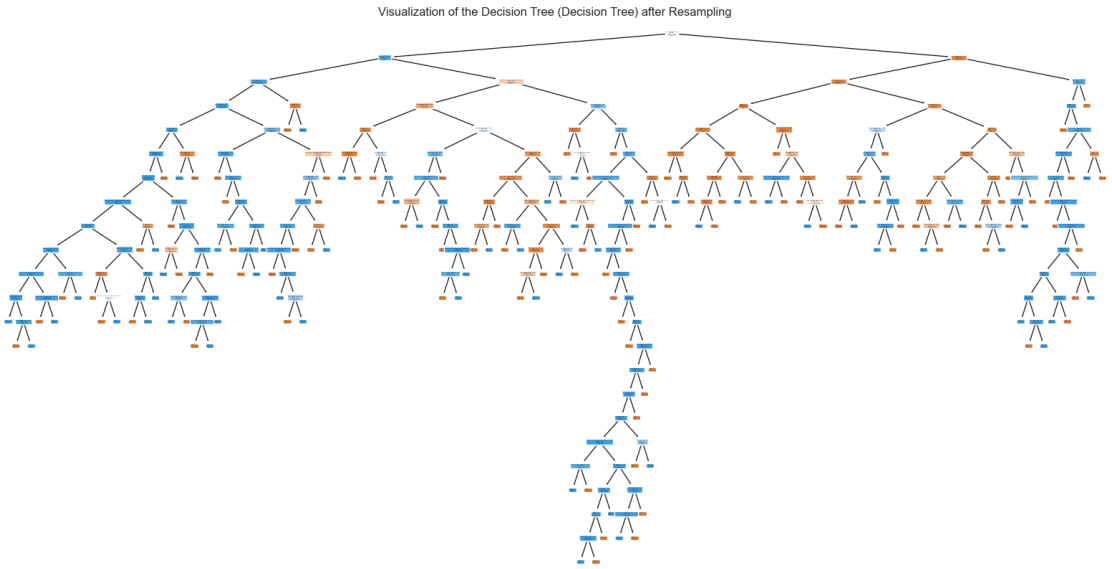
```
    'XGBoost': {
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [3, 6, 9, 12],
        'learning_rate': [0.01, 0.1, 0.3],
        'subsample': [0.7, 1.0],
        'colsample_bytree': [0.7, 1.0],
    },
}

# 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='accuracy',␣
 ↪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 score for {model_name}: {grid_search.best_score_}\n")

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

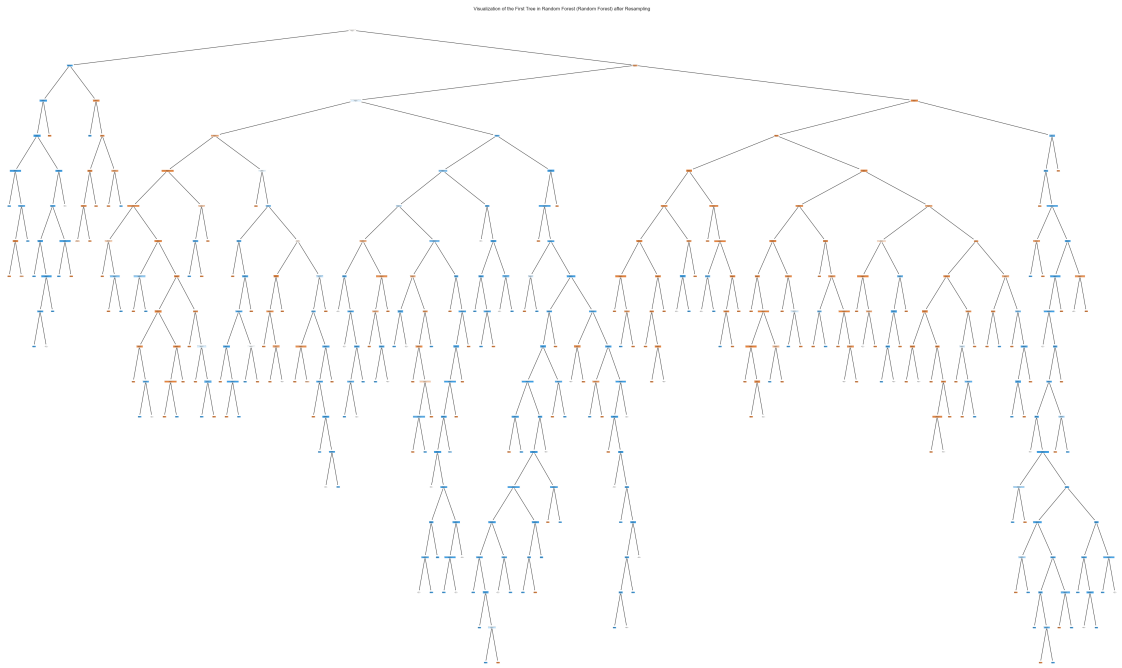
```
Performing GridSearchCV for Decision Tree…
Fitting 5 folds for each of 45 candidates, totalling 225 fits
Best parameters for Decision Tree: {'max_depth': 30, 'min_samples_leaf': 1,
'min_samples_split': 2}
Best score for Decision Tree: 0.9986819884455038

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': 2, 'min_samples_split': 5, 'n_estimators': 150}
Best score for Random Forest: 0.9996418481775573

Performing GridSearchCV for XGBoost…
Fitting 5 folds for each of 192 candidates, totalling 960 fits

D:\New Programming Projects (Pycharm, VS Code, e.t.c.)\AML\AML\venv\Lib\site-
packages\numpy\ma\core.py:2891: RuntimeWarning: invalid value encountered in
cast
  _data = np.array(data, dtype=dtype, copy=copy,

Best parameters for XGBoost: {'colsample_bytree': 0.7, 'learning_rate': 0.1,
'max_depth': 12, 'n_estimators': 150, 'subsample': 1.0}
Best score for XGBoost: 0.999699152715434
```

[169]:
```
# 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, X_resampled_smote_enn):
    for model_name, model in rsmpl_best_models.items():

        # 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=model.classes_.astype(str), 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=(50, 30))
            plot_tree(tree_to_plot, filled=True,␣
 ↪feature_names=X_resampled_smote_enn.columns.tolist(),
                      class_names=model.classes_.astype(str), 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)  # 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 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_after_resampling(rsmpl_best_models, X_resampled_smote_enn)
```





```
<Figure size 2000x1000 with 0 Axes>
```

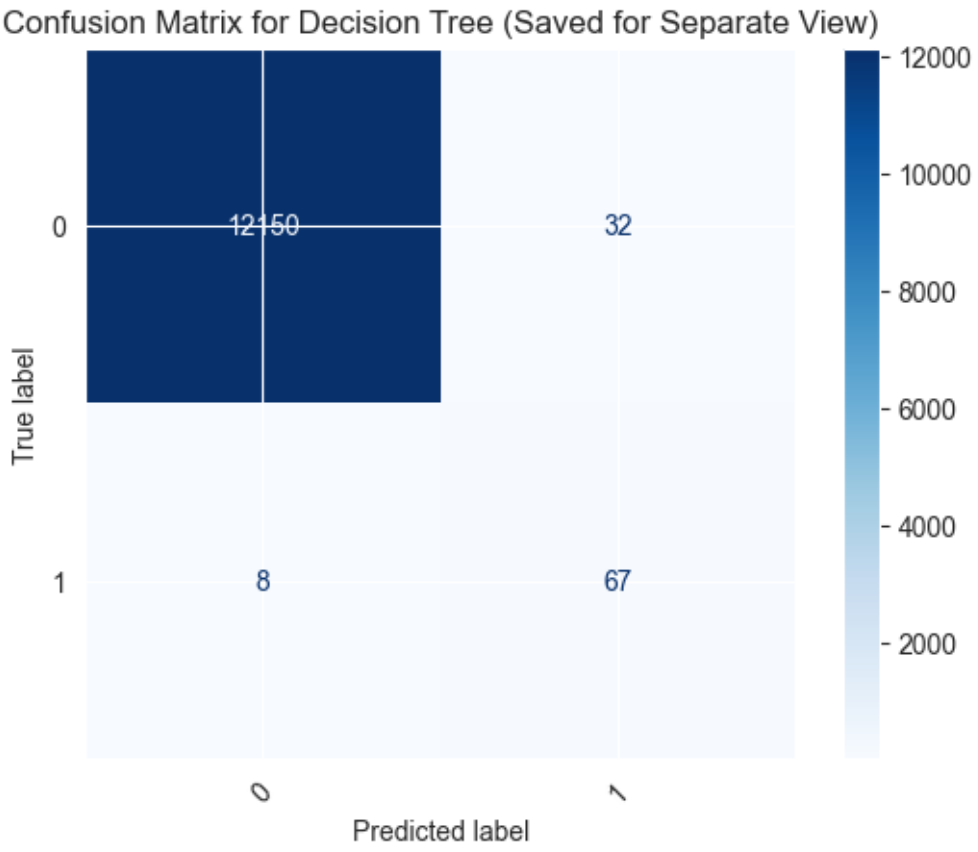
3.4 Evaluating and Validating model after Resampling & Hyperparameter Tuning

[170]:
```
# 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)
```

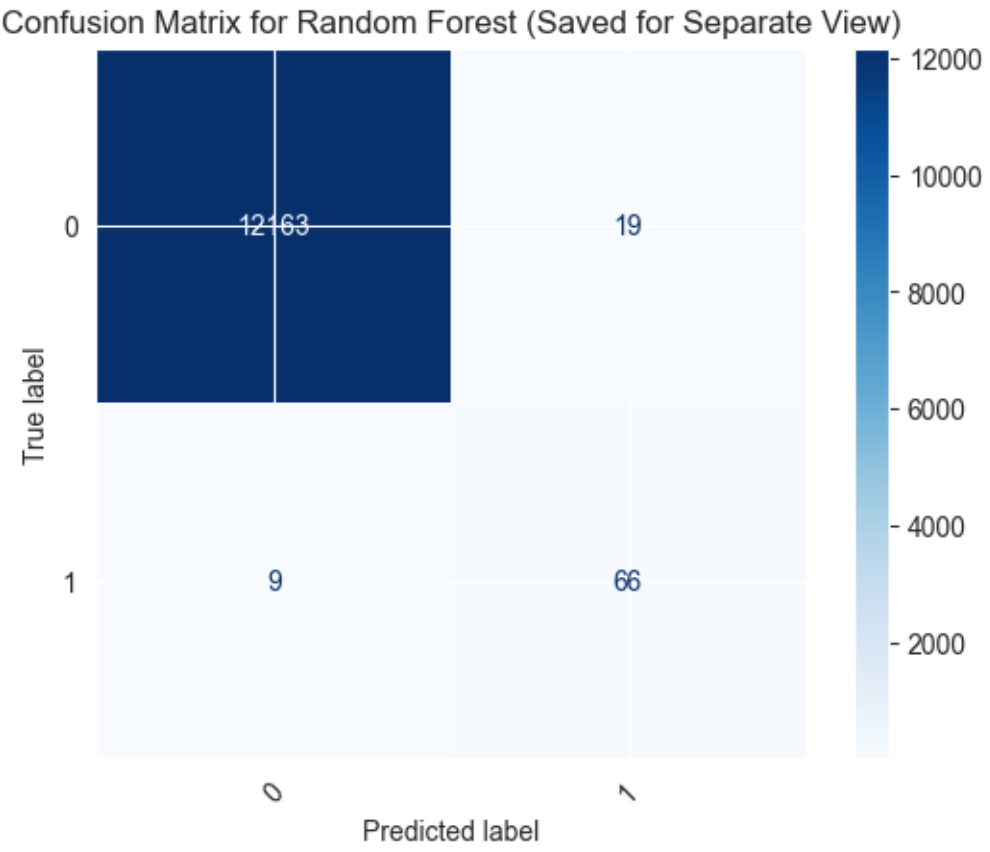
```
    # Compute metrics for Class 0
    accuracy = accuracy_score(y_test, y_pred)
    precision_0 = precision_score(y_test, y_pred, pos_label=0)
    recall_0 = recall_score(y_test, y_pred, pos_label=0)
    f1_0 = f1_score(y_test, y_pred, pos_label=0)

    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} (Saved for Separate View)")
    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)
```

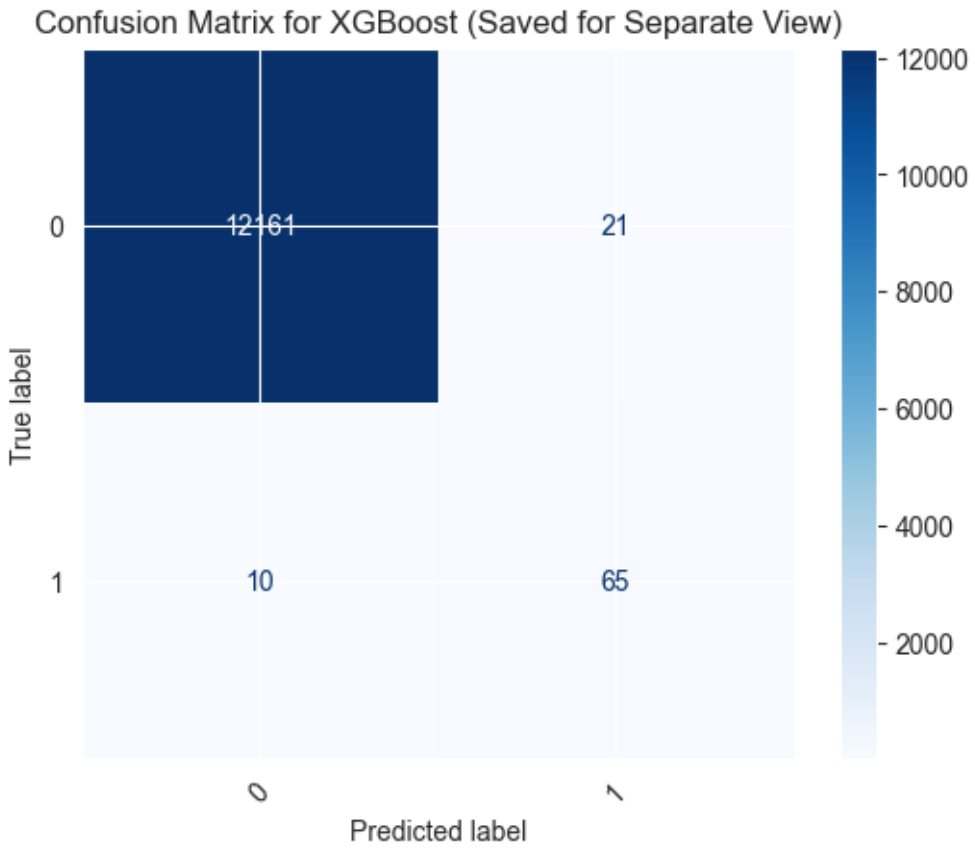
```
Evaluating Decision Tree after Hyperparameter Tuning and Resampling…

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

           0       1.00      1.00      1.00     12182
           1       0.68      0.89      0.77        75

    accuracy                           1.00     12257
   macro avg       0.84      0.95      0.88     12257
weighted avg       1.00      1.00      1.00     12257
```

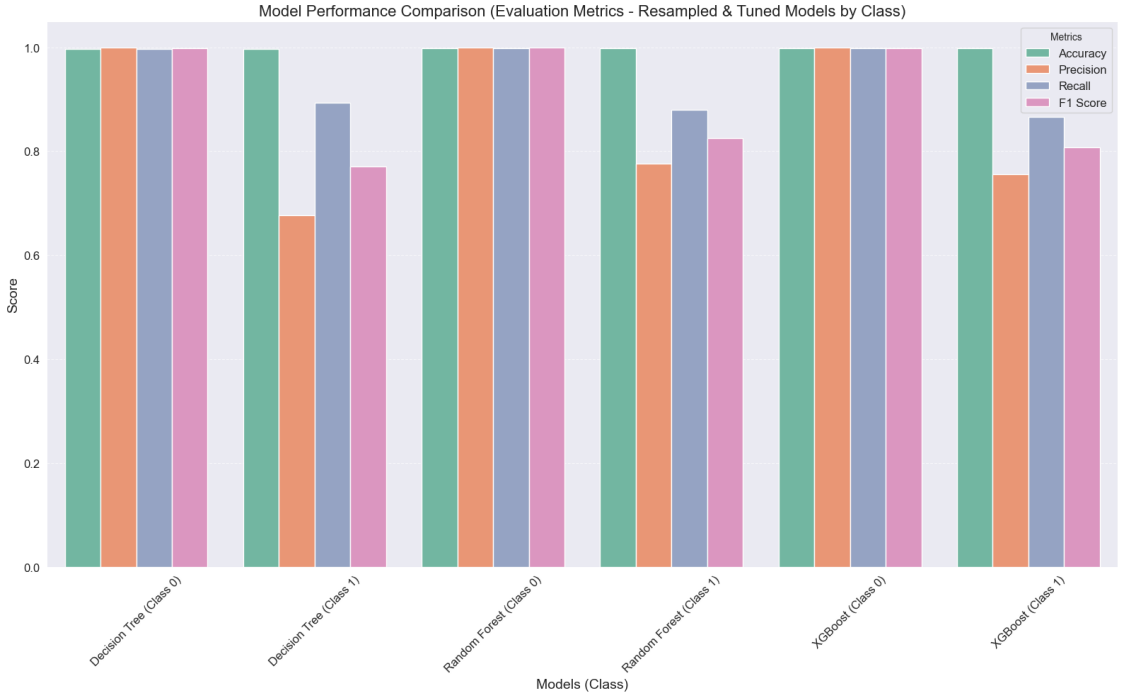



```
Evaluating Random Forest after Hyperparameter Tuning and Resampling…

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

           0       1.00      1.00      1.00     12182
           1       0.78      0.88      0.82        75

    accuracy                           1.00     12257
   macro avg       0.89      0.94      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       0.76      0.87      0.81        75

    accuracy                           1.00     12257
   macro avg       0.88      0.93      0.90     12257
weighted avg       1.00      1.00      1.00     12257
```


Confusion Matrix for XGBoost (Saved for Separate View)
True label
Predicted label
0
1
12161
21
10
65
12000
10000
8000
6000
4000
2000


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

```
Detailed Metrics for Each Resampled & Tuned Model and Class:
                      Model     Metric     Value
0   Decision Tree (Class 0)   Accuracy  0.996737
1   Decision Tree (Class 0)  Precision  0.999342
2   Decision Tree (Class 0)     Recall  0.997373
3   Decision Tree (Class 0)   F1 Score  0.998357
4   Decision Tree (Class 1)   Accuracy  0.996737
5   Decision Tree (Class 1)  Precision  0.676768
6   Decision Tree (Class 1)     Recall  0.893333
7   Decision Tree (Class 1)   F1 Score  0.770115
8   Random Forest (Class 0)   Accuracy  0.997716
9   Random Forest (Class 0)  Precision  0.999261
10  Random Forest (Class 0)     Recall  0.998440
11  Random Forest (Class 0)   F1 Score  0.998850
12  Random Forest (Class 1)   Accuracy  0.997716
13  Random Forest (Class 1)  Precision  0.776471
14  Random Forest (Class 1)     Recall  0.880000
15  Random Forest (Class 1)   F1 Score  0.825000
16        XGBoost (Class 0)   Accuracy  0.997471
17        XGBoost (Class 0)  Precision  0.999178
18        XGBoost (Class 0)     Recall  0.998276
19        XGBoost (Class 0)   F1 Score  0.998727
20        XGBoost (Class 1)   Accuracy  0.997471
21        XGBoost (Class 1)  Precision  0.755814
22        XGBoost (Class 1)     Recall  0.866667
23        XGBoost (Class 1)   F1 Score  0.807453
```

[171]:
```python
# Iterate through resampled models 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_accuracy = accuracy_score(y_train, y_train_pred)  # Compute training␣
 ↪accuracy

    # Predict on the test set
    y_test_pred = model.predict(X_test)  # Predict using the model on test data
    test_accuracy = accuracy_score(y_test, y_test_pred)  # Compute test accuracy

    # Print training and test accuracy for the current model
    print(f"Training Accuracy for {model_name}: {train_accuracy:.4f}")
```

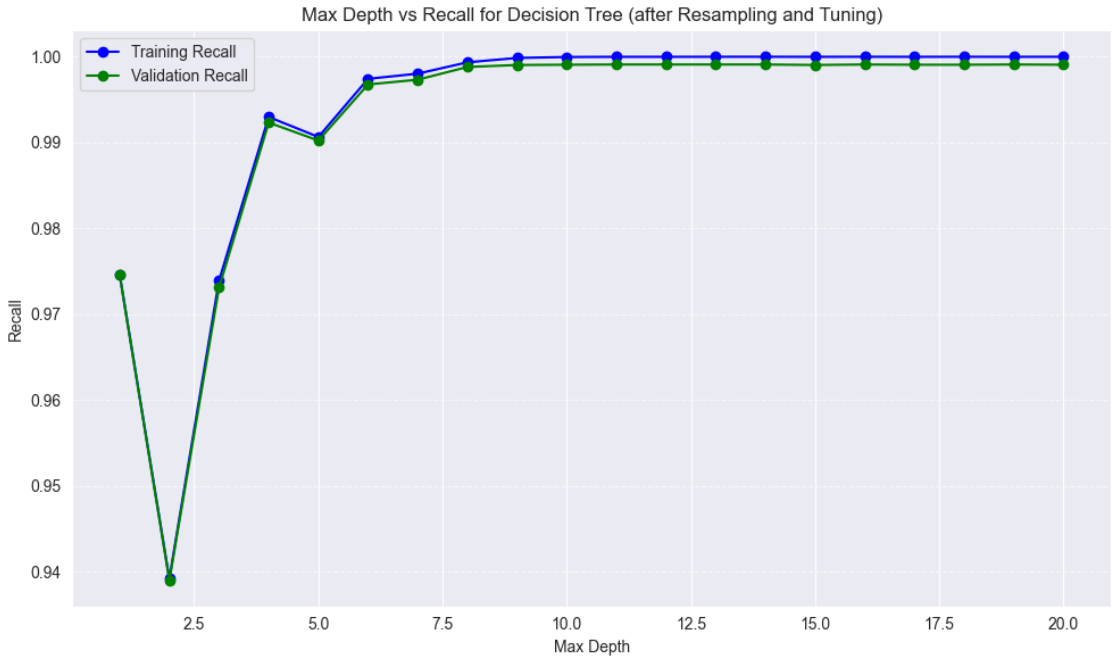
```
    print(f"Test Accuracy for {model_name}: {test_accuracy:.4f}")
```

```
Evaluating Decision Tree after Hyperparameter Tuning and Resampling…
Training Accuracy for Decision Tree: 0.9977
Test Accuracy for Decision Tree: 0.9967

Evaluating Random Forest after Hyperparameter Tuning and Resampling…
Training Accuracy for Random Forest: 0.9979
Test Accuracy for Random Forest: 0.9977

Evaluating XGBoost after Hyperparameter Tuning and Resampling…
Training Accuracy for XGBoost: 0.9975
Test Accuracy for XGBoost: 0.9975
```

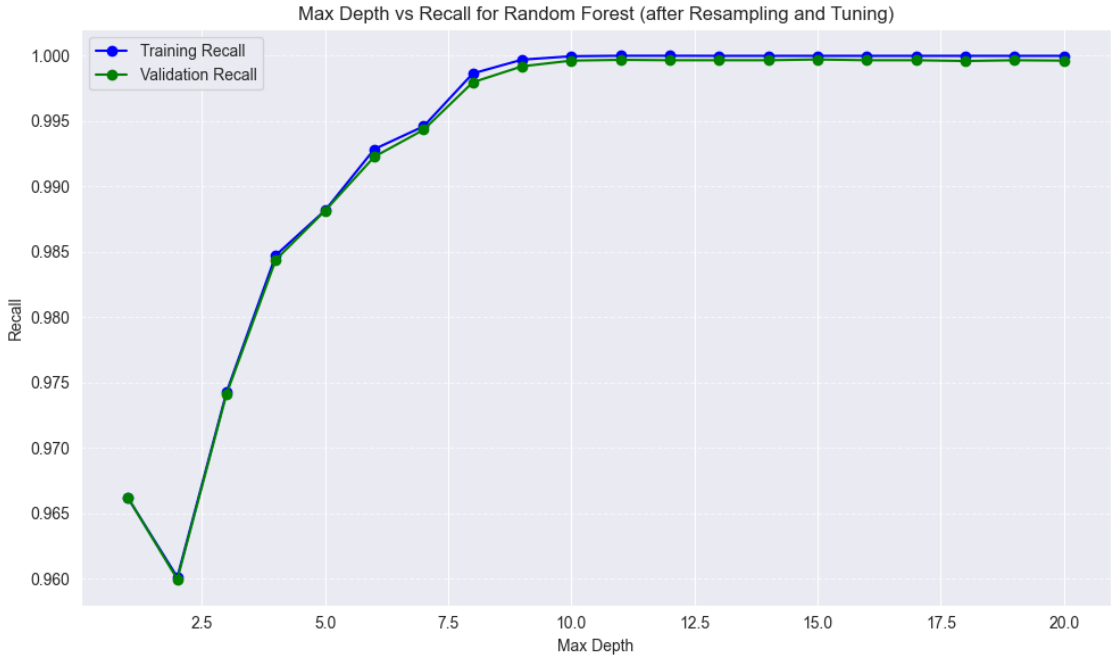
[172]:
```
# Iterate through resampled models to plot learning curves after hyperparameter␣
 ↪tuning and resampling
for model_name, model in rsmpl_best_models.items():
    print(f"\nGenerating Learning Curves for {model_name}...")

    if model_name in ["Decision Tree", "Random Forest"]:
        # Generate max_depth vs Recall for Decision Tree and Random Forest
        max_depths = range(1, 21)  # Example range for max_depth
        train_scores_recall = []
        validation_scores_recall = []

        for depth in max_depths:
            model.set_params(max_depth=depth)  # Set max_depth
            train_sizes, train_scores, validation_scores = learning_curve(
                model, X_resampled_smote_enn, y_resampled_smote_enn, cv=5,␣
 ↪scoring='recall', n_jobs=-1
            )
            train_scores_recall.append(train_scores.mean(axis=1)[-1])  #␣
 ↪Average recall for the largest training size
            validation_scores_recall.append(validation_scores.mean(axis=1)[-1])␣
 ↪ # Average recall for the largest training size

        # Plot max_depth vs Recall
        plt.figure(figsize=(10, 6))
        plt.plot(max_depths, train_scores_recall, label='Training Recall',␣
 ↪marker='o', color='blue')
        plt.plot(max_depths, validation_scores_recall, label='Validation␣
 ↪Recall', marker='o', color='green')
        plt.xlabel('Max Depth')
        plt.ylabel('Recall')
        plt.title(f'Max Depth vs Recall for {model_name} (after Resampling and␣
 ↪Tuning)')
```

```
        plt.legend(loc='best')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.show()

    elif model_name == "XGBoost":
        # Generate n_estimators vs F1-Score for XGBoost
        n_estimators = range(50, 301, 50)  # Example range for n_estimators
        train_scores_f1 = []
        validation_scores_f1 = []

        for n in n_estimators:
            model.set_params(n_estimators=n)  # Set n_estimators
            train_sizes, train_scores, validation_scores = learning_curve(
                model, X_resampled_smote_enn, y_resampled_smote_enn, cv=5,␣
↪scoring='f1', n_jobs=-1
            )
            train_scores_f1.append(train_scores.mean(axis=1)[-1])  # Average␣
↪F1-score for the largest training size
            validation_scores_f1.append(validation_scores.mean(axis=1)[-1])  #␣
↪Average F1-score for the largest training size

        # Plot n_estimators vs F1-Score
        plt.figure(figsize=(10, 6))
        plt.plot(n_estimators, train_scores_f1, label='Training F1-Score',␣
↪marker='o', color='blue')
        plt.plot(n_estimators, validation_scores_f1, label='Validation␣
↪F1-Score', marker='o', color='green')
        plt.xlabel('Number of Estimators')
        plt.ylabel('F1-Score')
        plt.title(f'Number of Estimators vs F1-Score for {model_name} (after␣
↪Resampling and Tuning)')
        plt.legend(loc='best')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.show()
```

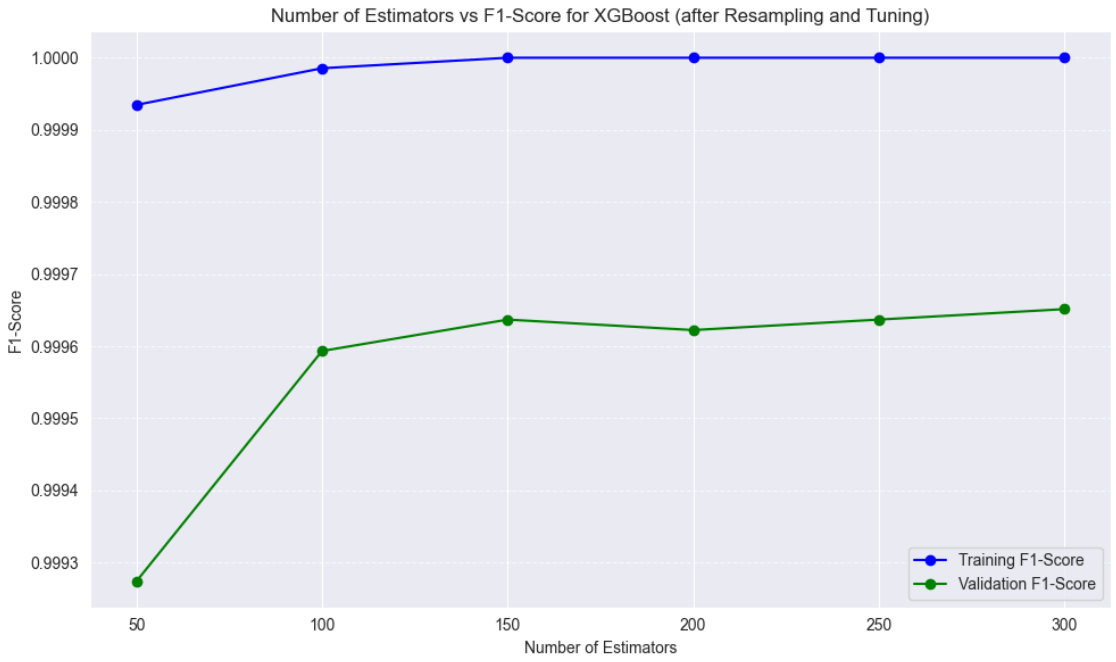
```
Generating Learning Curves for Decision Tree…
```




```
Generating Learning Curves for Random Forest…
```



```
Generating Learning Curves for XGBoost…
```



[173]:
```
# 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
    cross_val_scores = cross_val_score(model, X_train, y_train,␣
 ↪cv=stratified_kfold, scoring='accuracy')

    # 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 after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation Scores for Decision Tree: [0.99632853 0.99714441
```

```
0.99755235 0.99592059 0.99632853]
Mean Stratified Cross-Validation Score for Decision Tree: 0.9967

Performing Stratified Cross-Validation for Random Forest after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation Scores for Random Forest: [0.99850422 0.99809627
0.99823225 0.99700843 0.99809627]
Mean Stratified Cross-Validation Score for Random Forest: 0.9980

Performing Stratified Cross-Validation for XGBoost after Resampling and
Hyperparameter Tuning…
Stratified Cross-Validation Scores for XGBoost: [0.99850422 0.99782431
0.99823225 0.99687245 0.99823225]
Mean Stratified Cross-Validation Score for XGBoost: 0.9979
```

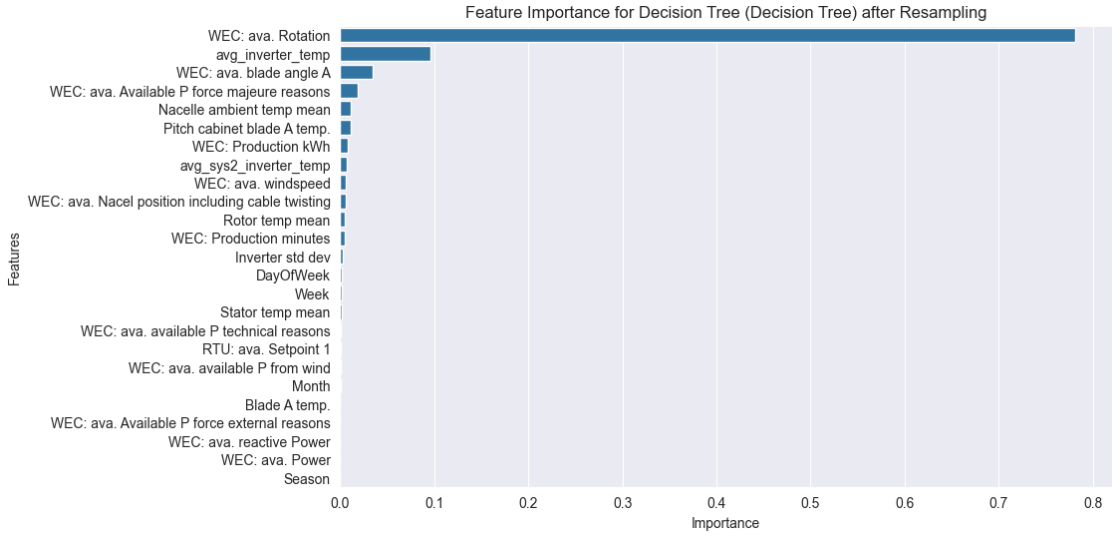
3.5 Feature Importance after Resampling & Hyperparameter Tuning

[174]:
```
# 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
```

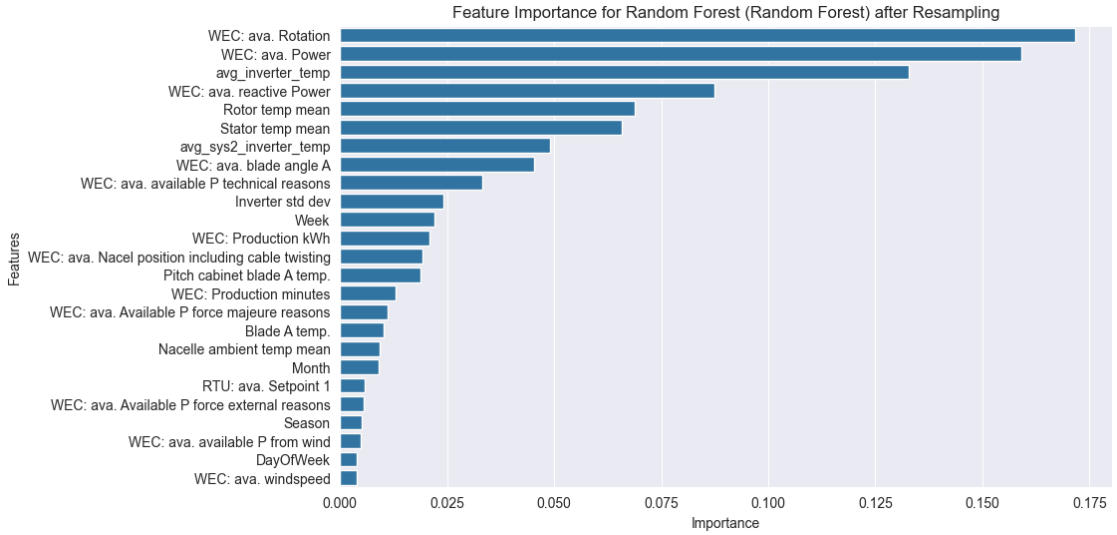
```
plot_and_print_feature_importance_after_resampling(rsmpl_best_models,␣
  ↪X_resampled_smote_enn)
```

```
Feature Importance for Decision Tree after Resampling:
                                                Feature  Importance
1                                  WEC: ava. Rotation    0.781641
20                                  avg_inverter_temp    0.095783
11                               WEC: ava. blade angle A    0.034407
9          WEC: ava. Available P force majeure reasons    0.019114
24                            Nacelle ambient temp mean    0.011205
12                          Pitch cabinet blade A temp.    0.010912
4                                  WEC: Production kWh    0.008463
21                               avg_sys2_inverter_temp    0.006968
0                                  WEC: ava. windspeed    0.005894
3    WEC: ava. Nacel position including cable twisting    0.005795
22                                   Rotor temp mean    0.004605
5                               WEC: Production minutes    0.004558
15                                  Inverter std dev    0.003014
18                                         DayOfWeek    0.002039
17                                              Week    0.001454
23                                  Stator temp mean    0.001311
8              WEC: ava. available P technical reasons    0.001131
14                                 RTU: ava. Setpoint 1    0.000556
7                      WEC: ava. available P from wind    0.000298
16                                             Month    0.000290
13                                   Blade A temp.    0.000178
10        WEC: ava. Available P force external reasons    0.000140
6                               WEC: ava. reactive Power    0.000104
2                                  WEC: ava. Power    0.000082
19                                             Season    0.000059
```



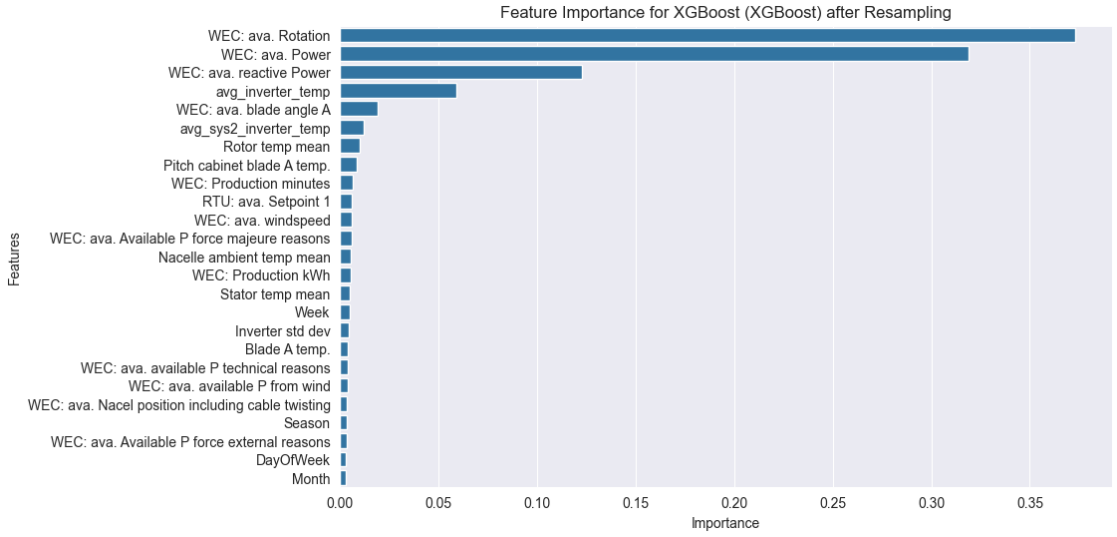
Feature Importance for Random Forest after Resampling:

	Feature	Importance
1	WEC: ava. Rotation	0.171751
2	WEC: ava. Power	0.159086
20	avg_inverter_temp	0.132750
6	WEC: ava. reactive Power	0.087534
22	Rotor temp mean	0.068793
23	Stator temp mean	0.065792
21	avg_sys2_inverter_temp	0.048957
11	WEC: ava. blade angle A	0.045359
8	WEC: ava. available P technical reasons	0.033161
15	Inverter std dev	0.024090
17	Week	0.022106
4	WEC: Production kWh	0.020807
3	WEC: ava. Nacel position including cable twisting	0.019182
12	Pitch cabinet blade A temp.	0.018871
5	WEC: Production minutes	0.013058
9	WEC: ava. Available P force majeure reasons	0.011148
13	Blade A temp.	0.010223
24	Nacelle ambient temp mean	0.009296
16	Month	0.008916
14	RTU: ava. Setpoint 1	0.005708
10	WEC: ava. Available P force external reasons	0.005444
19	Season	0.005166
7	WEC: ava. available P from wind	0.004915
18	DayOfWeek	0.003982
0	WEC: ava. windspeed	0.003905



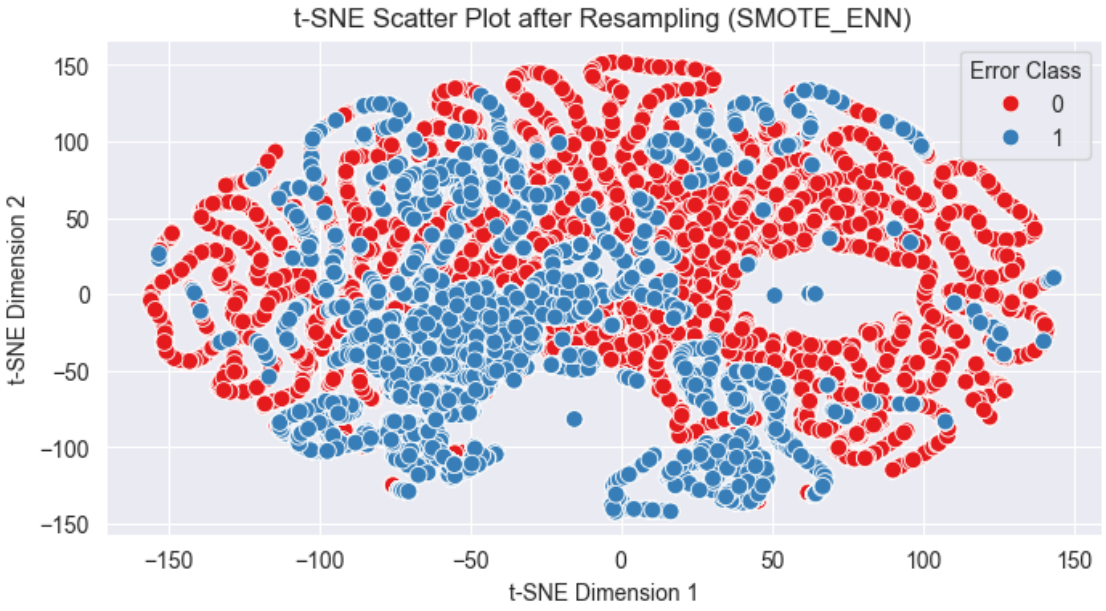
Feature Importance for XGBoost after Resampling:

	Feature	Importance
1	WEC: ava. Rotation	0.373230
2	WEC: ava. Power	0.318850
6	WEC: ava. reactive Power	0.122779
20	avg_inverter_temp	0.058875
11	WEC: ava. blade angle A	0.019191
21	avg_sys2_inverter_temp	0.012006
22	Rotor temp mean	0.009854
12	Pitch cabinet blade A temp.	0.008663
5	WEC: Production minutes	0.006417
14	RTU: ava. Setpoint 1	0.006123
0	WEC: ava. windspeed	0.005811
9	WEC: ava. Available P force majeure reasons	0.005694
24	Nacelle ambient temp mean	0.005453
4	WEC: Production kWh	0.005192
23	Stator temp mean	0.005012
17	Week	0.004665
15	Inverter std dev	0.004369
13	Blade A temp.	0.004138
8	WEC: ava. available P technical reasons	0.003949
7	WEC: ava. available P from wind	0.003811
3	WEC: ava. Nacel position including cable twisting	0.003372
19	Season	0.003302
10	WEC: ava. Available P force external reasons	0.003273
18	DayOfWeek	0.003124
16	Month	0.002848



[ ]:

[175]:
```
# --- 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()
```



[ ]:

[ ]: