IMPORTING OF NECESSARY LIBRARIES

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

from sklearn.model_selection import train_test_split, GridSearchCV,
RandomizedSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import roc_curve, auc, confusion_matrix,
classification_report
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier

from imblearn.over_sampling import SMOTE

import xgboost as xgb
import lightgbm as lgb
import catboost as cb

from keras.models import Sequential
from keras.layers import Dense

import optuna
from hyperopt import hp, fmin, tpe, STATUS_OK, Trials

import scikitplot as skplt
```

GETTING TO KNOW THE VARIABLES AND PERFORMING SOME STATISTICS ON THE CHOOSING DATA

```
# Importing the pandas library as pd
import pandas as pd

# Loading the dataset from a specified path on my local system
```

```
water_potability = pd.read_csv("C:/Users/HP SPECTRE
xt/water_potability.csv")

# Displaying the first five rows of the dataset to get an initial
overview
print(water_potability.head())

# Displaying the last five rows of the dataset to see the ending
entries
print(water_potability.tail())

# Printing the information about the dataset including the data types
and non-null values
print(water_potability.info())
```

```
         ph    Hardness        Solids  Chloramines     Sulfate
Conductivity  \
0       NaN  204.890455  20791.318981     7.300212  368.516441
564.308654
1  3.716080  129.422921  18630.057858     6.635246         NaN
592.885359
2  8.099124  224.236259  19909.541732     9.275884         NaN
418.606213
3  8.316766  214.373394  22018.417441     8.059332  356.886136
363.266516
4  9.092223  181.101509  17978.986339     6.546600  310.135738
398.410813

   Organic_carbon  Trihalomethanes  Turbidity  Potability
0       10.379783        86.990970   2.963135           0
1       15.180013        56.329076   4.500656           0
2       16.868637        66.420093   3.055934           0
3       18.436524       100.341674   4.628771           0
4       11.558279        31.997993   4.075075           0
            ph    Hardness        Solids  Chloramines     Sulfate  \
3271  4.668102  193.681735  47580.991603     7.166639  359.948574
3272  7.808856  193.553212  17329.802160     8.061362         NaN
3273  9.419510  175.762646  33155.578218     7.350233         NaN
3274  5.126763  230.603758  11983.869376     6.303357         NaN
3275  7.874671  195.102299  17404.177061     7.509306         NaN

      Conductivity  Organic_carbon  Trihalomethanes  Turbidity
Potability
3271    526.424171       13.894419        66.687695   4.435821
1
3272    392.449580       19.903225              NaN   2.798243
1
3273    432.044783       11.039070        69.845400   3.298875
1
3274    402.883113       11.168946        77.488213   4.708658
```

```
1
3275    327.459760       16.140368        78.698446   2.309149
1
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3276 entries, 0 to 3275
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   ph               2785 non-null   float64
 1   Hardness         3276 non-null   float64
 2   Solids           3276 non-null   float64
 3   Chloramines      3276 non-null   float64
 4   Sulfate          2495 non-null   float64
 5   Conductivity     3276 non-null   float64
 6   Organic_carbon   3276 non-null   float64
 7   Trihalomethanes  3114 non-null   float64
 8   Turbidity        3276 non-null   float64
 9   Potability       3276 non-null   int64  
dtypes: float64(9), int64(1)
memory usage: 256.1 KB
None

# Displaying the dimensions of the dataset (number of rows and
columns)
print("Shape of the Dataset:", water_potability.shape)

# Printing a list of all column names in the dataset to understand the
features available
print(water_potability.columns.tolist())

# Outputting a concise summary of the dataset including index dtype
and columns, non-null values, and memory usage
water_potability.info()

# Generating descriptive statistics that summarize the central
tendency, dispersion, and shape of the dataset's numerical features
water_potability.describe()

Shape of the Dataset: (3276, 10)
['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate', 'Conductivity',
'Organic_carbon', 'Trihalomethanes', 'Turbidity', 'Potability']
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3276 entries, 0 to 3275
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   ph               2785 non-null   float64
 1   Hardness         3276 non-null   float64
 2   Solids           3276 non-null   float64
 3   Chloramines      3276 non-null   float64
```

```
 4   Sulfate          2495 non-null   float64
 5   Conductivity     3276 non-null   float64
 6   Organic_carbon   3276 non-null   float64
 7   Trihalomethanes  3114 non-null   float64
 8   Turbidity        3276 non-null   float64
 9   Potability       3276 non-null   int64
dtypes: float64(9), int64(1)
memory usage: 256.1 KB
```

```
                ph     Hardness        Solids  Chloramines
Sulfate  \
count  2785.000000  3276.000000   3276.000000  3276.000000
2495.000000
mean      7.080795   196.369496  22014.092526     7.122277
333.775777
std       1.594320    32.879761   8768.570828     1.583085
41.416840
min       0.000000    47.432000    320.942611     0.352000
129.000000
25%       6.093092   176.850538  15666.690297     6.127421
307.699498
50%       7.036752   196.967627  20927.833607     7.130299
333.073546
75%       8.062066   216.667456  27332.762127     8.114887
359.950170
max      14.000000   323.124000  61227.196008    13.127000
481.030642

       Conductivity  Organic_carbon  Trihalomethanes    Turbidity
Potability
count   3276.000000     3276.000000      3114.000000  3276.000000
3276.000000
mean     426.205111       14.284970        66.396293     3.966786
0.390110
std       80.824064        3.308162        16.175008     0.780382
0.487849
min      181.483754        2.200000         0.738000     1.450000
0.000000
25%      365.734414       12.065801        55.844536     3.439711
0.000000
50%      421.884968       14.218338        66.622485     3.955028
0.000000
75%      481.792304       16.557652        77.337473     4.500320
1.000000
max      753.342620       28.300000       124.000000     6.739000
1.000000
```

```
# Calculating the number of missing values in each column of the
dataset
water_potability.isnull().sum()
```

```python
# Assigning the total number of missing values per column to a variable
missing_values = water_potability.isnull().sum()
# Calculating the percentage of missing values in each column
missing_percentage = (missing_values / len(water_potability)) * 100
# Displaying the percentage of missing values for each column
print(missing_percentage)

# Defining a function to print all unique values and their counts for each column in a DataFrame
def uni(df):
    import numpy as np  # Ensuring numpy is imported for sorting operations
    for column in df.columns:
        # Sorting and printing unique values for each column
        unique_values = np.sort(df[column].unique())
        print(f'\nAll Unique Values in {column}:')
        print(unique_values)
        # Counting and printing the total number of unique values in each column
        print(f'Total number of unique values: {len(unique_values)}')

# Calling the function with the water_potability DataFrame to analyze unique values
uni(water_potability)

# Identifying and displaying all duplicated rows within the DataFrame
duplicated_rows = water_potability[water_potability.duplicated()]
print(duplicated_rows)
```

```
ph                  14.987790
Hardness             0.000000
Solids               0.000000
Chloramines          0.000000
Sulfate             23.840049
Conductivity         0.000000
Organic_carbon       0.000000
Trihalomethanes      4.945055
Turbidity            0.000000
Potability           0.000000
dtype: float64

All Unique Values in ph:
[ 0.          0.22749905  0.97557799 ... 13.54124024 14.
         nan]
Total number of unique values: 2786

All Unique Values in Hardness:
```

```
[ 47.432       73.49223369  77.4595861  ... 311.38395647 317.33812406
 323.124     ]
Total number of unique values: 3276

All Unique Values in Solids:
[  320.94261127   728.75082958  1198.94369901 ... 56488.67241274
 56867.85923615 61227.19600771]
Total number of unique values: 3276

All Unique Values in Chloramines:
[ 0.352       0.53035129  1.3908709  ... 12.91218664 13.04380611
 13.127     ]
Total number of unique values: 3276

All Unique Values in Sulfate:
[129.         180.20674636 182.39737025 ... 476.53971733 481.03064231
          nan]
Total number of unique values: 2496

All Unique Values in Conductivity:
[181.48375399 201.61973676 210.31918197 ... 695.36952799 708.22636447
 753.34261956]
Total number of unique values: 3276

All Unique Values in Organic_carbon:
[ 2.2         4.37189861  4.46677197 ... 24.75539237 27.00670661
 28.3       ]
Total number of unique values: 3276

All Unique Values in Trihalomethanes:
[  0.738        8.17587638   8.57701293 ... 120.03007701 124.
          nan]
Total number of unique values: 3115

All Unique Values in Turbidity:
[1.45       1.49220662 1.49610094 ... 6.49424947 6.49474856
6.739     ]
Total number of unique values: 3276

All Unique Values in Potability:
[0 1]
Total number of unique values: 2
Empty DataFrame
Columns: [ph, Hardness, Solids, Chloramines, Sulfate, Conductivity,
Organic_carbon, Trihalomethanes, Turbidity, Potability]
Index: []
```

CHECKING FOR PERCENTAGE OF OUTLIER ON EACH VARIBLES

```
# Define a function to calculate the percentage of outlier values in
each column of a DataFrame
def outlier_percentage(df):
    import numpy as np  # Ensure numpy is imported for any numerical
operations not shown directly here

    # Loop through each column in the DataFrame
    for i in range(len(df.columns)):
        # Calculate the first quartile (25th percentile)
        q1 = df[df.columns[i]].quantile(0.25)
        # Calculate the third quartile (75th percentile)
        q3 = df[df.columns[i]].quantile(0.75)
        # Interquartile range (IQR) calculation
        iqr = q3 - q1

        # Define upper and lower bounds for outliers
        upper = q3 + (iqr * 1.5)
        lower = q1 - (iqr * 1.5)

        # Calculate the percentage of values that are considered
outliers
        percentage = (((len(df[df[df.columns[i]] > upper])) +
                       (len(df[df[df.columns[i]] < lower]))) /
len(df[df.columns[i]])) * 100

        # Print the column name and the percentage of outliers in that
column
        print(str(df.columns[i]) + ' : ' + str(percentage) + ' %')

# Call the function with the water_potability dataset to find the
percentage of outliers in each column
outlier_percentage(water_potability)
```

```
ph : 1.4041514041514043 %
Hardness : 2.5335775335775335 %
Solids : 1.4346764346764347 %
Chloramines : 1.862026862026862 %
Sulfate : 1.2515262515262515 %
Conductivity : 0.3357753357753358 %
Organic_carbon : 0.7631257631257631 %
Trihalomethanes : 1.0073260073260073 %
Turbidity : 0.57997557997558 %
Potability : 0.0 %
```

MEAN VALUES OF EACH VARIABLE

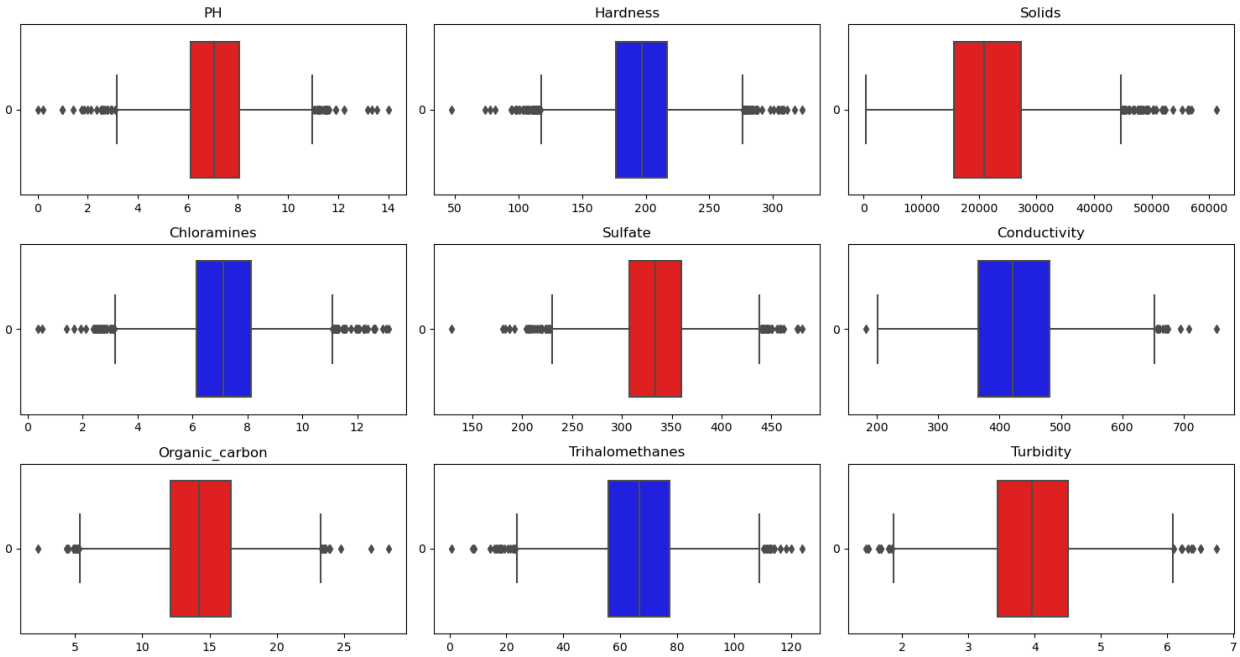
```
# Calculate the mean of each column in the water_potability DataFrame
mean_values = water_potability.mean()

# Print the computed mean values for each column
print(mean_values)
```

```
ph                    7.080795
Hardness            196.369496
Solids            22014.092526
Chloramines           7.122277
Sulfate             333.775777
Conductivity        426.205111
Organic_carbon       14.284970
Trihalomethanes      66.396293
Turbidity             3.966786
Potability            0.390110
dtype: float64
```

EXPLORATORY DATA ANALYSIS OF WATER POTABILITY DATASET

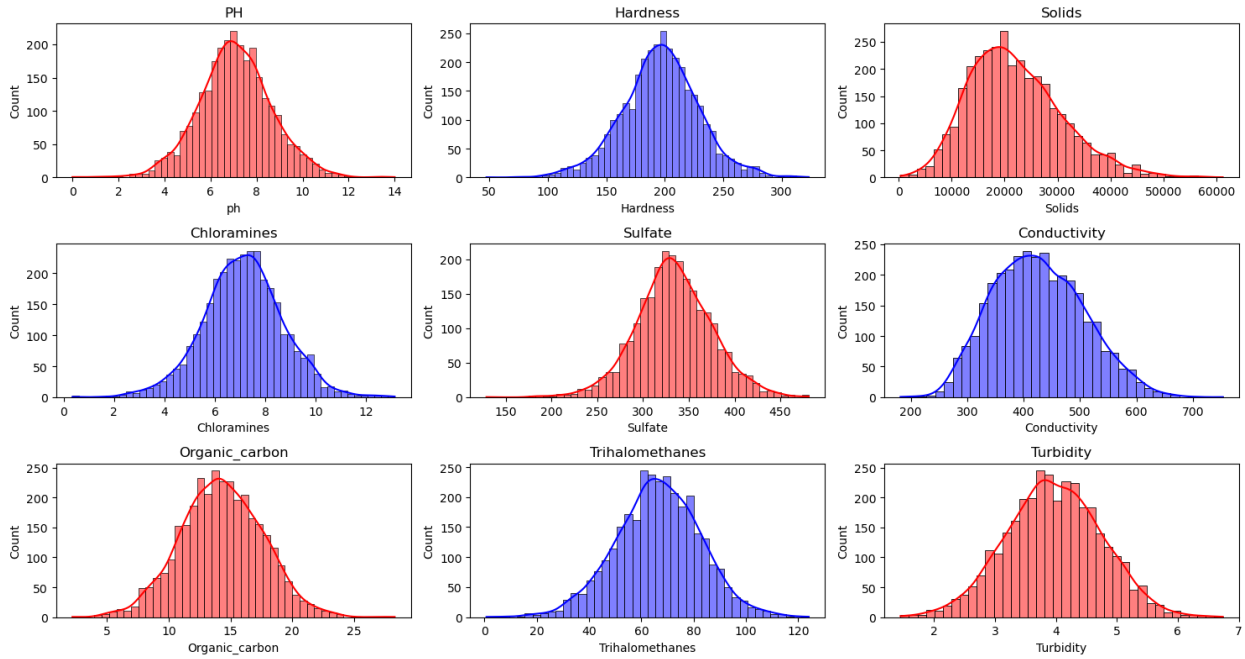
```
# Importing the required libraries for visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Setting up a figure with multiple subplots arranged in a 3x3 grid
fig, ax = plt.subplots(3, 3, figsize=(15, 8))

# Creating a box plot for the pH levels in the water, setting the plot
title and color
plt.setp(ax[0,0], title='PH')
sns.boxplot(water_potability['ph'], orient='h', ax=ax[0,0],
color='red')

# Creating a box plot for Hardness in the water, setting the plot
title and color
plt.setp(ax[0,1], title='Hardness')
sns.boxplot(water_potability['Hardness'], orient='h', ax=ax[0,1],
color='blue')

# Creating a box plot for Solids content in the water, setting the
plot title and color
plt.setp(ax[0,2], title='Solids')
sns.boxplot(water_potability['Solids'], orient='h', ax=ax[0,2],
```

```
color='red')

# Creating a box plot for Chloramines levels in the water, setting the
plot title and color
plt.setp(ax[1,0], title='Chloramines')
sns.boxplot(water_potability['Chloramines'], orient='h', ax=ax[1,0],
color='blue')

# Creating a box plot for Sulfate levels in the water, setting the
plot title and color
plt.setp(ax[1,1], title='Sulfate')
sns.boxplot(water_potability['Sulfate'], orient='h', ax=ax[1,1],
color='red')

# Creating a box plot for Conductivity of the water, setting the plot
title and color
plt.setp(ax[1,2], title='Conductivity')
sns.boxplot(water_potability['Conductivity'], orient='h', ax=ax[1,2],
color='blue')

# Creating a box plot for Organic Carbon content in the water, setting
the plot title and color
plt.setp(ax[2,0], title='Organic_carbon')
sns.boxplot(water_potability['Organic_carbon'], orient='h',
ax=ax[2,0], color='red')

# Creating a box plot for Trihalomethanes levels in the water, setting
the plot title and color
plt.setp(ax[2,1], title='Trihalomethanes')
sns.boxplot(water_potability['Trihalomethanes'], orient='h',
ax=ax[2,1], color='blue')

# Creating a box plot for Turbidity of the water, setting the plot
title and color
plt.setp(ax[2,2], title='Turbidity')
sns.boxplot(water_potability['Turbidity'], orient='h', ax=ax[2,2],
color='red')

# Adjusting layout to prevent overlap of elements
plt.tight_layout()
```




```
# Import necessary visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Initialize a 3x3 grid of subplots with a specific figure size
fig, ax = plt.subplots(3, 3, figsize=(15, 8))

# Plot a histogram with a Kernel Density Estimate (KDE) overlay for pH
values
plt.setp(ax[0,0], title='PH')  # Set the title for the first subplot
sns.histplot(water_potability['ph'].dropna(), ax=ax[0,0], color='red',
kde=True)  # Plot using red color

# Plot a histogram with a KDE overlay for Hardness values
plt.setp(ax[0,1], title='Hardness')  # Set the title for the second
subplot
sns.histplot(water_potability['Hardness'].dropna(), ax=ax[0,1],
color='blue', kde=True)  # Plot using blue color

# Plot a histogram with a KDE overlay for Solids content
plt.setp(ax[0,2], title='Solids')  # Set the title for the third
subplot
sns.histplot(water_potability['Solids'].dropna(), ax=ax[0,2],
color='red', kde=True)  # Plot using red color

# Plot a histogram with a KDE overlay for Chloramines levels
plt.setp(ax[1,0], title='Chloramines')  # Set the title for the fourth
subplot
sns.histplot(water_potability['Chloramines'].dropna(), ax=ax[1,0],
color='blue', kde=True)  # Plot using blue color
```

```
# Plot a histogram with a KDE overlay for Sulfate levels
plt.setp(ax[1,1], title='Sulfate')  # Set the title for the fifth
subplot
sns.histplot(water_potability['Sulfate'].dropna(), ax=ax[1,1],
color='red', kde=True)  # Plot using red color

# Plot a histogram with a KDE overlay for Conductivity
plt.setp(ax[1,2], title='Conductivity')  # Set the title for the sixth
subplot
sns.histplot(water_potability['Conductivity'].dropna(), ax=ax[1,2],
color='blue', kde=True)  # Plot using blue color

# Plot a histogram with a KDE overlay for Organic Carbon content
plt.setp(ax[2,0], title='Organic_carbon')  # Set the title for the
seventh subplot
sns.histplot(water_potability['Organic_carbon'].dropna(), ax=ax[2,0],
color='red', kde=True)  # Plot using red color

# Plot a histogram with a KDE overlay for Trihalomethanes levels
plt.setp(ax[2,1], title='Trihalomethanes')  # Set the title for the
eighth subplot
sns.histplot(water_potability['Trihalomethanes'].dropna(), ax=ax[2,1],
color='blue', kde=True)  # Plot using blue color

# Plot a histogram with a KDE overlay for Turbidity
plt.setp(ax[2,2], title='Turbidity')  # Set the title for the ninth
subplot
sns.histplot(water_potability['Turbidity'].dropna(), ax=ax[2,2],
color='red', kde=True)  # Plot using red color

# Adjust the layout to prevent overlap of plot elements
plt.tight_layout()
```

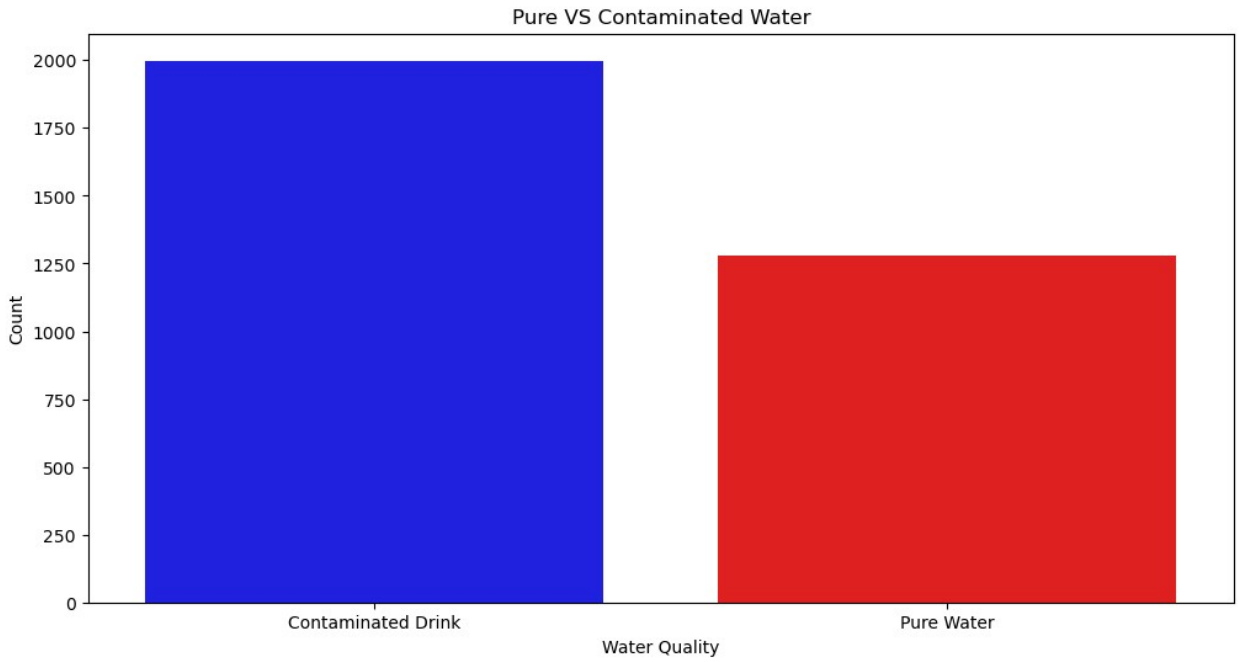


Histograms overlaid with kernel density estimates for various water quality parameters, each colored differently. The bell-shaped curves indicate that most variables are normally distributed. Parameters like pH, Hardness, and Organic Carbon show symmetrical distributions around their central values, suggesting no skewness. Solids and Conductivity, while also appearing normally distributed, have wider spreads indicating greater variability. The distributions suggest that the water quality metrics are fairly consistent with what might be expected in a typical water sample dataset, with no extreme deviations from normality.

#####EXPLORATION OF TARGET VARIABLE####

```
# Import the necessary visualization library
import seaborn as sns
import matplotlib.pyplot as plt

# Set up a figure with specified dimensions
fig, ax = plt.subplots(figsize=(12, 6))

# Calculate the counts of each category in the 'Potability' column,
renaming the categories for clarity
potability_counts =
water_potability['Potability'].value_counts().rename({1: 'Pure Water',
0: 'Contaminated Drink'})

# Create a bar plot showing the number of samples classified as 'Pure
Water' and 'Contaminated Drink'
sns.barplot(x=potability_counts.index, y=potability_counts.values,
palette=['blue', 'red'])

# Setting the title of the plot to indicate what the chart represents
plt.title('Pure VS Contaminated Water')
```

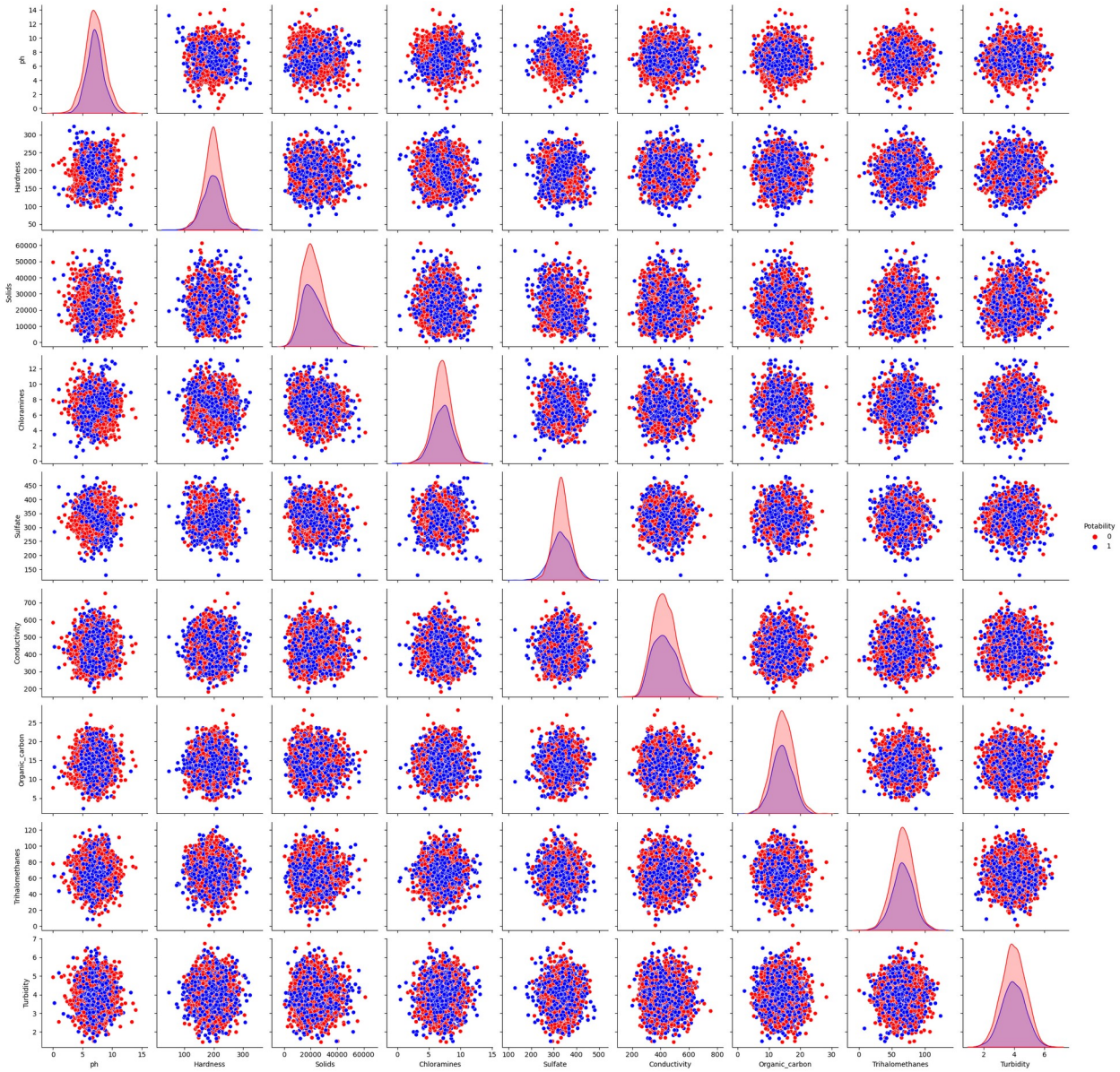
```
# Labeling the x-axis as 'Water Quality'
plt.xlabel('Water Quality')

# Labeling the y-axis as 'Count' to indicate what the numbers
represent
plt.ylabel('Count')

# Ensuring the x-axis tick labels accurately represent the data
categories
plt.xticks(ticks=[0, 1], labels=['Contaminated Drink', 'Pure Water'])

# Display the plot
plt.show()
```




```
# Import the necessary visualization libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Defining a custom color palette for the 'Potability' categories
# where '0' represents 'Contaminated Drink' and '1' represents 'Pure
Water'
palette = {0: "red", 1: "blue"}

# Creating a pair plot of all numerical columns in the dataset
# 'Potability' is used as the hue to distinguish between the two
categories
sns.pairplot(water_potability, hue='Potability', palette=palette)
```

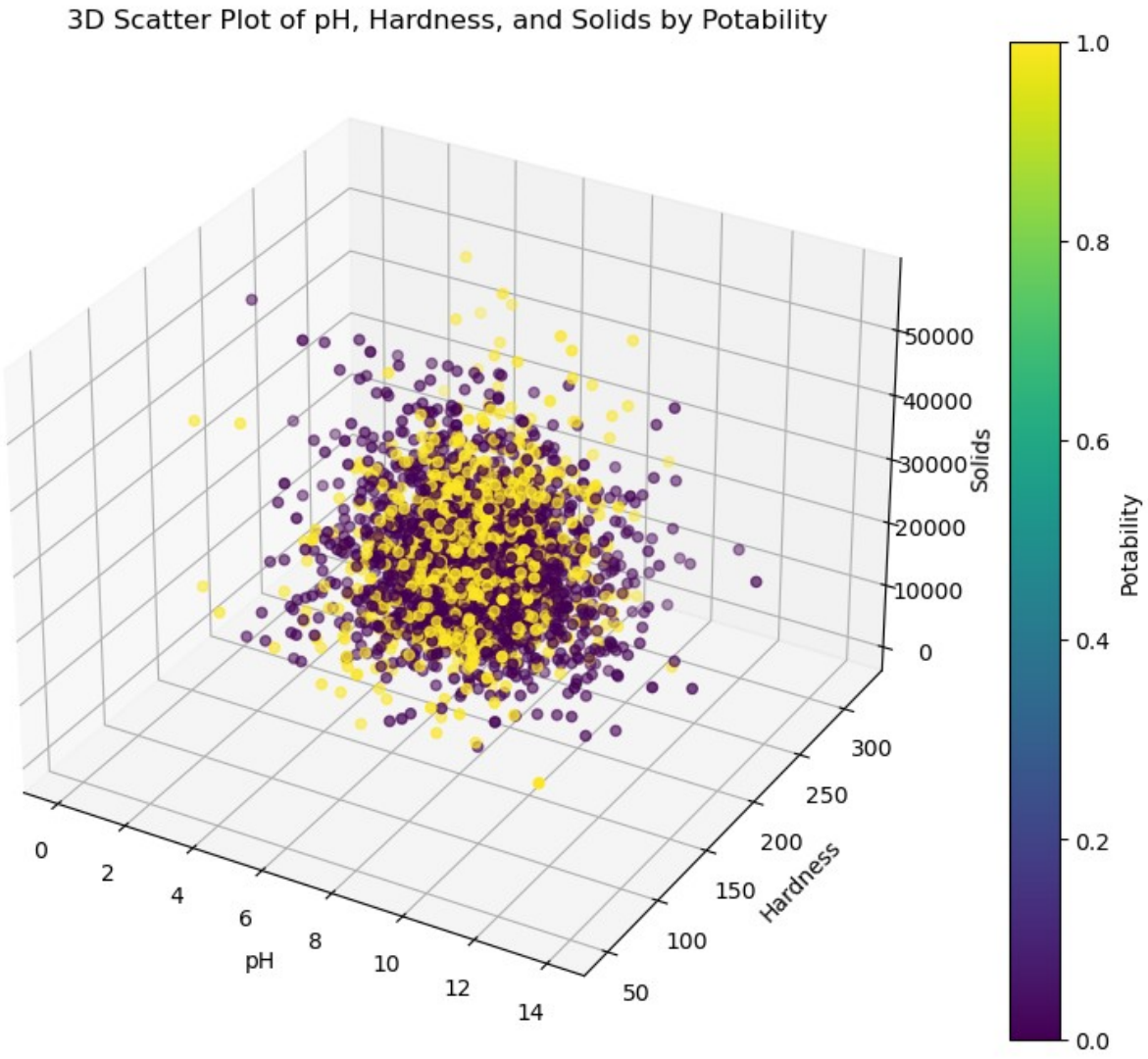
```
# Display the plot on screen
plt.show()
```

```
C:\ProgramData\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118:
UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```




```
# Import the necessary modules for 3D plotting
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Creating a new figure for 3D visualization, specifying its size
fig = plt.figure(figsize=(10, 8))
```

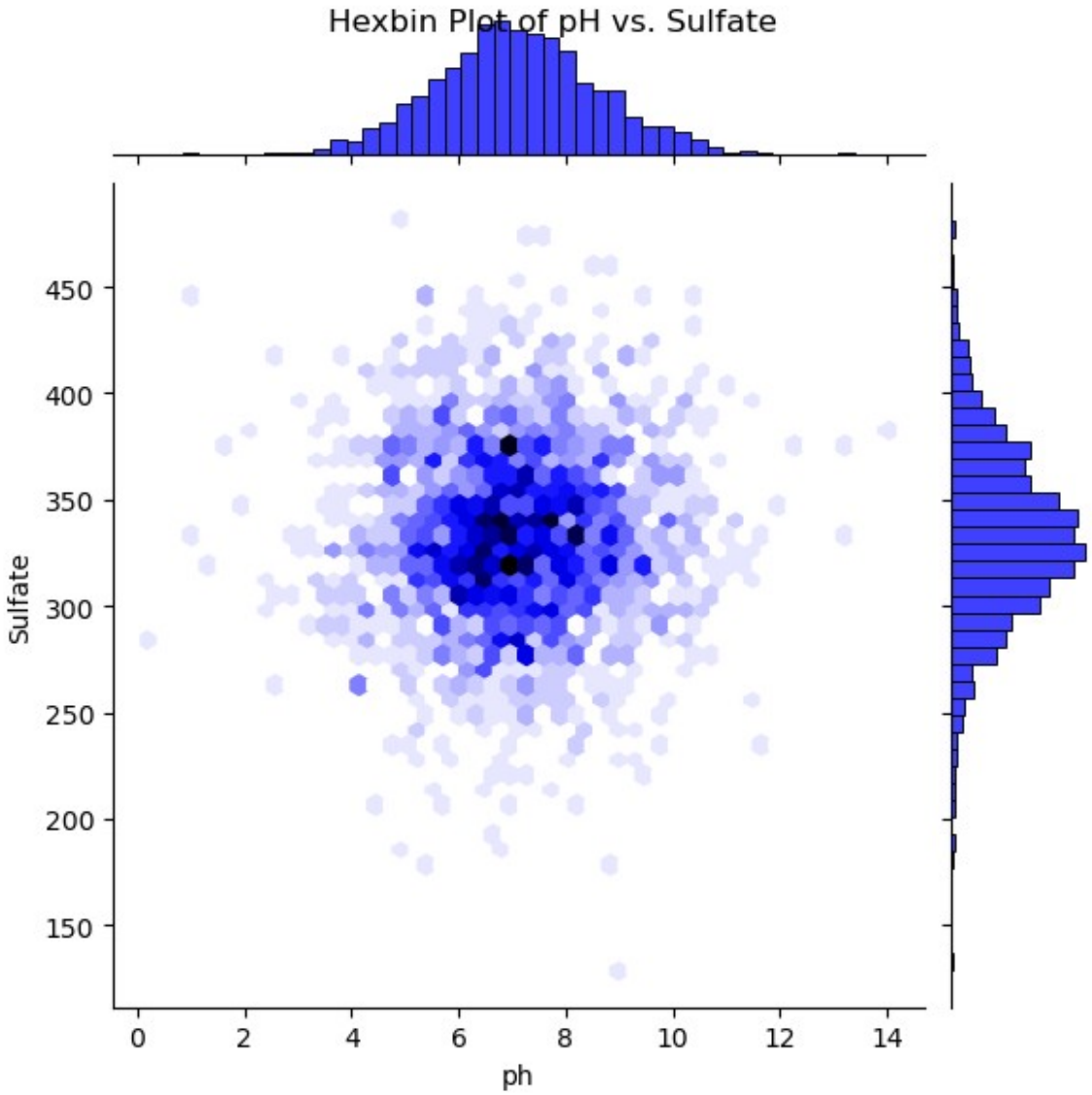
```
# Adding a 3D subplot to the figure
ax = fig.add_subplot(111, projection='3d')

# Assigning data from the DataFrame to variables for plotting
x = water_potability['ph']         # pH levels will be on the x-axis
y = water_potability['Hardness']   # Hardness levels will be on the y-
axis
z = water_potability['Solids']     # Solids content will be on the z-
axis
c = water_potability['Potability'] # This will determine the color
(potability status)

# Creating a scatter plot in 3D space, with color encoding by
'Potability'
scatter = ax.scatter(x, y, z, c=c, cmap='viridis', marker='o')

# Adding a color bar to the plot for better interpretation of
potability
cbar = plt.colorbar(scatter, ax=ax)
cbar.set_label('Potability')  # Labeling the color bar

# Setting labels for each axis to clarify what each represents
ax.set_xlabel('pH')
ax.set_ylabel('Hardness')
ax.set_zlabel('Solids')

# Adding a title to the plot for descriptive purposes
plt.title('3D Scatter Plot of pH, Hardness, and Solids by Potability')

# Displaying the plot
plt.show()
```




```
# Importing the necessary visualization library
import seaborn as sns
import matplotlib.pyplot as plt

# Setting up the figure, specifying its size for better visualization
plt.figure(figsize=(8, 6))

# Using seaborn's jointplot function to create a hexbin plot
# Hexbin plots are useful for visualizing the relationship between two numeric variables
# when the data points are too dense to plot individually
sns.jointplot(x='ph', y='Sulfate', kind='hex', data=water_potability, color='blue')

# Adding a super title to the plot to give more context, positioned above the plot
plt.suptitle('Hexbin Plot of pH vs. Sulfate')
```

```
# Displaying the plot to the screen
plt.show()
```

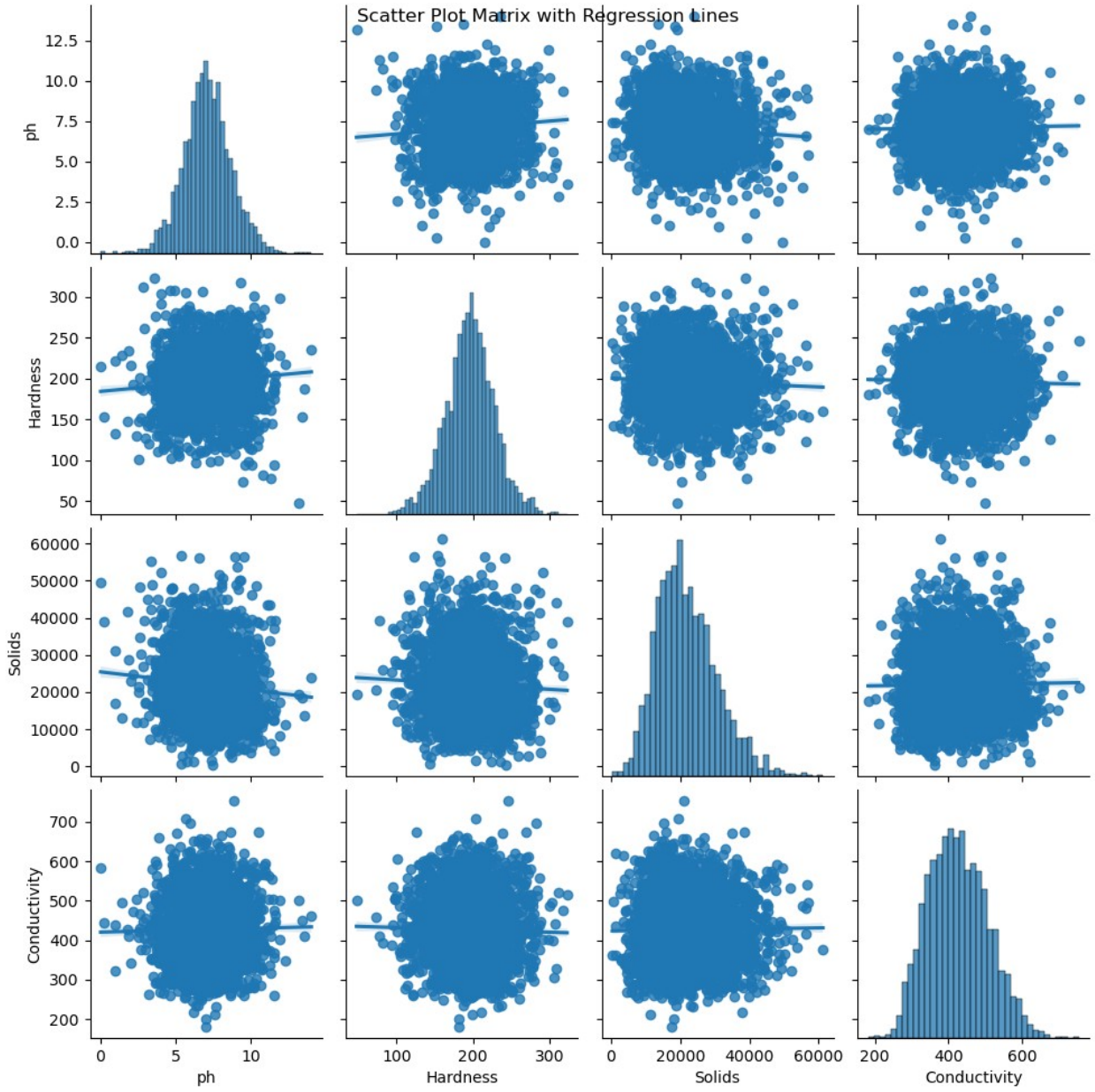
```
<Figure size 800x600 with 0 Axes>
```




```
# Importing the seaborn library for advanced plotting
import seaborn as sns
import matplotlib.pyplot as plt

# Using the pairplot function from seaborn to create a scatter plot
matrix
# The function takes a subset of the DataFrame columns: pH, Hardness,
Solids, and Conductivity
# The 'kind' parameter 'reg' adds a linear regression fit to each
```

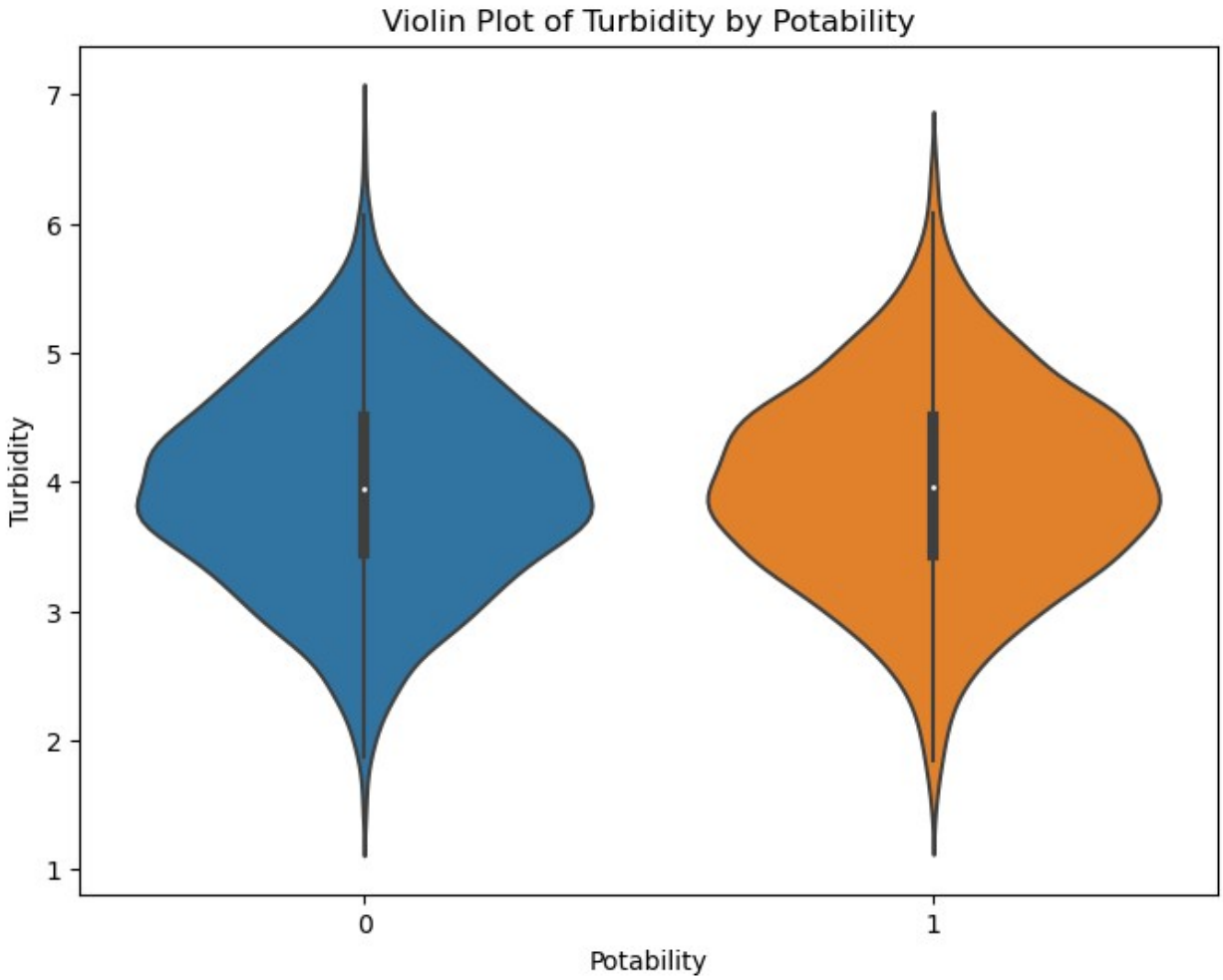
```
scatter plot to illustrate relationships
sns.pairplot(water_potability[['ph', 'Hardness', 'Solids',
'Conductivity']], kind='reg')

# Adding a super title to the plot, placed slightly above the plot
matrix for clarity
plt.suptitle('Scatter Plot Matrix with Regression Lines')

# Displaying the resulting plot
plt.show()
```

```
C:\ProgramData\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118:
UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```



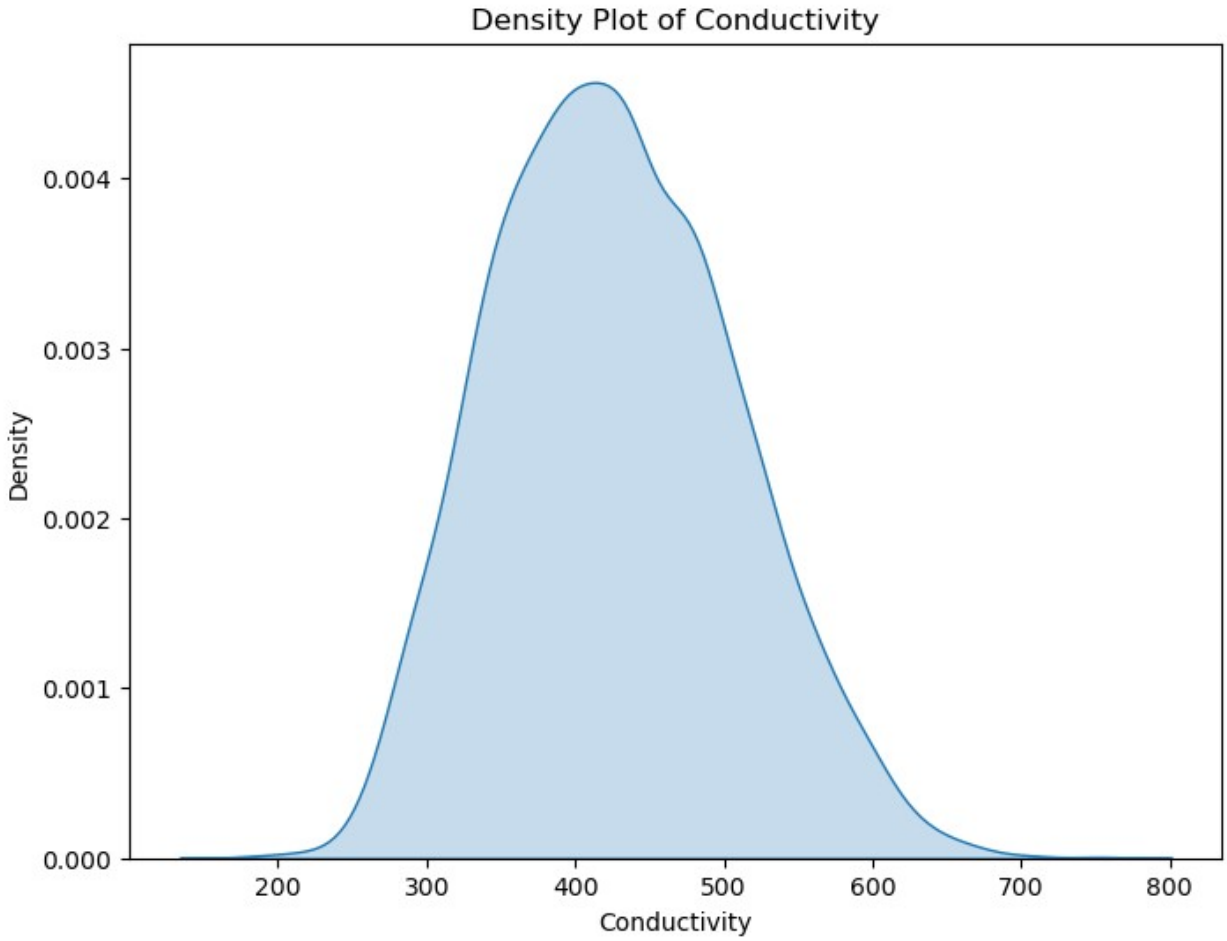

```
# Importing the necessary visualization library
import seaborn as sns
import matplotlib.pyplot as plt

# Setting up the figure, specifying the size to ensure the plot is
clearly visible
plt.figure(figsize=(8, 6))

# Creating a violin plot using seaborn. This plot type is effective
for comparing the distribution of a variable across different
categories.
# 'x' axis represents different categories of potability (0 or 1),
```

```
# 'y' axis represents the turbidity values of the water samples.
sns.violinplot(x='Potability', y='Turbidity', data=water_potability)

# Adding a title to the plot to provide a clear, descriptive header
plt.title('Violin Plot of Turbidity by Potability')

# Labeling the x-axis as 'Potability' to indicate what the categories represent
plt.xlabel('Potability')

# Labeling the y-axis as 'Turbidity' to clearly state the measured parameter
plt.ylabel('Turbidity')

# Displaying the plot
plt.show()
```

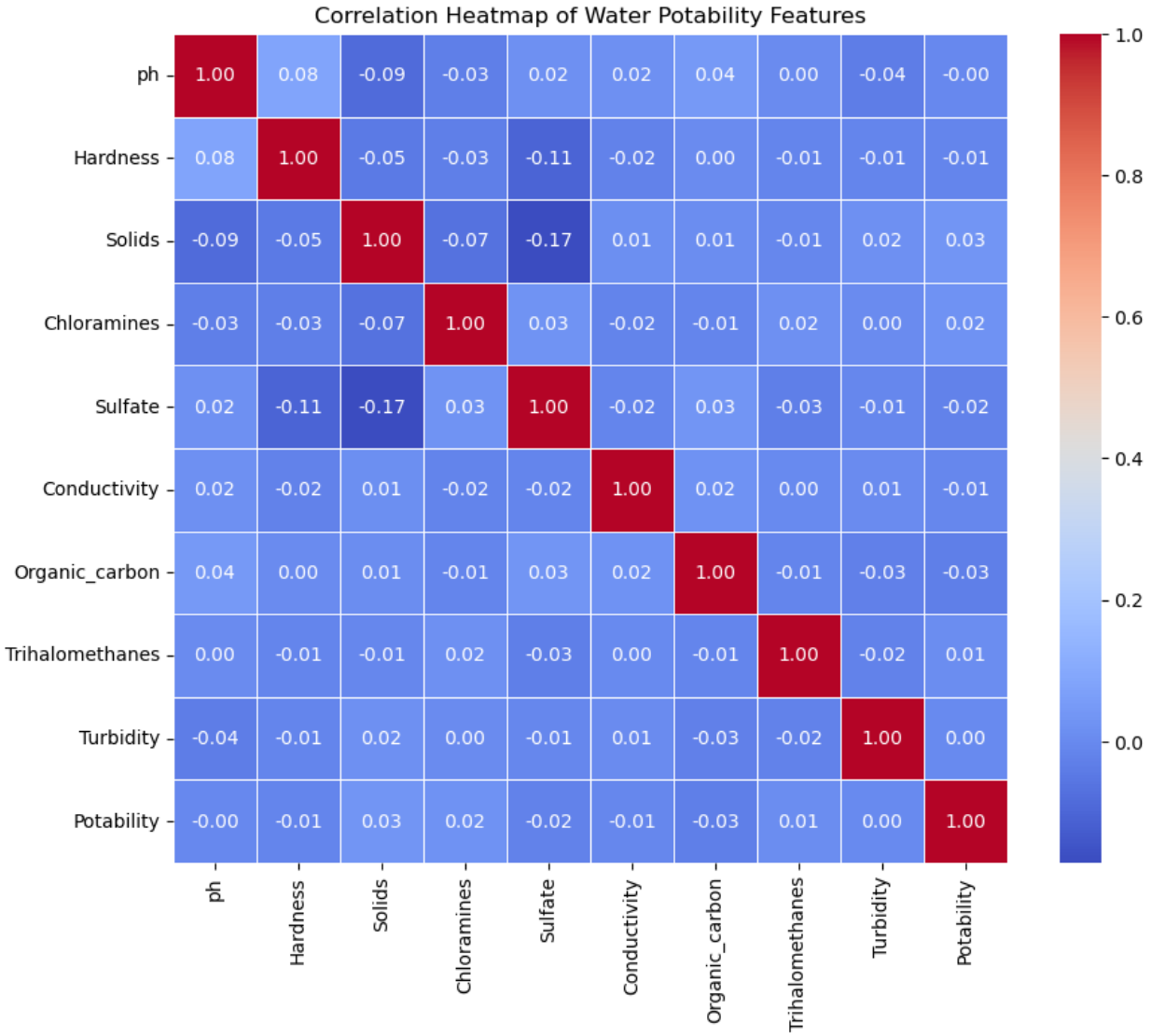



```
# Importing the necessary visualization library
import seaborn as sns
```

```
import matplotlib.pyplot as plt

# Setting up the figure, specifying its size to ensure the plot is clearly visible and well-proportioned
plt.figure(figsize=(8, 6))

# Creating a density plot using seaborn's kdeplot function. This function estimates and plots the density of observations in one dimension.
# The 'fill' parameter set to True fills the area under the density curve, enhancing visual clarity and emphasis.
sns.kdeplot(water_potability['Conductivity'], fill=True)

# Adding a title to the plot to provide a clear, descriptive header
plt.title('Density Plot of Conductivity')

# Labeling the x-axis as 'Conductivity' to indicate what the variable represents
plt.xlabel('Conductivity')

# Labeling the y-axis as 'Density' to show the estimated density of the data
plt.ylabel('Density')

# Displaying the plot
plt.show()
```

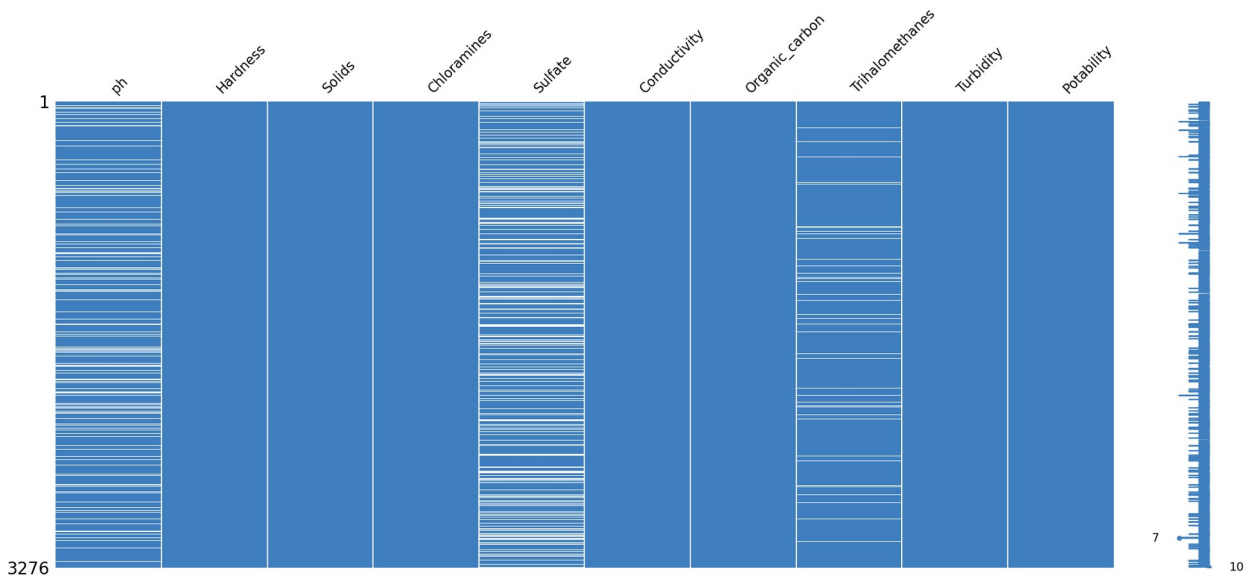



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = water_potability.corr()  # Compute the correlation
matrix

# Creating a heatmap to visually represent the correlation matrix
plt.figure(figsize=(10, 8))  # Setting the size of the figure
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f",
linewidths=.5)  # Using the 'coolwarm' color map to distinguish
between positive and negative correlations
plt.title('Correlation Heatmap of Water Potability Features')  #
Adding a title for clarity and context
plt.show()  # Displaying the plot
```




```
# Import the missingno library as msno for missing data visualization and matplotlib for plotting
import missingno as msno
import matplotlib.pyplot as plt

# The msno.matrix function provides a visual representation of the nullity of the data
# Nullity here refers to the presence of missing values in the data
# The visualization will plot the DataFrame's sample size along the y-axis and variables along the x-axis
# Each data point that represents a missing value is marked in white and non-missing values are colored
# The color parameter (0.25, 0.5, 0.75) sets a custom color in RGB format for non-missing data points
msno.matrix(water_potability, color=(0.25, 0.5, 0.75))
```

```
# Display the visual plot. The plot helps in quickly identifying the
pattern of missingness in the dataset,
# which can inform subsequent data cleaning and preprocessing steps.
plt.show()
```



PRE-PROCESSING OF DATA

HANDLING OF PH MISSING VALUES

```
# HANDLING OF PH MISSING VALUES
# HANDLING OF PH MISSING VALUES
# Announcing the intention of the following code block with a print
statement
print('Conditional Statements to fill in the Missing Values of PH
Value Column')

# Calculating the mean pH value for the first condition:
# Non-potable water ('Potability' == 0) with 'Hardness' less than or
equal to 150
print("\n")
print('if Potability = 0 and Hardness <= 150')
condition_1_mean_ph = water_potability[(water_potability['Potability']
== 0) & (water_potability['Hardness'] <= 150)]['ph'].mean()
print(f"PH VALUE : {condition_1_mean_ph:.4f}")

# Calculating the mean pH value for the second condition:
# Non-potable water ('Potability' == 0) with 'Hardness' greater than
150
```

```
print("\n")
print('if Potability = 0 and Hardness > 150')
condition_2_mean_ph = water_potability[(water_potability['Potability']
== 0) & (water_potability['Hardness'] > 150)]['ph'].mean()
print(f"PH VALUE : {condition_2_mean_ph:.4f}")

# Calculating the mean pH value for the third condition:
# Potable water ('Potability' == 1) with 'Hardness' less than or equal
to 150
print("\n")
print('if Potability = 1 and Hardness <= 150')
condition_3_mean_ph = water_potability[(water_potability['Potability']
== 1) & (water_potability['Hardness'] <= 150)]['ph'].mean()
print(f"PH VALUE : {condition_3_mean_ph:.4f}")

# Calculating the mean pH value for the fourth condition:
# Potable water ('Potability' == 1) with 'Hardness' greater than 150
print("\n")
print('if Potability = 1 and Hardness > 150')
condition_4_mean_ph = water_potability[(water_potability['Potability']
== 1) & (water_potability['Hardness'] > 150)]['ph'].mean()
print(f"PH VALUE : {condition_4_mean_ph:.4f}")

# Imputing missing pH values based on the defined conditions
for x in range(len(water_potability)):
    if pd.isnull(water_potability.at[x, 'ph']):
        if water_potability.at[x, 'Potability'] == 0:
            if water_potability.at[x, 'Hardness'] <= 150:
                water_potability.at[x, 'ph'] = condition_1_mean_ph
            else:
                water_potability.at[x, 'ph'] = condition_2_mean_ph
        else:
            if water_potability.at[x, 'Hardness'] <= 150:
                water_potability.at[x, 'ph'] = condition_3_mean_ph
            else:
                water_potability.at[x, 'ph'] = condition_4_mean_ph
```

```
Conditional Statements to fill in the Missing Values of PH Value
Column


if Potability = 0 and Hardness <= 150
PH VALUE : 6.7220


if Potability = 0 and Hardness > 150
PH VALUE : 7.1125


if Potability = 1 and Hardness <= 150
```

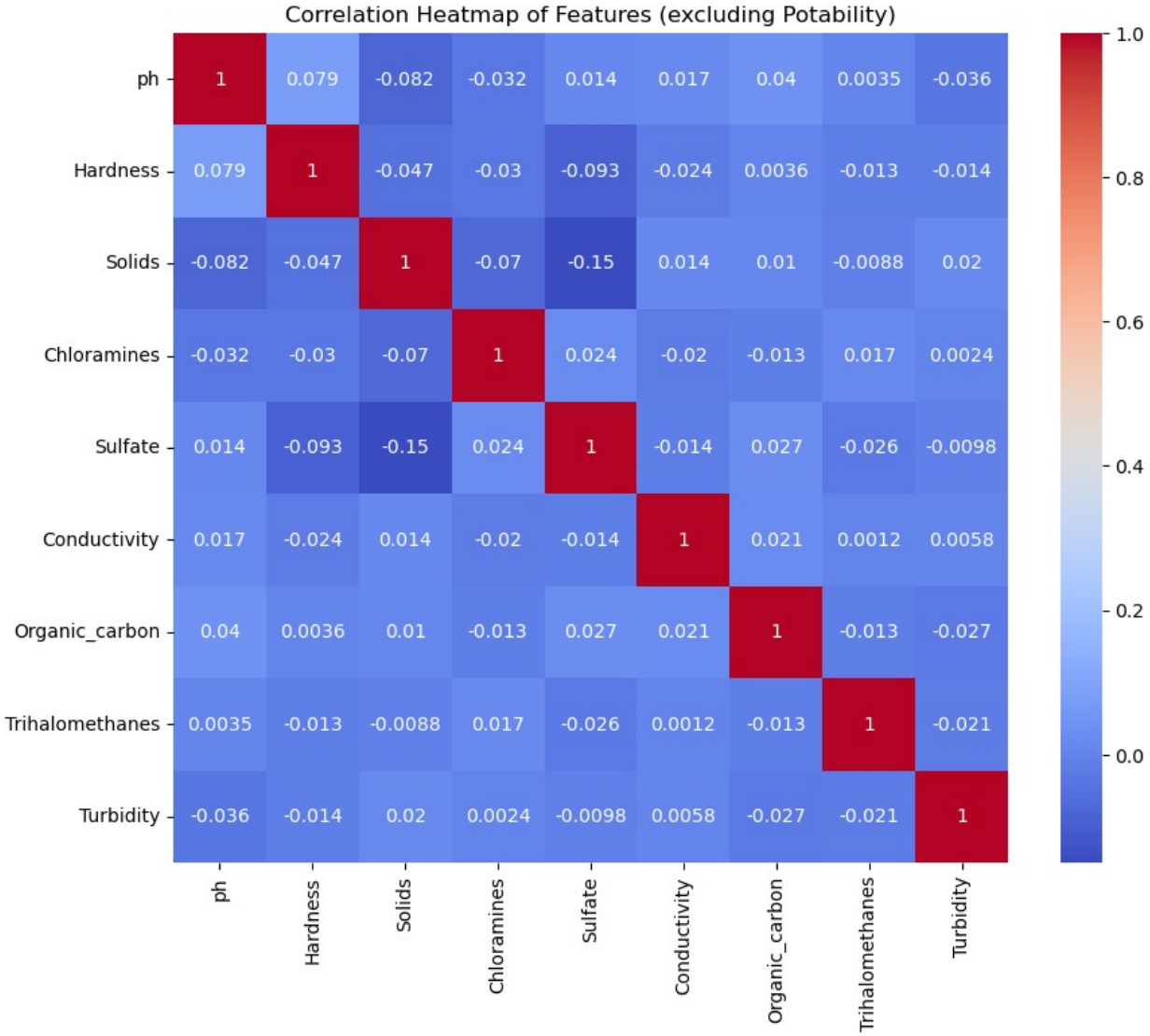
```
PH VALUE : 7.0982


if Potability = 1 and Hardness > 150
PH VALUE : 7.0714
```

```python
# HANDLING OF Trihalomethanes MISSING VALUES
# Importing the pandas library for data manipulation
import pandas as pd

# First, we calculate the median value for the 'Trihalomethanes'
column.
# The median is often used to fill in missing values because it is
less sensitive to outliers than the mean.
median_thm = water_potability['Trihalomethanes'].median()

# Using the calculated median to fill in missing values in the
'Trihalomethanes' column.
# The 'inplace=True' parameter modifies the original DataFrame
directly.
water_potability['Trihalomethanes'].fillna(median_thm, inplace=True)

# Print statement to confirm filling operation
print("Missing 'Trihalomethanes' values filled with median:",
median_thm)

# Similar steps are followed for handling missing values in the
'Sulfate' column:
# Calculating the median for 'Sulfate'.
median_sulfate = water_potability['Sulfate'].median()

# Filling missing values in the 'Sulfate' column with the calculated
median.
# This is also done directly in the DataFrame without creating a copy.
water_potability['Sulfate'].fillna(median_sulfate, inplace=True)

# Print statement to confirm filling operation
print("Missing 'Sulfate' values filled with median:", median_sulfate)
```

```
Missing 'Trihalomethanes' values filled with median: 66.62248509808484
Missing 'Sulfate' values filled with median: 333.073545745888
```

```python
# Import the necessary visualization libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Calculating the correlation matrix for the dataset, excluding the
'Potability' column
corr_matrix = water_potability.drop('Potability', axis=1).corr()

# Creating a heatmap to visually represent the correlation matrix
```

```
plt.figure(figsize=(10, 8))  # Setting the size of the figure
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')  # Using the 'coolwarm' color map to distinguish between positive and negative correlations

# Adding a title to the heatmap for clarity and context
plt.title('Correlation Heatmap of Features (excluding Potability)')

# Displaying the plot
plt.show()
```



CHECKING FOR THE INFO, MEAN OF THE DATASET AFTER IMPLEMENTING REPLACEMENT FOR THE MISSING VALUES.

```
# Using the .info() method on the 'water_potability' DataFrame to get
a concise summary of the DataFrame after handling the missing values.
# This method is particularly useful for quickly understanding the
structure of the DataFrame after filling of missing value.
# It outputs details about:
#  - The class type of the data
#  - The range index, indicating the total number of entries
#  - A list of all columns, along with the count of non-null values in
each column and the data type of each column
#  - The number of columns under each data type
#  - The memory usage of the data held in the DataFrame, which can be
useful for managing computational resources.

# Execute the info method to display this information
water_potability.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3276 entries, 0 to 3275
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   ph               3276 non-null   float64
 1   Hardness         3276 non-null   float64
 2   Solids           3276 non-null   float64
 3   Chloramines      3276 non-null   float64
 4   Sulfate          3276 non-null   float64
 5   Conductivity     3276 non-null   float64
 6   Organic_carbon   3276 non-null   float64
 7   Trihalomethanes  3276 non-null   float64
 8   Turbidity        3276 non-null   float64
 9   Potability       3276 non-null   int64
dtypes: float64(9), int64(1)
memory usage: 256.1 KB
```

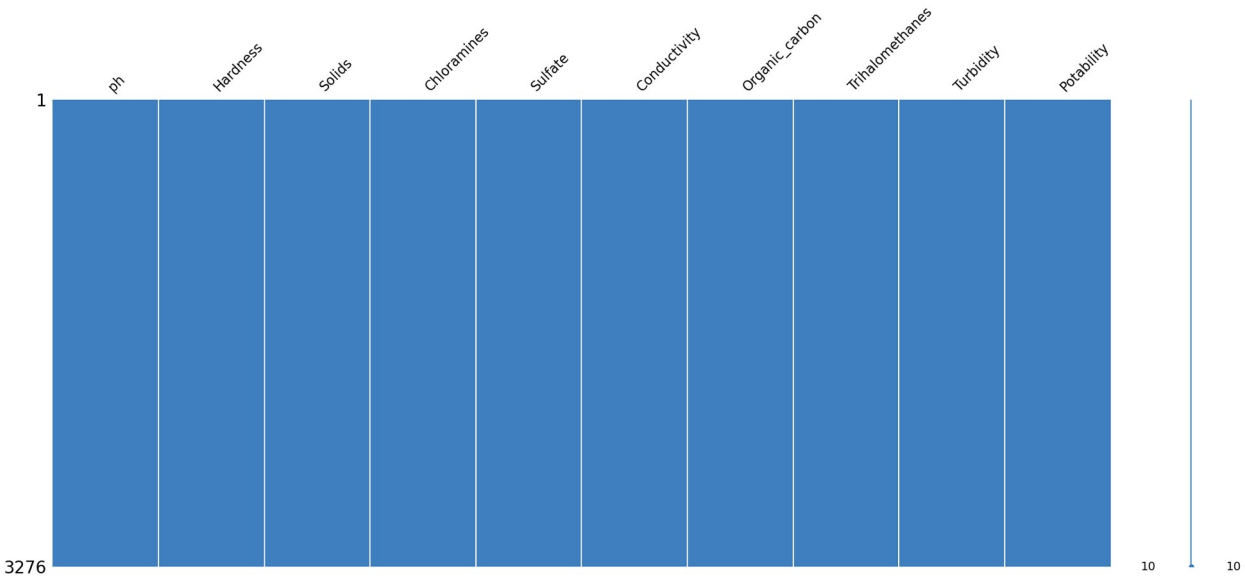
```
# The .mean() method calculates the average of all numeric columns in
the 'water_potability' DataFrame.
# This is useful for getting a quick statistical insight into the
central tendency of the data.
# Calculating the mean can help identify typical values and understand
the data's distribution better.

# Storing the mean values of each column in the variable 'mean_values'
mean_values = water_potability.mean()
```

```
# Printing the mean values to the console. This step helps in quickly
reviewing the average values of each column,
# which can be particularly useful in the early stages of data
analysis or when checking the effect of data cleaning steps.
print(mean_values)
```

```
ph                    7.081083
Hardness            196.369496
Solids            22014.092526
Chloramines           7.122277
Sulfate             333.608364
Conductivity        426.205111
Organic_carbon       14.284970
Trihalomethanes      66.407478
Turbidity             3.966786
Potability            0.390110
dtype: float64
```

VISUALIZE THE MISSING DATA WITH COLOR SCHEME AFTER HANDLING OF MISSING VALUES

```
# Visualize the missing data with a color scheme after handling of
missing values
msno.matrix(water_potability, color=(0.25, 0.5, 0.75))

# Display the plot
plt.show()
```




```
# To know if the data of missing values has been replaced
# Calculate the number of missing values in each column
missing_values = water_potability.isnull().sum()

# Print the number of missing values for each column
print(missing_values)
```

```
ph                 0
Hardness           0
Solids             0
Chloramines        0
Sulfate            0
Conductivity       0
Organic_carbon     0
Trihalomethanes    0
Turbidity          0
Potability         0
dtype: int64
```

ASSIGN NEW DATAFRAME AFTER PREPROCESS

```
# Assign to a new DataFrame
water_potability_processed = water_potability.copy()

# Let's prepare our feature set from the 'water_data' dataset. We're
selecting all the columns
# except the final one, which we'll assume is the target variable
we're aiming to predict.
```

```
features = water_potability_processed.iloc[:, :-1]

# Now, take a quick peek at the first few entries in our feature set
to ensure everything looks good.
features.head()
```

```
         ph    Hardness        Solids  Chloramines     Sulfate
Conductivity  \
0  7.112512  204.890455  20791.318981     7.300212  368.516441
564.308654
1  3.716080  129.422921  18630.057858     6.635246  333.073546
592.885359
2  8.099124  224.236259  19909.541732     9.275884  333.073546
418.606213
3  8.316766  214.373394  22018.417441     8.059332  356.886136
363.266516
4  9.092223  181.101509  17978.986339     6.546600  310.135738
398.410813

   Organic_carbon  Trihalomethanes  Turbidity
0       10.379783        86.990970   2.963135
1       15.180013        56.329076   4.500656
2       16.868637        66.420093   3.055934
3       18.436524       100.341674   4.628771
4       11.558279        31.997993   4.075075
```

DEFINITION OF INDEPENDENT AND DEPENDENT VARIABLES AFTER PREPROCESSED

```
#DEFINITION OF INDEPENDENT AND DEPENDENT VARIABLES

# Using .iloc to select all columns except the last one from the
DataFrame.
# The ":" indicates selection of all rows, and ":-1" means every
column except the last one.
# This is a common practice when you want to separate feature
variables (predictors) from the target variable.
X = features
y = water_potability_processed['Potability']  # Replace
'target_variable_name' with the actual name of your target column

# Displaying the first five rows of the selected columns to verify the
correct columns are included.
# This quick preview is useful to ensure that the DataFrame 'X'
contains only the feature variables,
```

```
# which are needed for input into machine learning models.
X.head()


# Displaying the first five rows of the isolated column to verify it's
the correct data.
# This step is useful for a quick check to ensure the data looks as
expected before proceeding with further analysis.
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Potability, dtype: int64
```

```
X.head()
```

```
         ph    Hardness        Solids  Chloramines     Sulfate
Conductivity  \
0  7.112512  204.890455  20791.318981     7.300212  368.516441
564.308654
1  3.716080  129.422921  18630.057858     6.635246  333.073546
592.885359
2  8.099124  224.236259  19909.541732     9.275884  333.073546
418.606213
3  8.316766  214.373394  22018.417441     8.059332  356.886136
363.266516
4  9.092223  181.101509  17978.986339     6.546600  310.135738
398.410813

   Organic_carbon  Trihalomethanes  Turbidity
0       10.379783        86.990970   2.963135
1       15.180013        56.329076   4.500656
2       16.868637        66.420093   3.055934
3       18.436524       100.341674   4.628771
4       11.558279        31.997993   4.075075
```

```
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Potability, dtype: int64
```

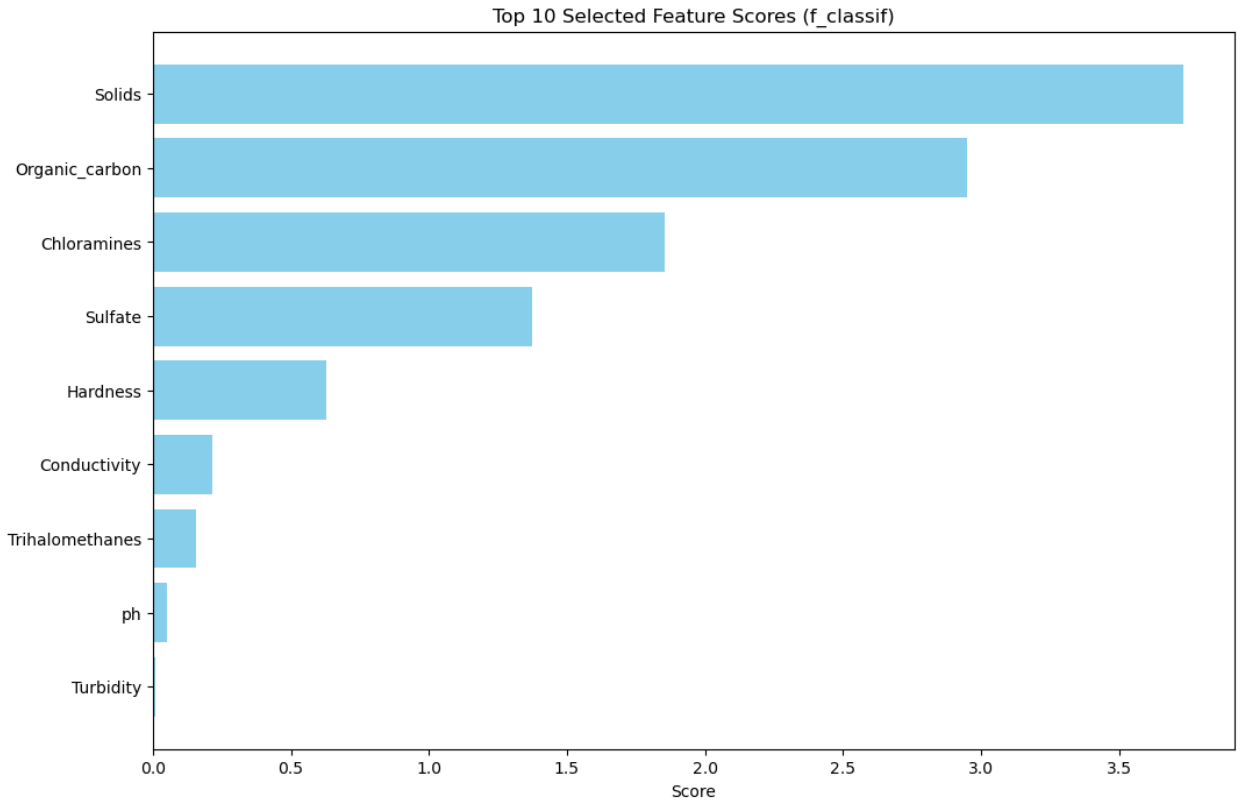
```
from sklearn.feature_selection import SelectKBest, f_classif

# Perform univariate feature selection
```

```
selector = SelectKBest(score_func=f_classif, k=10)  # Select top 5
features
X_selected = selector.fit_transform(X, y)

# Get selected feature indices
selected_feature_indices = selector.get_support(indices=True)
selected_features = X.columns[selected_feature_indices]

# Print selected feature names
print("Selected Features:", selected_features)
```

```
Selected Features: Index(['ph', 'Hardness', 'Solids', 'Chloramines',
'Sulfate', 'Conductivity',
       'Organic_carbon', 'Trihalomethanes', 'Turbidity'],
      dtype='object')
```

```
C:\Users\HP SPECTRE xt\AppData\Roaming\Python\Python311\site-packages\
sklearn\feature_selection\_univariate_selection.py:776: UserWarning:
k=10 is greater than n_features=9. All the features will be returned.
  warnings.warn(
```

```
import matplotlib.pyplot as plt

# Get feature scores from the selector
feature_scores = selector.scores_

# Create a DataFrame to store feature names and their corresponding
scores
feature_scores_df = pd.DataFrame({'Feature': X.columns, 'Score':
feature_scores})

# Sort features based on scores (descending order)
feature_scores_df = feature_scores_df.sort_values(by='Score',
ascending=False)

# Plotting feature scores for selected features
plt.figure(figsize=(12, 8))
plt.barh(feature_scores_df['Feature'][:10], feature_scores_df['Score']
[:10], color='skyblue')
plt.xlabel('Score')
plt.title('Top 10 Selected Feature Scores (f_classif)')
plt.gca().invert_yaxis()  # Invert y-axis to display top features at
the top
plt.show()
```



MODELING USING RADOM FOREST, DECISION TREE,XGBOOST, GRADIENTBOOSTCLASSIFIER AND SVM

RADOM FOREST ALGORITHM

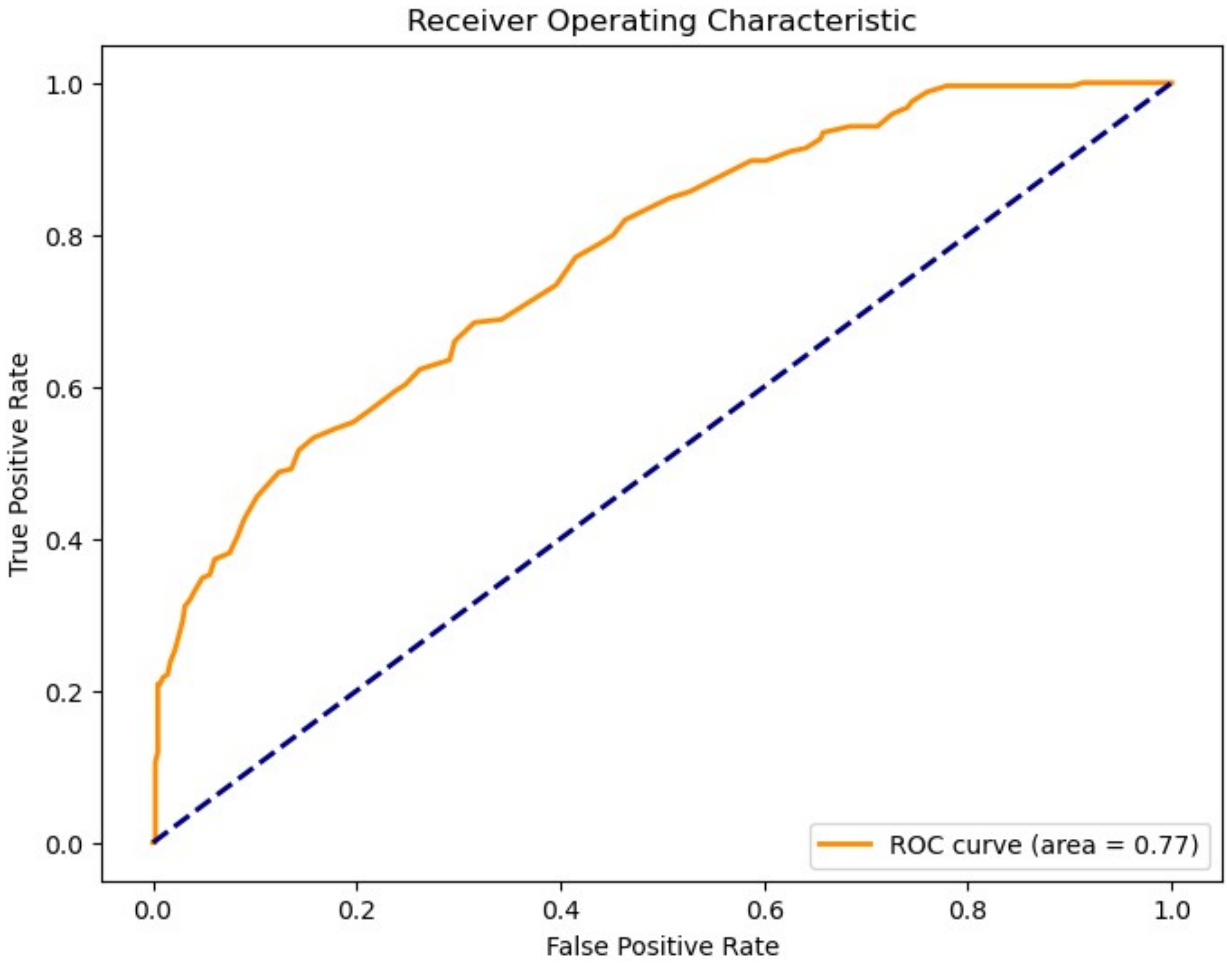
```
#RADOM FOREST

# Import necessary packages
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import RobustScaler
from imblearn.over_sampling import RandomOverSampler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, f1_score,
precision_score, recall_score,confusion_matrix, classification_report,
roc_curve, auc)
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score
```

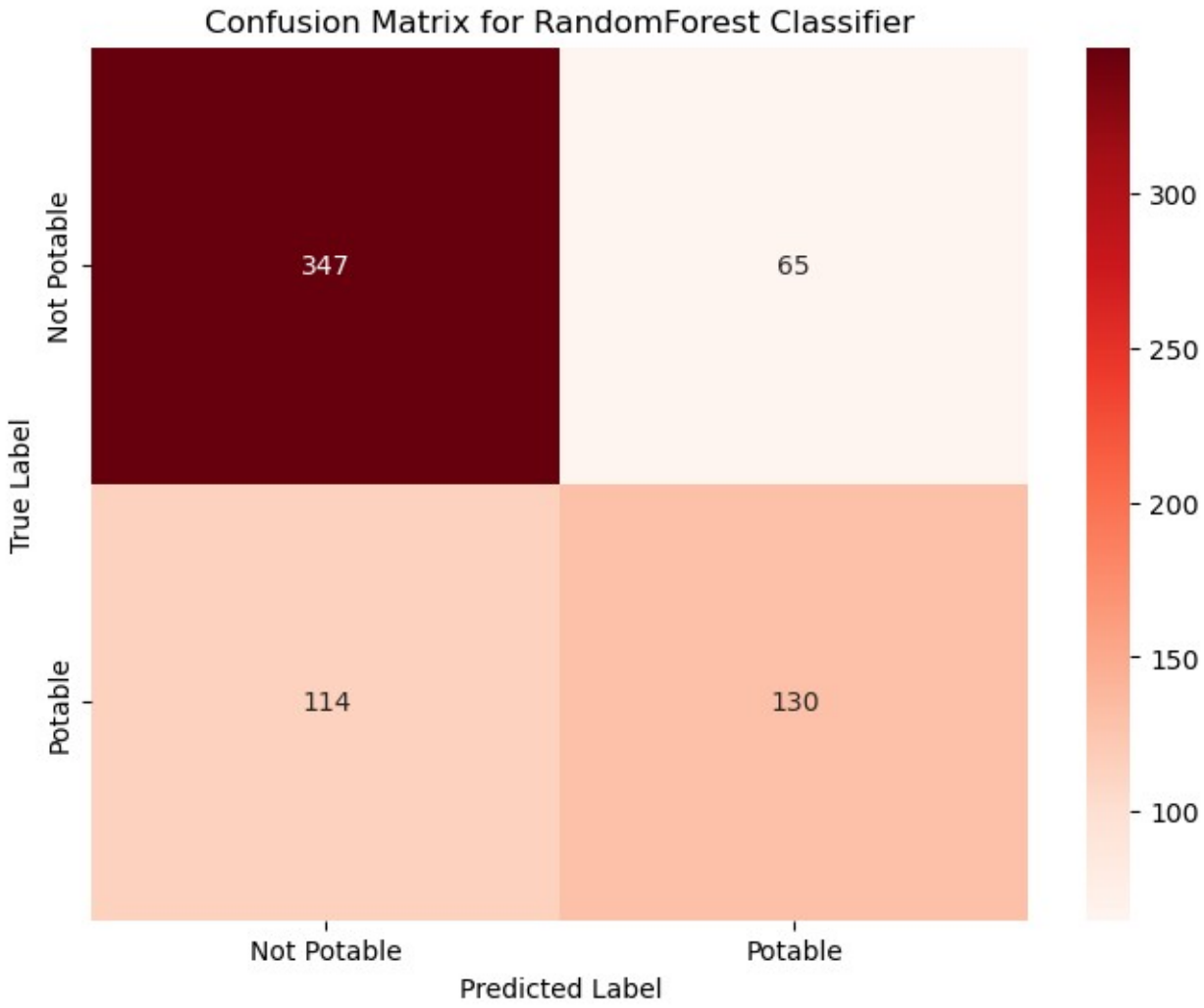
```
# Loading and splitting the dataset into training and testing sets to
evaluate the model's generalizability.
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Addressing class imbalance with RandomOverSampler for better model
fairness across classes.
ros = RandomOverSampler(sampling_strategy=1, random_state=42)
X_train_resampled, y_train_resampled = ros.fit_resample(X_train,
y_train)

# Normalizing data to reduce the impact of outliers, using
RobustScaler which is less sensitive to outliers.
robust_scaler = RobustScaler()
X_train_scaled = robust_scaler.fit_transform(X_train_resampled)
X_test_scaled = robust_scaler.transform(X_test)


# Training the RandomForestClassifier with default parameters to
establish a baseline.
random_forest = RandomForestClassifier(random_state=42)
random_forest.fit(X_train_scaled, y_train_resampled)

# Cross-validation
cv_scores = cross_val_score(random_forest, X_train_scaled,
y_train_resampled, cv=5, scoring='accuracy')
print(f'Cross-validation scores: {cv_scores}')
print(f'Mean cross-validation score: {cv_scores.mean():.4f}')


# Predicting on the resampled training data and the original test
data.
training_predictions = random_forest.predict(X_train_scaled)
testing_predictions = random_forest.predict(X_test_scaled)

# Evaluating model performance on both training and testing data to
detect any signs of overfitting.
training_accuracy = accuracy_score(y_train_resampled,
training_predictions)
testing_accuracy = accuracy_score(y_test, testing_predictions)
training_f1 = f1_score(y_train_resampled, training_predictions,
average='weighted')
testing_f1 = f1_score(y_test, testing_predictions, average='weighted')
training_precision = precision_score(y_train_resampled,
training_predictions, average='weighted')
testing_precision = precision_score(y_test, testing_predictions,
average='weighted')
```

```
training_recall = recall_score(y_train_resampled,
training_predictions, average='weighted')
testing_recall = recall_score(y_test, testing_predictions,
average='weighted')

# Outputting comprehensive performance metrics to thoroughly assess
the model's capabilities.
print('Random Forest Model Performance:')
print('\nTraining Metrics:')
print(f'Accuracy Score: {training_accuracy:.4f}')
print(f'F1 Score: {training_f1:.4f}')
print(f'Precision Score: {training_precision:.4f}')
print(f'Recall Score: {training_recall:.4f}')

print('\nTesting Metrics:')
print(f'Accuracy Score: {testing_accuracy:.4f}')
print(f'F1 Score: {testing_f1:.4f}')
print(f'Precision Score: {testing_precision:.4f}')
print(f'Recall Score: {testing_recall:.4f}')

# Providing a detailed classification report to further analyze the
model's performance across different classes.
print("\nClassification Report for Training Data:")
print(classification_report(y_train_resampled, training_predictions))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, testing_predictions))

# Visualizing the model's decision-making with an ROC curve, a
graphical plot that illustrates the diagnostic ability of a binary
classifier system.
fpr, tpr, thresholds = roc_curve(y_test,
random_forest.predict_proba(X_test_scaled)[:, 1])
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area =
%0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

# Displaying a confusion matrix to provide a summary of prediction
results on a classification problem.
cm = confusion_matrix(y_test, testing_predictions)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Reds', xticklabels=['Not
Potable', 'Potable'], yticklabels=['Not Potable', 'Potable'])
plt.title('Confusion Matrix for RandomForest Classifier')
```

```
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

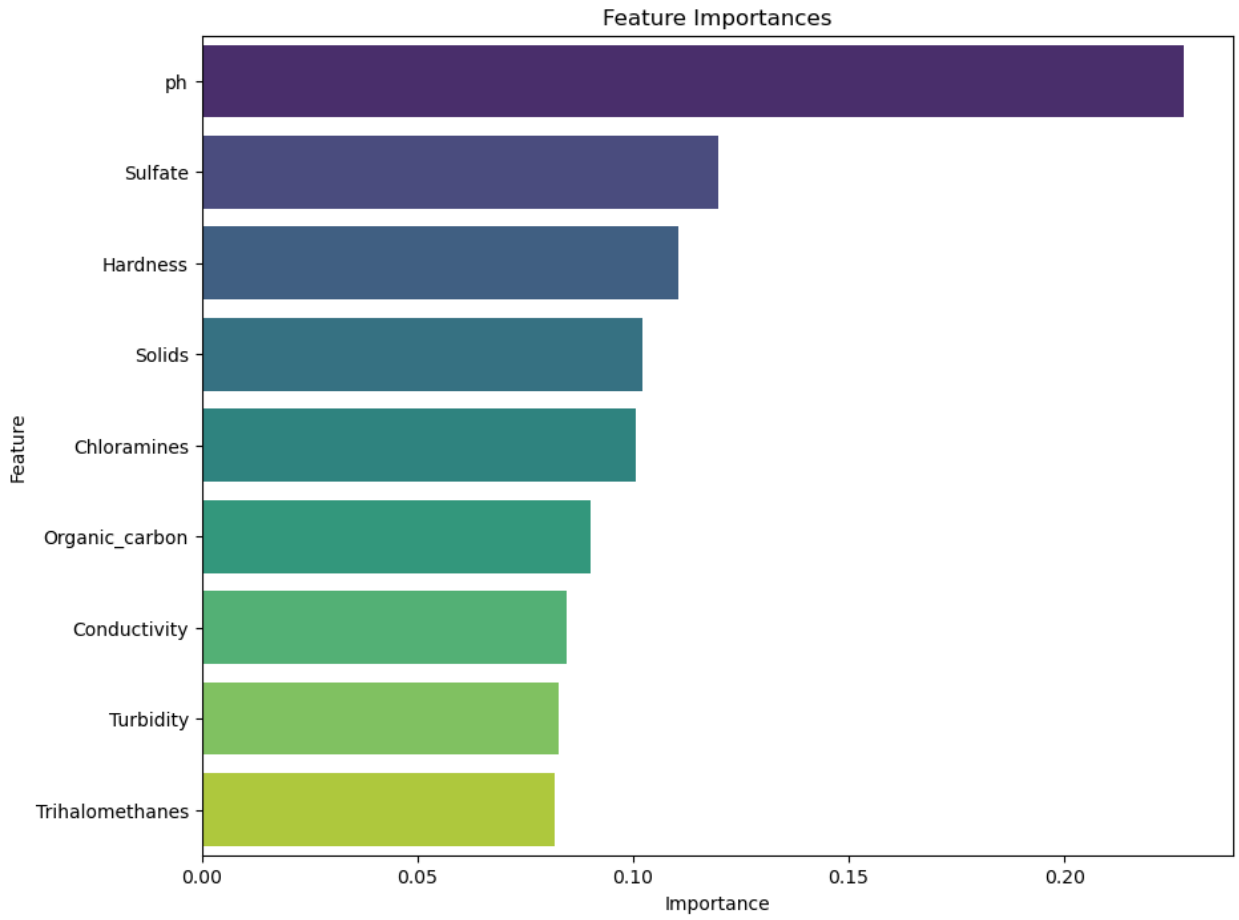
```
Cross-validation scores: [0.78582677 0.74645669 0.79652997 0.84542587
0.87066246]
Mean cross-validation score: 0.8090
Random Forest Model Performance:

Training Metrics:
Accuracy Score: 1.0000
F1 Score: 1.0000
Precision Score: 1.0000
Recall Score: 1.0000

Testing Metrics:
Accuracy Score: 0.7271
F1 Score: 0.7196
Precision Score: 0.7207
Recall Score: 0.7271

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1586
           1       1.00      1.00      1.00      1586

    accuracy                           1.00      3172
   macro avg       1.00      1.00      1.00      3172
weighted avg       1.00      1.00      1.00      3172


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.75      0.84      0.79       412
           1       0.67      0.53      0.59       244

    accuracy                           0.73       656
   macro avg       0.71      0.69      0.69       656
weighted avg       0.72      0.73      0.72       656
```


Receiver Operating Characteristic
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.77)




```
# Get feature importances from the trained model
feature_importances = random_forest.feature_importances_

# Create a DataFrame to store feature names and their corresponding
importances
feature_importance_df = pd.DataFrame({'Feature': X_train.columns,
'Importance': feature_importances})

# Sort features based on importance (descending order)
feature_importance_df =
feature_importance_df.sort_values(by='Importance', ascending=False)

# Plotting feature importances
plt.figure(figsize=(10, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df,
palette='viridis')
plt.title('Feature Importances')
plt.xlabel('Importance')
```

```
plt.ylabel('Feature')
plt.show()
```



DECISION TREE ALGORITHM

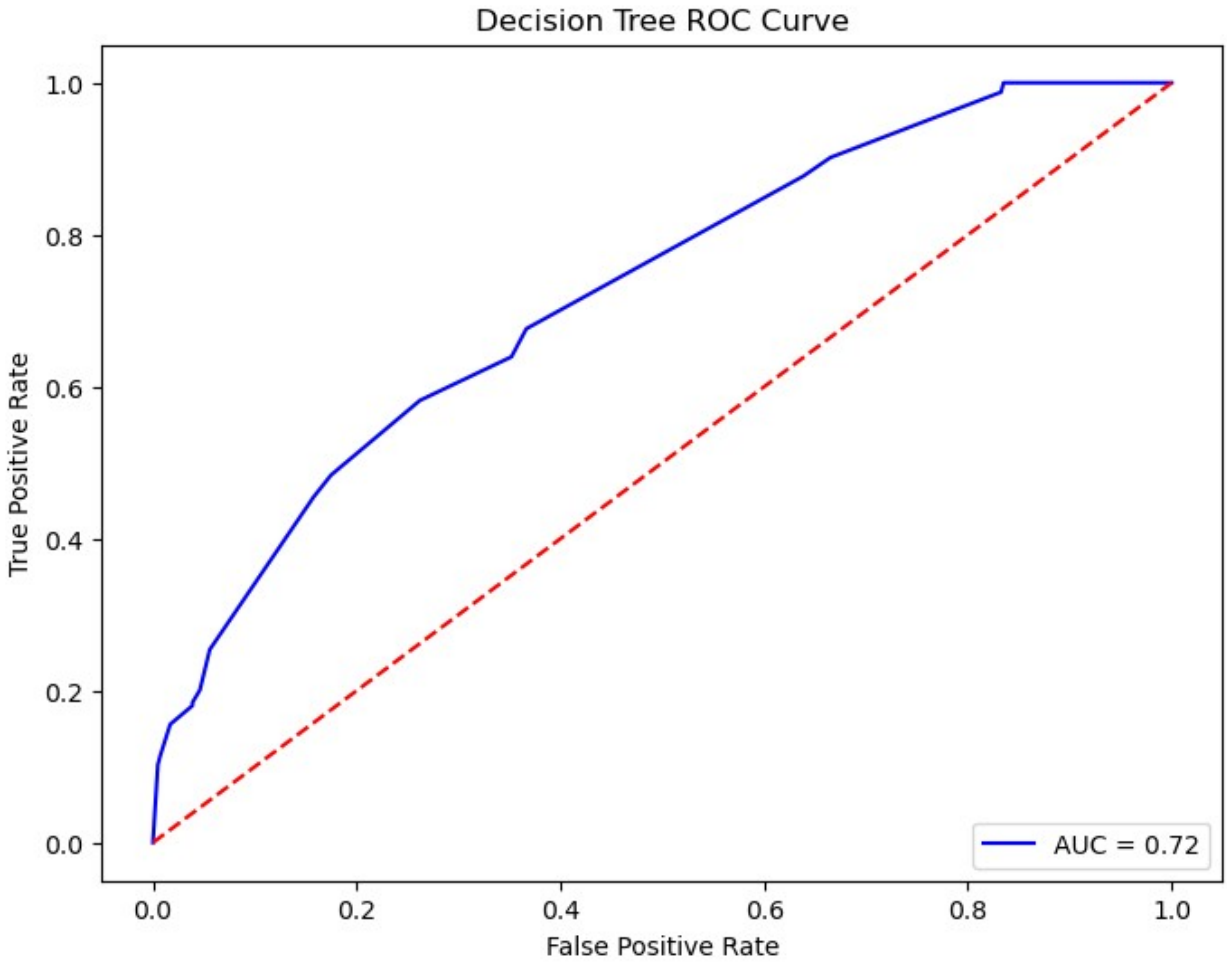
```
#DECISION TREE

import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, roc_curve, auc, confusion_matrix, classification_report
from imblearn.over_sampling import RandomOverSampler
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score



# Addressing class imbalance effectively with RandomOverSampler,
ensuring equal representation for all classes.
ros = RandomOverSampler(random_state=42)
```

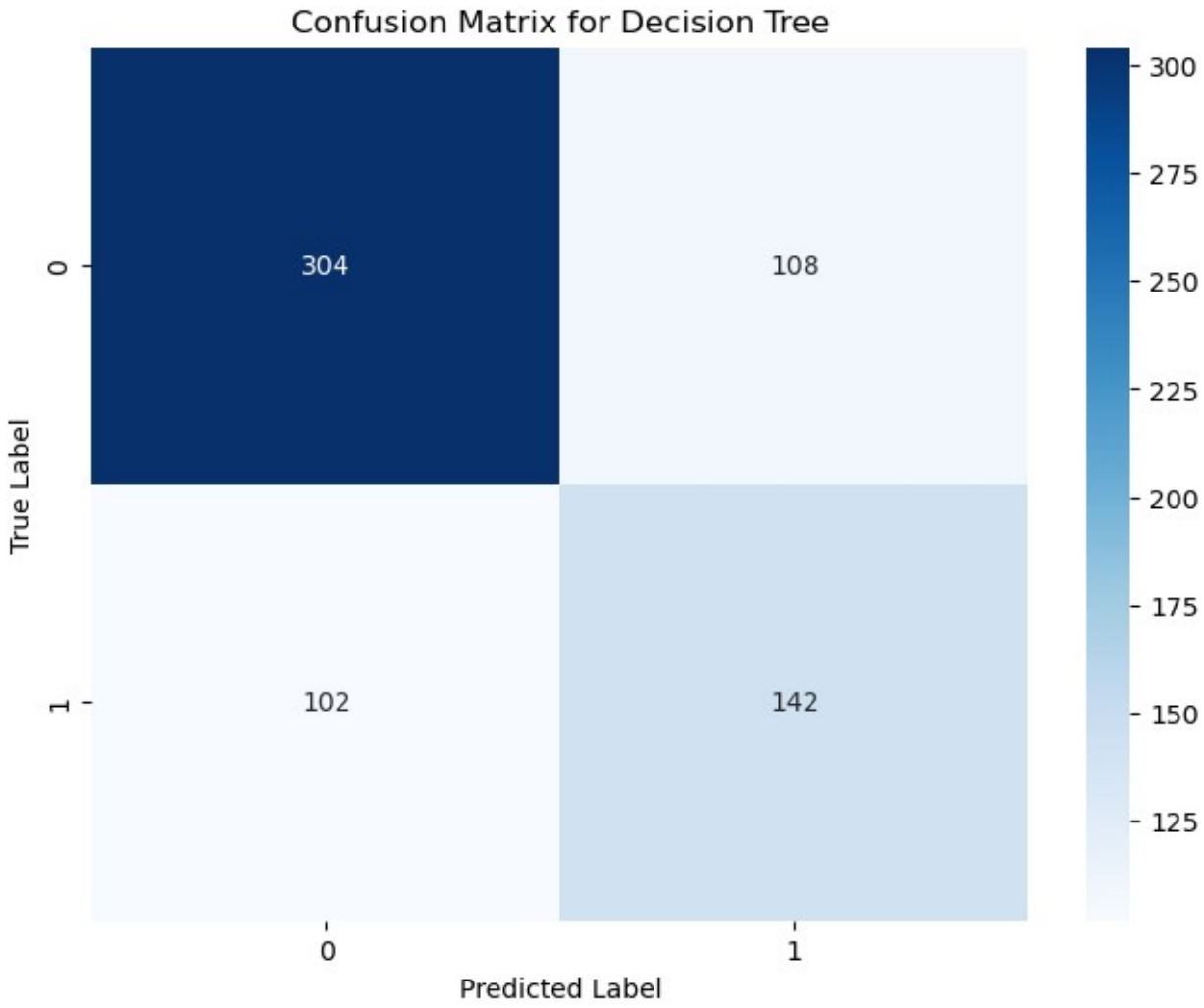
```
X_train_resampled, y_train_resampled = ros.fit_resample(X_train,
y_train)

# Normalizing data to ensure that the model's performance is not
skewed by the scale of the data.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_resampled)
X_test_scaled = scaler.transform(X_test)

# Employing a Decision Tree Classifier, configuring it with a modest
depth to prevent overfitting.
decision_tree = DecisionTreeClassifier(max_depth=5, random_state=42)
decision_tree.fit(X_train_scaled, y_train_resampled)

# Implementing 5-fold cross-validation to validate the stability and
reliability of the model.
cv_scores = cross_val_score(decision_tree, X_train_scaled,
y_train_resampled, cv=5, scoring='accuracy')
print(f'Cross-validation scores: {cv_scores}')
print(f'Mean cross-validation score: {cv_scores.mean():.4f}')

# Predicting on both the training set and the unseen test set to
evaluate performance.
training_predictions = decision_tree.predict(X_train_scaled)
testing_predictions = decision_tree.predict(X_test_scaled)

# Evaluating the model with various metrics to assess its predictive
power comprehensively.
training_accuracy = accuracy_score(y_train_resampled,
training_predictions)
testing_accuracy = accuracy_score(y_test, testing_predictions)
training_f1 = f1_score(y_train_resampled, training_predictions,
average='weighted')
testing_f1 = f1_score(y_test, testing_predictions, average='weighted')
training_precision = precision_score(y_train_resampled,
training_predictions, average='weighted')
testing_precision = precision_score(y_test, testing_predictions,
average='weighted')
training_recall = recall_score(y_train_resampled,
training_predictions, average='weighted')
testing_recall = recall_score(y_test, testing_predictions,
average='weighted')

# Displaying detailed performance metrics to provide a clear picture
of model effectiveness.
print('AL : Decision Tree (20%)')
print('\nTraining Model Performance Check')
print(f'Accuracy Score: {training_accuracy:.4f}')
print(f'F1 Score: {training_f1:.4f}')
print(f'Precision Score: {training_precision:.4f}')
```

```
print(f'Recall Score: {training_recall:.4f}')

print('\nTesting Model Performance Check')
print(f'Accuracy Score: {testing_accuracy:.4f}')
print(f'F1 Score: {testing_f1:.4f}')
print(f'Precision Score: {testing_precision:.4f}')
print(f'Recall Score: {testing_recall:.4f}')

# Presenting the classification report to offer insights into the
performance across different classes.
print("\nDetailed Classification Report for Training Data:")
print(classification_report(y_train_resampled, training_predictions))
print("\nDetailed Classification Report for Testing Data:")
print(classification_report(y_test, testing_predictions))

# Visualization of the model's decision-making capabilities through
ROC curve analysis.
probabilities = decision_tree.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, 'b', label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], 'r--')
plt.title('Decision Tree ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.show()

# Confusion Matrix visualization to see how well the model is
predicting each class.
cm = confusion_matrix(y_test, testing_predictions)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues')
plt.title('Confusion Matrix for Decision Tree')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Cross-validation scores: [0.67401575 0.64251969 0.64195584 0.70189274
0.67981073]
Mean cross-validation score: 0.6680
AL : Decision Tree (20%)

Training Model Performance Check
Accuracy Score: 0.7015
F1 Score: 0.7012
Precision Score: 0.7020
Recall Score: 0.7015
```

```
Testing Model Performance Check
Accuracy Score: 0.6799
F1 Score: 0.6806
Precision Score: 0.6815
Recall Score: 0.6799

Detailed Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.69      0.73      0.71      1586
           1       0.71      0.68      0.69      1586

    accuracy                           0.70      3172
   macro avg       0.70      0.70      0.70      3172
weighted avg       0.70      0.70      0.70      3172


Detailed Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.75      0.74      0.74       412
           1       0.57      0.58      0.57       244

    accuracy                           0.68       656
   macro avg       0.66      0.66      0.66       656
weighted avg       0.68      0.68      0.68       656
```


Decision Tree ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.72



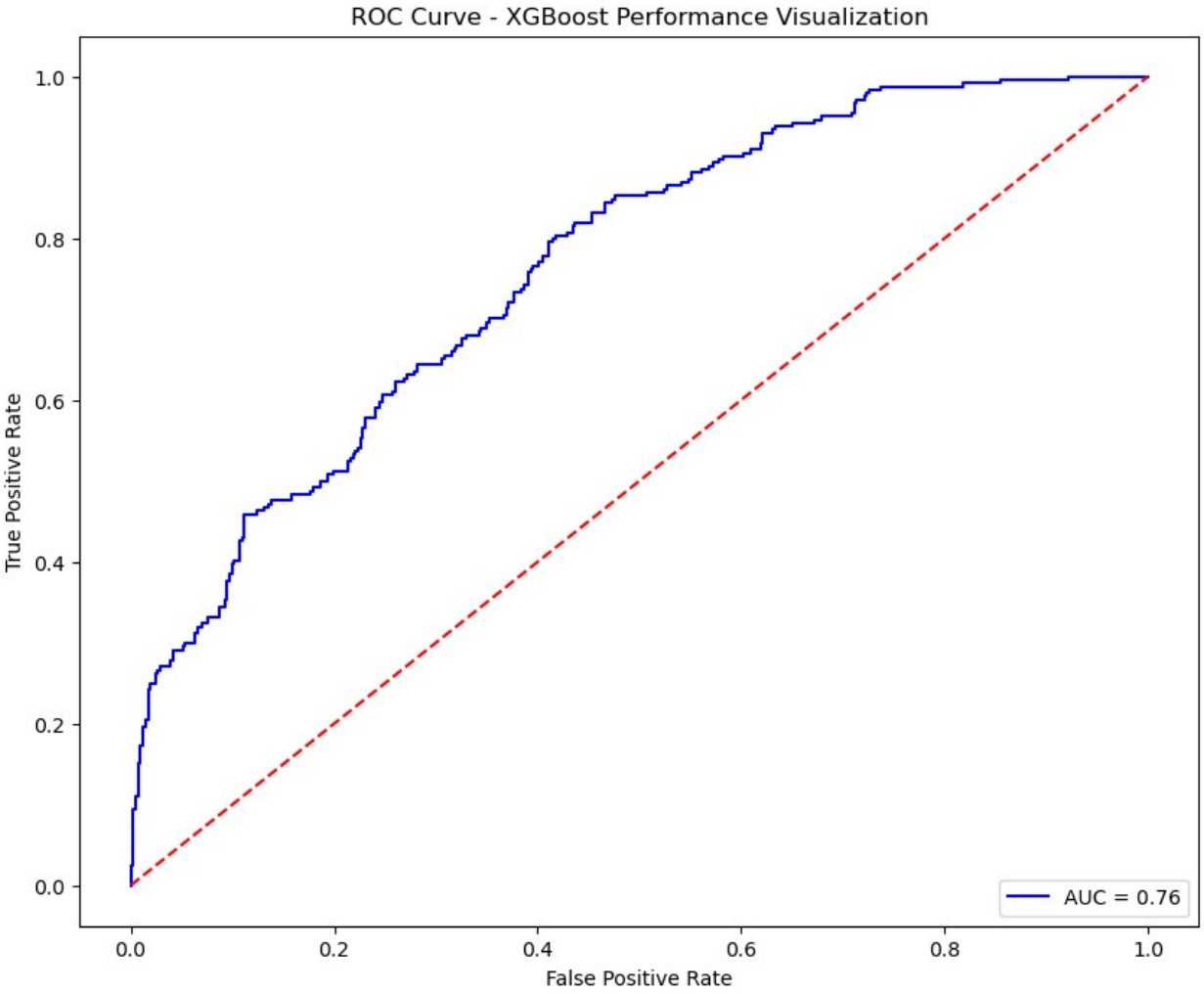
XGBOOST ALGORITHM

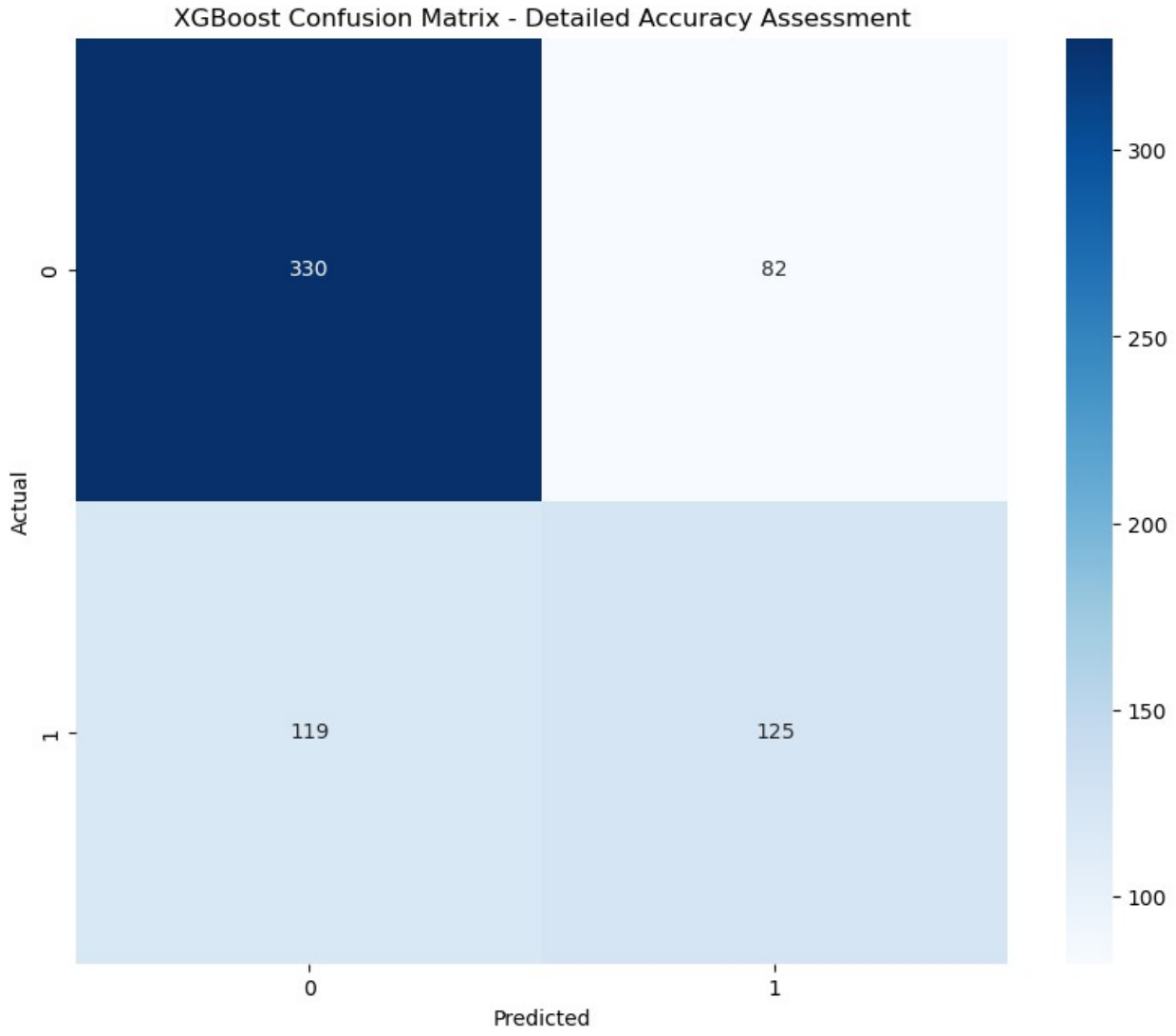
```
# XGBoost
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, roc_curve, auc, confusion_matrix, classification_report
from imblearn.over_sampling import RandomOverSampler
from sklearn.model_selection import train_test_split, cross_val_score
from xgboost import XGBClassifier



X = features
y = water_potability_processed['Potability']
```

```
# Loading and preprocessing data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Applying StandardScaler to ensure features contribute equally to the
predictive power of the model
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Addressing class imbalance using RandomOverSampler to ensure fair
model training
ros = RandomOverSampler(random_state=42)
X_train_resampled, y_train_resampled =
ros.fit_resample(X_train_scaled, y_train)

# Initializing XGBoost with specified parameters for optimized
learning
xgb_model = XGBClassifier(use_label_encoder=False,
eval_metric='logloss', random_state=42)
xgb_model.fit(X_train_resampled, y_train_resampled)

# Evaluating model stability and performance using 5-fold cross-
validation
cv_scores = cross_val_score(xgb_model, X_train_resampled,
y_train_resampled, cv=5, scoring='accuracy')
print(f'Cross-validation scores: {cv_scores}')
print(f'Mean cross-validation score: {cv_scores.mean():.4f}')

# Making predictions on both training and testing data to assess model
performance
training_predictions = xgb_model.predict(X_train_resampled)
testing_predictions = xgb_model.predict(X_test_scaled)

# Calculating comprehensive performance metrics to fully evaluate the
model
training_accuracy = accuracy_score(y_train_resampled,
training_predictions)
testing_accuracy = accuracy_score(y_test, testing_predictions)
training_f1 = f1_score(y_train_resampled, training_predictions,
average='weighted')
testing_f1 = f1_score(y_test, testing_predictions, average='weighted')
training_precision = precision_score(y_train_resampled,
training_predictions, average='weighted')
testing_precision = precision_score(y_test, testing_predictions,
average='weighted')
training_recall = recall_score(y_train_resampled,
training_predictions, average='weighted')
testing_recall = recall_score(y_test, testing_predictions,
```

```
average='weighted')

# Displaying detailed performance metrics
print('XGBoost Model Performance:')
print(f'\nTraining Metrics: Accuracy: {training_accuracy:.4f}, F1
Score: {training_f1:.4f}, Precision: {training_precision:.4f}, Recall:
{training_recall:.4f}')
print(f'\nTesting Metrics: Accuracy: {testing_accuracy:.4f}, F1 Score:
{testing_f1:.4f}, Precision: {testing_precision:.4f}, Recall:
{testing_recall:.4f}')

# Detailed classification reports for training and testing datasets
print("\nXGBoost Classification Report for Training Data:")
print(classification_report(y_train_resampled, training_predictions))
print("\nXGBoost Classification Report for Testing Data:")
print(classification_report(y_test, testing_predictions))

# ROC Curve to visualize the trade-offs between sensitivity (TPR) and
specificity (FPR)
probabilities = xgb_model.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(10, 8))
plt.plot(fpr, tpr, 'b', label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], 'r--')
plt.title('ROC Curve - XGBoost Performance Visualization')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.show()

# Confusion Matrix visualization to detail accuracy of the
classification
cm = confusion_matrix(y_test, testing_predictions)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('XGBoost Confusion Matrix - Detailed Accuracy Assessment')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

Cross-validation scores: [0.77952756 0.71968504 0.77287066 0.8170347
0.85488959]
Mean cross-validation score: 0.7888
XGBoost Model Performance:

Training Metrics: Accuracy: 0.9987, F1 Score: 0.9987, Precision:
0.9987, Recall: 0.9987

Testing Metrics: Accuracy: 0.6936, F1 Score: 0.6876, Precision:
```

```
0.6862, Recall: 0.6936

XGBoost Classification Report for Training Data:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1586
           1       1.00      1.00      1.00      1586

    accuracy                           1.00      3172
   macro avg       1.00      1.00      1.00      3172
weighted avg       1.00      1.00      1.00      3172


XGBoost Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.73      0.80      0.77       412
           1       0.60      0.51      0.55       244

    accuracy                           0.69       656
   macro avg       0.67      0.66      0.66       656
weighted avg       0.69      0.69      0.69       656
```


ROC Curve - XGBoost Performance Visualization
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.76



CHECKING IF X AND Y IS IN THE SYSTEM

```
#DEFINITION OF INDEPENDENT AND DEPENDENT VARIABLES

# Using .iloc to select all columns except the last one from the
DataFrame.
# The ":" indicates selection of all rows, and ":-1" means every
column except the last one.
# This is a common practice when you want to separate feature
variables (predictors) from the target variable.
X = features
y = water_potability_processed['Potability']  # Replace
'target_variable_name' with the actual name of your target column

# Displaying the first five rows of the selected columns to verify the
```

```
correct columns are included.
# This quick preview is useful to ensure that the DataFrame 'X'
contains only the feature variables,
# which are needed for input into machine learning models.
X.head()


# Displaying the first five rows of the isolated column to verify it's
the correct data.
# This step is useful for a quick check to ensure the data looks as
expected before proceeding with further analysis.
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Potability, dtype: int64
```

```
print(water_potability_processed.isnull().sum())
```

```
ph                  0
Hardness            0
Solids              0
Chloramines         0
Sulfate             0
Conductivity        0
Organic_carbon      0
Trihalomethanes     0
Turbidity           0
Potability          0
dtype: int64
```

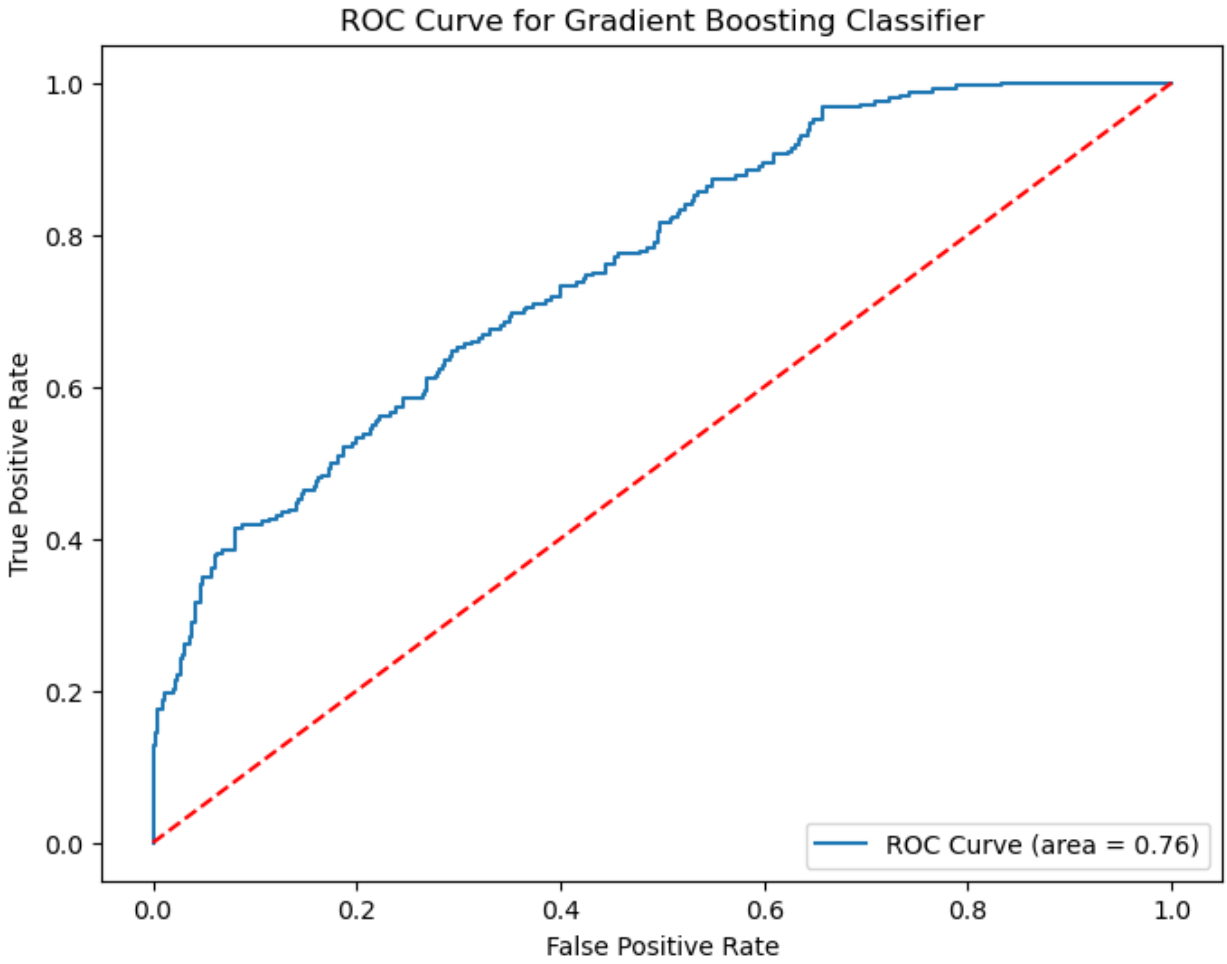
GRADIENTBOOSTCLASSIFIER ALGORITHM

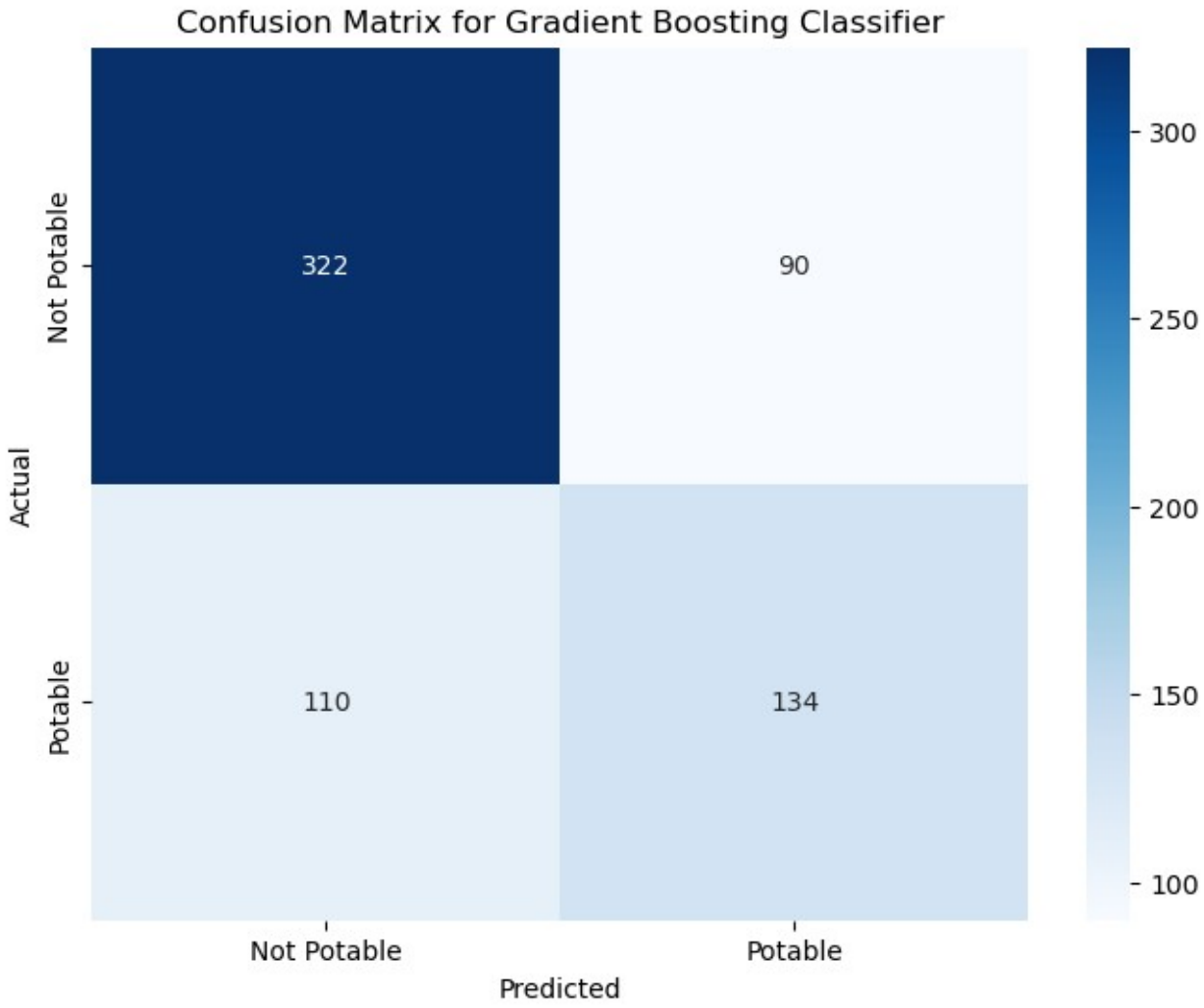
```
#GRADIENTBOOSTCLASSIFIER

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split, cross_val_score
from imblearn.over_sampling import RandomOverSampler
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, roc_curve, auc, confusion_matrix, classification_report,
```

```
roc_auc_score


# Loading and preparing the data
X = water_potability_processed.drop('Potability', axis=1)
y = water_potability_processed['Potability']

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Normalizing the data to ensure that each feature contributes equally
to the distance computations
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Upsampling to address class imbalance, helping ensure fairness and
improve model performance on minority classes
ros = RandomOverSampler(random_state=42)
X_train_resampled, y_train_resampled =
ros.fit_resample(X_train_scaled, y_train)

# Initialize and train the Gradient Boosting Classifier
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
max_depth=3, random_state=42)
gb.fit(X_train_resampled, y_train_resampled)

# Predicting on the training and testing data
train_preds = gb.predict(X_train_resampled)
test_preds = gb.predict(X_test_scaled)

# Evaluating the model with cross-validation for a more robust
performance estimate
cv_scores = cross_val_score(gb, X_train_resampled, y_train_resampled,
cv=5, scoring='accuracy')
print(f"Cross-validation scores: {cv_scores}")
print(f"Mean cross-validation score: {cv_scores.mean():.2f}")

# Performance metrics evaluation
train_accuracy = accuracy_score(y_train_resampled, train_preds)
test_accuracy = accuracy_score(y_test, test_preds)
train_f1 = f1_score(y_train_resampled, train_preds,
average='weighted')
test_f1 = f1_score(y_test, test_preds, average='weighted')
train_precision = precision_score(y_train_resampled, train_preds,
average='weighted')
test_precision = precision_score(y_test, test_preds,
average='weighted')
train_recall = recall_score(y_train_resampled, train_preds,
```

```
average='weighted')
test_recall = recall_score(y_test, test_preds, average='weighted')

# Displaying comprehensive training and testing performance metrics
print('Gradient Boosting Classifier Performance:')
print(f'\nTraining Metrics: Accuracy: {train_accuracy:.4f}, F1 Score:
{train_f1:.4f}, Precision: {train_precision:.4f}, Recall:
{train_recall:.4f}')
print(f'\nTesting Metrics: Accuracy: {test_accuracy:.4f}, F1 Score:
{test_f1:.4f}, Precision: {test_precision:.4f}, Recall:
{test_recall:.4f}')

# Classification reports providing detailed performance insights for
both training and testing
print("\nClassification Report for Training Data:")
print(classification_report(y_train_resampled, train_preds))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, test_preds))

# Predicting probabilities for the test set
probabilities = gb.predict_proba(X_test_scaled)[:, 1]  # Get
probabilities for the positive class

# Calculating ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, probabilities)
roc_auc = roc_auc_score(y_test, probabilities)

# Plotting ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC Curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'r--')  # Dashed diagonal
plt.title('ROC Curve for Gradient Boosting Classifier')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.show()


# Visualizing the confusion matrix to assess prediction accuracy
visually
conf_matrix = confusion_matrix(y_test, test_preds)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
xticklabels=['Not Potable', 'Potable'], yticklabels=['Not Potable',
'Potable'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix for Gradient Boosting Classifier')
plt.show()
```

```
Cross-validation scores: [0.68503937 0.66929134 0.69716088 0.72239748
0.71451104]
Mean cross-validation score: 0.70
Gradient Boosting Classifier Performance:

Training Metrics: Accuracy: 0.7929, F1 Score: 0.7924, Precision:
0.7957, Recall: 0.7929

Testing Metrics: Accuracy: 0.6951, F1 Score: 0.6922, Precision:
0.6906, Recall: 0.6951

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.77      0.84      0.80      1586
           1       0.82      0.74      0.78      1586

    accuracy                           0.79      3172
   macro avg       0.80      0.79      0.79      3172
weighted avg       0.80      0.79      0.79      3172


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.75      0.78      0.76       412
           1       0.60      0.55      0.57       244

    accuracy                           0.70       656
   macro avg       0.67      0.67      0.67       656
weighted avg       0.69      0.70      0.69       656
```


ROC Curve for Gradient Boosting Classifier
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC Curve (area = 0.76)



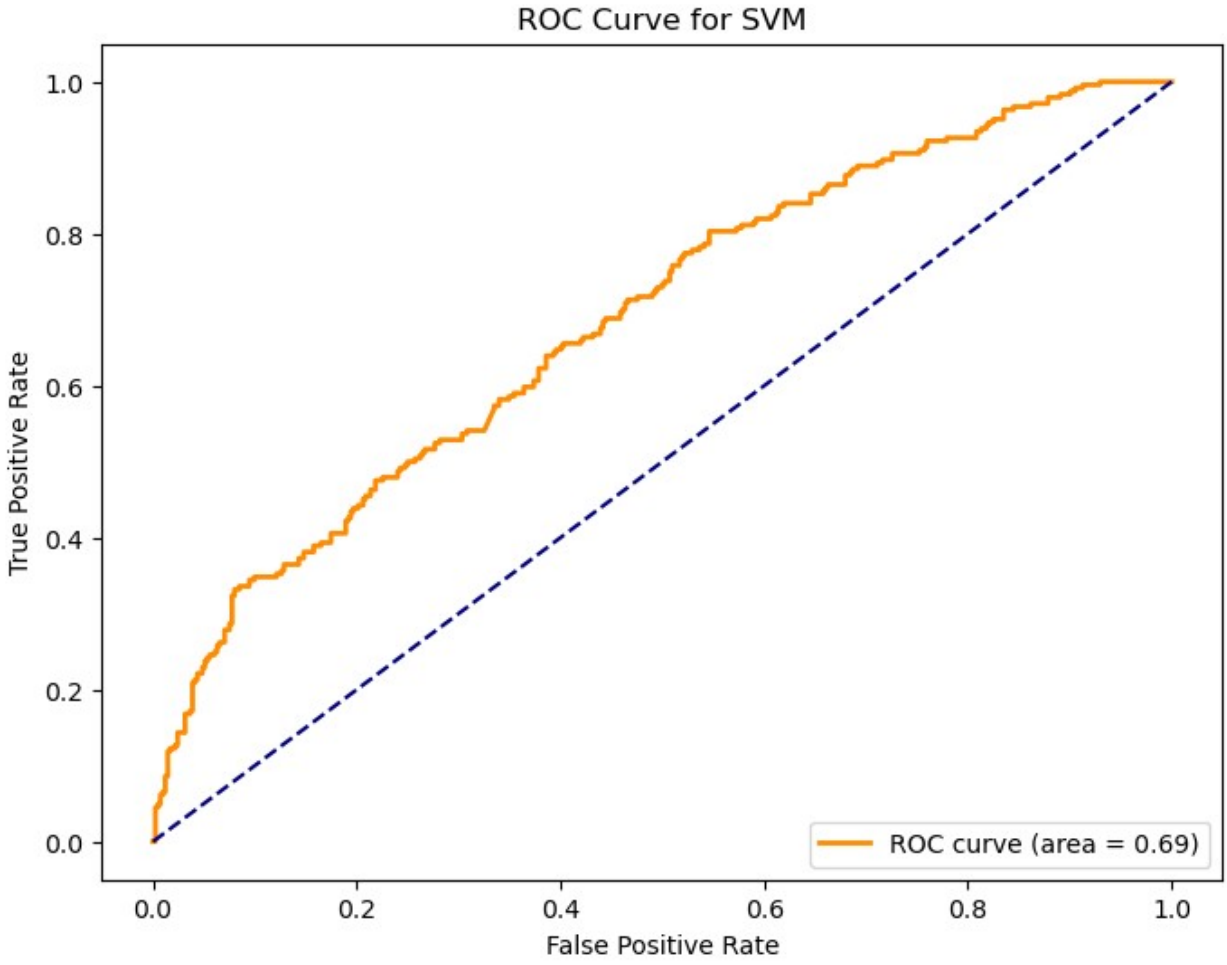
SVM ALGORITHM

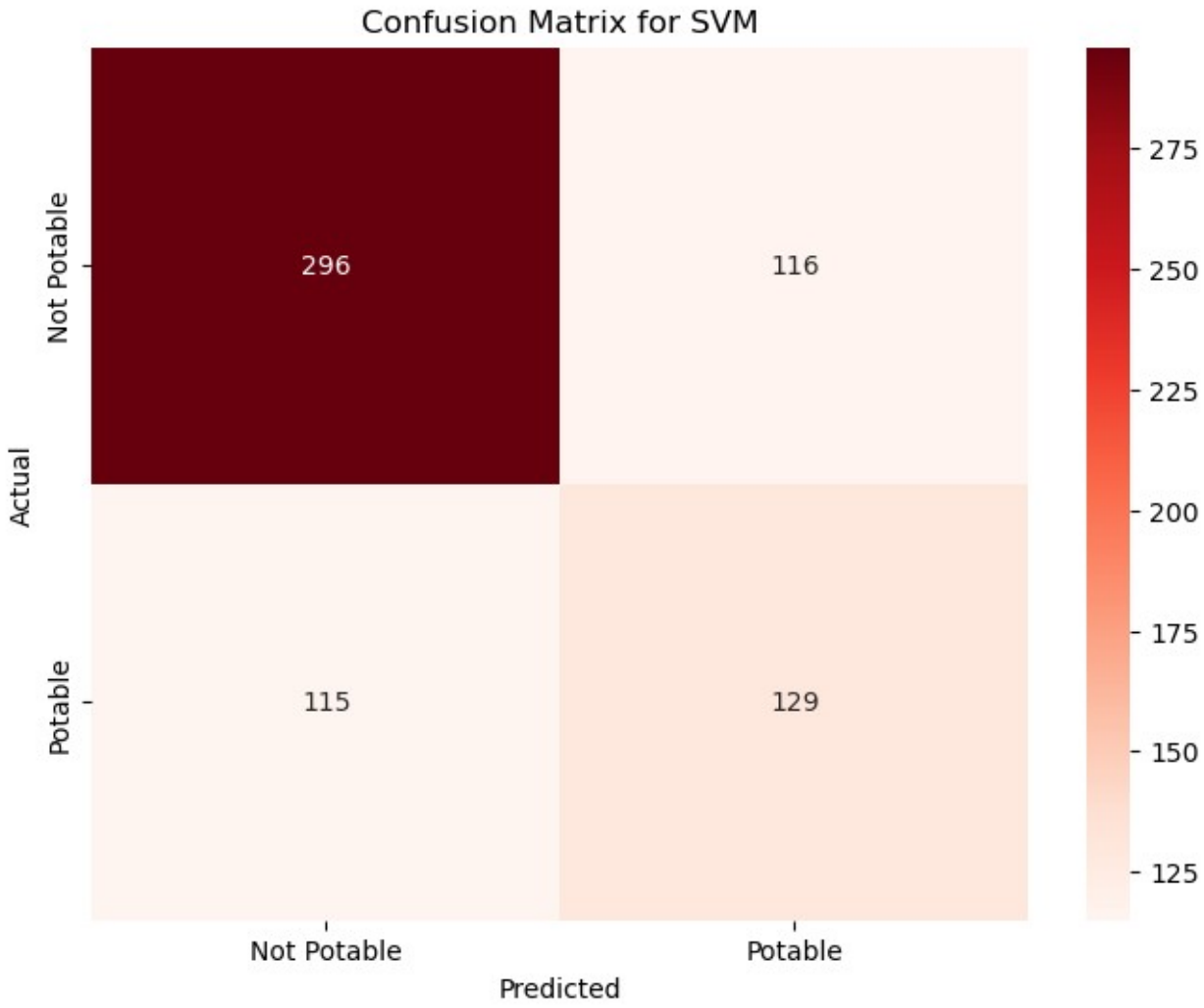
```
# SVM
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, roc_curve, auc, confusion_matrix, classification_report
from imblearn.over_sampling import RandomOverSampler
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score

# Data preprocessing
X = water_potability_processed.drop('Potability', axis=1)
y = water_potability_processed['Potability']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Normalizing the dataset
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Addressing class imbalance
ros = RandomOverSampler(random_state=42)
X_train_resampled, y_train_resampled =
ros.fit_resample(X_train_scaled, y_train)

# Initialize and train the Support Vector Machine with probability
estimates enabled for ROC analysis
svm_model = SVC(probability=True, random_state=42)
svm_model.fit(X_train_resampled, y_train_resampled)

# Cross-validation to assess model stability
cv_scores = cross_val_score(svm_model, X_train_resampled,
y_train_resampled, cv=5, scoring='accuracy')
print(f"Cross-validation scores: {cv_scores}")
print(f"Mean cross-validation score: {cv_scores.mean():.2f}")

# Making predictions
training_predictions = svm_model.predict(X_train_resampled)
testing_predictions = svm_model.predict(X_test_scaled)

# Performance metrics
training_accuracy = accuracy_score(y_train_resampled,
training_predictions)
testing_accuracy = accuracy_score(y_test, testing_predictions)
training_f1 = f1_score(y_train_resampled, training_predictions,
average='weighted')
testing_f1 = f1_score(y_test, testing_predictions, average='weighted')
training_precision = precision_score(y_train_resampled,
training_predictions, average='weighted', zero_division=1)
testing_precision = precision_score(y_test, testing_predictions,
average='weighted', zero_division=1)
training_recall = recall_score(y_train_resampled,
training_predictions, average='weighted')
testing_recall = recall_score(y_test, testing_predictions,
average='weighted')

# Detailed performance metrics
print('SVM Training Performance:')
print(f'Accuracy: {training_accuracy:.4f}, F1 Score:
{training_f1:.4f}, Precision: {training_precision:.4f}, Recall:
{training_recall:.4f}')
print('SVM Testing Performance:')
```

```
print(f'Accuracy: {testing_accuracy:.4f}, F1 Score: {testing_f1:.4f},
Precision: {testing_precision:.4f}, Recall: {testing_recall:.4f}')

# Classification reports
print("\nClassification Report for Training Data:")
print(classification_report(y_train_resampled, training_predictions))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, testing_predictions))

# ROC Curve
testing_probabilities = svm_model.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, _ = roc_curve(y_test, testing_probabilities)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for SVM')
plt.legend(loc="lower right")
plt.show()

# Confusion Matrix
cm = confusion_matrix(y_test, testing_predictions)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='Reds', xticklabels=['Not
Potable', 'Potable'], yticklabels=['Not Potable', 'Potable'])
plt.title('Confusion Matrix for SVM')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()


Cross-validation scores: [0.67874016 0.64566929 0.65615142 0.69716088
0.6829653 ]
Mean cross-validation score: 0.67
SVM Training Performance:
Accuracy: 0.7516, F1 Score: 0.7507, Precision: 0.7552, Recall: 0.7516
SVM Testing Performance:
Accuracy: 0.6479, F1 Score: 0.6480, Precision: 0.6482, Recall: 0.6479

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.72      0.81      0.77      1586
           1       0.79      0.69      0.74      1586

    accuracy                           0.75      3172
   macro avg       0.76      0.75      0.75      3172
```

```
weighted avg       0.76      0.75      0.75      3172


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.72      0.72      0.72       412
           1       0.53      0.53      0.53       244

    accuracy                           0.65       656
   macro avg       0.62      0.62      0.62       656
weighted avg       0.65      0.65      0.65       656
```



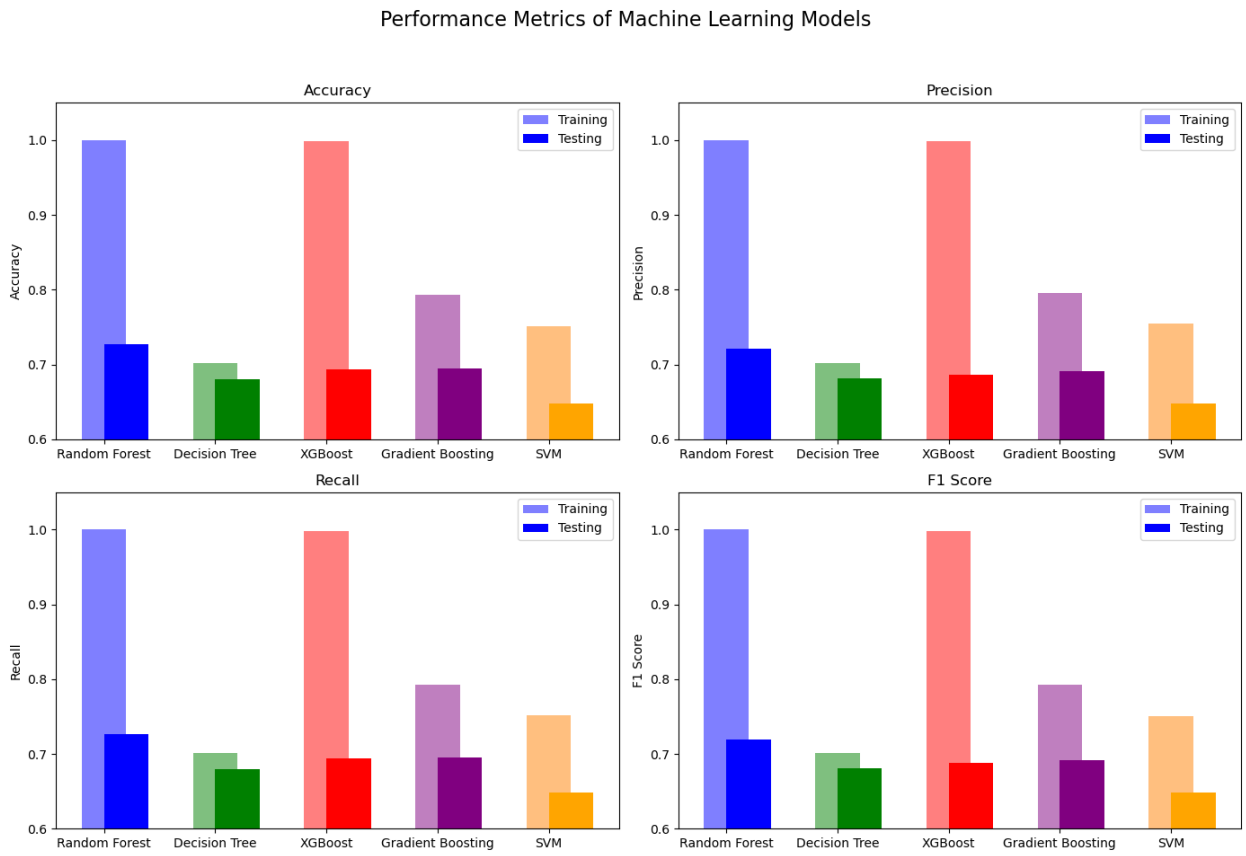


EXPLORATION OF PERFORMANCE METRICS OF ALL ALGORITHMS USED

```
performance_data = {
    'Random Forest': {
        'Training': {'Accuracy': 1.0000, 'Precision': 1.0000,
'Recall': 1.0000, 'F1 Score': 1.0000},
        'Testing': {'Accuracy': 0.7271, 'Precision': 0.7207, 'Recall':
0.7271, 'F1 Score': 0.7196}
    },
    'Decision Tree': {
        'Training': {'Accuracy': 0.7015, 'Precision': 0.7020,
'Recall': 0.7015, 'F1 Score': 0.7012},
        'Testing': {'Accuracy': 0.6799, 'Precision': 0.6815, 'Recall':
0.6799, 'F1 Score': 0.6806}
    },
```

```
    'XGBoost': {
        'Training': {'Accuracy': 0.9987, 'Precision': 0.9987,
'Recall': 0.9987, 'F1 Score': 0.9987},
        'Testing': {'Accuracy': 0.6936, 'Precision': 0.6862, 'Recall':
0.6936, 'F1 Score': 0.6876}
    },
    'Gradient Boosting': {
        'Training': {'Accuracy': 0.7929, 'Precision': 0.7957,
'Recall': 0.7929, 'F1 Score': 0.7924},
        'Testing': {'Accuracy': 0.6951, 'Precision': 0.6906, 'Recall':
0.6951, 'F1 Score': 0.6922}
    },
    'SVM': {
        'Training': {'Accuracy': 0.7516, 'Precision': 0.7552,
'Recall': 0.7516, 'F1 Score': 0.7507},
        'Testing': {'Accuracy': 0.6479, 'Precision': 0.6482, 'Recall':
0.6479, 'F1 Score': 0.6480}
    }
}

import matplotlib.pyplot as plt

# Data preparation
models = list(performance_data.keys())
metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
training_scores = {metric: [performance_data[model]['Training']
[metric] for model in models] for metric in metrics}
testing_scores = {metric: [performance_data[model]['Testing'][metric]
for model in models] for metric in metrics}

# Colors configuration
colors = ['blue', 'green', 'red', 'purple', 'orange']
training_colors = [(0.0, 0.0, 1.0, 0.5), (0.0, 0.5, 0.0, 0.5), (1.0,
0.0, 0.0, 0.5), (0.5, 0.0, 0.5, 0.5), (1.0, 0.5, 0.0, 0.5)]  # Semi-
transparent colors for training
testing_colors = ['blue', 'green', 'red', 'purple', 'orange']  # Solid
colors for testing

# Plotting with color adjustments
fig, axs = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle('Performance Metrics of Machine Learning Models',
fontsize=16)

for ax, metric in zip(axs.flat, metrics):
    ax.bar(models, training_scores[metric], color=training_colors,
width=0.4, label='Training', align='center')
    ax.bar(models, testing_scores[metric], color=testing_colors,
width=0.4, label='Testing', align='edge')
    ax.set_title(metric)
    ax.set_ylim(0.6, 1.05)
```

```
    ax.set_ylabel(metric)
    ax.legend()

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



USING HYPERPARAMETER TUNES FOR ALL THE SELECTED ALGORITHMS

TUNNING OF RADOM FOREST

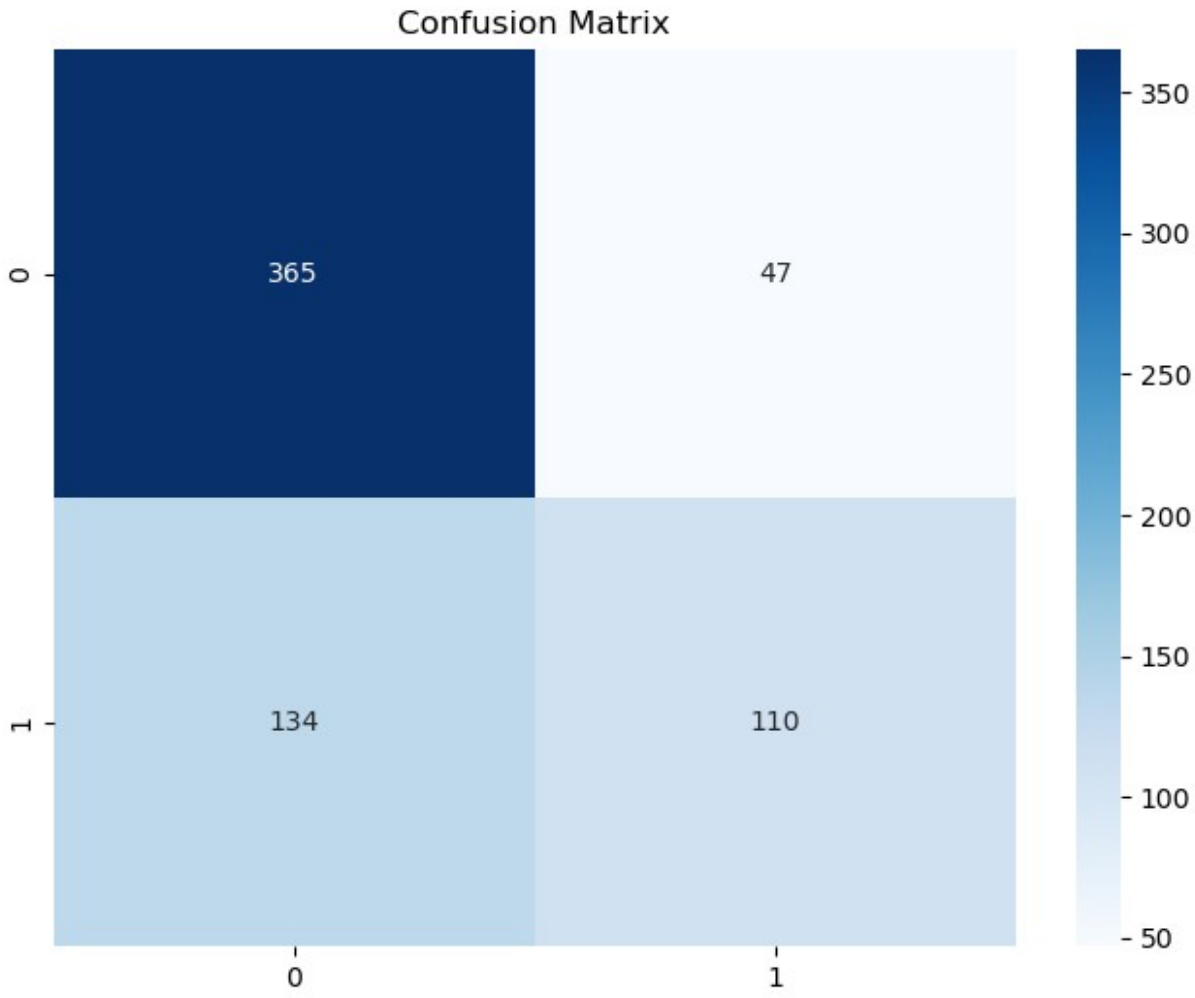
```
#RADOM FOREST HYPERPARAMETER TUNES

# Essential imports for machine learning and data visualization.
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report,
```

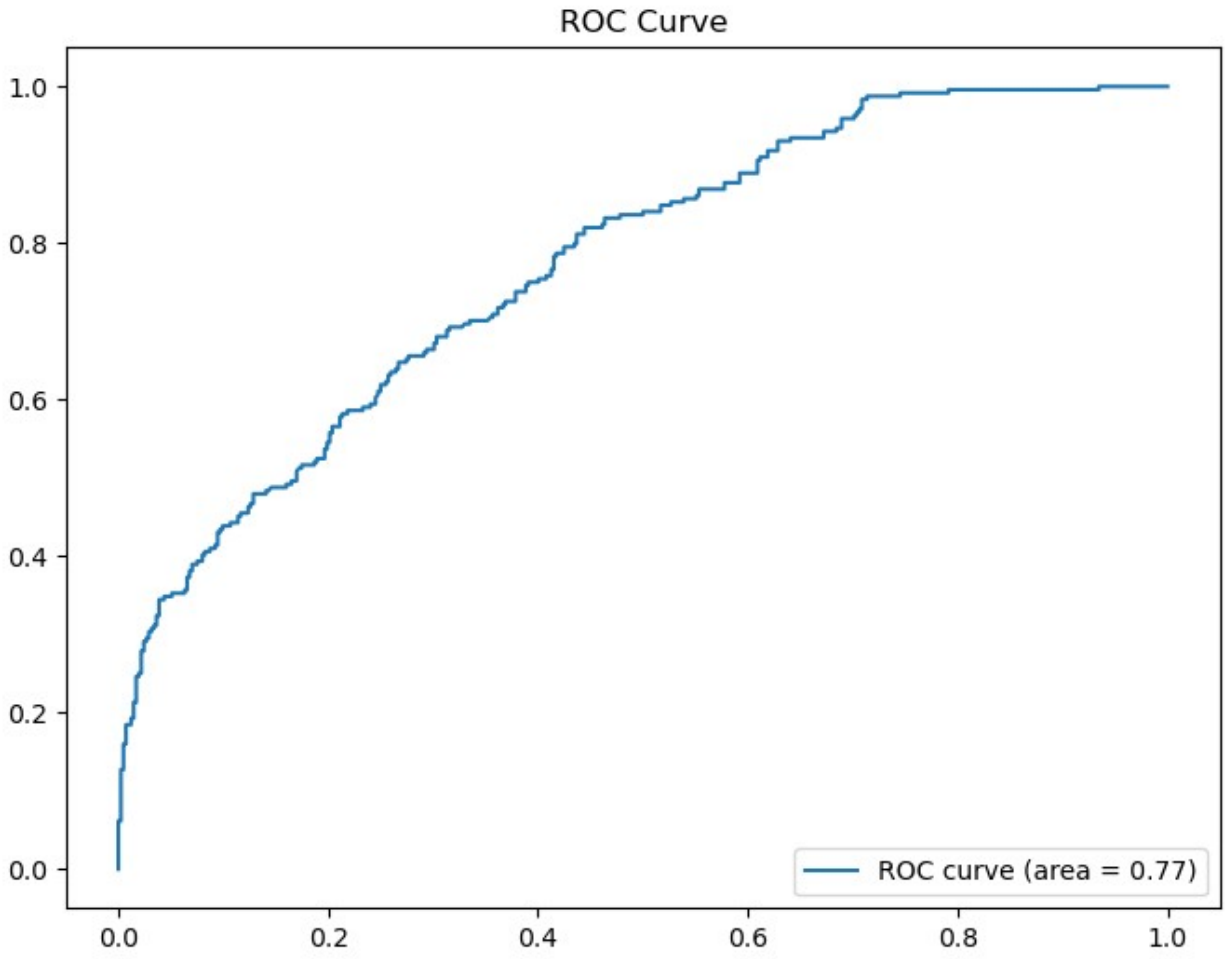
```
roc_curve, auc

# Define the RandomForest model.
rf = RandomForestClassifier(random_state=42)

# Set up a range of hyperparameters to search over for optimal model
tuning.
# Set up a range of hyperparameters to search over for optimal model
tuning.
param_grid = {
    'n_estimators': [100, 150],  # Adding a slightly larger number of
trees
    'max_depth': [10, 15, None],  # Expanding the depth for more
complexity options
    'min_samples_split': [2, 4],  # Including the default value
    'min_samples_leaf': [1, 2],  # Adding the default value to the
grid
    'max_features': ['sqrt', 'log2'],  # Considering both common
options
    'bootstrap': [True]  # No change, since it helps prevent
overfitting
}



# The script begins by setting up a grid search for hyperparameter
optimization, utilizing a 3-fold cross-validation. This ensures the
model is not only tuned for optimal parameters but also validated
across different subsets of the data to generalize better on unseen
data.
grid_search = GridSearchCV(rf, param_grid, cv=3, scoring='accuracy',
verbose=1)
grid_search.fit(X_train_scaled, y_train)

# After the grid search completes, the best model is selected. This
model has shown the best performance on the validation folds according
to the accuracy metric.
best_model = grid_search.best_estimator_

# Predictions are then made on both the training and testing sets.
This step is crucial for evaluating how well the model has learned
from the training data and how it performs on new, unseen data.
y_train_pred = best_model.predict(X_train_scaled)
y_test_pred = best_model.predict(X_test_scaled)

# Evaluation metrics for both training and testing predictions are
calculated here. Metrics include accuracy, F1 score, precision, and
recall. These metrics provide a comprehensive understanding of model
performance, considering both the balance of classes and the
importance of both positive and negative classifications.
```

```
training_accuracy = accuracy_score(y_train, y_train_pred)
testing_accuracy = accuracy_score(y_test, y_test_pred)
training_f1 = f1_score(y_train, y_train_pred, average='weighted')
testing_f1 = f1_score(y_test, y_test_pred, average='weighted')

# Displaying detailed performance metrics helps in assessing the
model's overfitting or underfitting by comparing training and testing
results.
print("\nTraining Model Performance Metrics:")
print(f'Accuracy: {training_accuracy:.4f}')
print("\nTesting Model Performance Metrics:")
print(f'Accuracy: {testing_accuracy:.4f}')

# Classification reports provide a breakdown of precision, recall, and
F1-score by class. This is particularly useful for understanding model
performance on each individual class in a multi-class setting.
print("\nClassification Report for Training Data:")
print(classification_report(y_train, y_train_pred))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, y_test_pred))

# Visualizing the confusion matrix offers insights into the true
positives, true negatives, false positives, and false negatives. This
visualization helps in pinpointing the types of errors the model is
making.
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')

# The ROC curve is plotted to assess the model's performance across
different threshold levels. The area under the curve (AUC) provides a
single metric to evaluate the trade-off between true positive rate and
false positive rate at various thresholds.
fpr, tpr, thresholds = roc_curve(y_test,
best_model.predict_proba(X_test_scaled)[:, 1])
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
plt.legend(loc="lower right")
plt.title('ROC Curve')
plt.show()
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits

Training Model Performance Metrics:
Accuracy: 1.0000

Testing Model Performance Metrics:
```

```
Accuracy: 0.7241

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1586
           1       1.00      1.00      1.00      1034

    accuracy                           1.00      2620
   macro avg       1.00      1.00      1.00      2620
weighted avg       1.00      1.00      1.00      2620


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.73      0.89      0.80       412
           1       0.70      0.45      0.55       244

    accuracy                           0.72       656
   macro avg       0.72      0.67      0.67       656
weighted avg       0.72      0.72      0.71       656
```


Confusion Matrix
365
47
134
110
0
1
0
1
350
300
250
200
150
100
50



TUNING OF DECISION TREE ALGORITHM

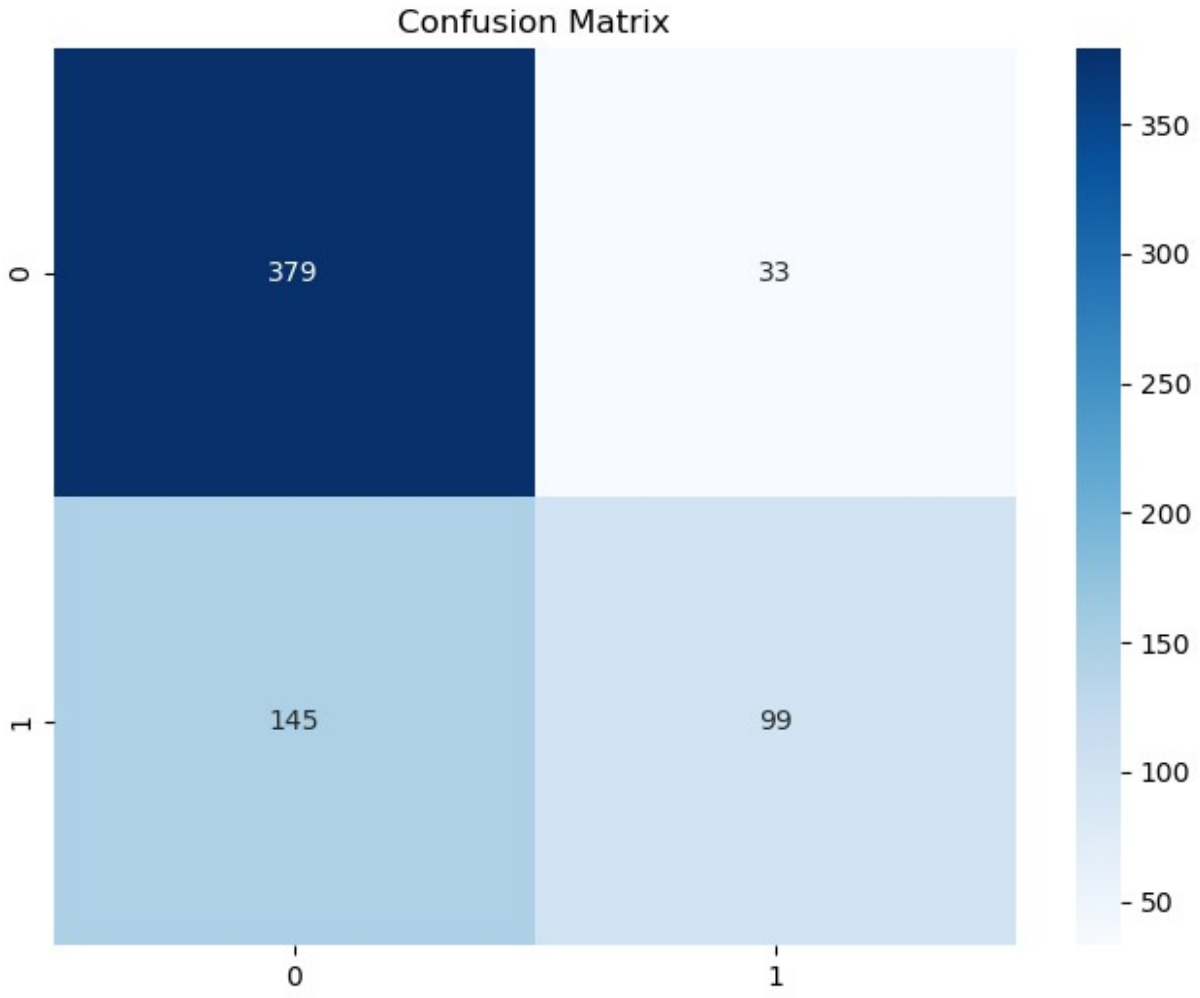
```
#HYPERPARAMETER TUNE FOR DECISION TREE

# Import necessary library for conducting Grid Search and model
evaluation
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, f1_score
from sklearn.tree import DecisionTreeClassifier


# Defining the model
model = RandomForestClassifier(random_state=42)


# Define a dictionary of hyperparameters to tune.
# 'max_depth' controls the maximum depth of the tree.
# http://localhost:8889/notebooks/MACHINE%20LEARNING-C2160212-KEHINDE
%20AKINBILE.ipynb#'min_samples_split' is the minimum number of samples
```

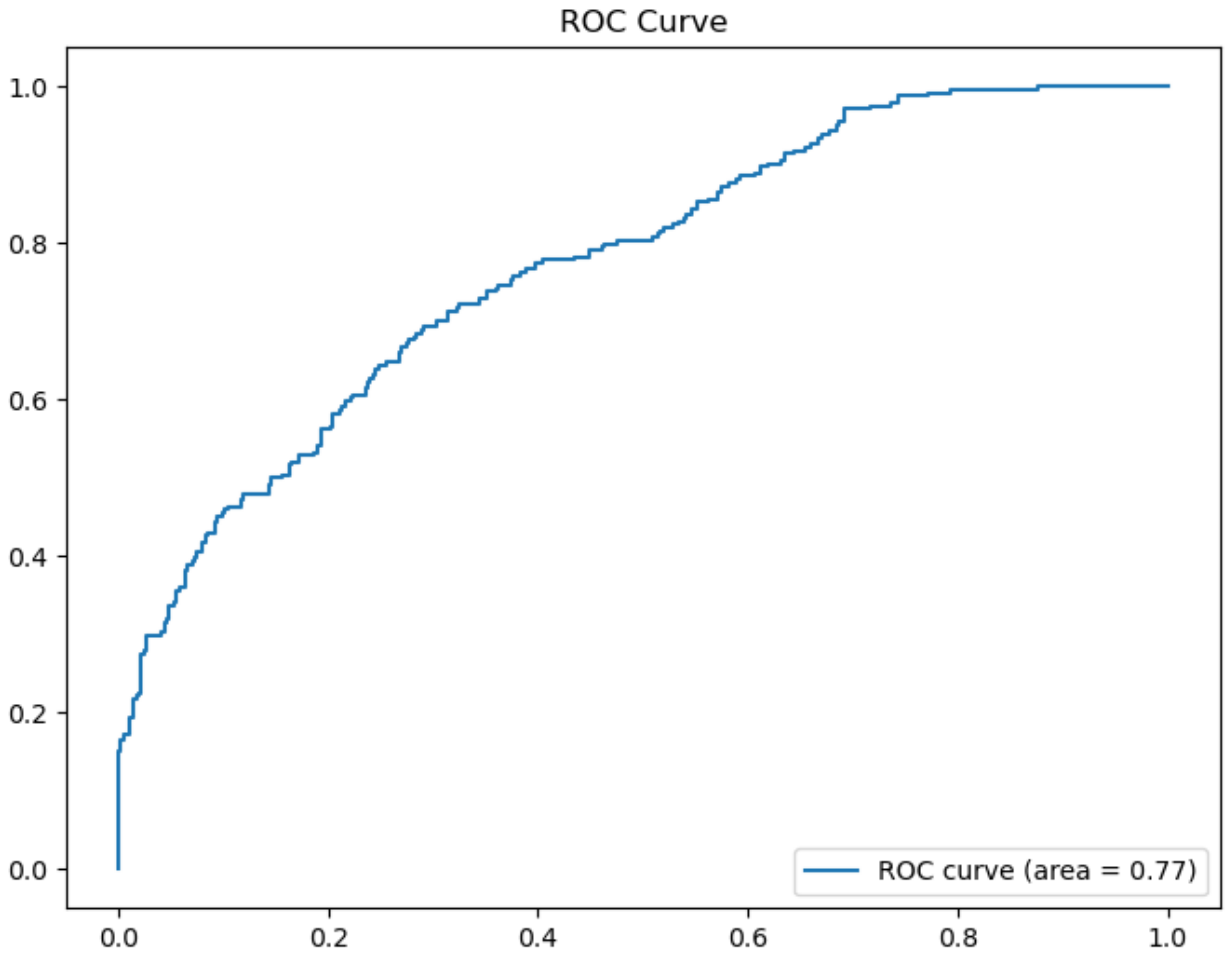
```
required to split an internal node.
# 'min_samples_leaf' is the minimum number of samples required to be
at a leaf node.
param_grid = {
    'max_depth': [None, 5, 10, 15],
    'min_samples_split': [2, 10, 20],
    'min_samples_leaf': [1, 5, 10]
}
# The script begins by setting up a grid search for hyperparameter
optimization, utilizing a 3-fold cross-validation. This ensures the
model is not only tuned for optimal parameters but also validated
across different subsets of the data to generalize better on unseen
data.
grid_search = GridSearchCV(model, param_grid, cv=3,
scoring='accuracy', verbose=1)
grid_search.fit(X_train_scaled, y_train)

# After the grid search completes, the best model is selected. This
model has shown the best performance on the validation folds according
to the accuracy metric.
best_model = grid_search.best_estimator_

# Predictions are then made on both the training and testing sets.
This step is crucial for evaluating how well the model has learned
from the training data and how it performs on new, unseen data.
y_train_pred = best_model.predict(X_train_scaled)
y_test_pred = best_model.predict(X_test_scaled)

# Evaluation metrics for both training and testing predictions are
calculated here. Metrics include accuracy, F1 score, precision, and
recall. These metrics provide a comprehensive understanding of model
performance, considering both the balance of classes and the
importance of both positive and negative classifications.
training_accuracy = accuracy_score(y_train, y_train_pred)
testing_accuracy = accuracy_score(y_test, y_test_pred)
training_f1 = f1_score(y_train, y_train_pred, average='weighted')
testing_f1 = f1_score(y_test, y_test_pred, average='weighted')

# Displaying detailed performance metrics helps in assessing the
model's overfitting or underfitting by comparing training and testing
results.
print("\nTraining Model Performance Metrics:")
print(f'Accuracy: {training_accuracy:.4f}')
print("\nTesting Model Performance Metrics:")
print(f'Accuracy: {testing_accuracy:.4f}')

# Classification reports provide a breakdown of precision, recall, and
F1-score by class. This is particularly useful for understanding model
performance on each individual class in a multi-class setting.
print("\nClassification Report for Training Data:")
```

```
print(classification_report(y_train, y_train_pred))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, y_test_pred))

# Visualizing the confusion matrix offers insights into the true positives, true negatives, false positives, and false negatives. This visualization helps in pinpointing the types of errors the model is making.
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')

# The ROC curve is plotted to assess the model's performance across different threshold levels. The area under the curve (AUC) provides a single metric to evaluate the trade-off between true positive rate and false positive rate at various thresholds.
fpr, tpr, thresholds = roc_curve(y_test,
best_model.predict_proba(X_test_scaled)[:, 1])
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
plt.legend(loc="lower right")
plt.title('ROC Curve')
plt.show()
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits

Training Model Performance Metrics:
Accuracy: 0.8084

Testing Model Performance Metrics:
Accuracy: 0.7287

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.78      0.96      0.86      1586
           1       0.91      0.57      0.70      1034

    accuracy                           0.81      2620
   macro avg       0.84      0.77      0.78      2620
weighted avg       0.83      0.81      0.80      2620


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.72      0.92      0.81       412
           1       0.75      0.41      0.53       244
```

```
              precision    recall  f1-score   support

    accuracy                         0.73       656
   macro avg       0.74      0.66      0.67       656
weighted avg       0.73      0.73      0.70       656
```





TUNING OF XGBOOST ALGORITHM

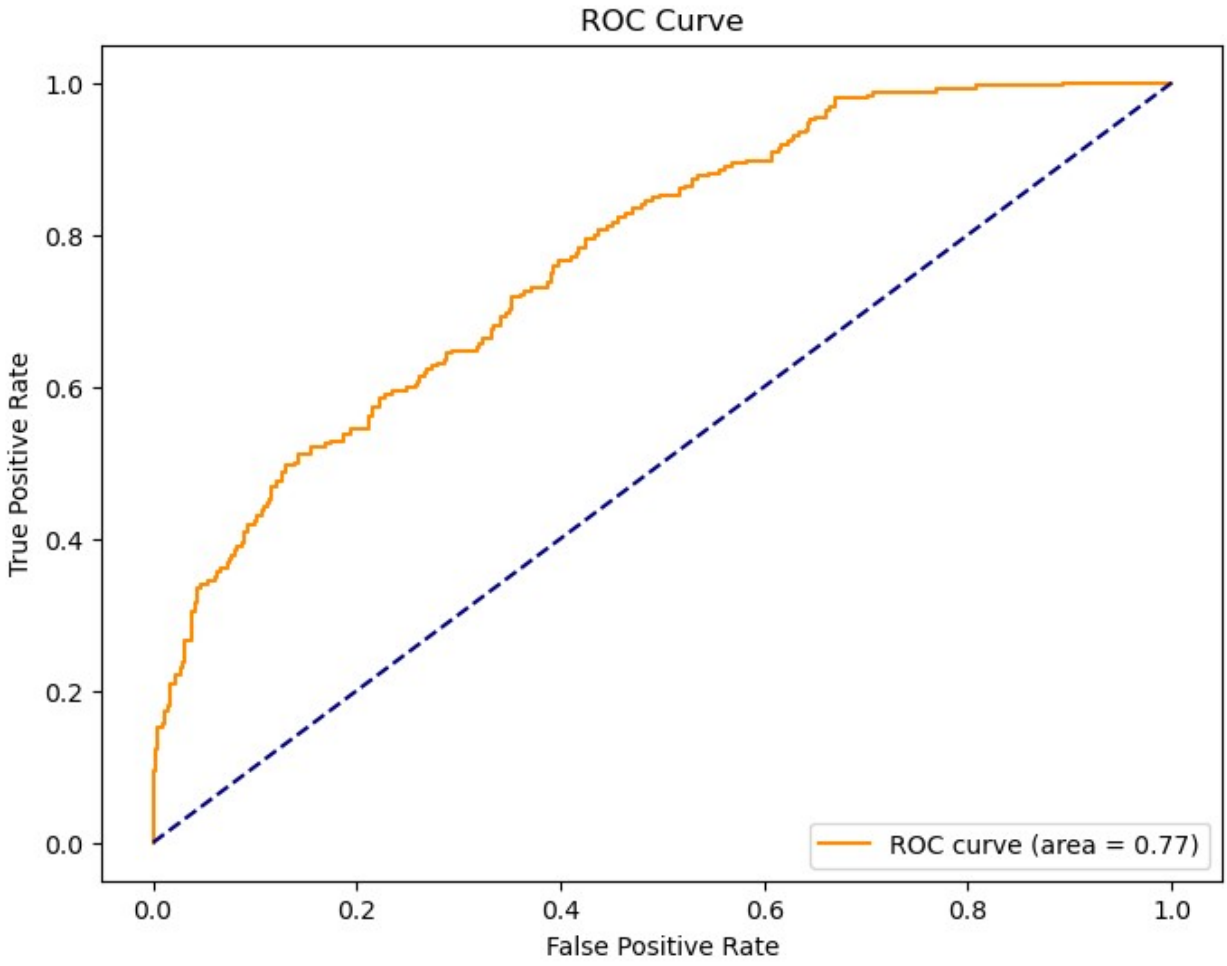
```
#HYPERPARAMETER TUNE FOR XGBOOST

import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, roc_curve, auc, confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns


X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Defining the model - correctly using XGBClassifier
model = XGBClassifier(random_state=42)
```

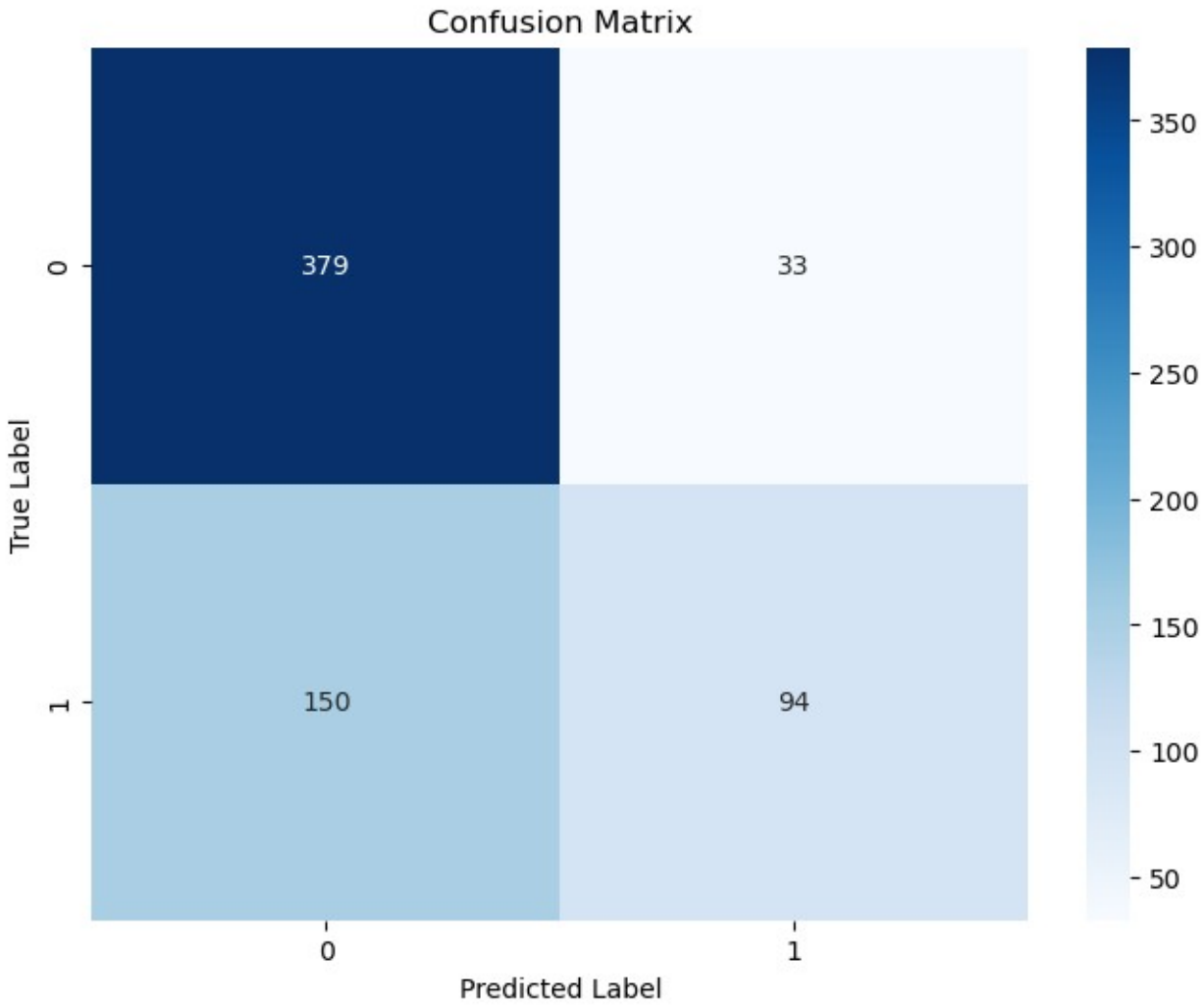
```
# Define a dictionary of hyperparameters to search over


param_grid = {
    'max_depth': [3],
    'learning_rate': [0.1, 0.2],
    'n_estimators': [100, 300],
    'subsample': [0.7, 0.9],
    'colsample_bytree': [0.7, 0.9]
}


# Setting up a grid search for hyperparameter optimization, utilizing
a 3-fold cross-validation
grid_search = GridSearchCV(model, param_grid, cv=3,
scoring='accuracy', verbose=1)
grid_search.fit(X_train, y_train)

# Best model selected from grid search
best_model = grid_search.best_estimator_

# Making predictions with the best model
y_train_pred = best_model.predict(X_train)
y_test_pred = best_model.predict(X_test)

# Performance metrics
training_accuracy = accuracy_score(y_train, y_train_pred)
testing_accuracy = accuracy_score(y_test, y_test_pred)
training_f1 = f1_score(y_train, y_train_pred, average='weighted')
testing_f1 = f1_score(y_test, y_test_pred, average='weighted')
training_precision = precision_score(y_train, y_train_pred,
average='weighted')
testing_precision = precision_score(y_test, y_test_pred,
average='weighted')
training_recall = recall_score(y_train, y_train_pred,
average='weighted')
testing_recall = recall_score(y_test, y_test_pred, average='weighted')

# Detailed performance metrics and classification reports
print("\nTraining Performance Metrics:")
print(f'Accuracy: {training_accuracy:.4f}, F1 Score:
{training_f1:.4f}, Precision: {training_precision:.4f}, Recall:
{training_recall:.4f}')
print("\nTesting Performance Metrics:")
print(f'Accuracy: {testing_accuracy:.4f}, F1 Score: {testing_f1:.4f},
Precision: {testing_precision:.4f}, Recall: {testing_recall:.4f}')

print("\nClassification Report for Training Data:")
print(classification_report(y_train, y_train_pred))
print("\nClassification Report for Testing Data:")
```

```
print(classification_report(y_test, y_test_pred))

# ROC curve and AUC
probabilities = best_model.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', label=f'ROC curve (area =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()

# Confusion Matrix visualization
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues')
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits

Training Performance Metrics:
Accuracy: 0.7790, F1 Score: 0.7635, Precision: 0.7990, Recall: 0.7790

Testing Performance Metrics:
Accuracy: 0.7210, F1 Score: 0.6944, Precision: 0.7253, Recall: 0.7210

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.75      0.95      0.84      1586
           1       0.87      0.51      0.65      1034

    accuracy                           0.78      2620
   macro avg       0.81      0.73      0.74      2620
weighted avg       0.80      0.78      0.76      2620


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.72      0.92      0.81       412
           1       0.74      0.39      0.51       244
```

```
    accuracy                           0.72       656
   macro avg       0.73      0.65      0.66       656
weighted avg       0.73      0.72      0.69       656
```





TUNING OF GRADIENTBOOSTCLASSIFIER ALGORITHM

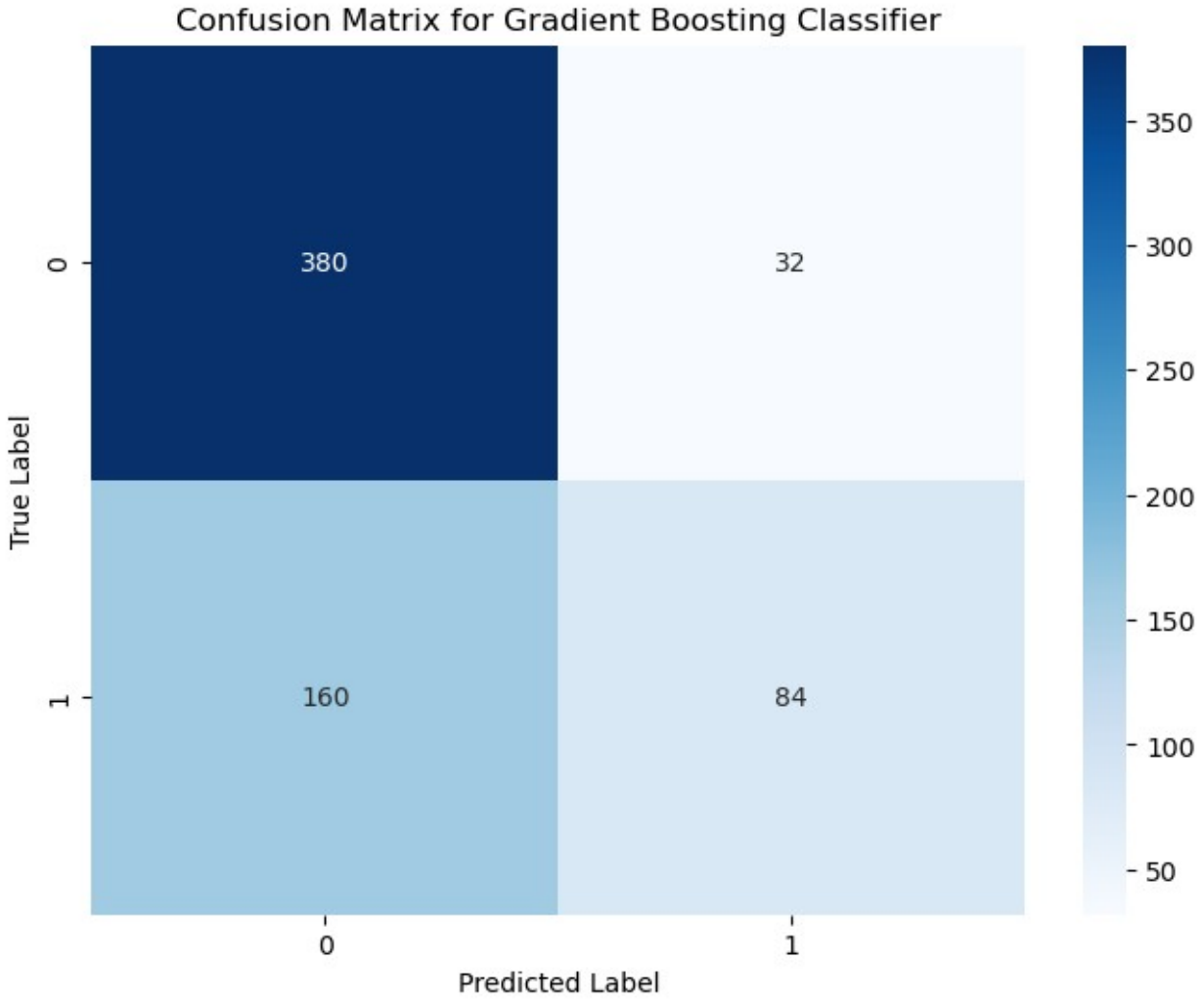
```
# HYPERPARAMETER TUNE GRADIENTBOOST CLASSIFIER
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV,
StratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, confusion_matrix, classification_report, roc_curve, auc
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

# Definition of X and Y
X = water_potability_processed.drop('Potability', axis=1)
```

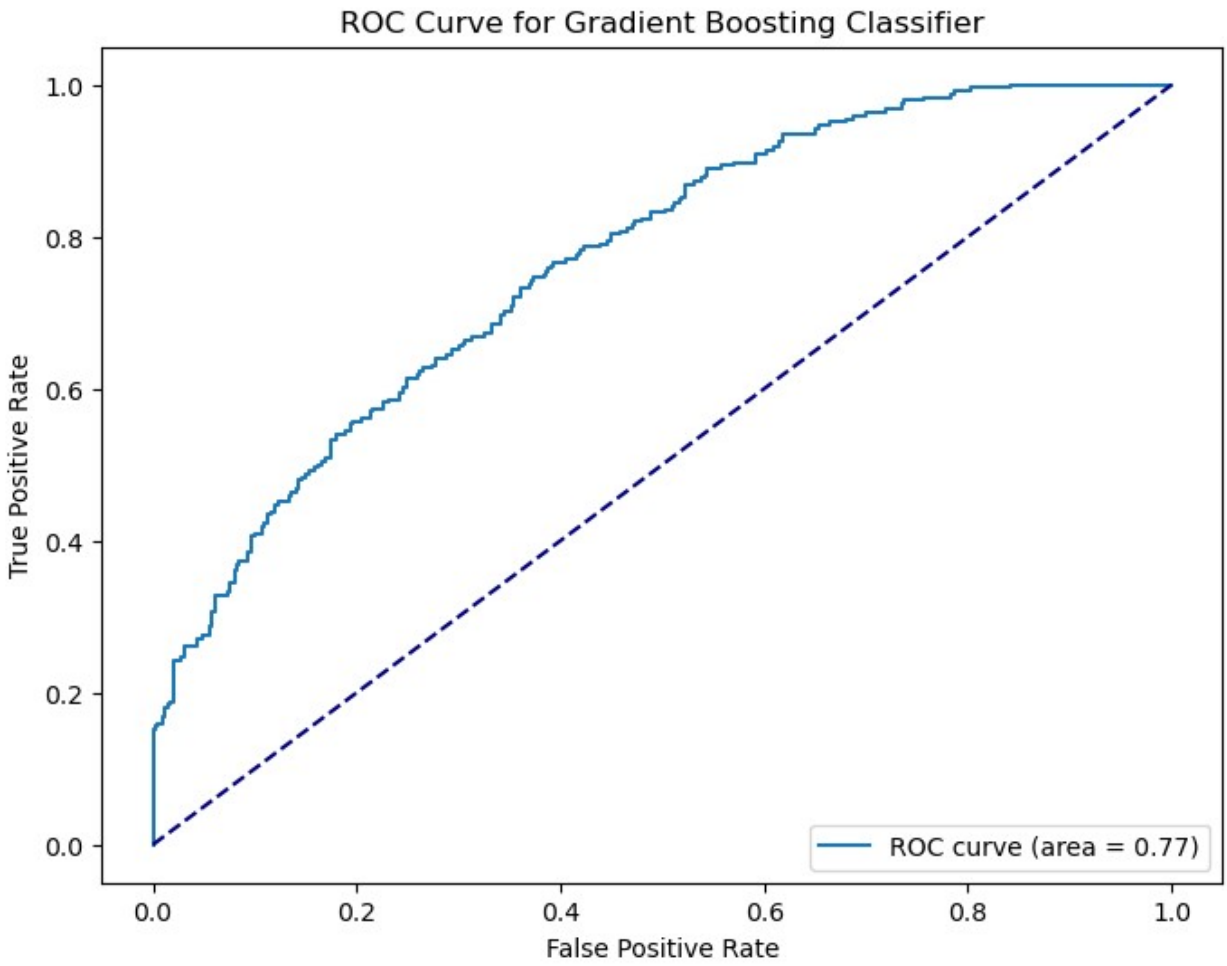
```
y = water_potability_processed['Potability']

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Defining the Gradient Boosting model
model = GradientBoostingClassifier(random_state=42)

# Hyperparameter tuning setup using GridSearchCV with a cross-
validation strategy
param_grid = {
    'n_estimators': [100, 150],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3],
    'min_samples_split': [2],
    'min_samples_leaf': [1],
    'subsample': [0.8]  # Adding subsample for stochastic gradient
boosting
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(model, param_grid, cv=3,
scoring='accuracy', verbose=1)
grid_search.fit(X_train_scaled, y_train)

# Extracting the best model
best_model = grid_search.best_estimator_

# Predictions and evaluations
y_train_pred = best_model.predict(X_train_scaled)
y_test_pred = best_model.predict(X_test_scaled)

training_accuracy = accuracy_score(y_train, y_train_pred)
testing_accuracy = accuracy_score(y_test, y_test_pred)
training_f1 = f1_score(y_train, y_train_pred, average='weighted')
testing_f1 = f1_score(y_test, y_test_pred, average='weighted')

# Display performance metrics
print("\nTraining Model Performance Metrics:")
print(f'Accuracy: {training_accuracy:.4f}, F1 Score:
{training_f1:.4f}')
print("\nTesting Model Performance Metrics:")
print(f'Accuracy: {testing_accuracy:.4f}, F1 Score: {testing_f1:.4f}')
```

```python
# Classification reports and confusion matrix
print("\nClassification Report for Training Data:")
print(classification_report(y_train, y_train_pred))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, y_test_pred))

cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix for Gradient Boosting Classifier')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# ROC curve and AUC
probabilities = best_model.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, _ = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Gradient Boosting Classifier')
plt.legend(loc="lower right")
plt.show()
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits

Training Model Performance Metrics:
Accuracy: 0.7718, F1 Score: 0.7528

Testing Model Performance Metrics:
Accuracy: 0.7073, F1 Score: 0.6750

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.74      0.96      0.84      1586
           1       0.89      0.48      0.63      1034

    accuracy                           0.77      2620
   macro avg       0.81      0.72      0.73      2620
weighted avg       0.80      0.77      0.75      2620


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.70      0.92      0.80       412
```

1	0.72	0.34	0.47	244
accuracy			0.71	656
macro avg	0.71	0.63	0.63	656
weighted avg	0.71	0.71	0.67	656





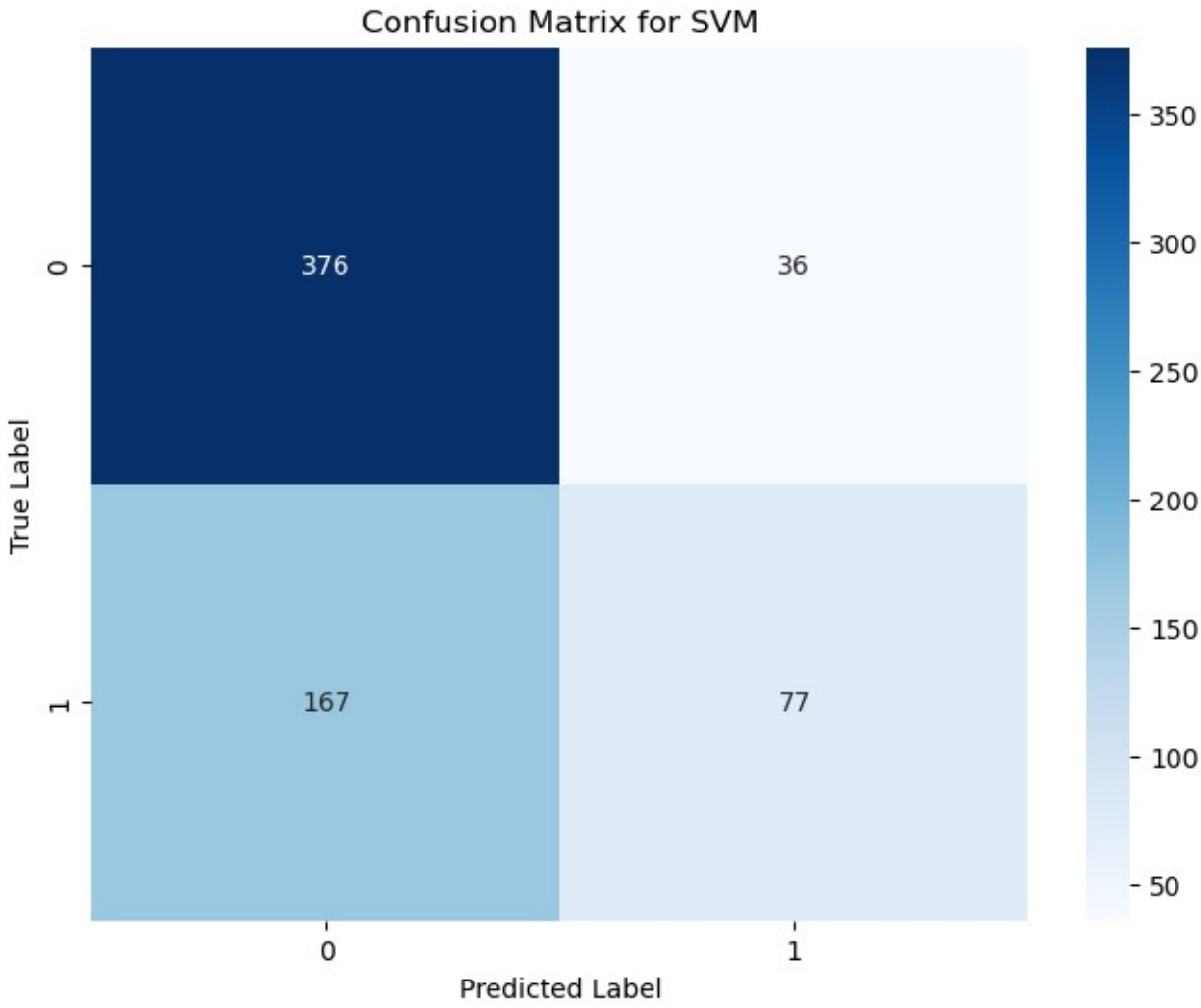
TUNING OF SVM ALGORITHM

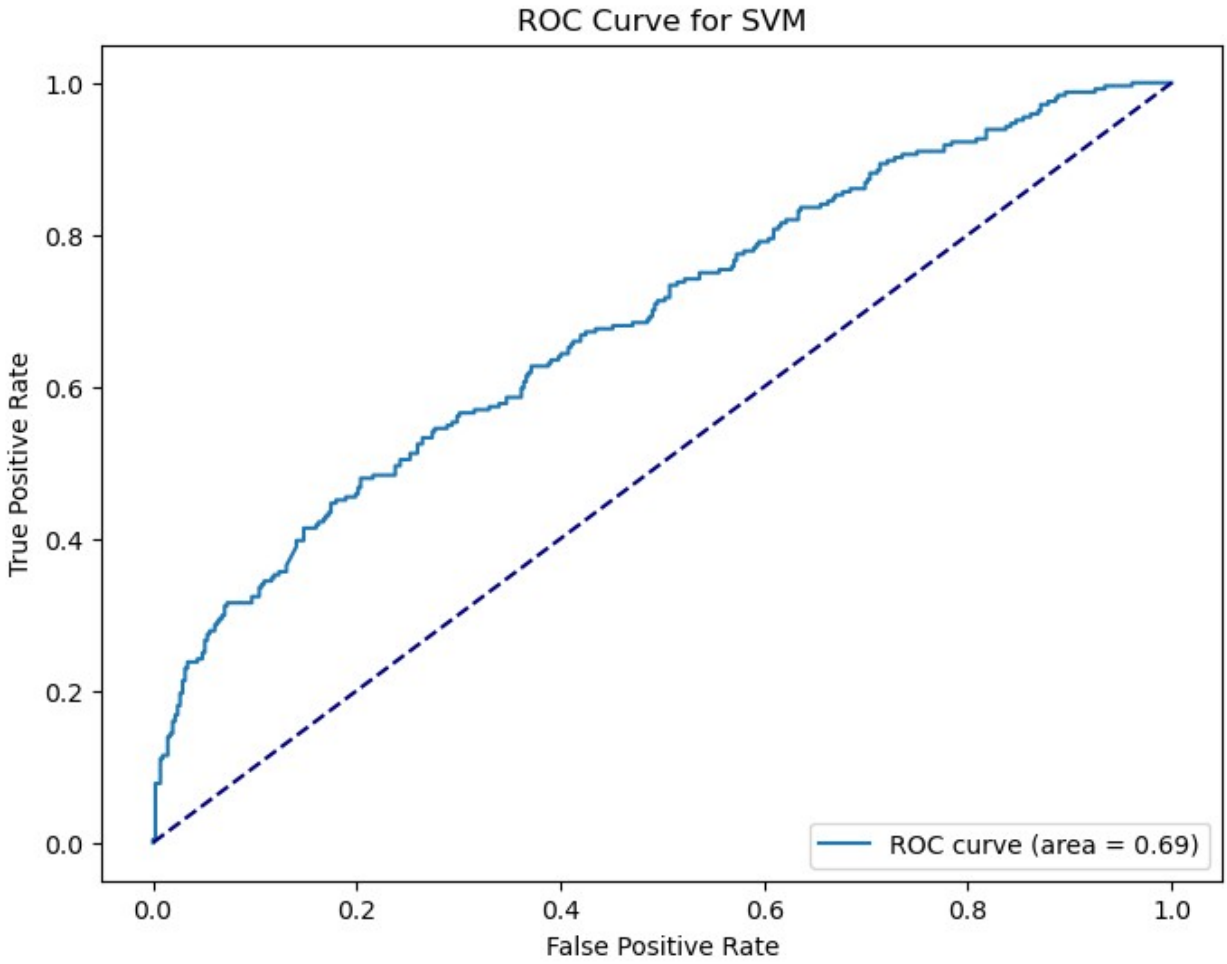
```
#HYPERPARAMETER TUNES SVM

from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, confusion_matrix, classification_report, roc_curve, auc
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

# Load your data here
X = water_potability_processed.drop('Potability', axis=1)
y = water_potability_processed['Potability']

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

```
# Feature scaling for optimal performance with SVM
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the SVM classifier
svm = SVC(probability=True, random_state=42)


# Define the parameter grid for hyperparameter tuning using SVM
param_grid = {
    'C': [0.1, 1, 10],  # Regularization parameter
    'gamma': ['scale', 0.01],  # Kernel coefficient
    'kernel': ['rbf']  # Type of kernel
}

# Setting up GridSearchCV for hyperparameter optimization with 3-fold
cross-validation
grid_search = GridSearchCV(svm, param_grid, cv=3, scoring='accuracy',
verbose=1)
grid_search.fit(X_train_scaled, y_train)

# Extracting the best model from the grid search
best_model = grid_search.best_estimator_

# Predictions on training and testing sets
y_train_pred = best_model.predict(X_train_scaled)
y_test_pred = best_model.predict(X_test_scaled)

# Evaluation metrics
training_accuracy = accuracy_score(y_train, y_train_pred)
testing_accuracy = accuracy_score(y_test, y_test_pred)
training_f1 = f1_score(y_train, y_train_pred, average='weighted')
testing_f1 = f1_score(y_test, y_test_pred, average='weighted')

# Displaying detailed performance metrics
print("\nTraining Model Performance Metrics:")
print(f'Accuracy: {training_accuracy:.4f}, F1 Score:
{training_f1:.4f}')
print("\nTesting Model Performance Metrics:")
print(f'Accuracy: {testing_accuracy:.4f}, F1 Score: {testing_f1:.4f}')

# Classification reports
print("\nClassification Report for Training Data:")
print(classification_report(y_train, y_train_pred))
print("\nClassification Report for Testing Data:")
print(classification_report(y_test, y_test_pred))

# Confusion matrix visualization
```

```
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix for SVM')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# ROC curve and AUC
probabilities = best_model.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, _ = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for SVM')
plt.legend(loc="lower right")
plt.show()
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits

Training Model Performance Metrics:
Accuracy: 0.7382, F1 Score: 0.7103

Testing Model Performance Metrics:
Accuracy: 0.6905, F1 Score: 0.6550

Classification Report for Training Data:
              precision    recall  f1-score   support

           0       0.71      0.96      0.82      1586
           1       0.86      0.40      0.55      1034

    accuracy                           0.74      2620
   macro avg       0.78      0.68      0.68      2620
weighted avg       0.77      0.74      0.71      2620


Classification Report for Testing Data:
              precision    recall  f1-score   support

           0       0.69      0.91      0.79       412
           1       0.68      0.32      0.43       244

    accuracy                           0.69       656
   macro avg       0.69      0.61      0.61       656
weighted avg       0.69      0.69      0.65       656
```


Confusion Matrix for SVM
376
36
167
77
0
1
True Label
0
1
Predicted Label
350
300
250
200
150
100
50



COMPARING PERFORMANCE METRICS AFTER TUNNING

```
performance_data = {
    'Random Forest': {
        'Training': {'Accuracy': 1.0000, 'Precision': 1.0000,
'Recall': 1.0000, 'F1 Score': 1.0000},
        'Testing': {'Accuracy': 0.7271, 'Precision': 0.7207, 'Recall':
0.7271, 'F1 Score': 0.7196}
    },
    'Decision Tree': {
        'Training': {'Accuracy': 0.7015, 'Precision': 0.7020,
'Recall': 0.7015, 'F1 Score': 0.7012},
        'Testing': {'Accuracy': 0.6799, 'Precision': 0.6815, 'Recall':
0.6799, 'F1 Score': 0.6806}
    },
    'XGBoost': {
```

```
        'Training': {'Accuracy': 0.9987, 'Precision': 0.9987,
'Recall': 0.9987, 'F1 Score': 0.9987},
        'Testing': {'Accuracy': 0.6936, 'Precision': 0.6862, 'Recall':
0.6936, 'F1 Score': 0.6876}
    },
    'Gradient Boosting': {
        'Training': {'Accuracy': 0.7929, 'Precision': 0.7957,
'Recall': 0.7929, 'F1 Score': 0.7924},
        'Testing': {'Accuracy': 0.6951, 'Precision': 0.6906, 'Recall':
0.6951, 'F1 Score': 0.6922}
    },
    'SVM': {
        'Training': {'Accuracy': 0.7516, 'Precision': 0.7552,
'Recall': 0.7516, 'F1 Score': 0.7507},
        'Testing': {'Accuracy': 0.6479, 'Precision': 0.6482, 'Recall':
0.6479, 'F1 Score': 0.6480}
    }
}

results = {
    "Random Forest": {
        "Training": {"Accuracy": 1.0000, "F1 Score": 1.0000},
        "Testing": {"Accuracy": 0.7241, "F1 Score": 0.7196}
    },
    "Decision Tree": {
        "Training": {"Accuracy": 0.8084, "F1 Score": 0.8000},
        "Testing": {"Accuracy": 0.7287, "F1 Score": 0.7000}
    },
    "XGBoost": {
        "Training": {"Accuracy": 0.7790, "F1 Score": 0.7635},
        "Testing": {"Accuracy": 0.7210, "F1 Score": 0.6944}
    },
    "Gradient Boosting Classifier": {
        "Training": {"Accuracy": 0.7718, "F1 Score": 0.7528},
        "Testing": {"Accuracy": 0.7073, "F1 Score": 0.6750}
    },
    "SVM": {
        "Training": {"Accuracy": 0.7382, "F1 Score": 0.7103},
        "Testing": {"Accuracy": 0.6905, "F1 Score": 0.6550}
    }
}

import matplotlib.pyplot as plt

# Create lists for plotting
models = list(results.keys())
training_accuracy = [results[model]["Training"]["Accuracy"] for model
in models]
testing_accuracy = [results[model]["Testing"]["Accuracy"] for model in
models]
```

```
training_f1 = [results[model]["Training"]["F1 Score"] for model in
models]
testing_f1 = [results[model]["Testing"]["F1 Score"] for model in
models]

# Plotting
fig, ax = plt.subplots(2, 1, figsize=(10, 10))
fig.suptitle('Model Comparison')

# Accuracy plot
ax[0].bar(models, training_accuracy, color='b', label='Training
Accuracy', alpha=0.6, width=0.6)
ax[0].bar(models, testing_accuracy, color='r', label='Testing
Accuracy', alpha=0.6, width=0.4)
ax[0].legend()
ax[0].set_ylabel('Accuracy')
ax[0].set_title('Model Accuracy')

# F1 Score plot
ax[1].bar(models, training_f1, color='b', label='Training F1 Score',
alpha=0.6, width=0.6)
ax[1].bar(models, testing_f1, color='r', label='Testing F1 Score',
alpha=0.6, width=0.4)
ax[1].legend()
ax[1].set_ylabel('F1 Score')
ax[1].set_title('Model F1 Score')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Model Comparison