Wine Quality Classification

November 30, 2023

1 Wine Quality Classification

1.1 Author

Joel Ruetas

1.2 Abstract

This study aims to evaluate red wine quality by analyzing its physicochemical properties. By identifying key measures that significantly i nfluence qu ality, I ai m to pr ovide a sy stematic assessment of red wine's quality attributes.

1.3 Background

This study is centered around the evaluation of red wine quality, a critical factor in the wine industry. The assessment of wine quality is traditionally based on either human sensory analysis or the examination of its physicochemical properties. These properties include crucial factors like pH, dissolved salts, sodium content, acidity, and density, which are known to influence the overall quality of the wine. As the market for high-quality wine continues to expand, there is an increasing demand for more efficient a nd r eliable m ethods t o p redict w ine q uality. Human sensory testing, while insightful, tends to be subjective and time-consuming. Therefore, there is a growing interest in utilizing machine learning techniques within wine informatics to offer a more objective and expedient approach. This involves classifying various wine attributes, including quality, through advanced data analysis, providing a new perspective on quality evaluation beyond traditional methods.

1.4 Objective

The goal of this research is to employ a binary classification m odel t o d istinguish b etween high and low-quality white wines. This classification w ill b e b ased o n s everal c ritical physicochemical properties of the wines. I am utilizing the red wine quality dataset available from the UCI Machine Learning Repository.

1.5 Data

I use the red wine quality dataset from the UCI Machine Learning Repository (http://www3.dsi.uminho.pt/pcortez/wine/winequality.zip) for this analysis.

1.6 References

Cortez,Paulo, Cerdeira,A., Almeida,F., Matos,T., and Reis,J.. (2009). *Wine Quality*. UCI Machine Learning Repository. https://doi.org/10.24432/C56S3T.

Giardina, C. (2022, May 11). *Wine rating system: Quality-to-price-ratio analysis and scoring.* VineRoutes Wine Magazine. https://vineroutes.com/wine-rating-system/#:~:text=Wines%20rated%2089%20and%20above,outstanding%20for%20its%20particular%20type

1.7 Import necessary libraries

[ ]:
```
import lightgbm as lgb
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pickle
import seaborn as sns

from sklearn.ensemble import ExtraTreesClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, auc, average_precision_score,␣
  ↪cohen_kappa_score, confusion_matrix, f1_score, matthews_corrcoef,␣
  ↪precision_recall_curve, precision_score, recall_score, roc_auc_score,␣
  ↪roc_curve
from sklearn.model_selection import GridSearchCV, learning_curve,␣
  ↪StratifiedShuffleSplit, train_test_split
from sklearn.preprocessing import PowerTransformer, StandardScaler
from sklearn.feature_selection import VarianceThreshold
from imblearn.over_sampling import SMOTE
```

1.8 Constants

[ ]:
```
random_state = 42
test_size = 0.2
```

1.9 Get Data

[ ]:
```
# Download and unzip the compressed file
!wget "http://www3.dsi.uminho.pt/pcortez/wine/winequality.zip"

!unzip winequality.zip
```

```
--2023-11-22 21:58:04--  http://www3.dsi.uminho.pt/pcortez/wine/winequality.zip
Resolving www3.dsi.uminho.pt (www3.dsi.uminho.pt)… 193.136.11.133
Connecting to www3.dsi.uminho.pt (www3.dsi.uminho.pt)|193.136.11.133|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 96005 (94K) [application/x-zip-compressed]
```

```
Saving to: ‘winequality.zip’

winequality.zip     100%[===================>]  93.75K   260KB/s    in 0.4s

2023-11-22 21:58:06 (260 KB/s) - ‘winequality.zip’ saved [96005/96005]

Archive:  winequality.zip
  inflating: winequality/winequality-names.txt
  inflating: winequality/winequality-names.txt.bak
  inflating: winequality/winequality-red.csv
  inflating: winequality/winequality-white.csv
```

[ ]:
```
# Read the CSV file
df_path = 'winequality/winequality-red.csv'
df = pd.read_csv(df_path, sep=';')

# View the first 5 rows
df.head()
```

[ ]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

1.9.1 Recode quality to a binary label:

In my analysis, I’ve redefined the quality rating of wines, originally scaled from 0 to 10, with 10 representing the highest quality. Upon examining the dataset, it’s observed that the ratings range from a minimum of 3 to a maximum of 9, with both the mean and median hovering around 6.

Guided by external wine rating standards, as outlined by sources like Vineroutes.com, wines with a rating of 7 or higher are generally considered of good quality. Based on this benchmark, I’ve simplified the rating system into a binary classification for a more straightforward analysis. In this new system, wines with a quality rating of 7 or above are labelled as **high quality** (assigned a value of 1), indicating their superior status. Conversely, wines with a rating of 6 or lower are categorized as **standard** (assigned a value of 0), reflecting their relatively average or below-average quality. This binary classification facilitates a more focused analysis on distinguishing higher-quality wines from the standard ones in the dataset.

```
[ ]: # Add binary classification label
     df['new_quality'] = np.where(df['quality'] > 6, 1, 0)

     # View the first 5 rows
     df.head()
```

```
[ ]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
     0            7.4              0.70         0.00             1.9      0.076
     1            7.8              0.88         0.00             2.6      0.098
     2            7.8              0.76         0.04             2.3      0.092
     3           11.2              0.28         0.56             1.9      0.075
     4            7.4              0.70         0.00             1.9      0.076

        free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
     0                 11.0                  34.0   0.9978  3.51       0.56
     1                 25.0                  67.0   0.9968  3.20       0.68
     2                 15.0                  54.0   0.9970  3.26       0.65
     3                 17.0                  60.0   0.9980  3.16       0.58
     4                 11.0                  34.0   0.9978  3.51       0.56

        alcohol  quality  new_quality
     0      9.4        5            0
     1      9.8        5            0
     2      9.8        5            0
     3      9.8        6            0
     4      9.4        5            0
```

```
[ ]: # Drop old quality column and rename new
     df = df.drop(columns = ['quality']) # Drop old column
     df = df.rename(columns = {'new_quality':'quality'}) # Rename new_quality back␣
      ↪to quality

     # View the first 5 rows
     df.head()
```

```
[ ]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
     0            7.4              0.70         0.00             1.9      0.076
     1            7.8              0.88         0.00             2.6      0.098
     2            7.8              0.76         0.04             2.3      0.092
     3           11.2              0.28         0.56             1.9      0.075
     4            7.4              0.70         0.00             1.9      0.076
```

```
   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        0
1      9.8        0
2      9.8        0
3      9.8        0
4      9.4        0
```

1.10 Exploratory Analysis Report

The subsequent section presents key findings and insights from an exploratory data analysis (EDA) process. The report encapsulates a wide range of data explorations, providing a visual and statistical overview of the dataset under study. Key highlights from this EDA report include:

1. **Missing Value Analysis:** Assessment of any missing data within the dataset and its potential implications on the analysis.
2. **Data Distribution and Trends:** Overview of the central tendencies, variability, and distribution patterns within the dataset.
3. **Correlation Analysis:** Investigation of potential relationships and correlations between different variables in the dataset, which could indicate underlying patterns or dependencies.
4. **Outlier Detection:** Identification of anomalies or outliers in the data that might affect the overall analysis or model performance.
5. **Comparative Analysis:** Comparative views of different subsets of data, allowing for a nuanced understanding of the dataset's characteristics.

These highlights form the crux of the EDA report, offering a foundational understanding of the dataset's characteristics and guiding subsequent analytical steps in the study.

```
[ ]: # Print the full summary
     df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

[ ]:
```
# Check if dataset has any nulls
df.isnull().any().any()
```

[ ]:
```
False
```

[ ]:
```
# Check for missing values
df.isnull().sum()
```

[ ]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

1.10.1 Dataset Characteristics:

The dataset I'm working with comprises a total of 12 variables, which include 11 numeric predictors that provide various quantitative insights into the dataset, and 1 categorical label. This categorical label has been transformed into a binary format for ease of analysis, where a value of 0 indicates **standard** quality wine and a value of 1 denotes **high quality** wine.

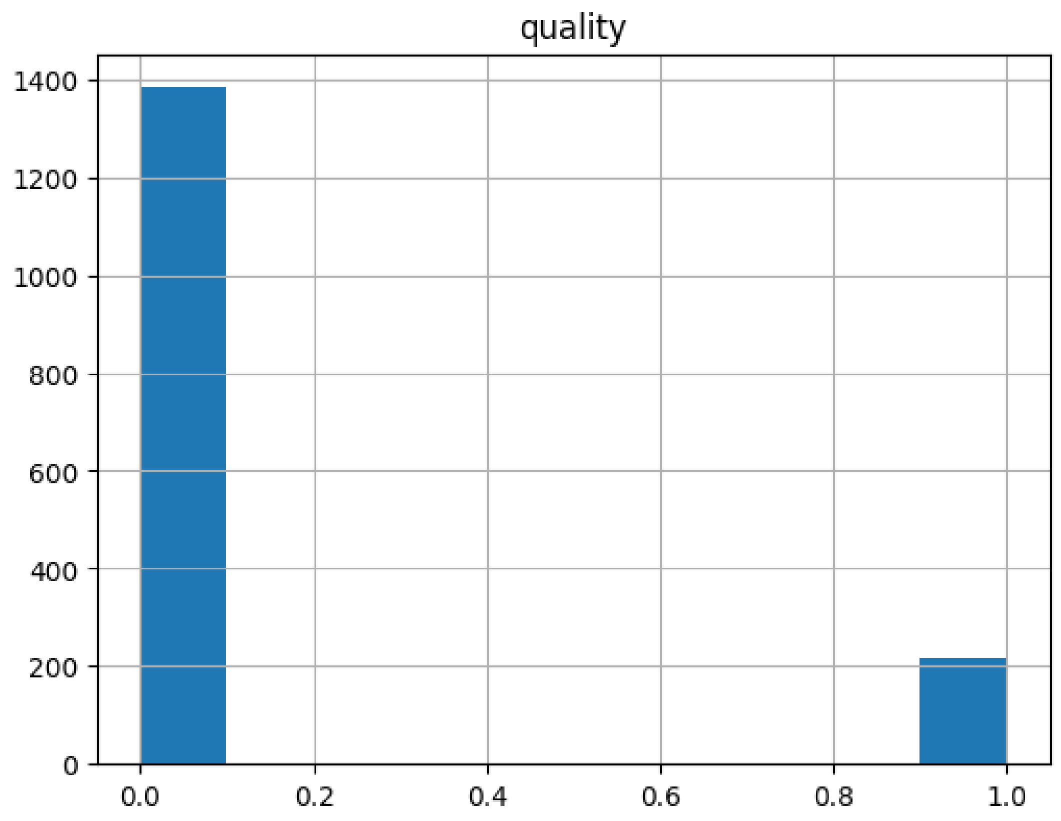
In terms of data volume, the dataset encompasses 1599 individual observations, each representing a unique set of data points across these 12 variables. This substantial number of observations ensures a comprehensive analysis and robustness in the patterns and insights derived.

A significant aspect of this dataset is its completeness. It contains no missing values, which is a noteworthy attribute in data analysis. The absence of missing values means that there will be no need for data imputation strategies, ensuring that the analysis is based on complete and undistorted information. This completeness enhances the reliability of any statistical or machine learning models applied to the dataset, as all the observations contribute valid and comprehensive data points for the analysis.

1.11 Distributions

```
[ ]: # Distribution of target variable
     df.hist(column = 'quality')
     df[['quality']].describe()
```

```
[ ]:            quality
     count  1599.000000
     mean      0.135710
     std       0.342587
     min       0.000000
     25%       0.000000
     50%       0.000000
     75%       0.000000
     max       1.000000
```



```
[ ]: # Determine proportion of high quality wines in dataset
     np.count_nonzero(df['quality'] == 1) / len(df['quality'])
```

[ ]: 0.1357098186366479

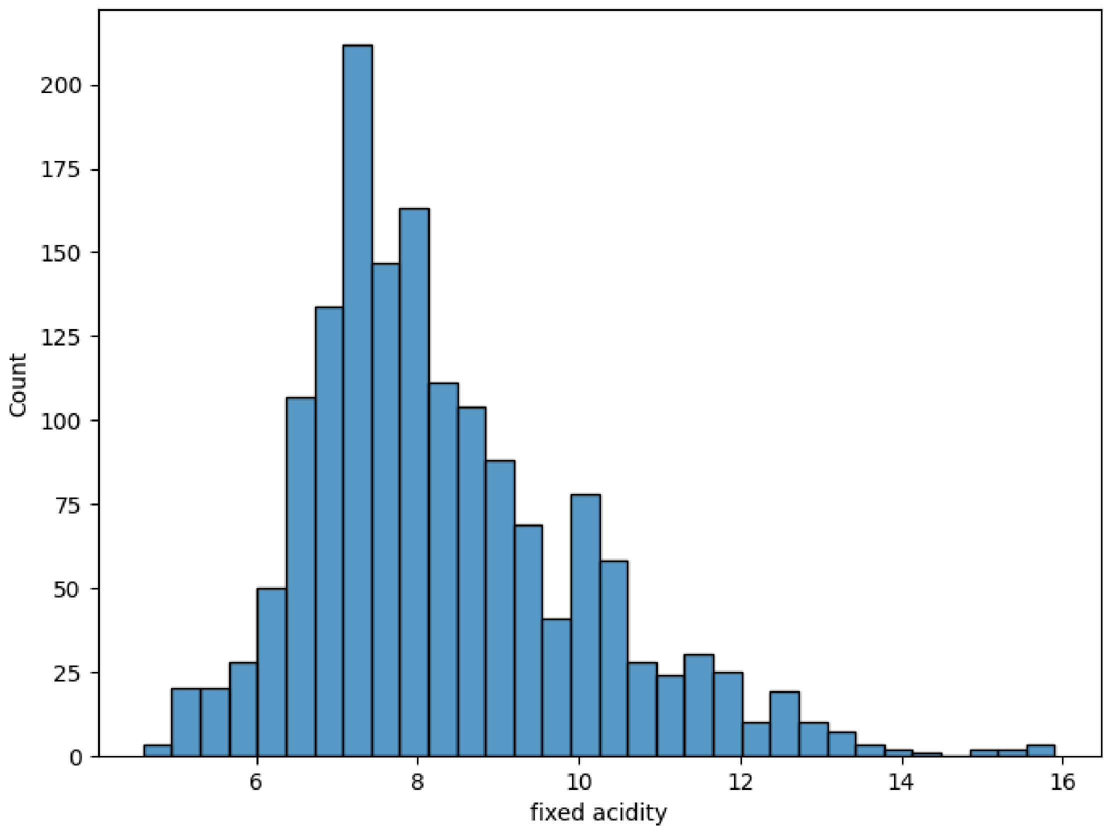
[ ]:
```
# Display the distribution of the numerical features
df.describe()
```

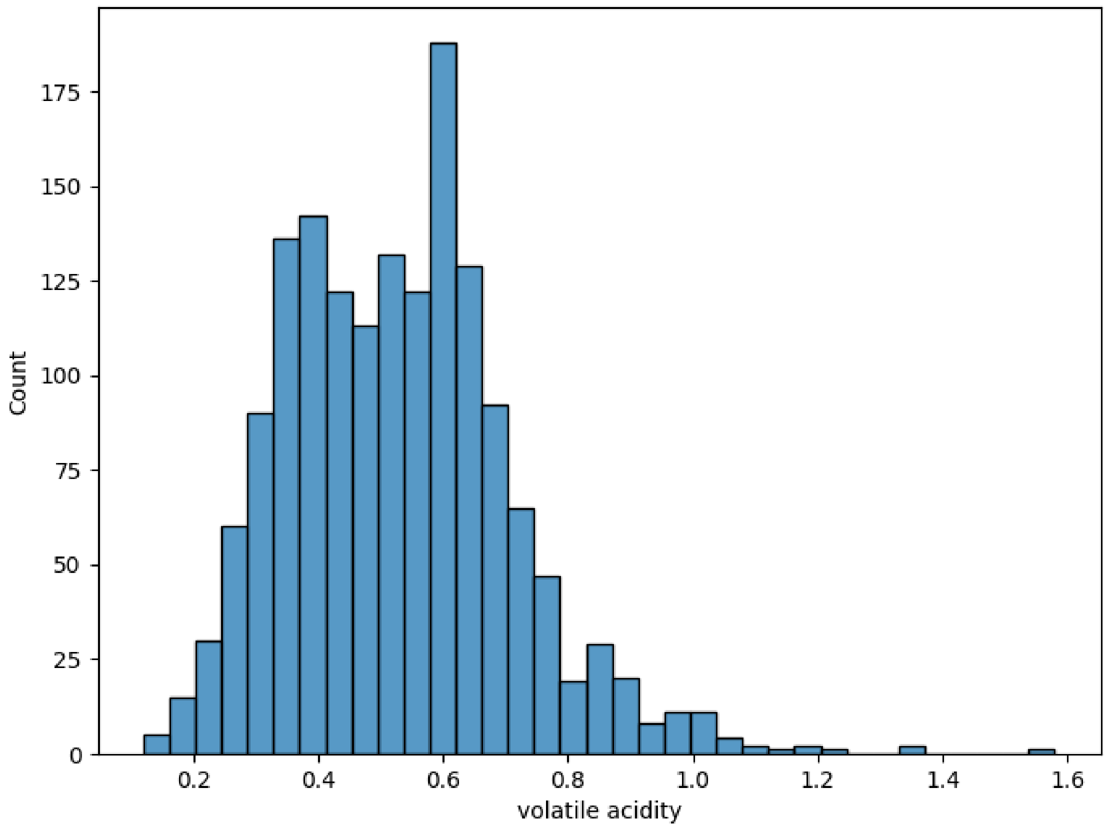
[ ]:
```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1599.000000       1599.000000  1599.000000     1599.000000
mean        8.319637          0.527821     0.270976        2.538806
std         1.741096          0.179060     0.194801        1.409928
min         4.600000          0.120000     0.000000        0.900000
25%         7.100000          0.390000     0.090000        1.900000
50%         7.900000          0.520000     0.260000        2.200000
75%         9.200000          0.640000     0.420000        2.600000
max        15.900000          1.580000     1.000000       15.500000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  1599.000000          1599.000000           1599.000000  1599.000000
mean      0.087467            15.874922             46.467792     0.996747
std       0.047065            10.460157             32.895324     0.001887
min       0.012000             1.000000              6.000000     0.990070
25%       0.070000             7.000000             22.000000     0.995600
50%       0.079000            14.000000             38.000000     0.996750
75%       0.090000            21.000000             62.000000     0.997835
max       0.611000            72.000000            289.000000     1.003690

                pH    sulphates      alcohol      quality
count  1599.000000  1599.000000  1599.000000  1599.000000
mean      3.311113     0.658149    10.422983     0.135710
std       0.154386     0.169507     1.065668     0.342587
min       2.740000     0.330000     8.400000     0.000000
25%       3.210000     0.550000     9.500000     0.000000
50%       3.310000     0.620000    10.200000     0.000000
75%       3.400000     0.730000    11.100000     0.000000
max       4.010000     2.000000    14.900000     1.000000
```

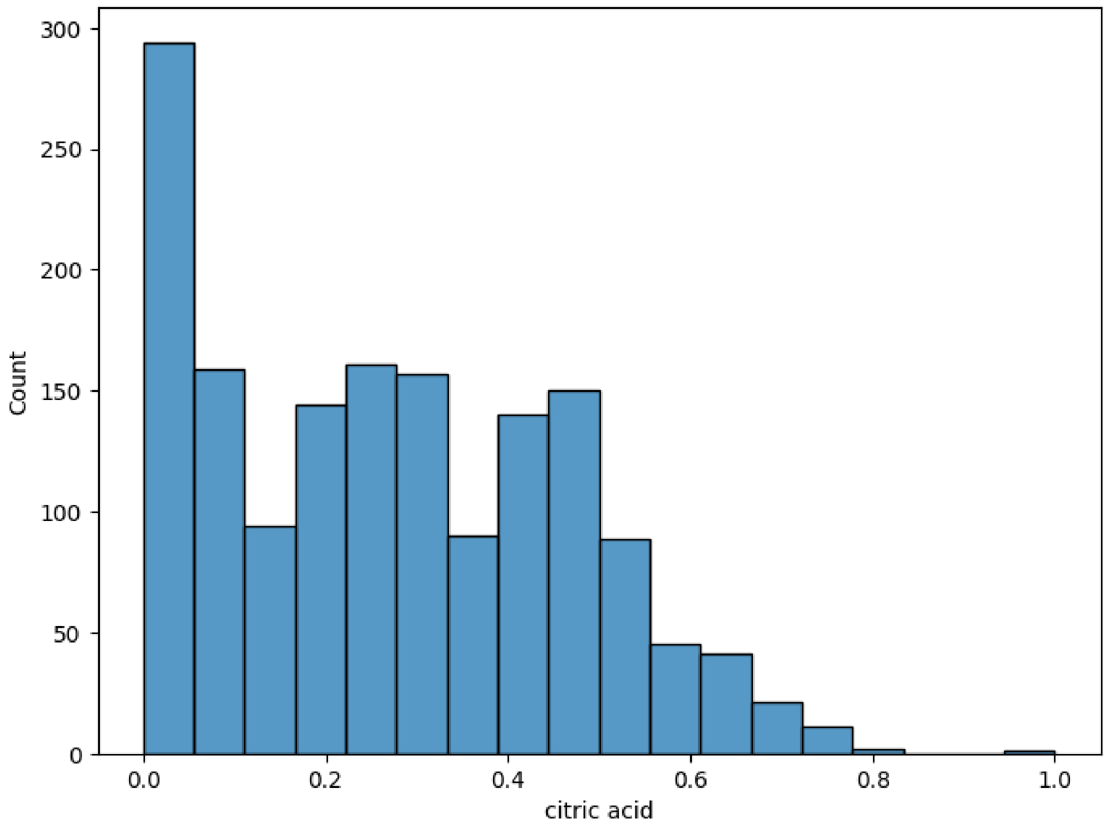
[ ]:
```
# Calculate the skewness of each column in the DataFrame
# Skewness is a measure of the asymmetry of the probability distribution of a␣
↪real-valued random variable
# A skewness value greater than 0 means that there is more weight in the right␣
↪tail of the distribution
# A skewness value less than 0 indicates more weight in the left tail of the␣
↪distribution
# A skewness value close to 0 (between -0.5 and 0.5) indicates a symmetric␣
↪distribution
df.skew()
```

[ ]:
```
fixed acidity           0.982751
volatile acidity        0.671593
citric acid             0.318337
residual sugar          4.540655
chlorides               5.680347
free sulfur dioxide     1.250567
total sulfur dioxide    1.515531
density                 0.071288
pH                      0.193683
sulphates               2.428672
alcohol                 0.860829
quality                 2.129363
dtype: float64
```

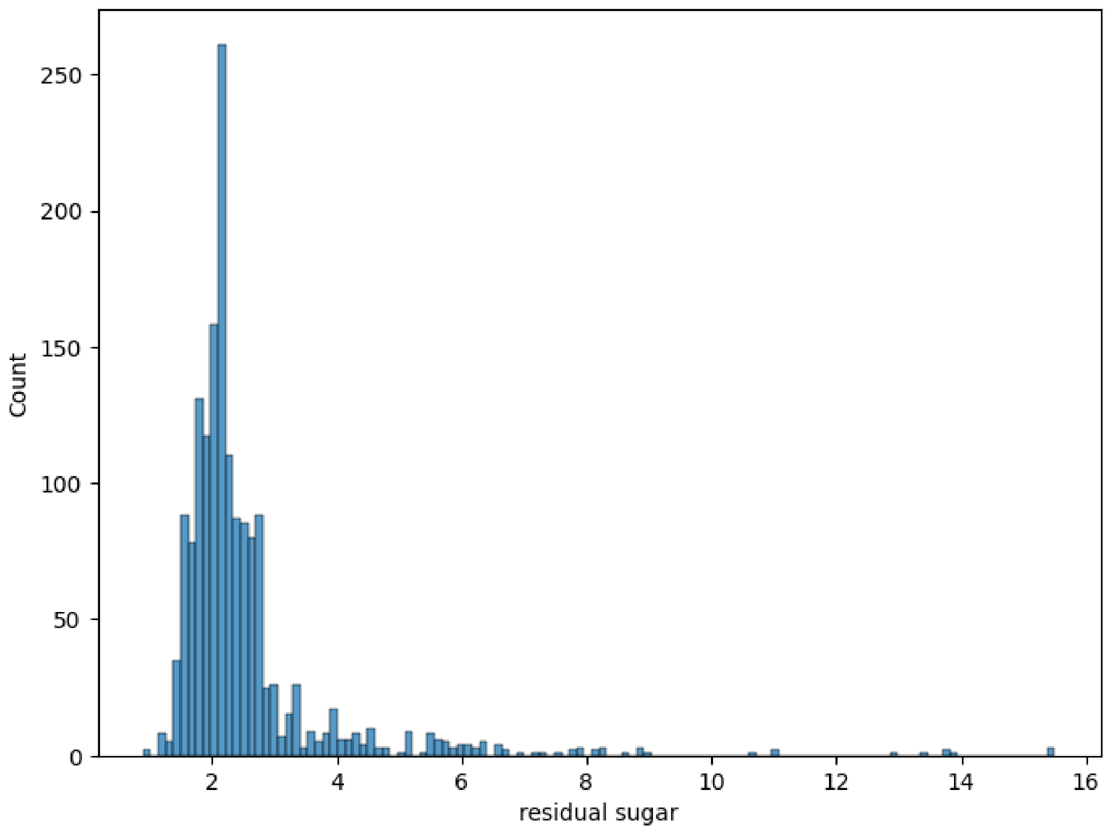
[ ]:
```python
# Extracting the list of column names from the DataFrame 'df'
cols = list(df.columns)

# Create a boxplot for every numeric feature (cols 0-11) to show outliers
for col in cols:
  # Creating a new figure with a specified size for each histogram
  plt.figure(figsize = (8, 6))

  # Creating a histogram for the current column using seaborn's histplot
  sns.histplot(df[col])

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


Count
fixed acidity
0
25
50
75
100
125
150
175
200
6
8
10
12
14
16

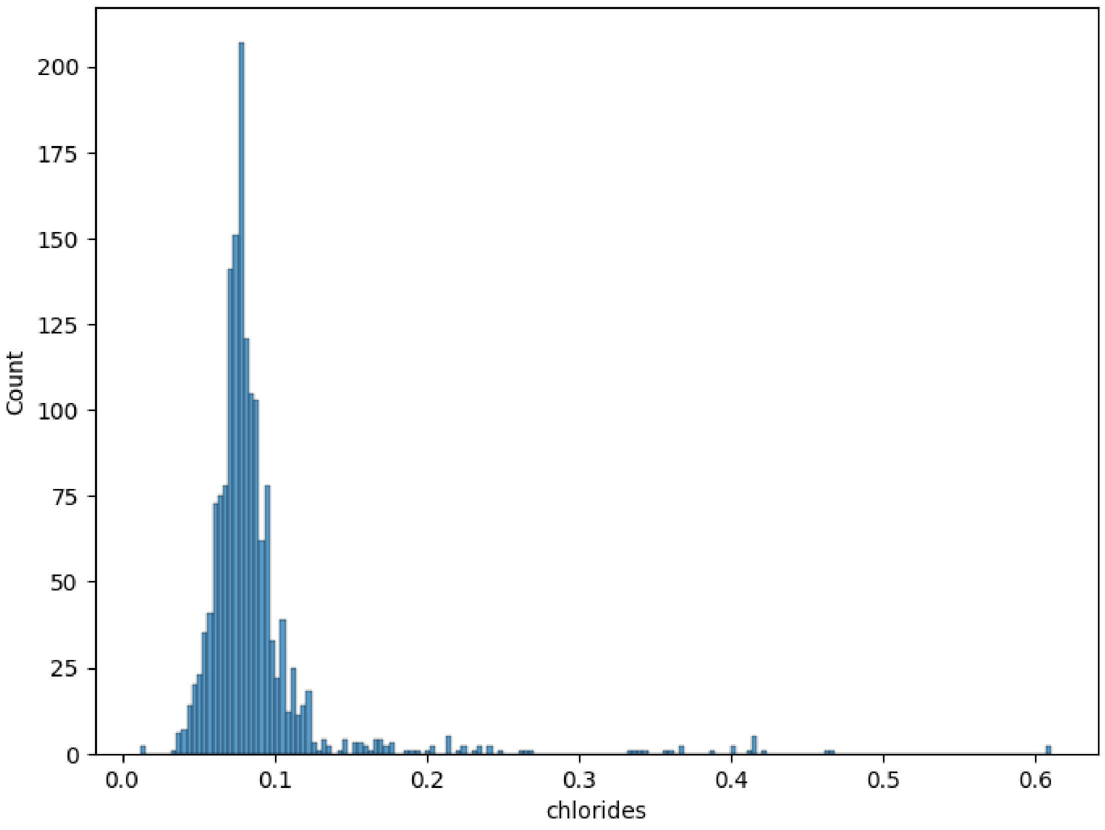

175
150
125
100
75
50
25
0
Count
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
volatile acidity

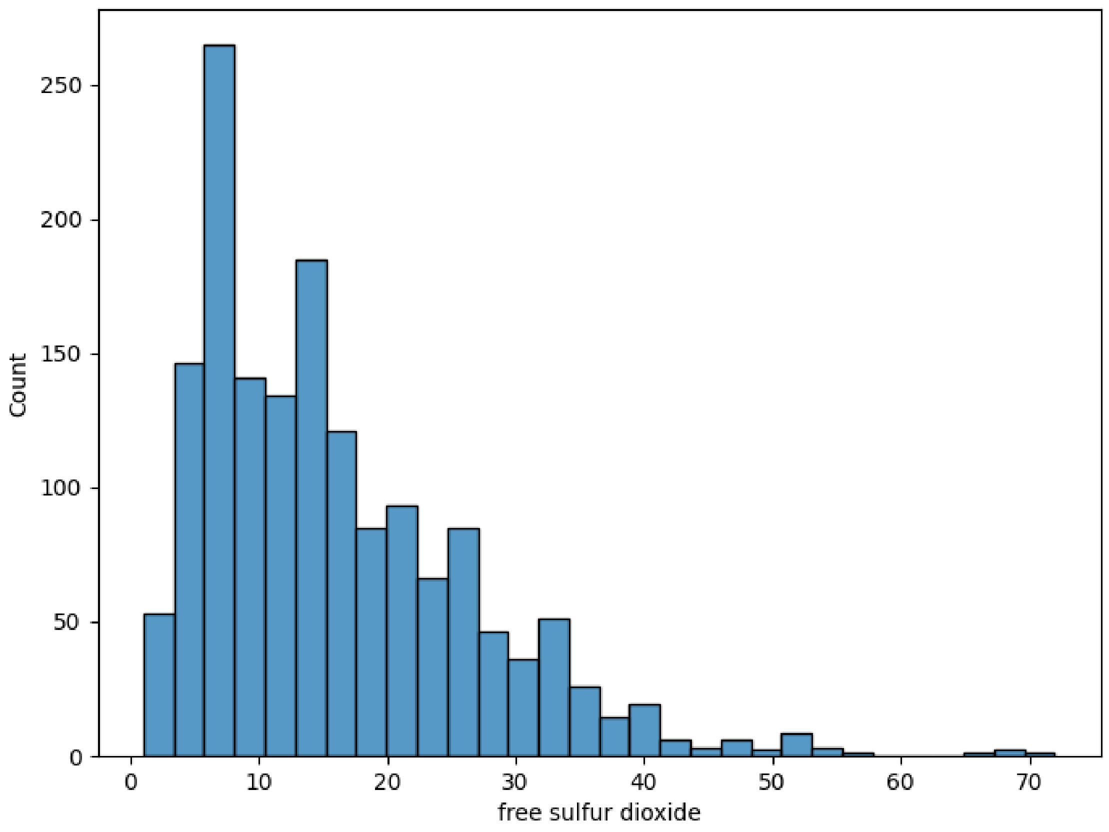


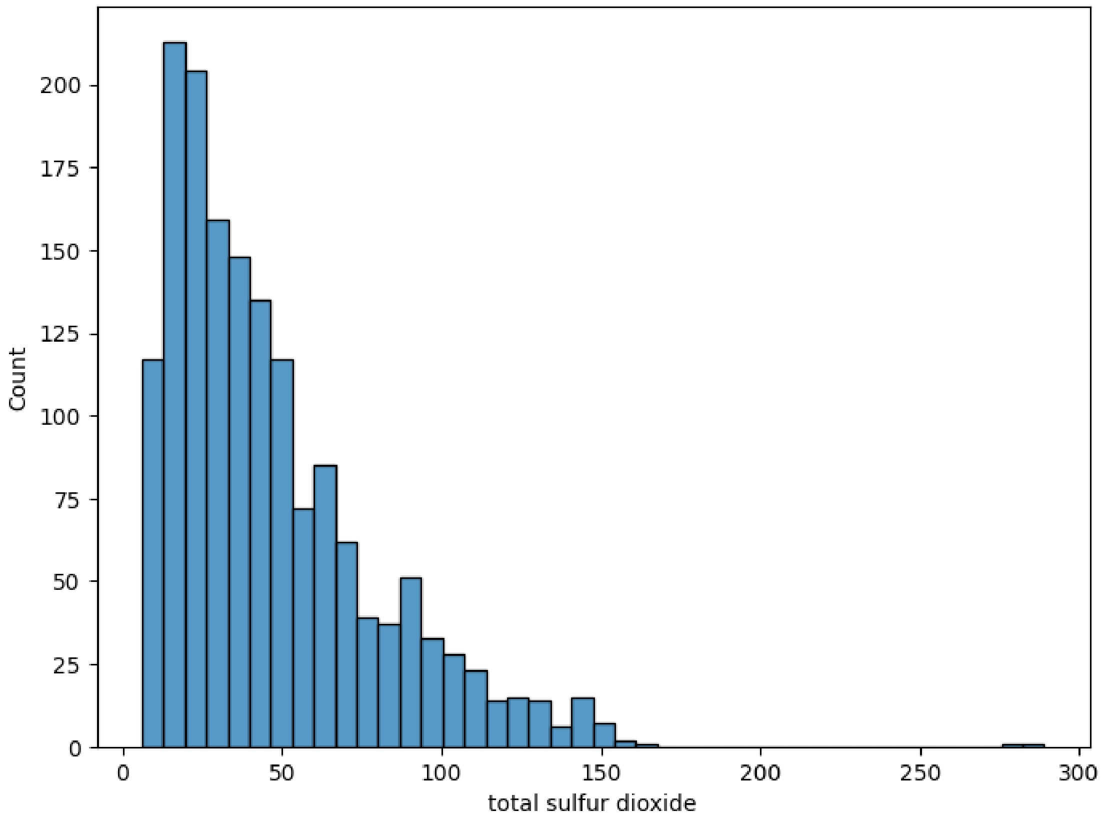
Count
300
250
200
150
100
50
0
0.0
0.2
0.4
0.6
0.8
1.0
citric acid

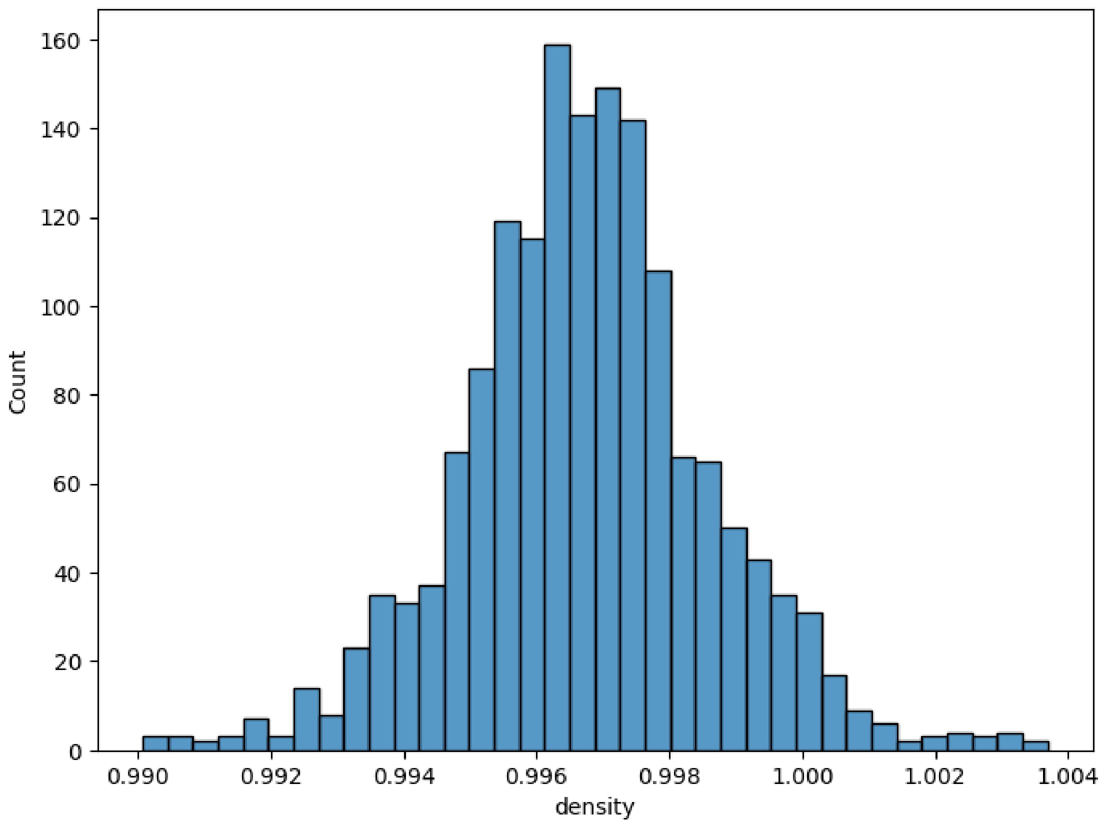


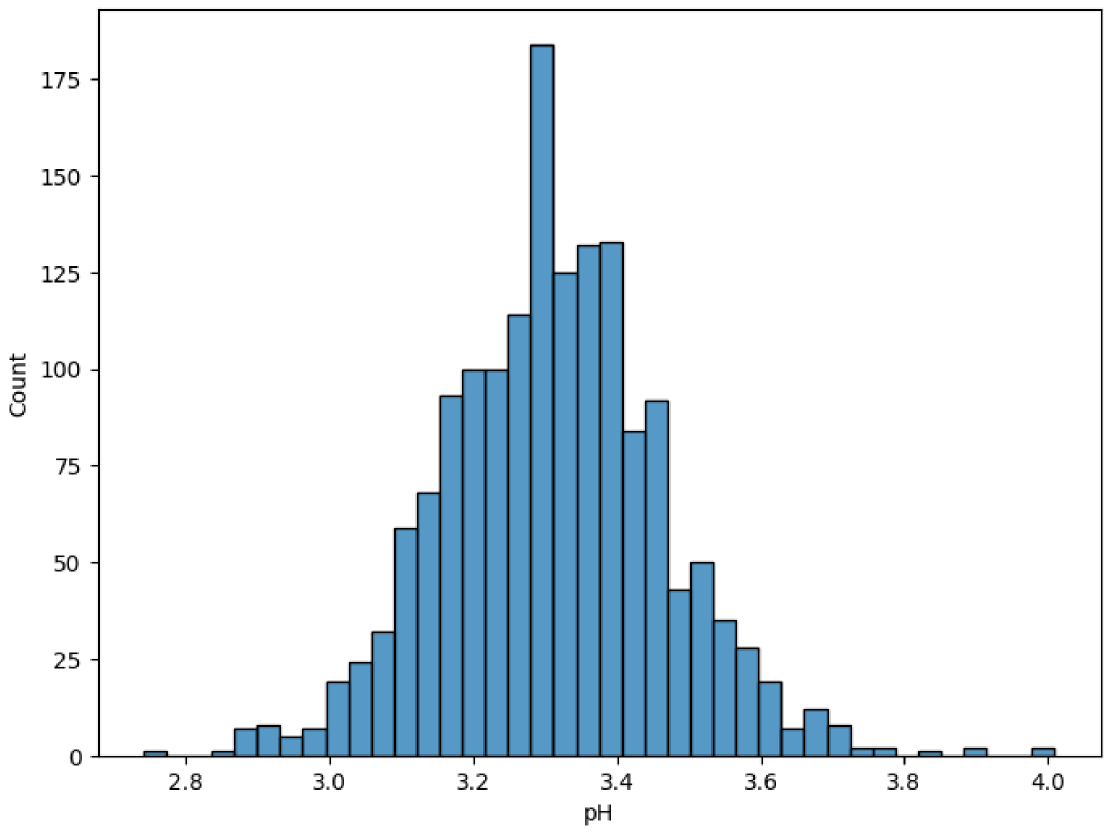
Count
residual sugar
0
50
100
150
200
250
2
4
6
8
10
12
14
16

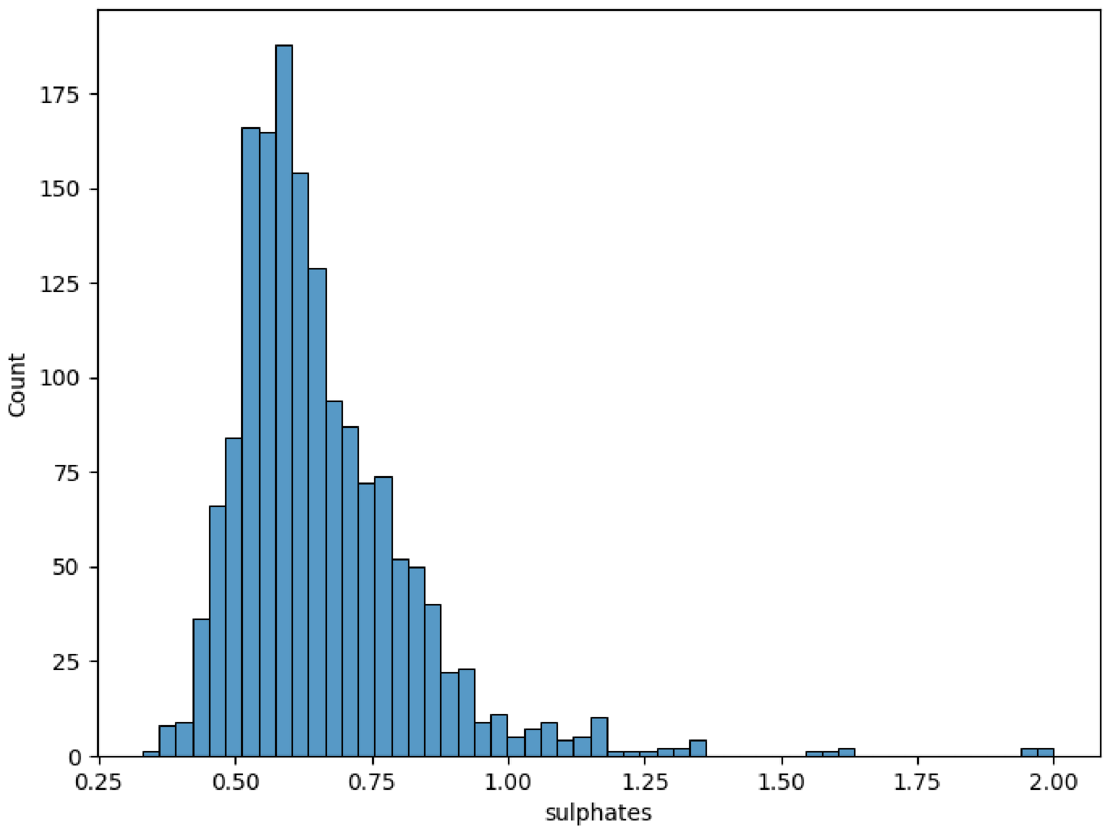

Count
chlorides
0
25
50
75
100
125
150
175
200
0.0
0.1
0.2
0.3
0.4
0.5
0.6

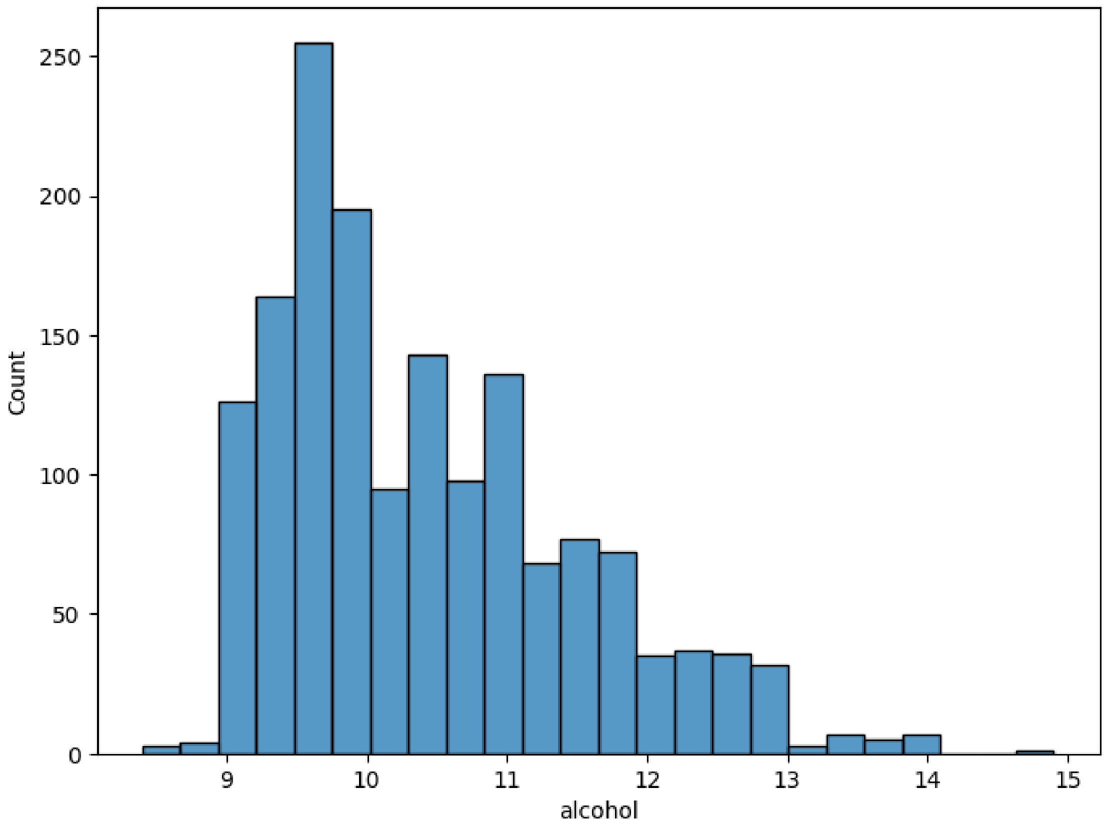

Count
0
50
100
150
200
250
0
10
20
30
40
50
60
70
free sulfur dioxide


Count
0
25
50
75
100
125
150
175
200
0
50
100
150
200
250
300
total sulfur dioxide

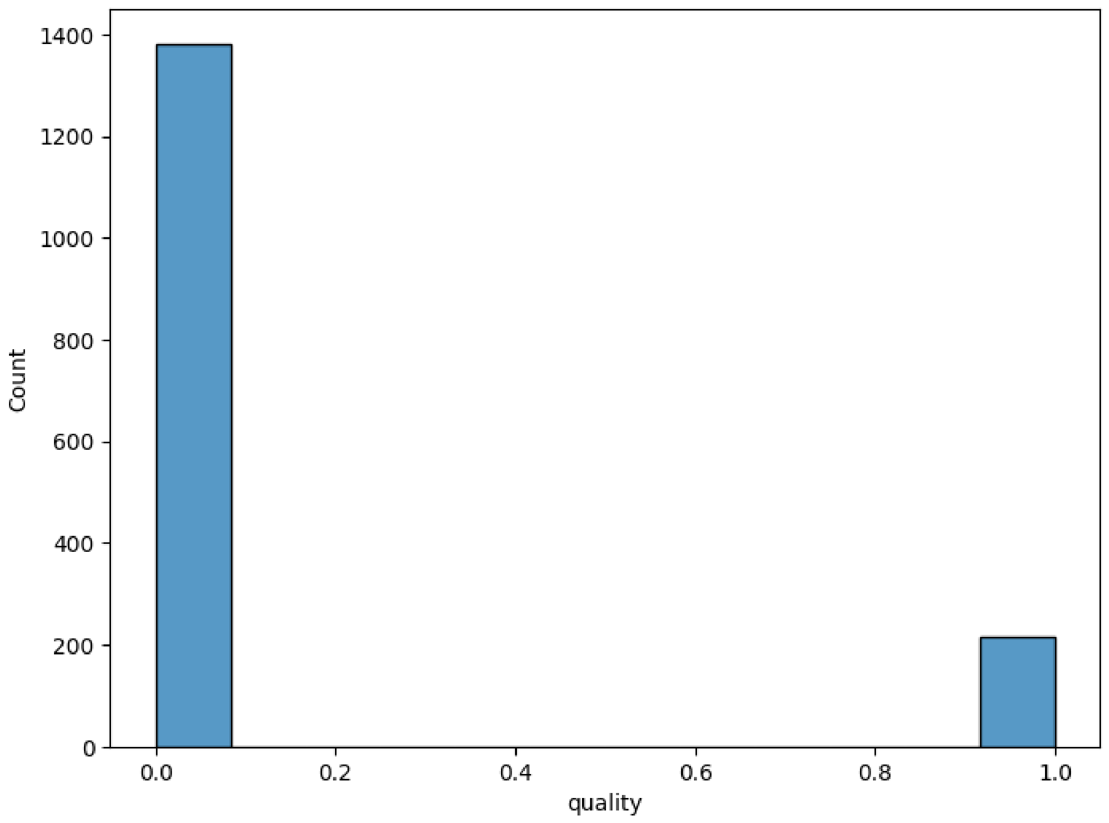

Count
density
0
20
40
60
80
100
120
140
160
0.990
0.992
0.994
0.996
0.998
1.000
1.002
1.004


175
150
125
100
75
50
25
0
Count
2.8
3.0
3.2
3.4
3.6
3.8
4.0
pH


Count
0
25
50
75
100
125
150
175
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
sulphates



Count
0
50
100
150
200
250
9
10
11
12
13
14
15
alcohol



1.11.1 Analysis of Distributions:

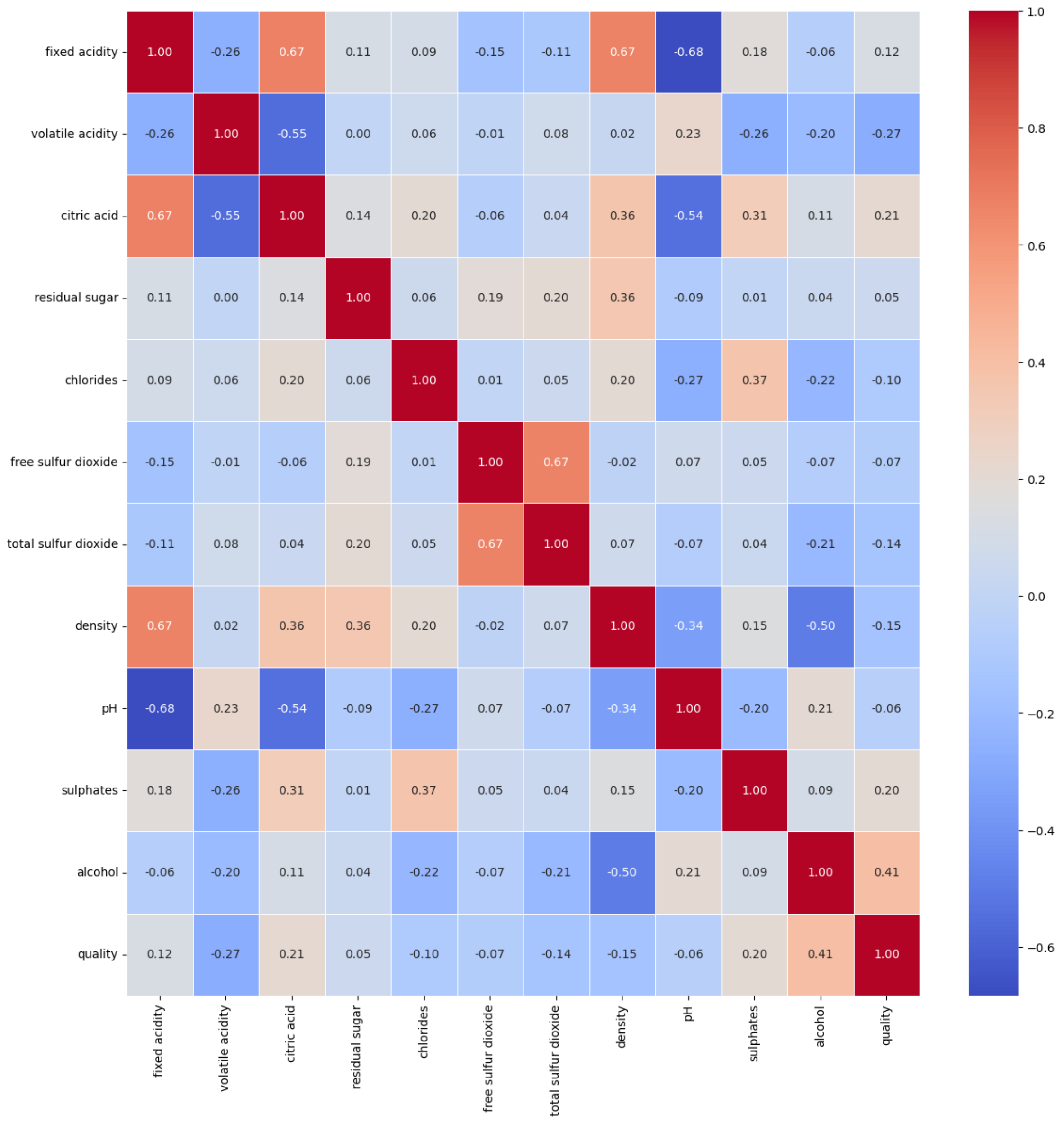
The dataset presents a significant imbalance in the label classes, with approximately 14% of the wines classified as **high quality** and the remaining 86% as **standard**. This disproportionate distribution suggests the need for careful consideration in our approach, particularly in terms of sampling methods. Specifically, I might explore strategies like undersampling to balance the classes more effectively. Additionally, this imbalance necessitates the selection of an appropriate performance metric that can accurately reflect the model's efficacy in dealing with such skewed class distributions.

Regarding the numeric predictors, I observe notable disparities in their scales. This variation in scale could potentially bias the model, overemphasizing certain predictors over others. To mitigate this risk and ensure a fair representation and impact of each predictor, it is imperative to standardize or normalize these features before proceeding with model training.

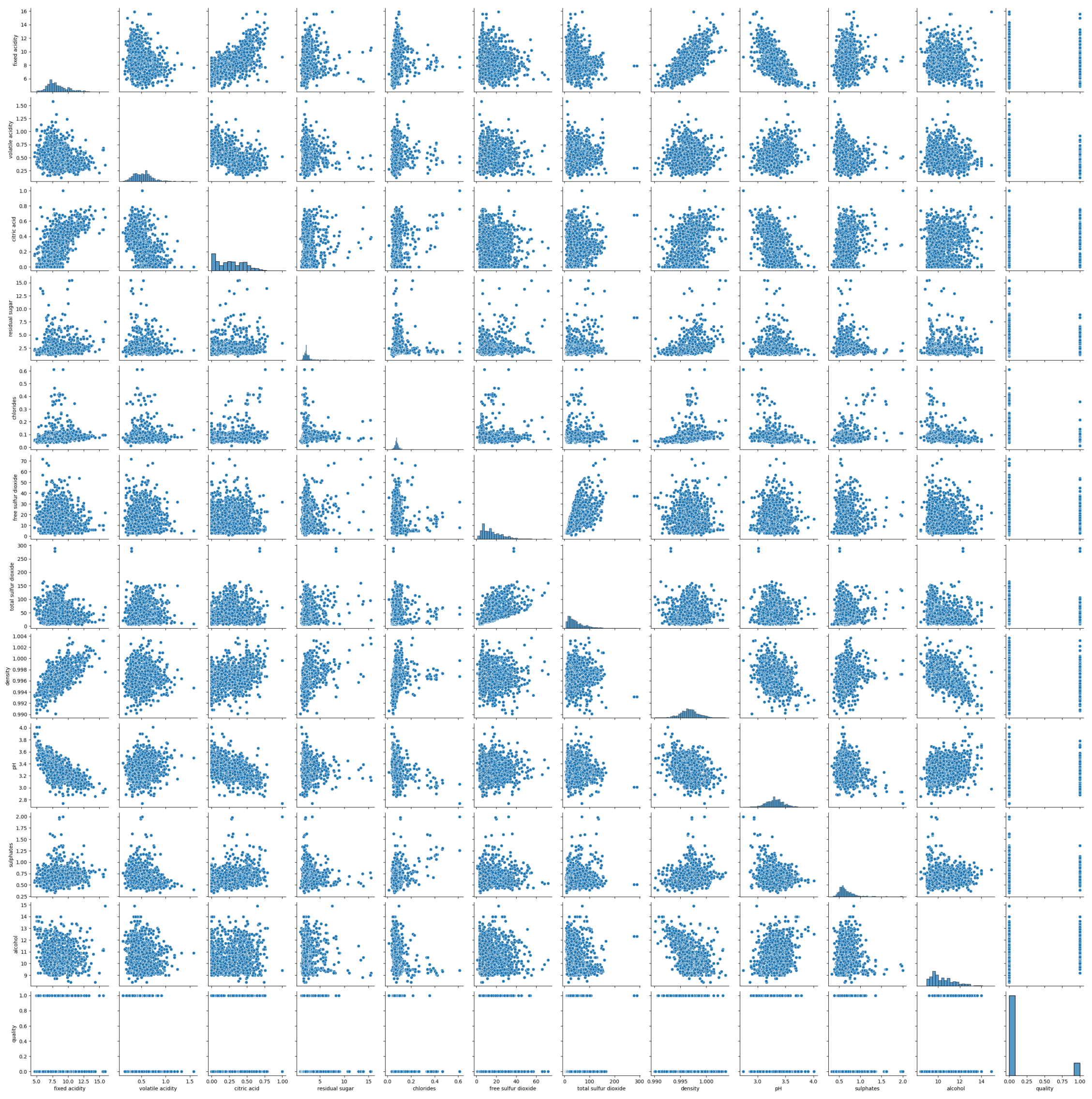
Furthermore, an examination of the distribution patterns of these predictors reveals mixed characteristics. While many exhibit distributions close to normal, several key predictors — notably alcohol content, residual sugar, and citric acid levels — display pronounced deviations from normality. These skewed distributions could affect the model's performance. To address this, I plan to implement appropriate data transformations in my experimental setup, aiming to approximate a more normal distribution for these variables. This step is crucial to enhance the robustness and accuracy of the models we intend to train.

1.12 Correlations

```
[ ]: # Create a Heatmap

     # Setting the size of the plot
     plt.figure(figsize = (15, 15))

     # Creating a heatmap to visualize the correlation matrix of the DataFrame 'df'
     # 'corr(numeric_only=True)' computes the correlation between numeric columns␣
      ↪only
     # 'annot=True' displays the correlation coefficients in the heatmap
     # 'fmt=".2f"' formats the correlation coefficients to two decimal places
     # 'linewidths=0.7' sets the width of the lines that will divide each cell in␣
      ↪the heatmap
     # 'cmap="coolwarm"' sets the color scheme of the heatmap to 'coolwarm',
     # which is a diverging colormap (from cool to warm colors)
     sns.heatmap(df.corr(numeric_only = True), annot = True, fmt = ".2f", linewidths␣
      ↪= 0.7, cmap = "coolwarm")

     # Displaying the heatmap
     plt.show() # Renders the heatmap in the output
```



```
[ ]: # Create a Pairplot

     # A pairplot displays pairwise relationships in a dataset
     # By default, this plot will show scatter plots for each pair of variables and␣
      ↪histograms on the diagonal
     sns.pairplot(df)

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



1.12.1 Analysis of Correlation Patterns:

In the dataset, I observe noteworthy correlation trends among various features, which are crucial for understanding the interdependencies between different wine characteristics. There are moderate to strong negative correlations identified between the alcohol content and the density of the wine, as well as between the pH level and fixed acidity. These negative correlations imply that as one variable increases, the other tends to decrease, suggesting an inverse relationship.

Additionally, there is a strong positive correlation observed between the density of the wine and its residual sugar content. This positive correlation indicates that these two variables tend to increase or decrease in tandem. A higher level of residual sugar often corresponds to an increase in the wine's density, and vice versa.

Given these correlation trends, an important consideration for my analysis is the issue of multicollinearity, which arises when independent variables in a regression model are highly correlated. Multicollinearity can distort the model's estimation and make the results less reliable. To address this, I will incorporate strategies in my experimental setup to mitigate the effects of multicollinearity before I proceed with training the models. These strategies might include techniques such as feature selection or regularization, which can help in reducing the impact of these correlations on the model's performance and ensure a more accurate and robust analysis.

1.13 Dataset Splitting Strategy:

My approach to splitting the dataset involves a meticulous process to ensure the integrity and effectiveness of the model evaluation. Initially, I allocated 10% of the dataset as a test set, which will remain untouched until the final stages of the model development. This test set is essentially a reserve of unseen data, crucial for assessing the model's performance in real-world scenarios.

To prevent any potential data leakage, which could bias the model, this 10% the test set is segregated right at the outset, before I engage in any form of feature engineering. The process of separating this subset involves a random shuffling method, ensuring that the test set is a representative sample of the overall dataset.

The bulk of the dataset, constituting the remaining 90%, is then prepared for further processing. This larger portion will undergo feature engineering to enhance and refine the dataset for better model performance. Post feature engineering, this part of the dataset is further divided into two distinct sets: the training set and the validation set.

A critical aspect of this division is the use of stratified sampling to split the data. Stratified sampling ensures that both the training and validation sets mirror the original distribution of the target labels in the dataset. This method is particularly important given the earlier noted imbalance in label classes. By maintaining a consistent proportion of **high quality** and **standard** labels in both sets, we ensure that the model is trained and validated on datasets that accurately reflect the real-world distribution of wine quality. This approach helps in building a model that is well-tuned and generalizable to diverse data scenarios.

[ ]:
```
# Splitting the DataFrame into two parts: 90% for modeling and 10% for unseen␣
↪predictions

# Randomly sampling 90% of the data from 'df' for modeling purposes
# 'frac=0.90' specifies the fraction of data to sample (90%)
# 'random_state=42' ensures that the random sampling is reproducible
data = df.sample(frac = 0.90, random_state = random_state)

# Creating a separate dataset for unseen data (the remaining 10%)
# This is done by dropping the indices of 'data' from 'df', leaving the␣
↪unsampled 10%
data_unseen = df.drop(data.index)

# Resetting the index of the 'data' DataFrame
# 'inplace=True' modifies the DataFrame in place
# 'drop=True' prevents the old index from being added as a new column in the␣
↪DataFrame
```

```
data.reset_index(inplace = True, drop = True)

# Resetting the index of the 'data_unseen' DataFrame in the same manner
data_unseen.reset_index(inplace = True, drop = True)

# Printing the shape (number of rows and columns) of the 'data' DataFrame
# This shows the size of the dataset that will be used for modeling
print('Data for Modeling: ' + str(data.shape))

# Printing the shape of the 'data_unseen' DataFrame
# This indicates the size of the dataset set aside for final model testing
print('Unseen Data For Predictions: ' + str(data_unseen.shape))
```

```
Data for Modeling: (1439, 12)
Unseen Data For Predictions: (160, 12)
```

[ ]:
```
# Display the first 5 rows of the 'data' DataFrame
data.head()
```

[ ]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.7              0.56         0.08            2.50      0.114
1            7.8              0.50         0.17            1.60      0.082
2           10.7              0.67         0.22            2.70      0.107
3            8.5              0.46         0.31            2.25      0.078
4            6.7              0.46         0.24            1.70      0.077

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 14.0                  46.0   0.9971  3.24       0.66
1                 21.0                 102.0   0.9960  3.39       0.48
2                 17.0                  34.0   1.0004  3.28       0.98
3                 32.0                  58.0   0.9980  3.33       0.54
4                 18.0                  34.0   0.9948  3.39       0.60

   alcohol  quality
0      9.6        0
1      9.5        0
2      9.9        0
3      9.8        0
4     10.6        0
```

[ ]:
```
# Display the first 5 rows of the 'data_unseen' DataFrame
data_unseen.head()
```

[ ]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.8              0.61         0.29             1.6      0.114
1            8.9              0.62         0.18             3.8      0.176
2            8.9              0.22         0.48             1.8      0.077
```

```
3            7.6              0.39         0.31             2.3      0.082
4            5.2              0.32         0.25             1.8      0.103

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                  9.0                  29.0   0.9974  3.26       1.56
1                 52.0                 145.0   0.9986  3.16       0.88
2                 29.0                  60.0   0.9968  3.39       0.53
3                 23.0                  71.0   0.9982  3.52       0.65
4                 13.0                  50.0   0.9957  3.38       0.55

   alcohol  quality
0      9.1        0
1      9.2        0
2      9.4        0
3      9.7        0
4      9.2        0
```

1.14 Data Cleaning Strategy:

My data cleaning approach is designed to refine and prepare the dataset for effective analysis. In this dataset, I encounter primarily numeric features, except for the target label, which is already one-hot encoded. This simplifies my process, as no additional processing is required for categorical variables. Another advantage is the absence of missing values in our data, eliminating the need for imputation strategies or the removal of rows due to incomplete data.

However, during the exploratory data analysis, I have identified two notable aspects that require attention. Firstly, the numeric features exhibit varying scales, which can potentially skew the model's interpretation and valuation of these features. Secondly, several numeric features are non-normally distributed, which could impact the model's performance. Both these issues will be addressed comprehensively in the experiment setup phase, ensuring that the data is optimally conditioned for modelling.

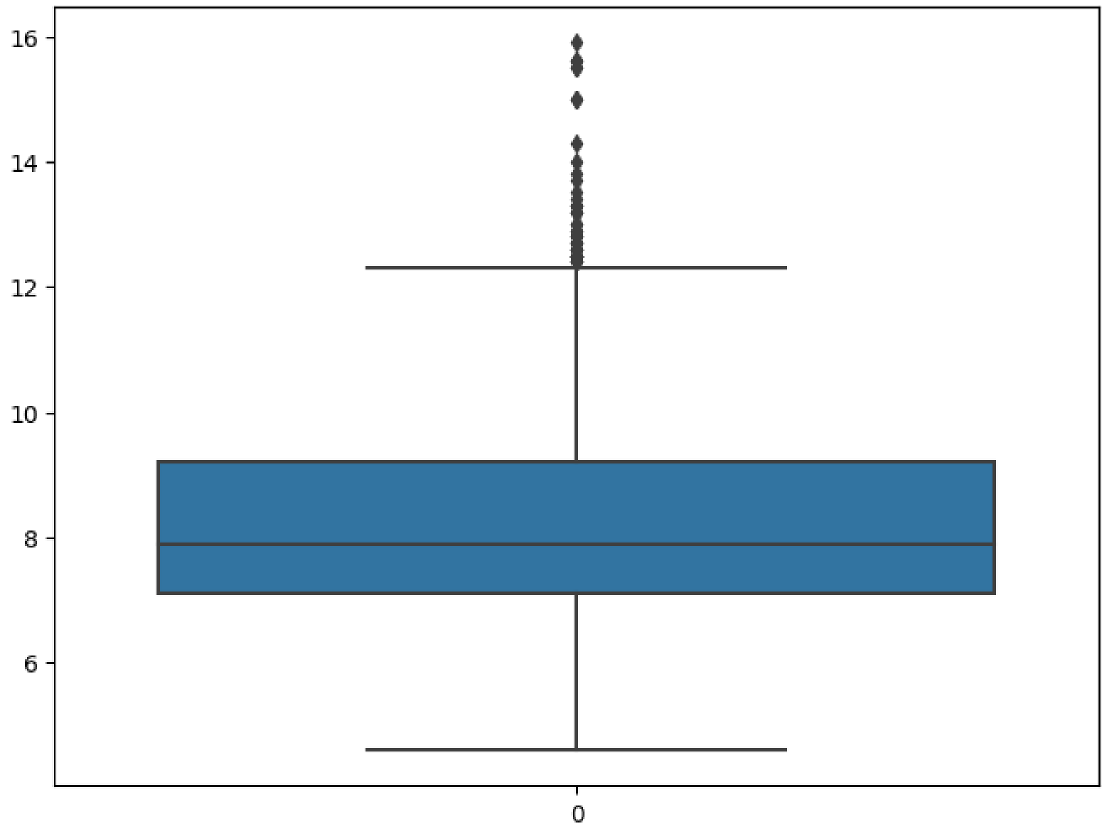
1.14.1 Outlier Analysis Approach:

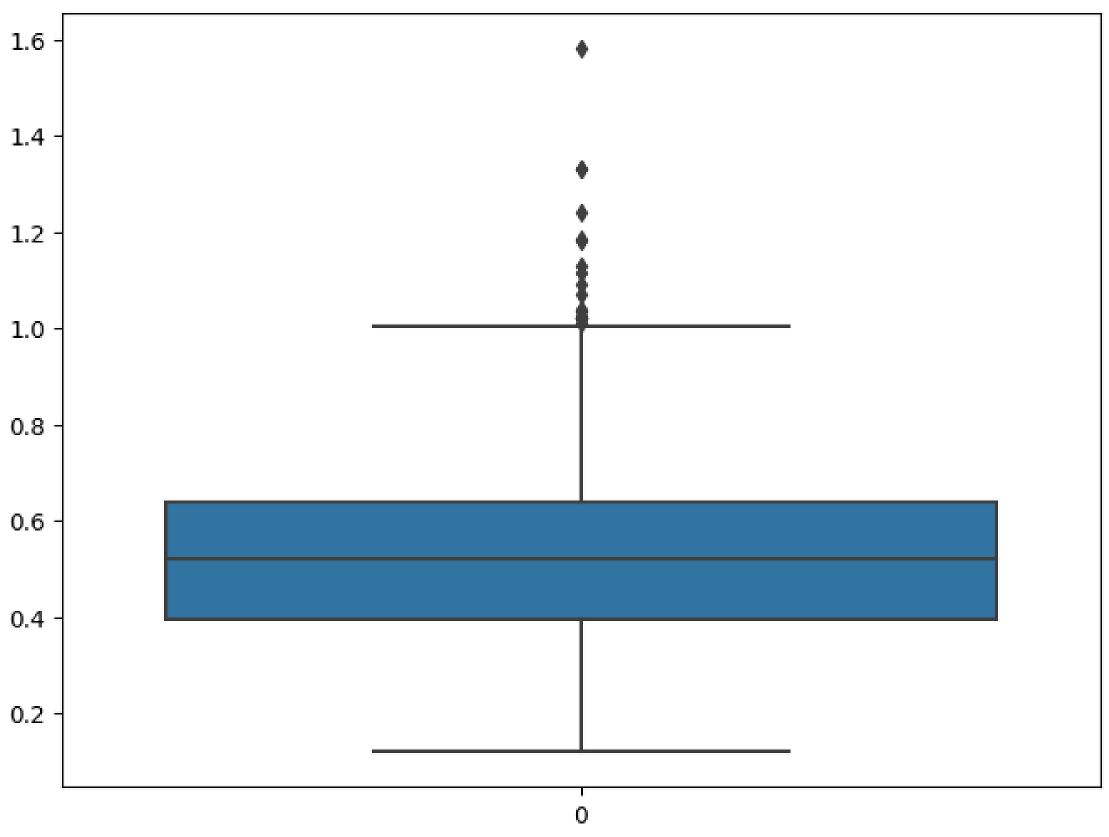
My strategy for dealing with outliers is contingent on the dataset's characteristics and the extent of outlier presence. I consider two scenarios:

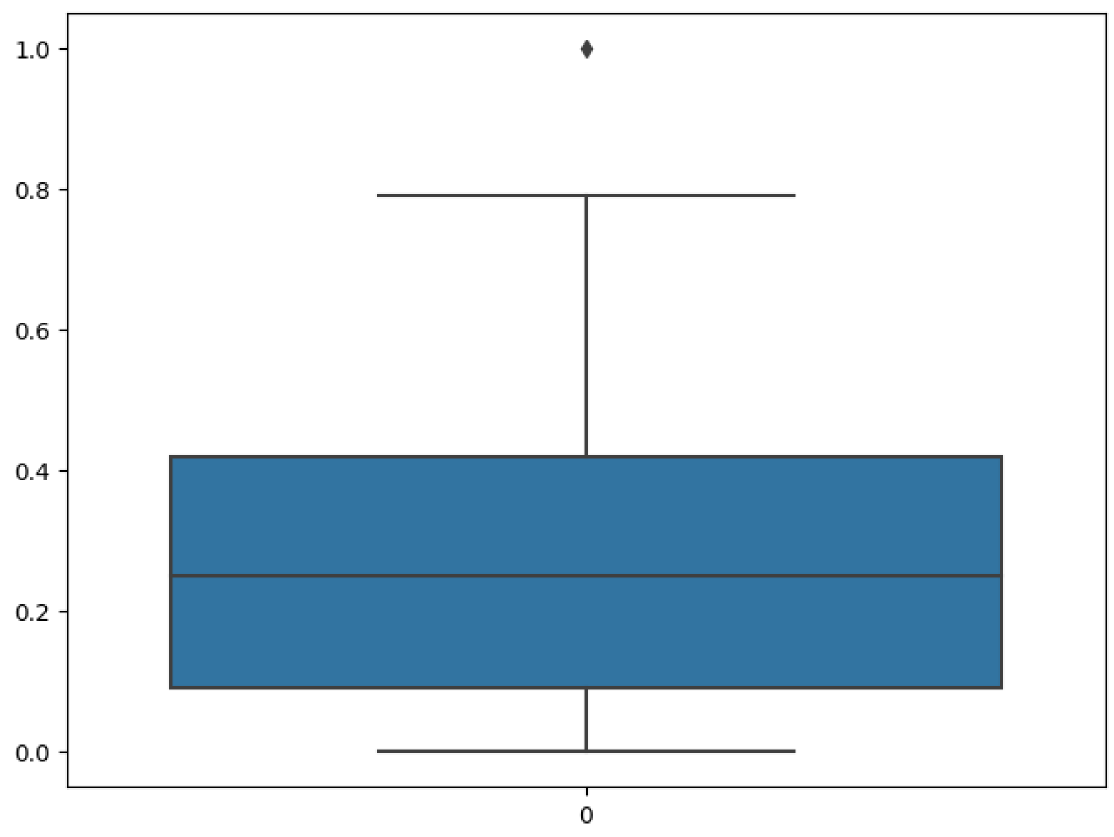
1. **Abundant Data with Sparse Outliers:** In cases where the dataset is sizeable but contains only a few rows with outlier values in any column, my approach leans towards removing these rows. This decision is based on the premise that the elimination of a small number of outliers will not significantly impact the dataset's integrity yet will enhance the model's accuracy.
2. **Limited Data with Numerous Outliers:** Conversely, in situations where our dataset is relatively small and a larger proportion of rows exhibit outlier values, a more nuanced approach is required. Here, I opt to cap the values at the 5th and 95th percentiles. This method effectively reduces the impact of extreme outliers while preserving the bulk of the data, ensuring that our model is trained on a dataset that is representative of the broader population, albeit with moderated extremities.

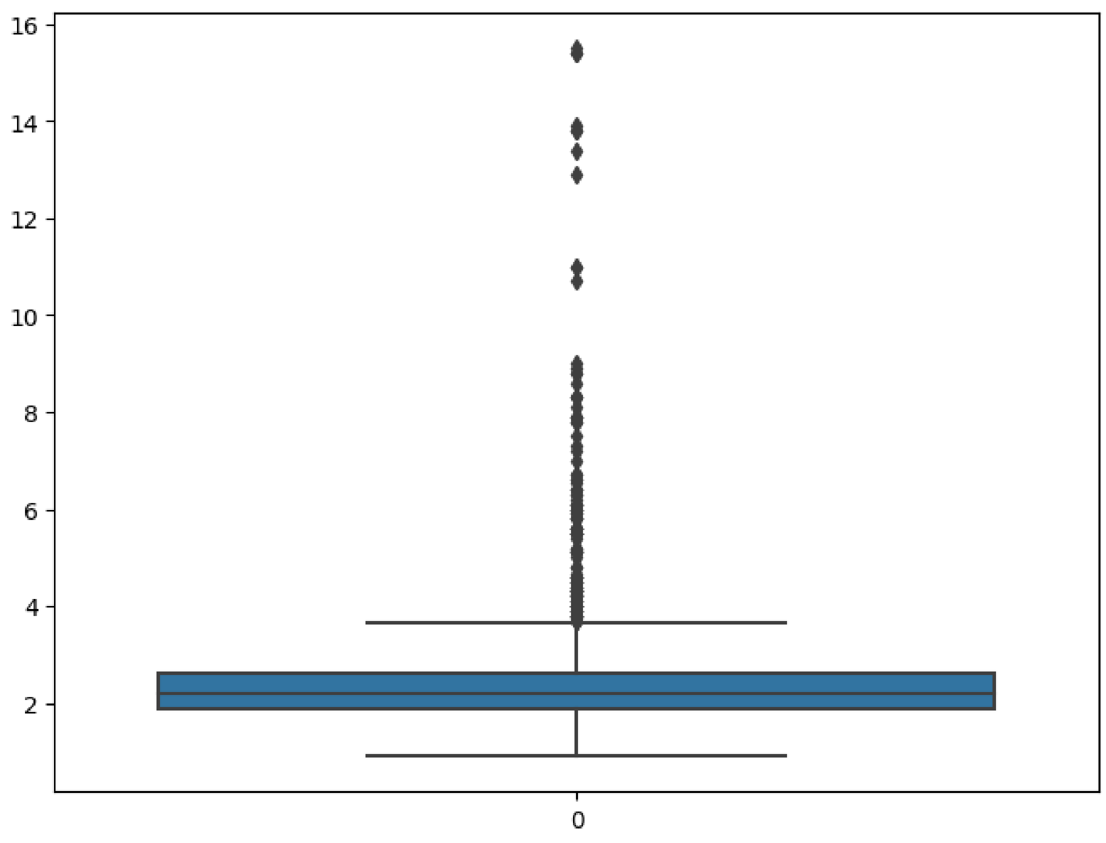
This two-pronged strategy for outlier management is designed to balance the need for a clean, representative dataset with the preservation of as much data as possible, thereby ensuring robustness in the subsequent modelling phase.

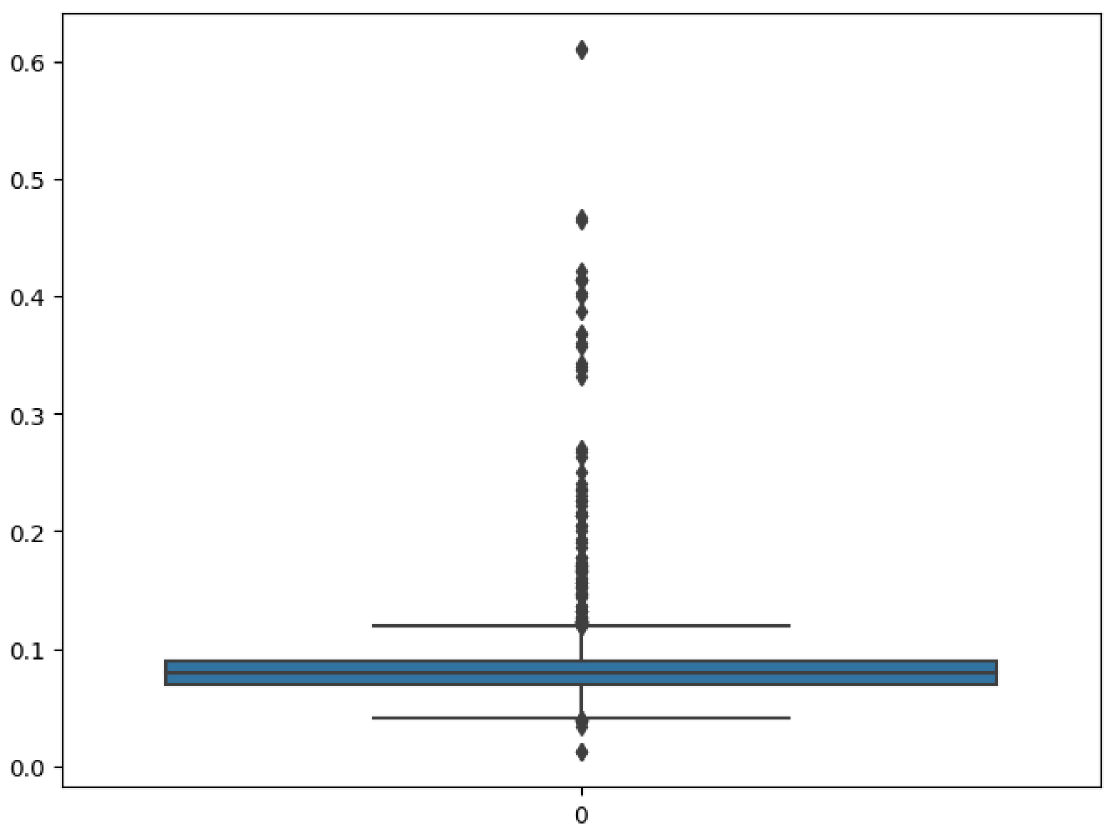
```
[ ]: # Outlier analysis
cols = list(data.columns)

# Create boxplot for every numeric feature (cols 0-11) to show outliers
for col in cols[0:-1]:

  # Set the size of the plot
  plt.figure(figsize = (8, 6))

  # Create the boxplot for the current column
  sns.boxplot(data[col])

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

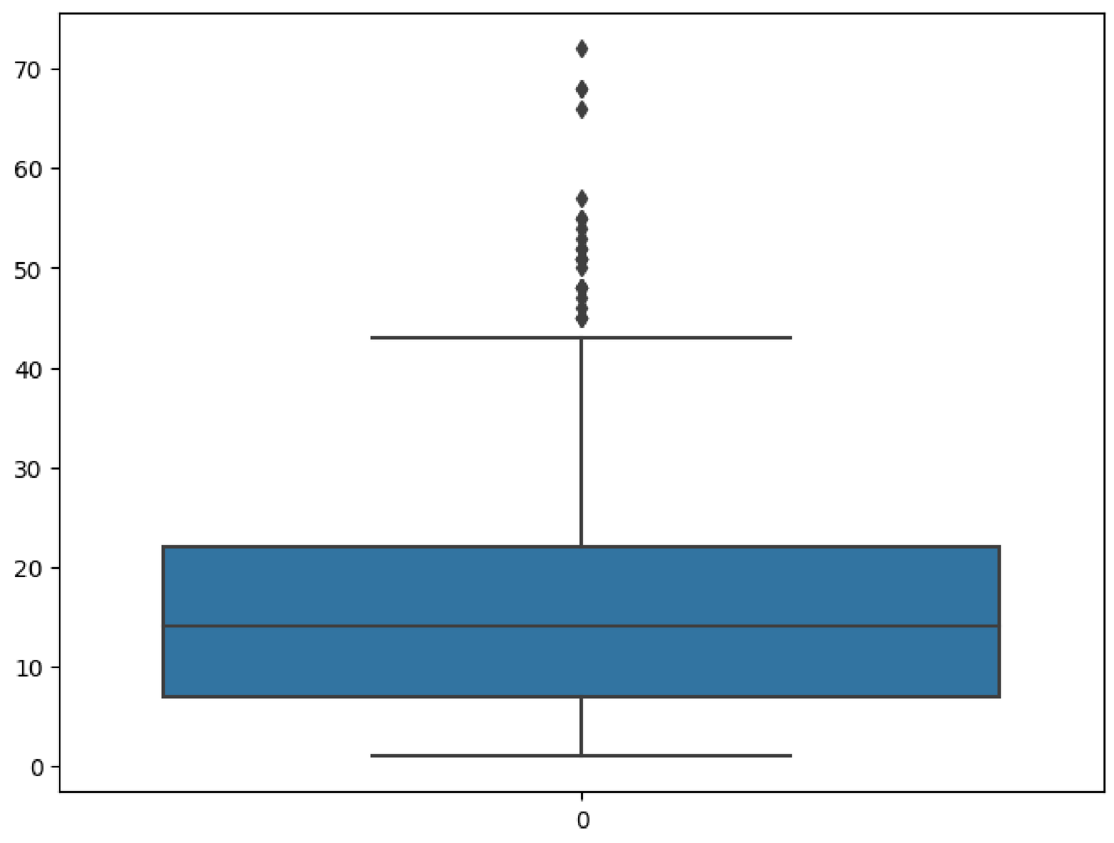


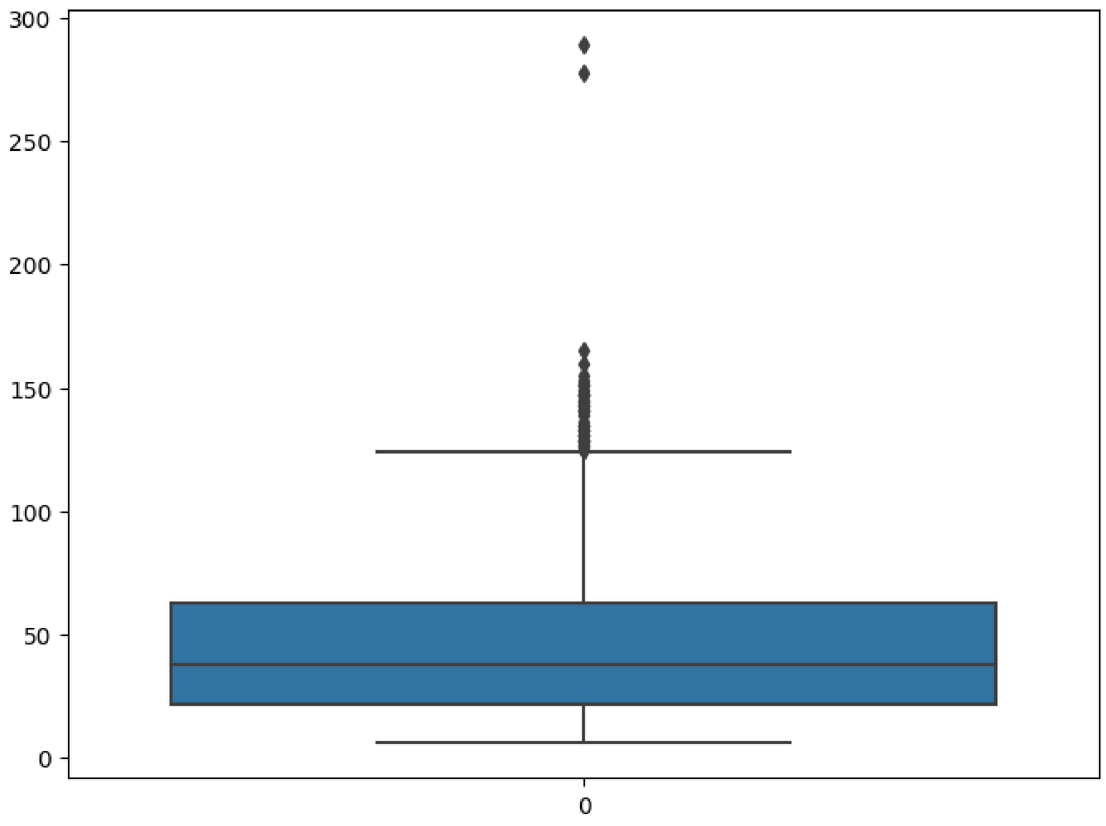

1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0

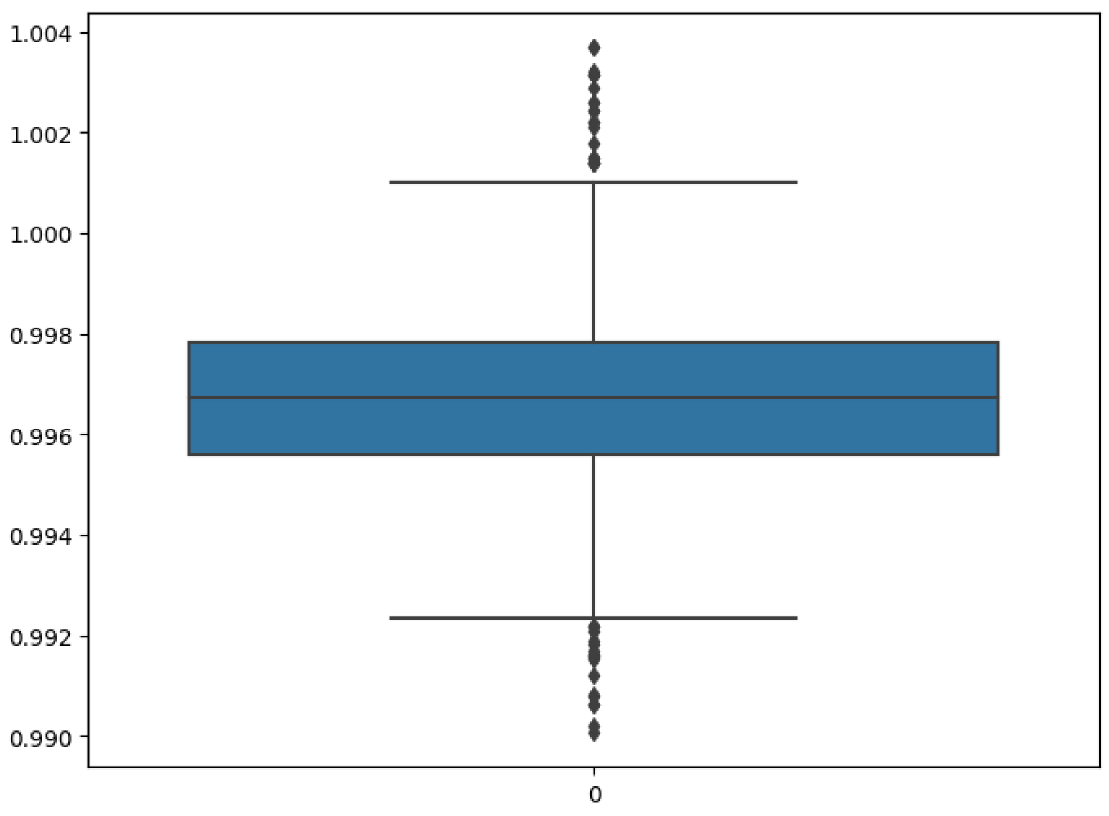

1.0
0.8
0.6
0.4
0.2
0.0
0

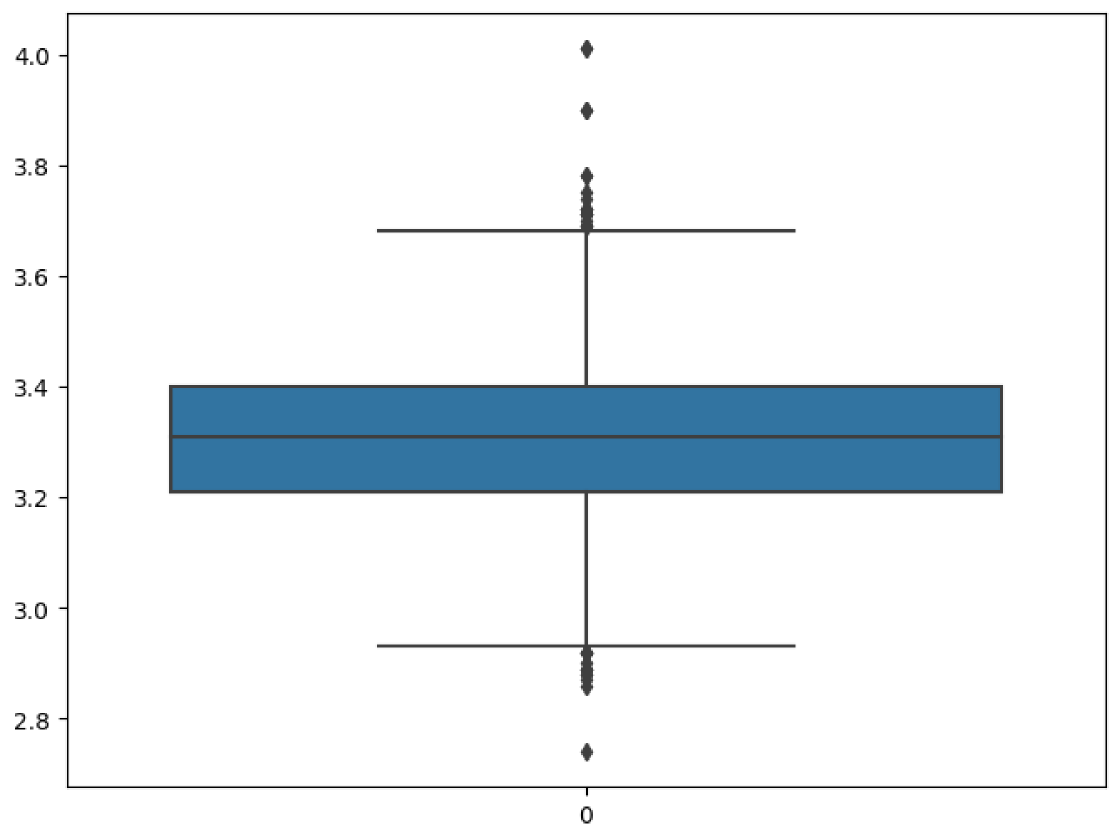

16
14
12
10
8
6
4
2
0

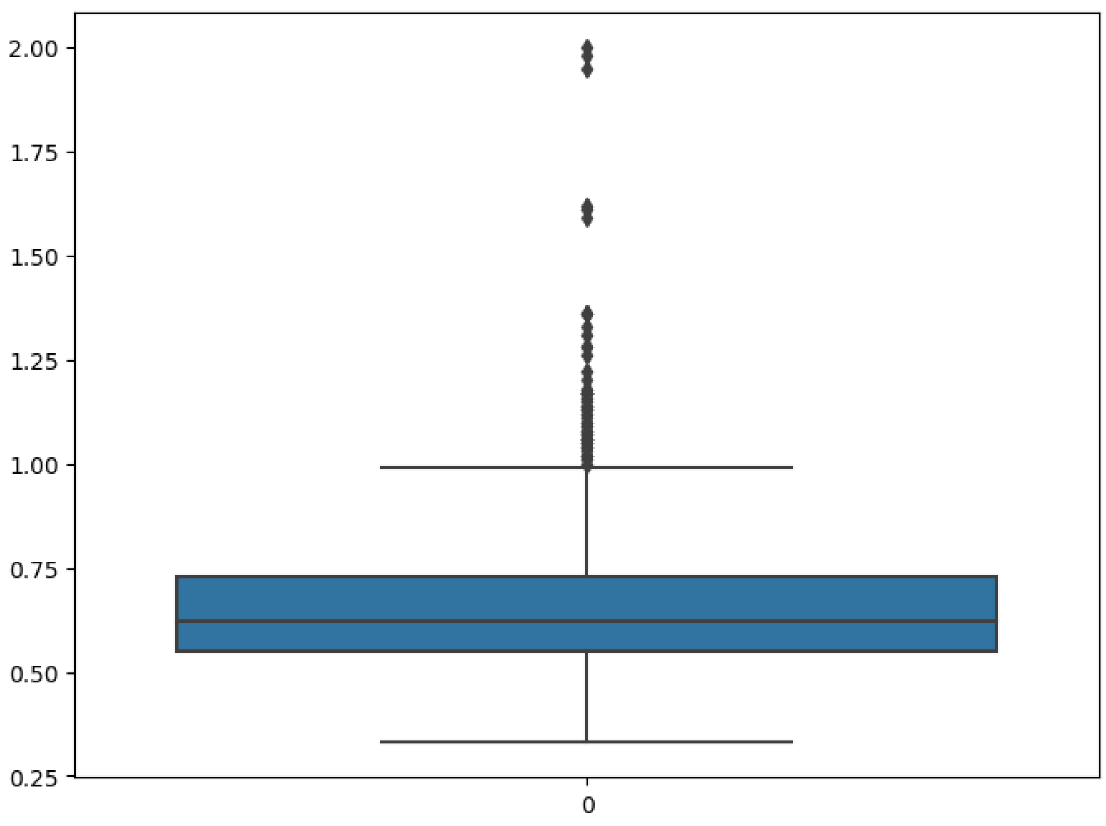

0.6
0.5
0.4
0.3
0.2
0.1
0.0
0

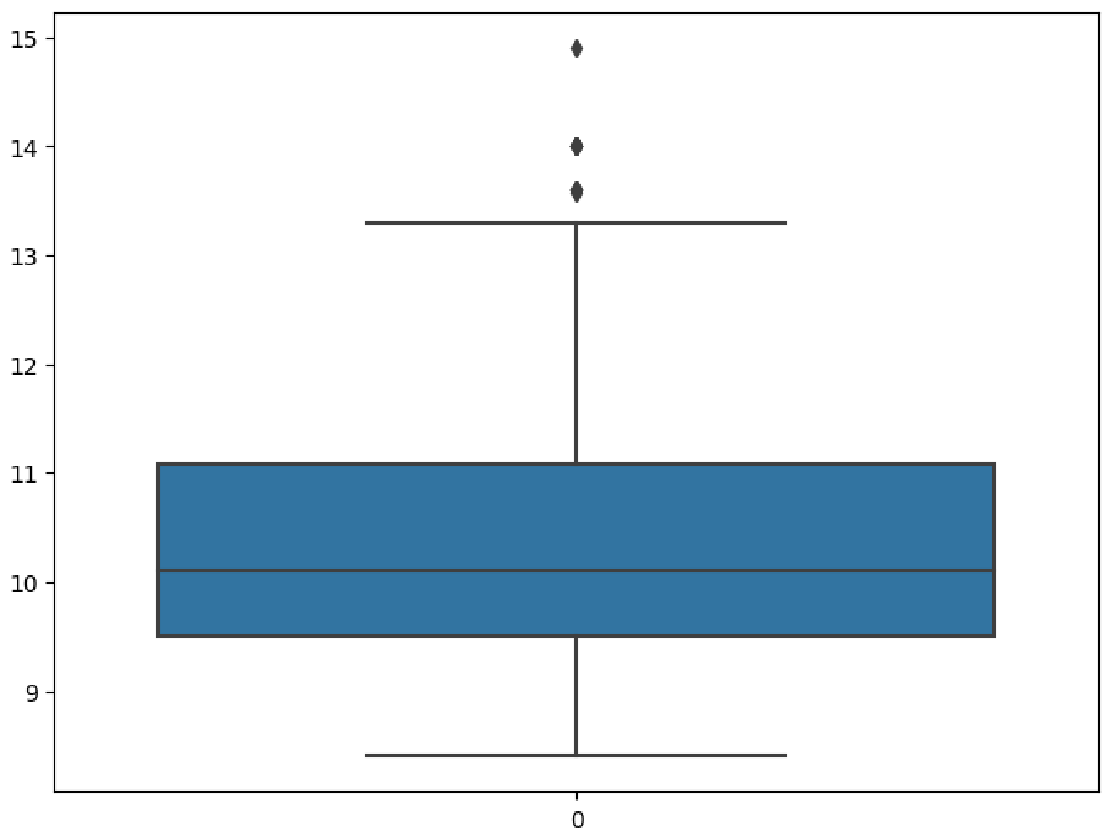

70
60
50
40
30
20
10
0
0


300
250
200
150
100
50
0
0


1.004
1.002
1.000
0.998
0.996
0.994
0.992
0.990
0


4.0
3.8
3.6
3.4
3.2
3.0
2.8
0


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0



1.14.2 Analysis of Outliers from Boxplots:

Upon reviewing the boxplots generated for each predictor variable, it becomes evident that a considerable number of these variables are characterized by the presence of outliers. Specifically, certain predictors, such as chlorides, exhibit a notably high frequency of outlier values. This abundance of outliers across various predictors presents a significant challenge in data preprocessing.

The sheer volume of these outliers poses a dilemma: if I choose to remove all rows containing outlier values, I risk significantly diminishing our dataset. Such a reduction could compromise the dataset's representativeness and the robustness of any subsequent analyses or model training. This is particularly critical given that an extensive removal of data points might lead to the loss of valuable information that could be crucial for understanding the underlying patterns and relationships within the dataset.

Considering this, my approach is to implement a capping strategy. I will cap the values at the 5th and 95th percentiles. This method involves adjusting extreme outlier values to these percentile thresholds, effectively reducing the impact of the most extreme cases while preserving most of the data. By doing so, I aim to maintain the integrity and the size of our dataset as much as possible. This approach strikes a balance between mitigating the influence of outliers and retaining a comprehensive set of data for analysis, ensuring that our models are trained on a dataset that is both clean and representative of the broader range of values.

```
[ ]: # Creating a temporary copy of the 'data' DataFrame
     # This is done to avoid accidentally modifying the original DataFrame
     # The temporary copy allows for comparison and verification after data␣
      ↪processing
     tmp = data

     # Assigning the original 'data' DataFrame to a new variable 'data_clean'
     # This step indicates the intention to perform data cleaning or preprocessing␣
      ↪on 'data_clean'
     # It provides clarity and maintains the original data in 'tmp' for reference or␣
      ↪comparison
     data_clean = data
```

```
[ ]: # Generating and displaying descriptive statistics for the 'data' DataFrame
     data.describe()
```

```
[ ]:        fixed acidity  volatile acidity  citric acid  residual sugar  \
     count    1439.000000       1439.000000  1439.000000     1439.000000
     mean        8.301459          0.528311     0.268506        2.535302
     std         1.738644          0.178718     0.193119        1.443800
     min         4.600000          0.120000     0.000000        0.900000
     25%         7.100000          0.395000     0.090000        1.900000
     50%         7.900000          0.520000     0.250000        2.200000
     75%         9.200000          0.640000     0.420000        2.600000
     max        15.900000          1.580000     1.000000       15.500000

              chlorides  free sulfur dioxide  total sulfur dioxide      density  \
     count  1439.000000          1439.000000           1439.000000  1439.000000
     mean      0.087767            15.974635             46.640723     0.996748
     std       0.048272            10.495904             33.052438     0.001894
     min       0.012000             1.000000              6.000000     0.990070
     25%       0.070000             7.000000             22.000000     0.995600
     50%       0.079000            14.000000             38.000000     0.996720
     75%       0.090000            22.000000             63.000000     0.997825
     max       0.611000            72.000000            289.000000     1.003690

                     pH    sulphates      alcohol      quality
     count  1439.000000  1439.000000  1439.000000  1439.000000
     mean      3.311598     0.657596    10.398992     0.137596
     std       0.154815     0.166836     1.054741     0.344595
     min       2.740000     0.330000     8.400000     0.000000
     25%       3.210000     0.550000     9.500000     0.000000
     50%       3.310000     0.620000    10.100000     0.000000
     75%       3.400000     0.730000    11.083333     0.000000
     max       4.010000     2.000000    14.900000     1.000000
```

```
[ ]: # Extracting a list of all column names from the DataFrame 'data'
     cols = list(data.columns)

     # Looping through each column (except the last one) to create a boxplot and cap␣
      ↪outliers
     for col in cols[0:-1]:
         # Calculating the upper limit, which is approximately the 95th percentile
         # This is done by adding three standard deviations to the mean
         upper_limit = tmp[col].mean() + 3 * tmp[col].std()

         # Calculating the lower limit, which is approximately the 5th percentile
         # This is done by subtracting three standard deviations from the mean
         lower_limit = tmp[col].mean() - 3 * tmp[col].std()

         # Capping the values in 'data_clean' at the upper and lower limits
         # If a value in 'tmp' is greater than the upper limit, it's set to the␣
      ↪upper limit
         # If a value in 'tmp' is less than the lower limit, it's set to the lower␣
      ↪limit
         # Otherwise, the original value from 'tmp' is retained
         data_clean[col] = np.where(tmp[col] > upper_limit, upper_limit,  # If above␣
      ↪95th percentile, cap at upper limit
                                    np.where(tmp[col] < lower_limit, lower_limit,  #␣
      ↪If below 5th percentile, cap at lower limit
                                    tmp[col]))  # Retain original value if within␣
      ↪limits
```

```
[ ]: # Capped distributions. Verify by checking max and min
     data_clean.describe()
```

```
[ ]:        fixed acidity  volatile acidity  citric acid  residual sugar  \
     count    1439.000000       1439.000000  1439.000000     1439.000000
     mean        8.290890          0.527196     0.268400        2.470628
     std         1.700962          0.174393     0.192760        1.074141
     min         4.600000          0.120000     0.000000        0.900000
     25%         7.100000          0.395000     0.090000        1.900000
     50%         7.900000          0.520000     0.250000        2.200000
     75%         9.200000          0.640000     0.420000        2.600000
     max        13.517392          1.064467     0.847864        6.866703

              chlorides  free sulfur dioxide  total sulfur dioxide      density  \
     count  1439.000000          1439.000000           1439.000000  1439.000000
     mean      0.085060            15.871263             46.395393     0.996746
     std       0.031403            10.106061             31.886199     0.001873
     min       0.012000             1.000000              6.000000     0.991065
     25%       0.070000             7.000000             22.000000     0.995600
     50%       0.079000            14.000000             38.000000     0.996720
```

```
75%       0.090000            22.000000            63.000000      0.997825
max       0.232582            47.462346           145.798037      1.002431

                pH    sulphates      alcohol      quality
count  1439.000000  1439.000000  1439.000000  1439.000000
mean      3.311170     0.653922    10.395860     0.137596
std       0.152727     0.148889     1.044248     0.344595
min       2.847152     0.330000     8.400000     0.000000
25%       3.210000     0.550000     9.500000     0.000000
50%       3.310000     0.620000    10.100000     0.000000
75%       3.400000     0.730000    11.083333     0.000000
max       3.776045     1.158102    13.563214     1.000000
```

1.15 Training Classifier Models

Now that the data is clean, I can set up the classifier experiment pipeline. I address some final data cleaning issues in the experiment setup. For example, the values of the chlorides column range from 0.009 - 0.346 while residual sugar ranges from 0.6 - 65.8. This impacts performance on certain classifier algorithms; therefore I choose to z-score normalize the features to put them all on the same scale (-3 to +3). Second, I transform the features into a more Gaussian (normal) distribution. Next, I address the multicollinearity in the data. The threshold for multicollinearity was set to 0.7. Inter-correlated features that exceeded the 0.7 threshold were removed, and when two features are highly correlated with each other (for example, density and residual sugar) the feature that is least correlated with the target variable is removed. Finally, I address the imbalance in the target variable by employing the SMOTE algorithm which synthesizes new examples from the minority class, in this case, high-quality wines.

1.16 Remove multicollinearity

1.16.1 Split the data into features and target

[ ]:
```
X = data_clean.drop('quality', axis=1)
y = data_clean['quality']
```

1.16.2 Use StratifiedShuffleSplit for splitting

[ ]:
```
# Initialize StratifiedShuffleSplit
# This will create a single train/test split (as indicated by n_splits=1)
# 'test_size=0.2'
# 'random_state=42' ensures that the random sampling is reproducible
splitter = StratifiedShuffleSplit(n_splits = 1, test_size = test_size,␣
 ↪random_state = random_state)

# Splitting the dataset into training and testing sets using the indices␣
 ↪provided by splitter
# StratifiedShuffleSplit ensures that both the training and testing sets have
# the same proportion of each target class as the complete set
```

```
for train_index, test_index in splitter.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

# Printing the size of the training and testing sets to verify the split
print("Training set size:", X_train.shape)
print("Testing set size:", X_test.shape)

# Printing the distribution of the target variable in both the training and␣
 ↪testing sets
# 'normalize=True' in value_counts() gives the relative frequencies of unique␣
 ↪values
print("\nDistribution in Training Set:\n", y_train.value_counts(normalize =␣
 ↪True))
print("\nDistribution in Testing Set:\n", y_test.value_counts(normalize = True))
```

```
Training set size: (1151, 11)
Testing set size: (288, 11)

Distribution in Training Set:
 0    0.862728
1    0.137272
Name: quality, dtype: float64

Distribution in Testing Set:
 0    0.861111
1    0.138889
Name: quality, dtype: float64
```

1.16.3 Power Transformation

[ ]:
```
# Initialize the PowerTransformer
# PowerTransformer applies a power transformation to each feature to make the␣
 ↪data more Gaussian-like
# This is particularly useful for modeling issues related to heteroscedasticity␣
 ↪or other situations
# where normality is desired
power_transformer = PowerTransformer()

# Fit the power transformer to the training data and then transform it
# 'fit_transform' first computes the necessary transformation parameters (e.g.,␣
 ↪mean and std. deviation for Gaussian)
# and then applies the transformation to the training data
X_train = power_transformer.fit_transform(X_train)

# Transform the test data using the same transformer
# The test data is transformed using the same parameters computed from the␣
↪training data
# It's important to only transform (and not fit) the test data to avoid data␣
↪leakage
X_test = power_transformer.transform(X_test)
```

1.16.4 Normalization (Z-score Scaling)

[ ]:
```
# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the training data and then transform it
# This standardizes the training data, i.e., scales it so that it has a mean of␣
↪0 and a standard deviation of 1
X_train_scaled = scaler.fit_transform(X_train)

# Transform the test data using the same scaler
# It's important to use the same scaler for both training and test data to␣
↪ensure consistency
# Note that we only transform the test data, we do not fit the scaler to it
X_test_scaled = scaler.transform(X_test)
```

1.16.5 Removing Multicollinearity

[ ]:
```
# This is a simplified approach for handling multicollinearity.
# In practice, deciding which variables to drop in case of high correlation␣
↪might require a more sophisticated method.
# A high correlation between two or more variables means they are providing␣
↪similar information to the model.

# Compute the absolute value of the correlation matrix for the training data
corr_matrix = pd.DataFrame(X_train).corr().abs()

# Select the upper triangle of the correlation matrix
# This is done to consider each pair of correlations only once
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k = 1).astype(np.
↪bool))

# Determine columns to drop based on a correlation threshold (e.g., 0.7)
# If any pair of variables has a correlation higher than 0.7, one of the␣
↪variables in the pair is marked for removal
to_drop = [column for column in upper.columns if any(upper[column] > 0.7)]

# Drop the marked columns from the training data
X_train = pd.DataFrame(X_train).drop(to_drop, axis = 1)
```

```
# Drop the same columns from the test data to maintain consistency
X_test = pd.DataFrame(X_test).drop(to_drop, axis = 1)
```

```
<ipython-input-29-b606d60b0838>:10: DeprecationWarning: `np.bool` is a
deprecated alias for the builtin `bool`. To silence this warning, use `bool` by
itself. Doing this will not modify any behavior and is safe. If you specifically
wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k =
1).astype(np.bool))
```

1.16.6 Apply SMOTE to the training set

[ ]:
```
# Initialize the SMOTE object
# SMOTE (Synthetic Minority Over-sampling Technique) is used to handle␣
 ↪imbalanced datasets
# 'random_state=42' ensures that the random sampling is reproducible
smote = SMOTE(random_state = random_state)

# Apply SMOTE to the scaled training data
# 'fit_resample' is used to fit the SMOTE method on the scaled training data␣
 ↪and to create synthetic samples
# This results in an augmented training set (X_train_smote, y_train_smote)␣
 ↪where the minority class is upsampled
X_train, y_train = smote.fit_resample(X_train, y_train)
```

1.17 Reusable Functions

[ ]:
```
def display_metrics(y_test, y_pred, y_pred_proba):
  # Calculate Accuracy
  accuracy = accuracy_score(y_test, y_pred)

  # Calculate Recall
  recall = recall_score(y_test, y_pred, average = 'binary') # Use 'binary' for␣
 ↪binary classification, 'macro' or 'weighted' for multiclass

  # Calculate Precision
  precision = precision_score(y_test, y_pred, average = 'binary') # Use␣
 ↪'binary' for binary classification, 'macro' or 'weighted' for multiclass

  # Calculate F1 Score
  f1 = f1_score(y_test, y_pred, average = 'binary') # Use 'binary' for binary␣
 ↪classification, 'macro' or 'weighted' for multiclass

  # Calculate AUC
```

```
    auc = roc_auc_score(y_test, y_pred_proba)

    # Calculate Cohen's Kappa
    kappa = cohen_kappa_score(y_test, y_pred)

    # Calculate Matthews Correlation Coefficient
    mcc = matthews_corrcoef(y_test, y_pred)

    # Print the calculated metrics
    print("Accuracy: ", accuracy)
    print("Precision: ", precision)
    print("Recall: ", recall)
    print("F1 Score: ", f1)
    print("AUC: ", auc)
    print("Cohen's Kappa Score: ", kappa)
    print("mcc: ", mcc)
```

```
[ ]: def display_confusion_matrix(y_test, y_pred):
        # Create the confusion matrix
        # The confusion matrix compares the actual target values with those predicted␣
      ↪by the model
        # 'y_test' are the true values, and 'y_pred' are the predictions made by the␣
      ↪model
        cm = confusion_matrix(y_test, y_pred)

        # Create a heatmap for visualizing the confusion matrix using seaborn
        # 'plt.figure' sets up the figure that seaborn will plot on
        # 'figsize' determines the size of the figure
        plt.figure(figsize = (10, 7))

        # 'sns.heatmap' creates a heatmap to represent the confusion matrix
        # 'cm' is the confusion matrix to be visualized
        # 'annot=True' enables annotations inside the squares of the heatmap
        # 'fmt='d'' formats the annotations to integer (digit) format
        # 'cmap='Greens'' sets the color map to 'Greens' for the heatmap
        sns.heatmap(cm, annot = True, fmt = 'd', cmap = 'Greens')

        # Labeling the axes of the heatmap
        # 'plt.xlabel' and 'plt.ylabel' label the x-axis and y-axis respectively
        plt.xlabel('Predicted')
        plt.ylabel('True')

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

```
[ ]: from sklearn.metrics import auc

     def display_roc_curve(fpr, tpr):
       # 'auc' computes the area under the ROC curve, a measure of the model's␣
      ↪ability to distinguish between classes
       roc_auc = auc(fpr, tpr)

       # Plotting the ROC curve
       # 'plt.figure' creates a new figure for plotting
       # 'figsize' sets the figure size
       plt.figure(figsize = (8, 6))

       # 'plt.plot' plots the ROC curve with the false positive rate on the x-axis␣
      ↪and the true positive rate on the y-axis
       # 'color' sets the color of the line, 'lw' sets the line width, and 'label'␣
      ↪provides a label for the plot legend
       plt.plot(fpr, tpr, color = 'darkorange', lw = 2, label = 'ROC curve (area =␣
      ↪%0.2f)' % roc_auc)

       # Plotting a diagonal dashed line for reference
       # This line represents a purely random classifier; a good model stays as far␣
      ↪away from this line as possible (towards the top-left corner)
       plt.plot([0, 1], [0, 1], color = 'navy', lw = 2, linestyle = '--')

       # Setting the limits for the x-axis and y-axis
       plt.xlim([0.0, 1.0])
       plt.ylim([0.0, 1.05])

       # Labeling the axes, giving the plot a title and adding a legend
       plt.xlabel('False Positive Rate')
       plt.ylabel('True Positive Rate')
       plt.title('Receiver Operating Characteristic')
       plt.legend(loc='lower right')

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

```
[ ]: def display_precision_recall_curve(y_test, y_pred, recall, precision):
       # Calculate the average precision score
       # This summarizes the precision-recall curve as the weighted mean of␣
      ↪precisions at each threshold
       # The increase in recall from the previous threshold is used as the weight
       average_precision = average_precision_score(y_test, y_pred)

       # Plotting the precision-recall curve
       # 'plt.figure' creates a new figure for plotting
       # 'figsize' sets the figure size
```

```
plt.figure(figsize = (8, 6))

# 'plt.step' creates a step plot, which is appropriate for precision-recall␣
↪curves
# 'recall' is on the x-axis, 'precision' is on the y-axis
# 'where=post' indicates that the step change happens at the x-value
# 'label' provides a label for the plot legend
plt.step(recall, precision, where = 'post', label = 'Average precision = %0.
↪2f' % average_precision)

# Labeling the axes and setting their limits
# Precision and recall values range from 0 to 1
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])

# Giving the plot a title and adding a legend
plt.title('Precision-Recall curve')
plt.legend(loc = 'upper right')

# Displaying the plot
# 'plt.show()' renders the plot on the screen
plt.show()
```

1.18 LightGBM Model

```
[ ]: # Create and train LightGBM model
     lgm_model = lgb.LGBMClassifier()
     lgm_model.fit(X_train, y_train)
```

```
[LightGBM] [Info] Number of positive: 993, number of negative: 993
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.003182 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 2290
[LightGBM] [Info] Number of data points in the train set: 1986, number of used
features: 9
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
```

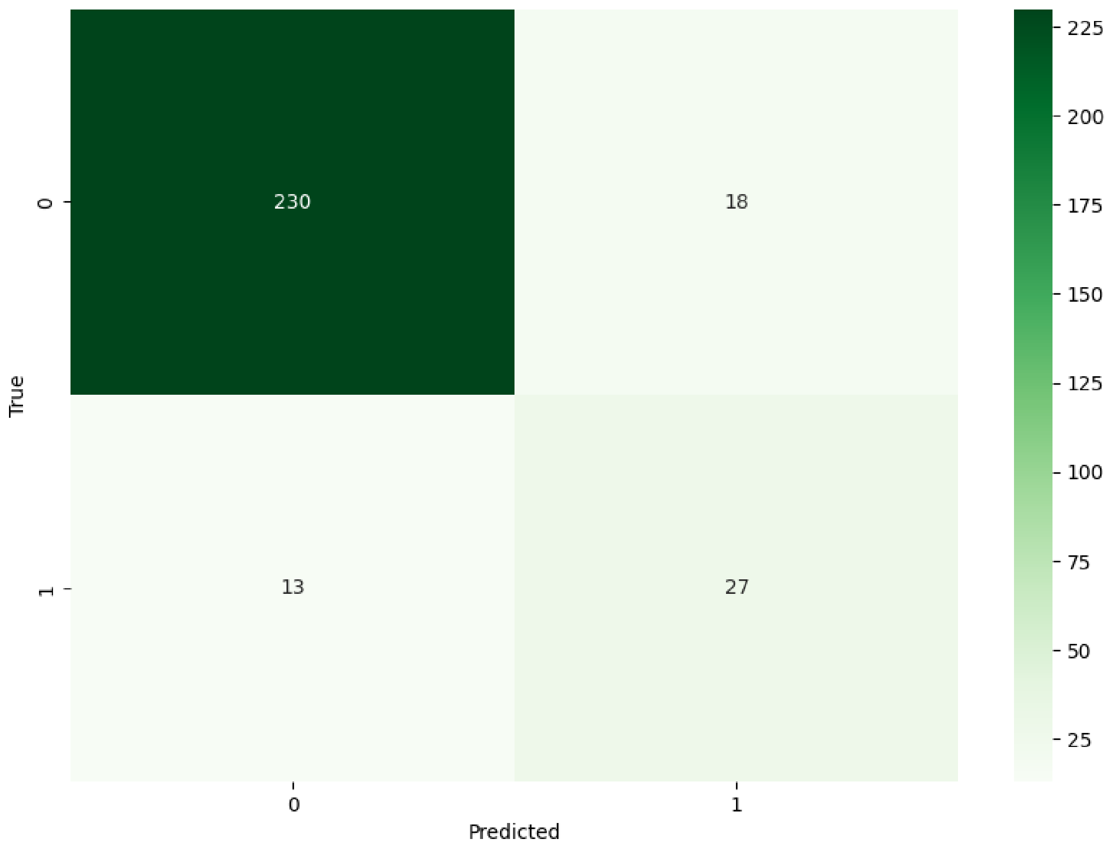
```
[ ]: LGBMClassifier()
```

1.18.1 Make predictions and evaluate the model

[ ]:
```
# Predict the labels for the test set
y_pred = lgm_model.predict(X_test)
y_pred_proba = lgm_model.predict_proba(X_test)[:, 1] # Probabilities for the␣
 ↪positive class
```

1.18.2 Confusion Matrix

[ ]:
```
display_confusion_matrix(y_test, y_pred)
```



1.18.3 Metrics

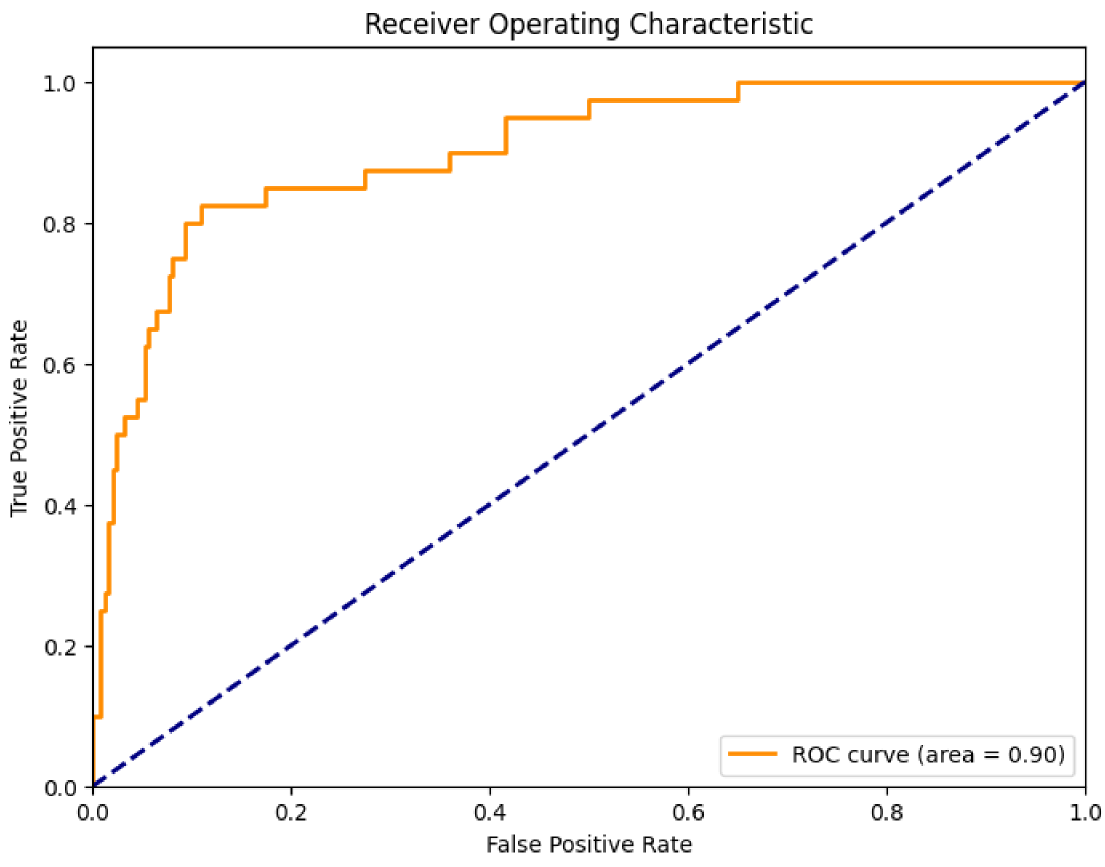
[ ]:
```
display_metrics(y_test, y_pred, y_pred_proba)
```

```
Accuracy:  0.8923611111111112
Precision:  0.6
Recall:  0.675
F1 Score:  0.6352941176470589
AUC:  0.9024193548387096
```

```
Cohen's Kappa Score:  0.5724137931034483
mcc:  0.5737799377508352
```

1.18.4 ROC Curve and AUC Score

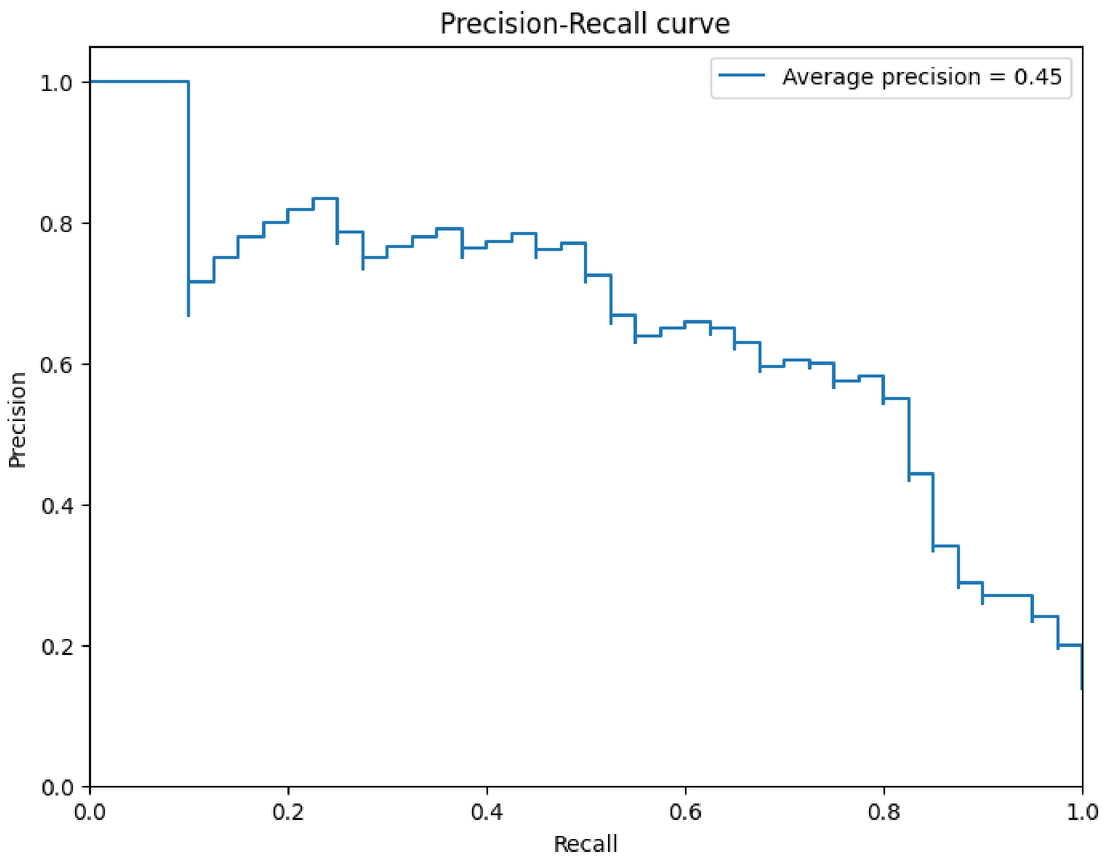
[ ]:
```
# Calculate the ROC curve and AUC score
# 'roc_curve' computes the receiver operating characteristic curve, which is a␣
↪plot of the true positive rate against the false positive rate
# 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
↪are the probabilities of the positive class
fpr, tpr, thresholds = roc_curve(y_test, lgm_model.predict_proba(X_test)[:,1])

display_roc_curve(fpr, tpr)
```



1.18.5 Precision-Recall Curve

[ ]:
```
# Calculate the precision-recall curve
# This curve shows the tradeoff between precision and recall for different␣
↪threshold values
```

```
# 'precision_recall_curve' computes precision and recall pairs for different␣
↪probability thresholds
# 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
↪are the probabilities of the positive class
precision, recall, _ = precision_recall_curve(y_test, lgm_model.
↪predict_proba(X_test)[:,1])

display_precision_recall_curve(y_test, y_pred, recall, precision)
```



1.19 Random Forest Model

[ ]:
```
# 'random_state=42' ensures that the random sampling is reproducible
rf_model = RandomForestClassifier(random_state = random_state)
rf_model.fit(X_train, y_train)
```

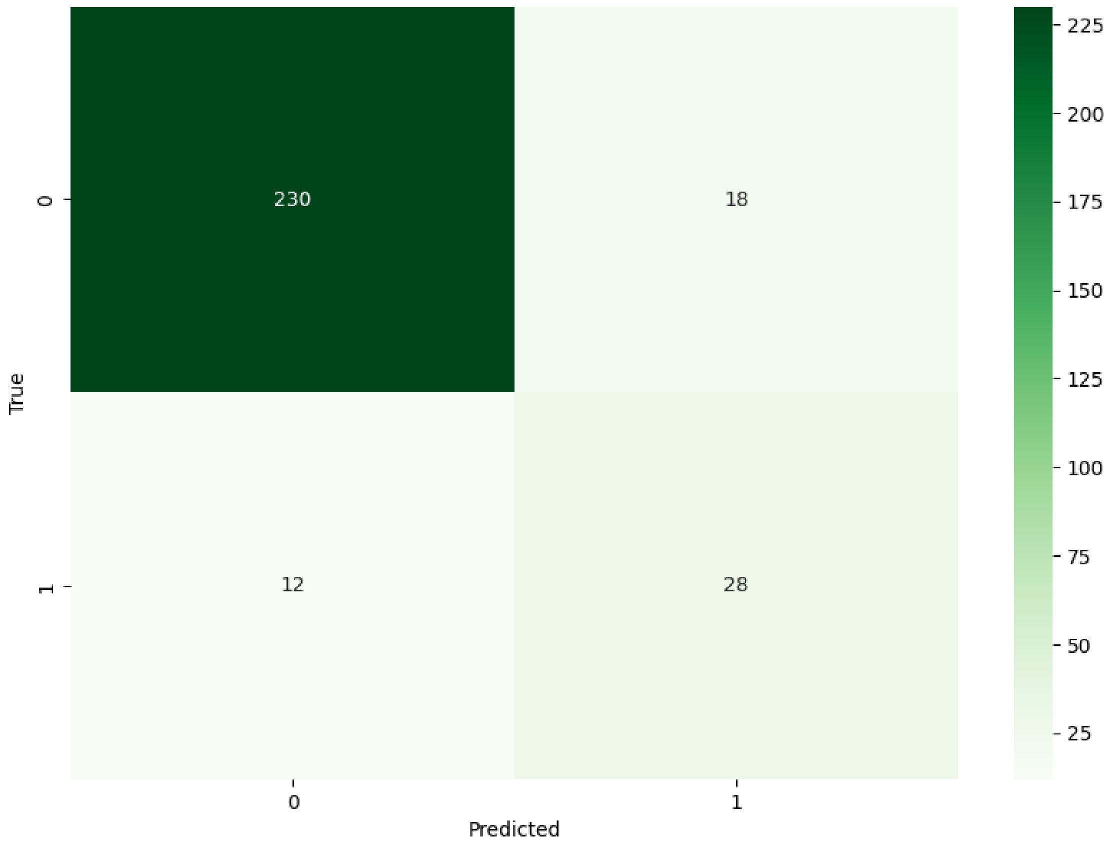
[ ]:
```
RandomForestClassifier(random_state=42)
```

[ ]:
```
# Predict the labels for the test set
y_pred = rf_model.predict(X_test)
```

```
y_pred_proba = rf_model.predict_proba(X_test)[:, 1] # Probabilities for the␣
  ↪positive class
```

1.19.1 Confusion Matrix

```
[ ]: display_confusion_matrix(y_test, y_pred)
```



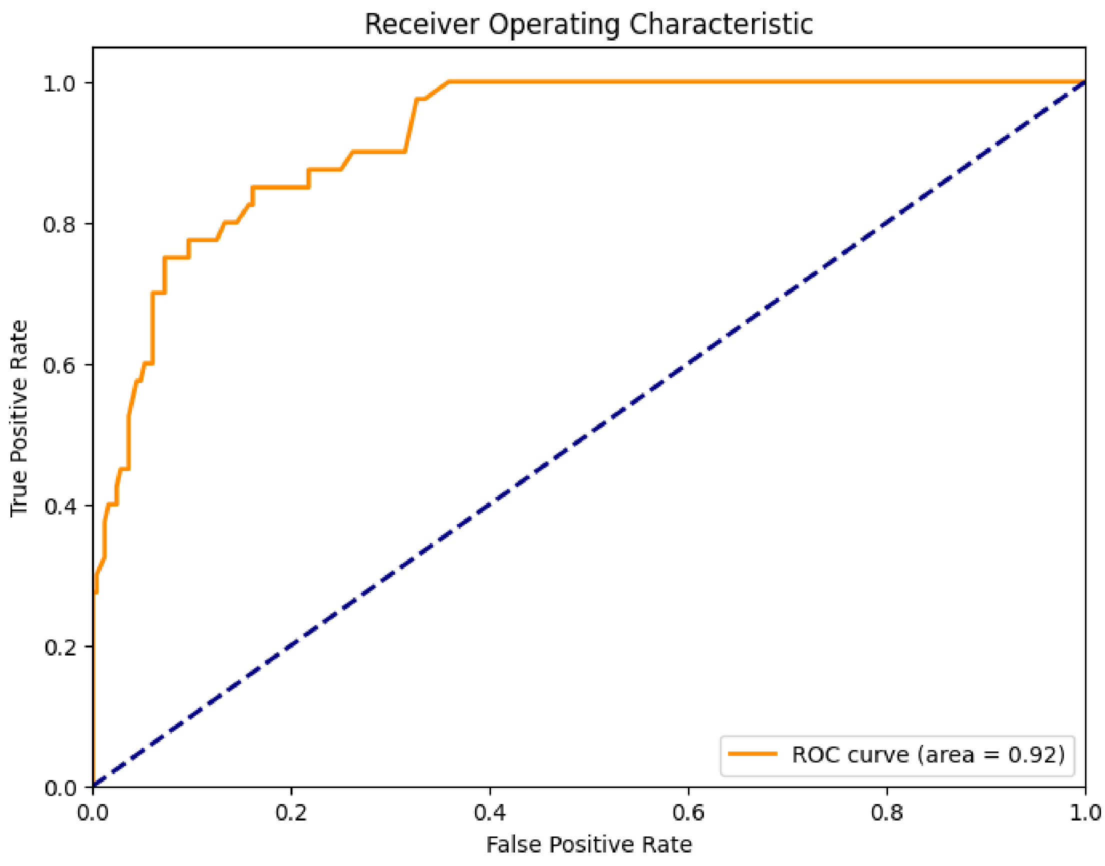
1.19.2 Metrics

```
[ ]: display_metrics(y_test, y_pred, y_pred_proba)
```

```
Accuracy:  0.8958333333333334
Precision:  0.6086956521739131
Recall:  0.7
F1 Score:  0.6511627906976744
AUC:  0.9237903225806451
Cohen's Kappa Score:  0.590288315629742
mcc:  0.5922801109562629
```

1.19.3 ROC Curve and AUC Score

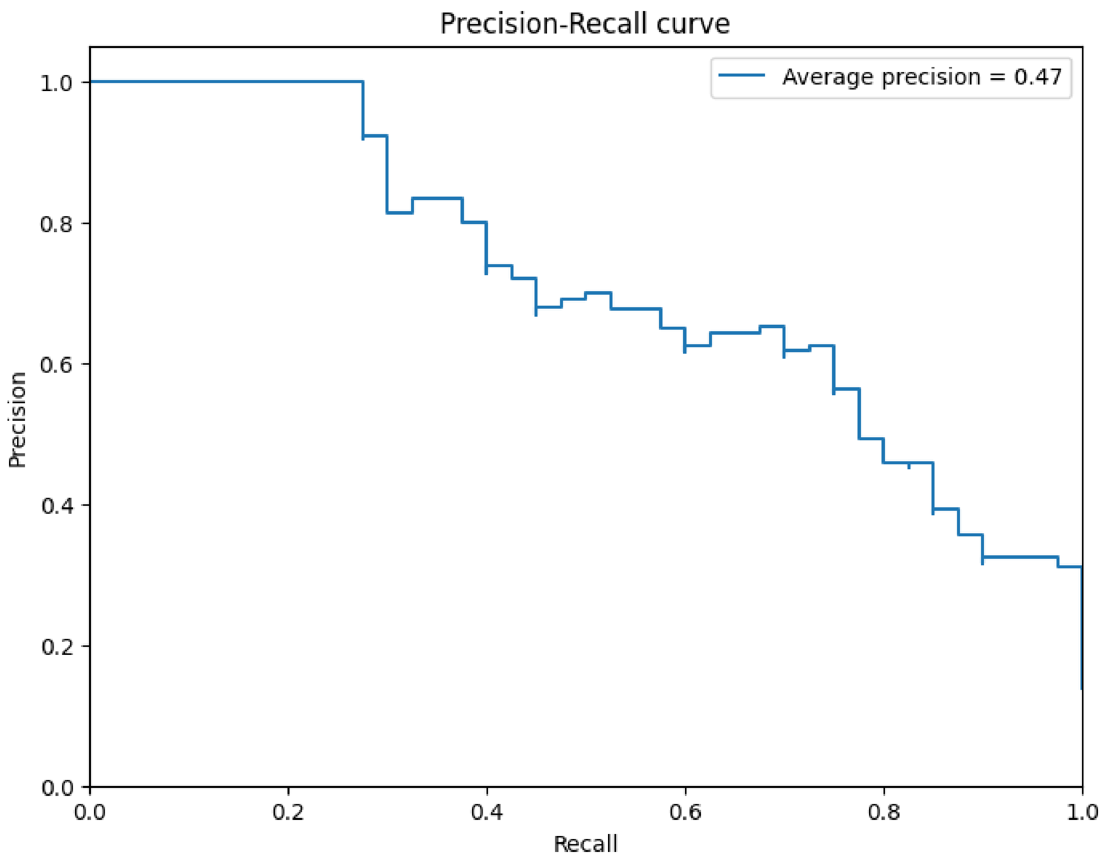
[ ]:
```
# Calculate the ROC curve and AUC score
# 'roc_curve' computes the receiver operating characteristic curve, which is a␣
↪plot of the true positive rate against the false positive rate
# 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
↪are the probabilities of the positive class
fpr, tpr, thresholds = roc_curve(y_test, rf_model.predict_proba(X_test)[:,1])

display_roc_curve(fpr, tpr)
```



1.19.4 Precision-Recall Curve

[ ]:
```
# Calculate the precision-recall curve
# This curve shows the tradeoff between precision and recall for different␣
↪threshold values
# 'precision_recall_curve' computes precision and recall pairs for different␣
↪probability thresholds
# 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
↪are the probabilities of the positive class
```

```
precision, recall, _ = precision_recall_curve(y_test, rf_model.
  ↪predict_proba(X_test)[:,1])

display_precision_recall_curve(y_test, y_pred, recall, precision)
```



1.20 Extra Trees Model

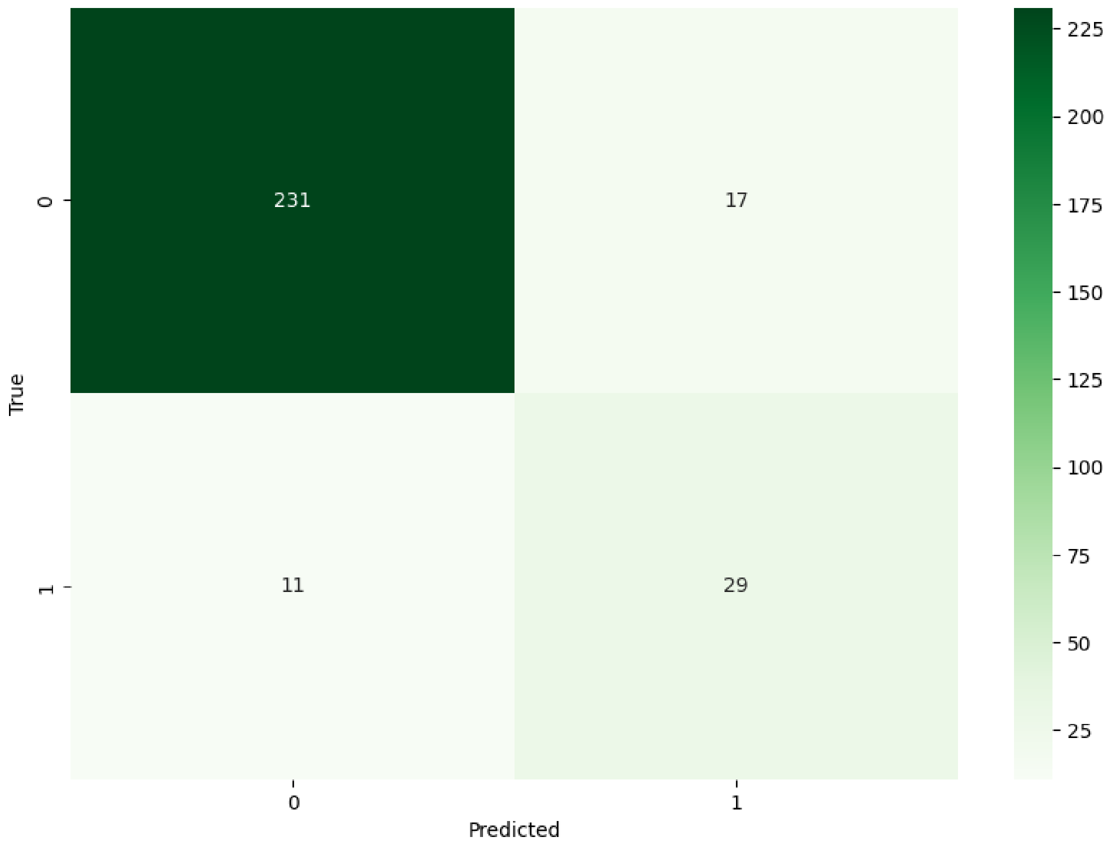
```
[ ]: et_model = ExtraTreesClassifier(n_estimators = 100, random_state = random_state)
     et_model.fit(X_train, y_train)
```

```
[ ]: ExtraTreesClassifier(random_state=42)
```

```
[ ]: # Predict the labels for the test set
     y_pred = et_model.predict(X_test)
     y_pred_proba = et_model.predict_proba(X_test)[:, 1] # Probabilities for the␣
      ↪positive class
```

1.20.1 Confusion Matrix

```
[ ]: display_confusion_matrix(y_test, y_pred)
```



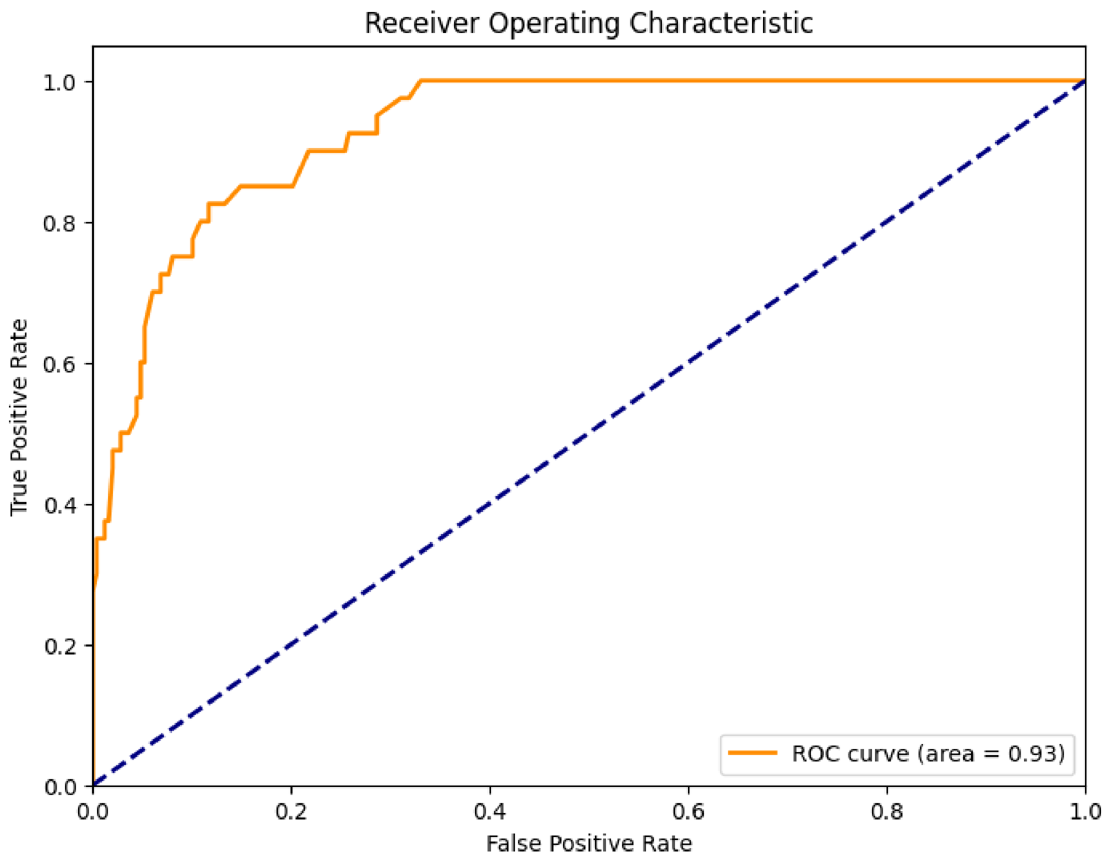
1.20.2 Metrics

```
[ ]: display_metrics(y_test, y_pred, y_pred_proba)
```

```
Accuracy:  0.9027777777777778
Precision:  0.6304347826086957
Recall:  0.725
F1 Score:  0.6744186046511628
AUC:  0.9320060483870967
Cohen's Kappa Score:  0.6176024279210925
mcc:  0.6196863885840591
```

1.20.3 ROC Curve and AUC Score

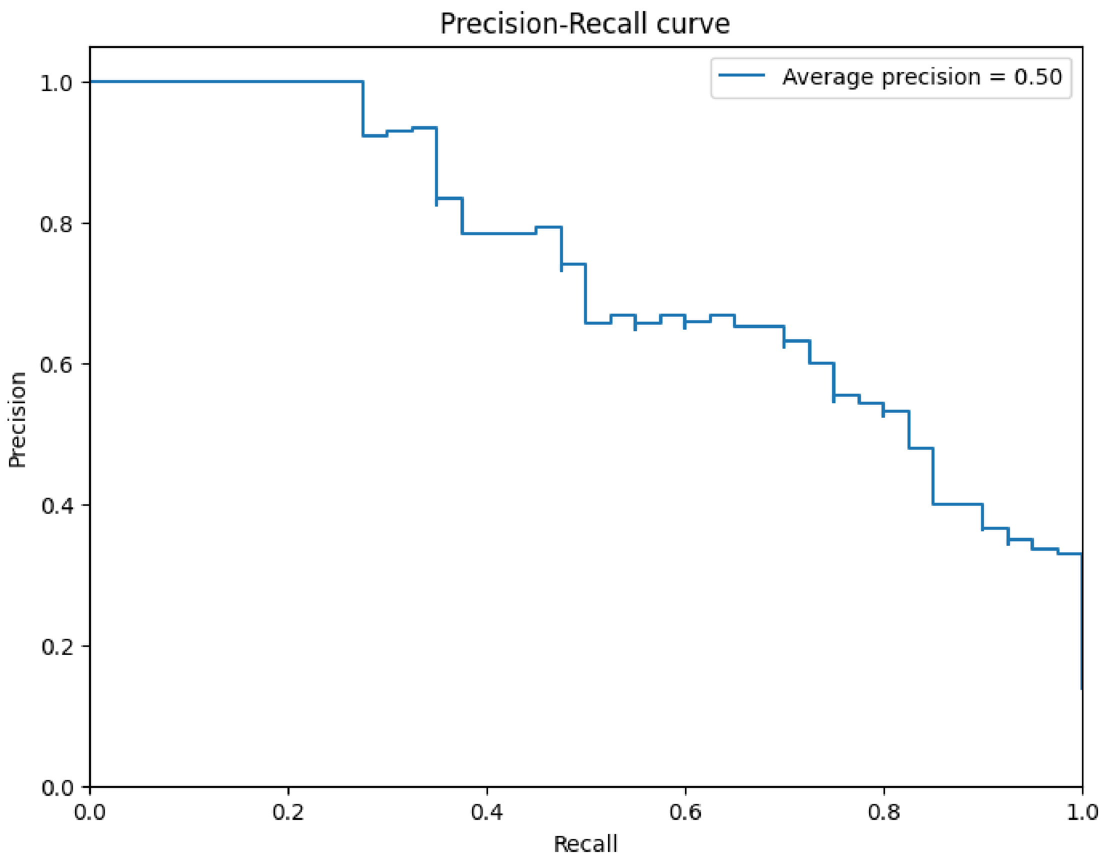
```
[ ]: # Calculate the ROC curve and AUC score
     # 'roc_curve' computes the receiver operating characteristic curve, which is a␣
      ↪plot of the true positive rate against the false positive rate
     # 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
      ↪are the probabilities of the positive class
     fpr, tpr, thresholds = roc_curve(y_test, et_model.predict_proba(X_test)[:,1])

     display_roc_curve(fpr, tpr)
```



1.20.4 Precision-Recall Curve

```
[ ]: # Calculate the precision-recall curve
     # This curve shows the tradeoff between precision and recall for different␣
      ↪threshold values
     # 'precision_recall_curve' computes precision and recall pairs for different␣
      ↪probability thresholds
     # 'y_test' are the true binary labels and 'model.predict_proba(X_test)[:,1]'␣
      ↪are the probabilities of the positive class
```

```
precision, recall, _ = precision_recall_curve(y_test, et_model.
  ↪predict_proba(X_test)[:,1])

display_precision_recall_curve(y_test, y_pred, recall, precision)
```



1.21 Check Performance on Validation Set

As previously mentioned, it's a common occurrence for machine learning models to exhibit better performance on their training data compared to unseen or validation data. This is primarily due to the models being trained and tuned specifically on the training dataset, potentially leading to overfitting. Overfitting occurs when a model learns not only the underlying patterns but also the noise specific to the training dataset, which doesn't generalize well to new data.

Therefore, before finalizing my decision on which classifier to use, it's crucial to assess how the models fare on a validation set. The validation set, which the models haven't encountered during training, serves as a proxy for new, unseen data. By evaluating the models on the validation set, we can gauge their ability to generalize and perform accurately on data they haven't seen before. This step is vital to ensure that we choose a model that not only performs well on the training data but also maintains a robust performance on data outside of the training dataset. This approach helps in minimizing the risk of selecting a model that's overfitted and, consequently, might perform poorly in real-world scenarios or when deployed in a production environment.

1.21.1 Split the data into features and target

```
[ ]: X_val = data_unseen.drop('quality', axis=1)
     y_val = data_unseen['quality']
```

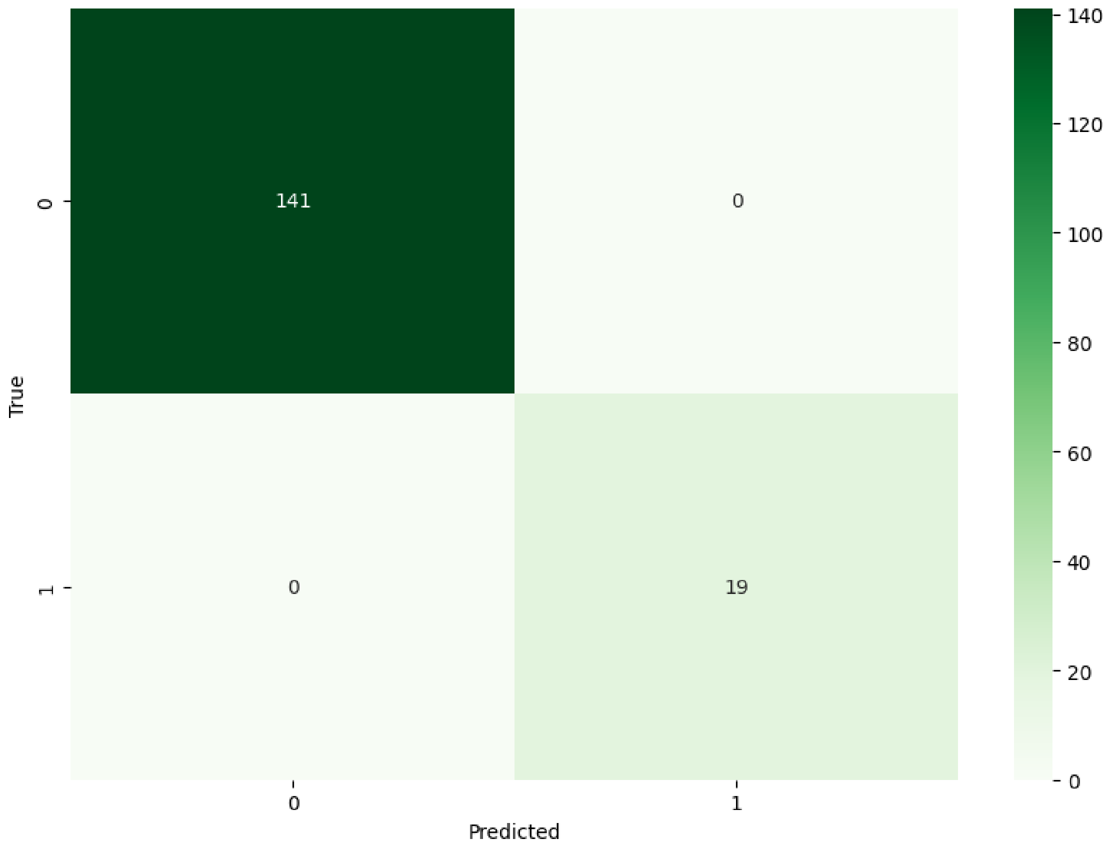
```
[ ]: et_model.fit(X_val, y_val)
```

```
[ ]: ExtraTreesClassifier(random_state=42)
```

```
[ ]: # Predict the labels for the test set
     y_pred = et_model.predict(X_val)
     y_pred_proba = et_model.predict_proba(X_val)[:, 1] # Probabilities for the␣
      ↪positive class
```

1.21.2 Confusion Matrix

```
[ ]: display_confusion_matrix(y_val, y_pred)
```



1.21.3 Metrics

```
[ ]: display_metrics(y_val, y_pred, y_pred_proba)
```

```
Accuracy:  1.0
Precision:  1.0
Recall:  1.0
F1 Score:  1.0
AUC:  1.0
Cohen's Kappa Score:  1.0
mcc:  1.0
```

1.21.4 Analysis of Metrics:

Accuracy (1.0): Perfect accuracy, meaning the model correctly classified all instances.

Precision (1.0): Implies no false positives; every instance predicted as positive was indeed positive.

Recall (1.0): Indicates no false negatives; the model identified all actual positives.

F1 Score (1.0): Demonstrates a perfect balance between precision and recall.

AUC (Area Under Curve) (1.0): Suggests perfect discrimination between positive and negative classes.

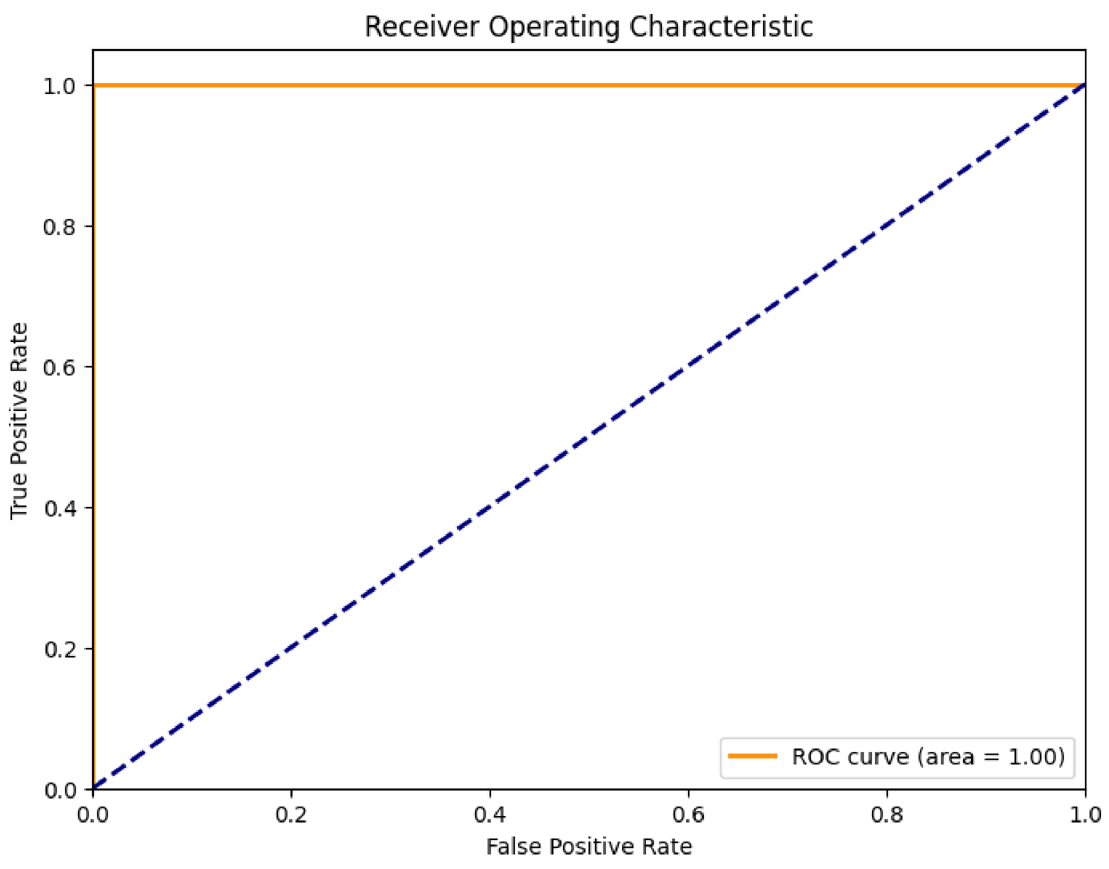
Cohen's Kappa Score (1.0): Indicates perfect agreement; the model's predictions are in complete harmony with the actual values, beyond chance.

MCC (Matthews Correlation Coefficient) (1.0): Shows a perfect correlation between observed and predicted classifications.

In summary, these scores reflect an ideal model with maximum effectiveness in classification, without any errors or misclassifications. This level of performance is rare and might warrant scrutiny for overfitting, data leakage, or an overly simplistic problem scenario.

1.21.5 ROC Curve and AUC Score

```
[ ]: # Calculate the ROC curve and AUC score
     # 'roc_curve' computes the receiver operating characteristic curve, which is a␣
      ↪plot of the true positive rate against the false positive rate
     # 'y_val' are the true binary labels and 'model.predict_proba(X_val)[:,1]' are␣
      ↪the probabilities of the positive class
     fpr, tpr, thresholds = roc_curve(y_val, et_model.predict_proba(X_val)[:,1])

     display_roc_curve(fpr, tpr)
```

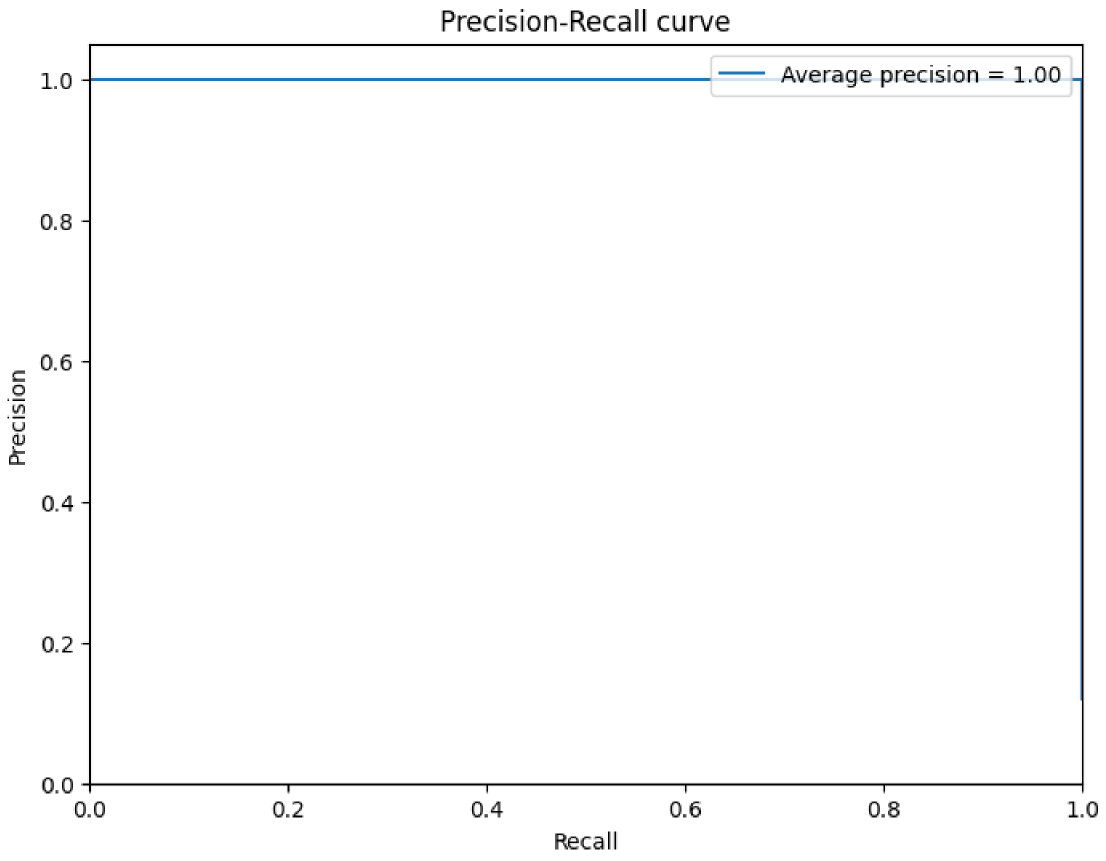


1.21.6 Analysis of ROC curve:

A ROC (Receiver Operating Characteristic) curve value of 1, the model demonstrates perfect classification ability.

1.21.7 Precision-Recall Curve

```
[ ]: # Calculate the precision-recall curve
     # This curve shows the tradeoff between precision and recall for different␣
      ↪threshold values
     # 'precision_recall_curve' computes precision and recall pairs for different␣
      ↪probability thresholds
     # 'y_val' are the true binary labels and 'model.predict_proba(X_val)[:,1]' are␣
      ↪the probabilities of the positive class
     precision, recall, _ = precision_recall_curve(y_val, et_model.
      ↪predict_proba(X_val)[:,1])

     display_precision_recall_curve(y_val, y_pred, recall, precision)
```



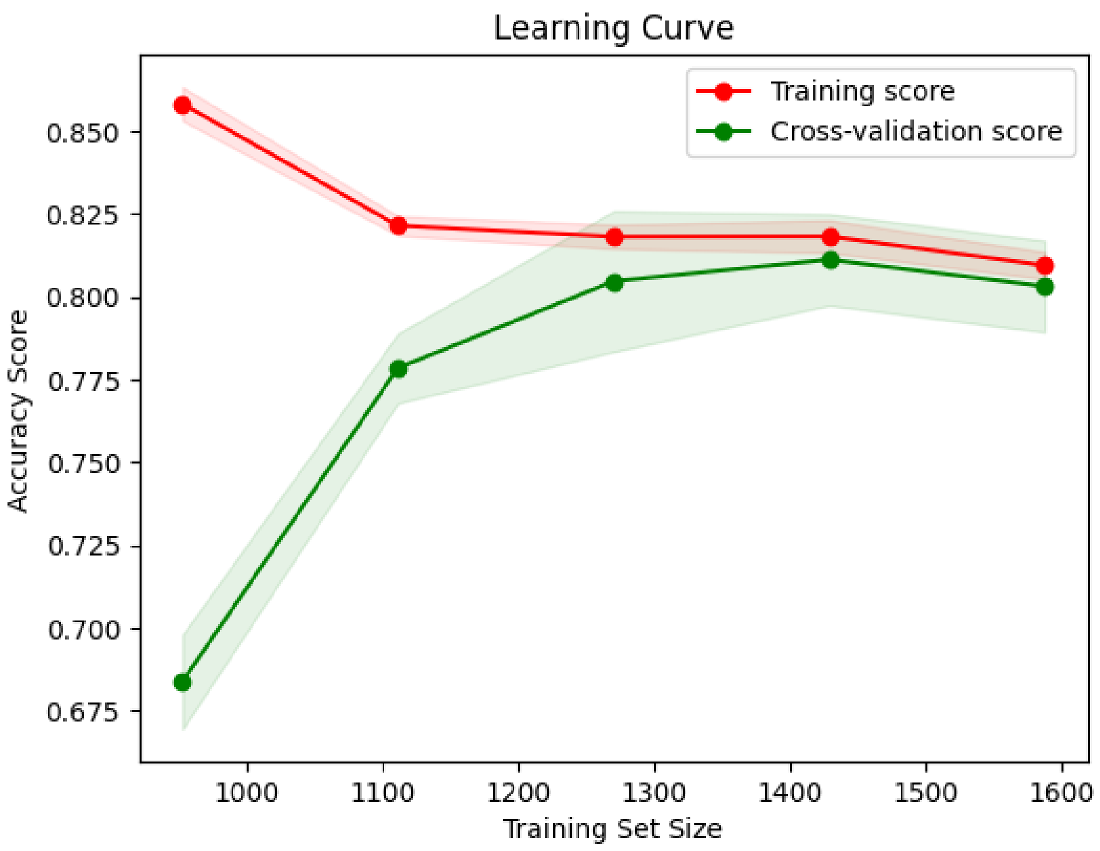
1.21.8 Analysis of Precision-Recall Curve:

With an average precision of 1 in the precision-recall curve, the model demonstrates the highest possible performance in terms of precision at all levels of recall.

1.21.9 Learning Curve

```
[ ]: # Initialize the classifier
lr_model = LogisticRegression(max_iter = 1000)

# Determine training sizes
train_sizes = np.linspace(0.1, 1.0, 10)

# Generate the learning curve
train_sizes, train_scores, test_scores = learning_curve(lr_model, X_train,␣
 ↪y_train, train_sizes = train_sizes, cv = 5)

# Calculate the average and standard deviation of the training and test set␣
 ↪scores
train_mean = np.mean(train_scores, axis = 1)
```

```
train_std = np.std(train_scores, axis = 1)
test_mean = np.mean(test_scores, axis = 1)
test_std = np.std(test_scores, axis = 1)

# Plot the learning curve
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std,␣
 ↪alpha = 0.1, color = "r")
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, alpha␣
 ↪= 0.1, color = "g")
plt.plot(train_sizes, train_mean, 'o-', color = "r", label = "Training score")
plt.plot(train_sizes, test_mean, 'o-', color = "g", label = "Cross-validation␣
 ↪score")

plt.title("Learning Curve")
plt.xlabel("Training Set Size")
plt.ylabel("Accuracy Score")
plt.legend(loc = "best")
plt.show()
```



1.21.10 Analysis of Learning Curve:

Convergence to Similar Values: Both training and validation curves converge at similar levels, signifying that the model effectively learns and generalizes across both seen and unseen data. This convergence is a clear indicator of neither underfitting nor overfitting.

Good Overall Performance: The model shows strong performance metrics on both training and validation sets. This suggests that it accurately captures and predicts the underlying patterns in the data.

Small Gap Between Curves: The minimal gap between training and validation performance is a positive sign, indicating balanced model complexity and an absence of overfitting. The model learns the training data well while retaining the ability to generalize to new, unseen data.

In summary, the model demonstrates a desirable balance in machine learning, characterized by its ability to generalize well without overfitting, as evidenced by the convergence of the learning curves and consistently good performance across both training and validation sets.

1.22 Save Model

```
[ ]: # Save the model to file
     filename = 'finalized_model.pkl'

     with open(filename, 'wb') as file:
       pickle.dump(et_model, file)
```

1.23 Load the Model

```
[ ]: with open(filename, 'rb') as file:
       loaded_model = pickle.load(file)

     loaded_model
```

```
[ ]: ExtraTreesClassifier(random_state=42)
```

1.24 Conclusions

In my study on red wine quality, I decided to use the Extra Trees model because it was the most accurate among all the models I tried. This model was really good at using the chemical details of the wines to predict their quality. It did a better job than the other models I tested.

Choosing the Extra Trees model was important for my project. It worked better than traditional ways of judging wine, like tasting, because it was more objective and used data to make decisions. This approach shows that using advanced models like Extra Trees can really help in the wine industry, especially when we need to make decisions based on the quality of the wine.

In short, my research shows that the Extra Trees model is a great tool for figuring out wine quality. It's a big step forward in using technology to improve how we understand and evaluate wine. This could lead to more use of machine learning in the wine industry and other areas where knowing the quality of a product is key.