Project2AI

July 30, 2025

1 CI - 0129 | University of Costa Rica

2 Red Wine Dataset

3 Problem Analysis

In this notebook, we will study the influence of 11 factors on red wine quality with the goal of taking the most relevant features as predictors for the wine's quality. These possible features are:

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide
- Total sulfur dioxide
- Density
- pH
- Sulphates
- Alcohol

To carry out this research, we'll use this Dataset available on Kaggle. Initially, the data will be imported and cleaned, the purpose of this is that in order to build a successful machine learning model, good and clean data is needed. Secondly, knowledge needs to be gathered about the dataset and we do this by using Exploratory Data Analysis. Finally, we train supervised machine learning models on this data that was cleaned beforehand, comparing the performance of these models and explaining the obtained results.

4 Data Analysis (Pre-Processing)

4.1 Dataset Load

Firsly we will import the necesary libraries for data downloading and handling. Then we will load the dataset:

[ ]:
```
import pandas as pd
```

[ ]:
```
```

```
wine_data = pd.read_csv('https://www.kaggle.com/api/v1/datasets/download/uciml/
 ↪red-wine-quality-cortez-et-al-2009?
 ↪dataset_version_number=2&file_name=winequality-red.csv')
wine_data.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
```

Now that the dataset was imported correctly, let's take a brief look into the types of the individual variables:

[ ]:
```
wine_data.dtypes
```

[ ]:
```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
dtype: object
```

It is important to note that `quality` in this original dataset has an integer format (`int64`) and sets the ground for multi-class classification. However, since the goal of the present study is to apply binary classification, this normal integer target variable will be converted into a binary target variable following this set of rules:

- **Bad wine:** quality values 3, 4 and 5.
- **Good wine:** quality values 6, 7 and 8.

This variable will be changed later on to comply with this set of rules previously described.

Looking at the column count:

[ ]:
```
len(wine_data.columns)
```

[ ]:
```
12
```

It can be seen that the dataset has 12 columns, 11 being the ones described at the start of the notebook, plus the target variable (the wine quality values themselves).

4.2 Target Variable Conversion

Since we're dealing with a binary classification problem, we'll make the target variable binary:

[ ]:
```
wine_data['quality'] = (wine_data['quality'] > 5).astype(int)
```

4.3 Initial Data Exploration

In order to have an idea of how the values are distributed, metrics like the median and the standard deviation of each column will be shown:

[ ]:
```
wine_data.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     0.534709
std       0.154386     0.169507     1.065668     0.498950
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     1.000000
75%       3.400000     0.730000    11.100000     1.000000
max       4.010000     2.000000    14.900000     1.000000
```

Analyzing the resulting table, it can be seen that the variables have very different scales between them. This means that the application of either normalization or standardization will be needed in order to improve the model's performance on this data.

So, the values fall into the following ranges:

- **fixed acidity:** 4.6-15.9
- **volatile acidity:** 0.12-1.58
- **citric acid:** 0-1
- **residual sugar:** 0.9-15.5
- **chlorides:** 0.012-0.611
- **free sulfur dioxide:** 1-72
- **total sulfur dioxide:** 6-289
- **density:** 0.99-1.003
- **pH:** 2.74-4.01
- **sulphates:** 0.33-2
- **alcohol:** 8.4-14.9
- **quality:** 0 or 1

Also, thanks to the `count` measurement, the amount of rows present in the dataset can be inferred, so the dataset has 1599 rows.

4.4 Handling Missing Values

Checking how many null values are there in the wine dataset:

[ ]:
```
wine_data.isna().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
```

Watching the output, it can be seen that there are no missing values in the wine dataset.

4.5 Exploratory Data Analysis

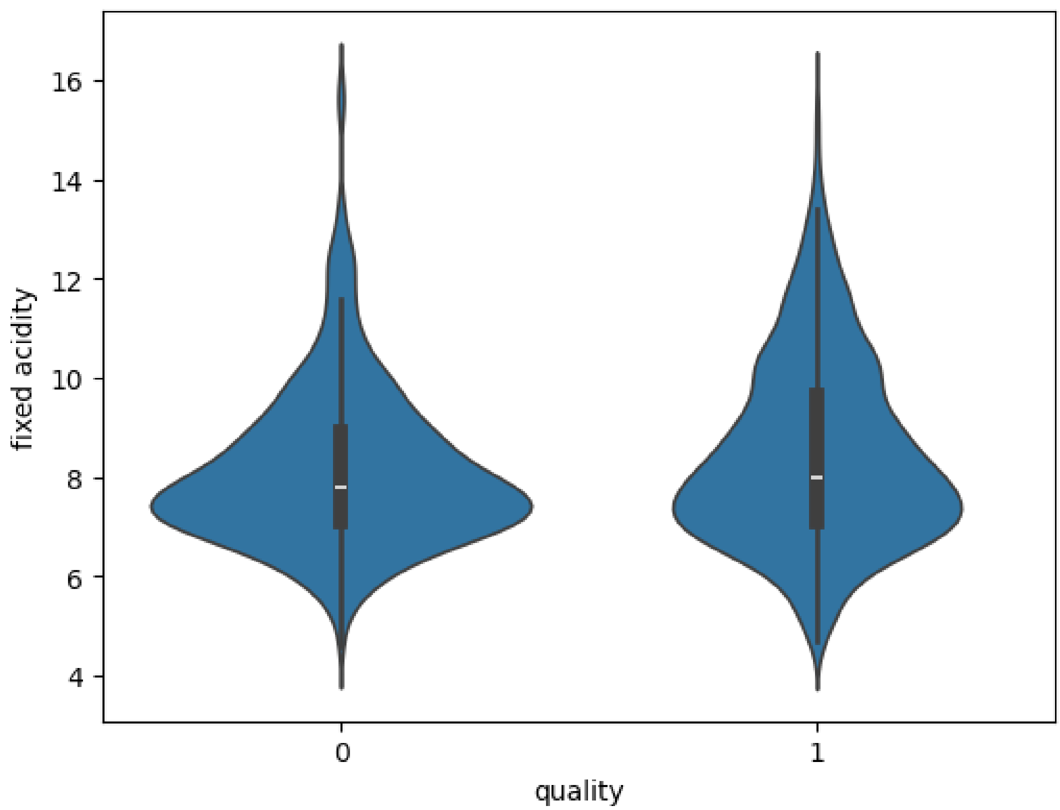
Learning about the data and its distributions can be helpful in order to extract a good performance out of the machine learning models.

4.5.1 Relationships Between Variables

First, we'll look at how the variables might influence the response variable:

Analyzing `Quality` and `Fixed Acidity`:

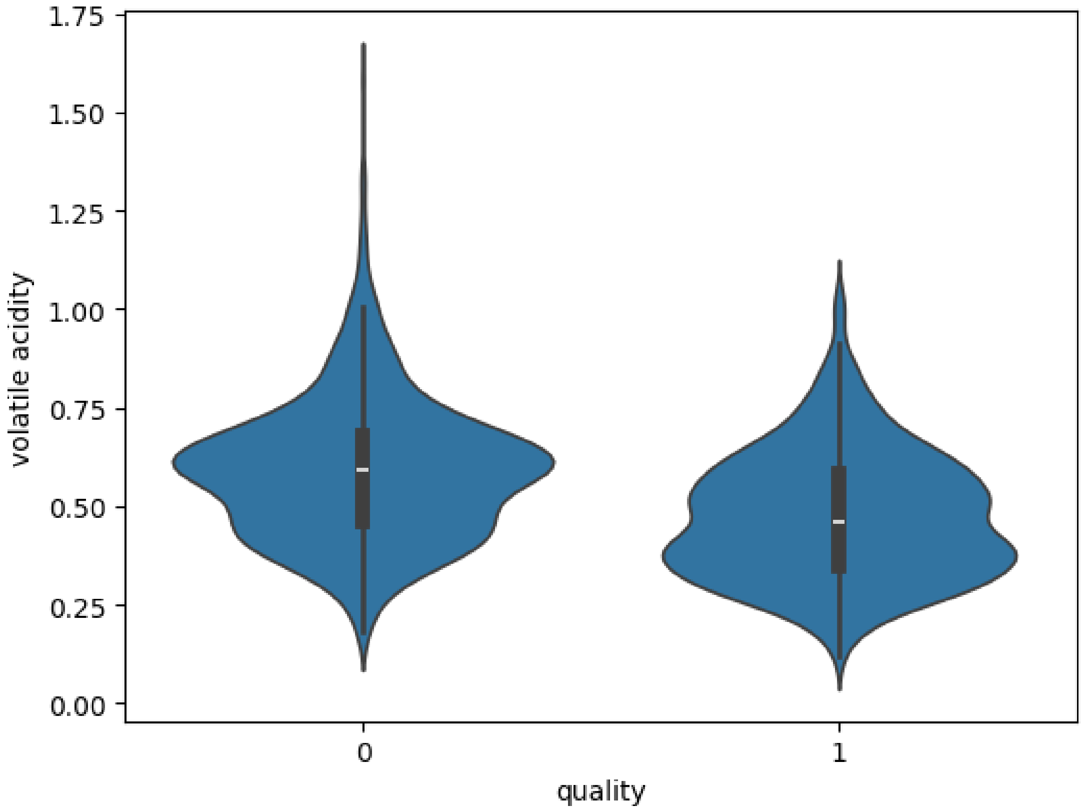
[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.violinplot(x='quality', y='fixed acidity', data=wine_data)
plt.show()
```



According to the graph, there isn't a straightforward relationship between the `Quality` and the `Fixed Acidity` of the wine.

Analyzing `Quality` and `Volatile Acidity`:

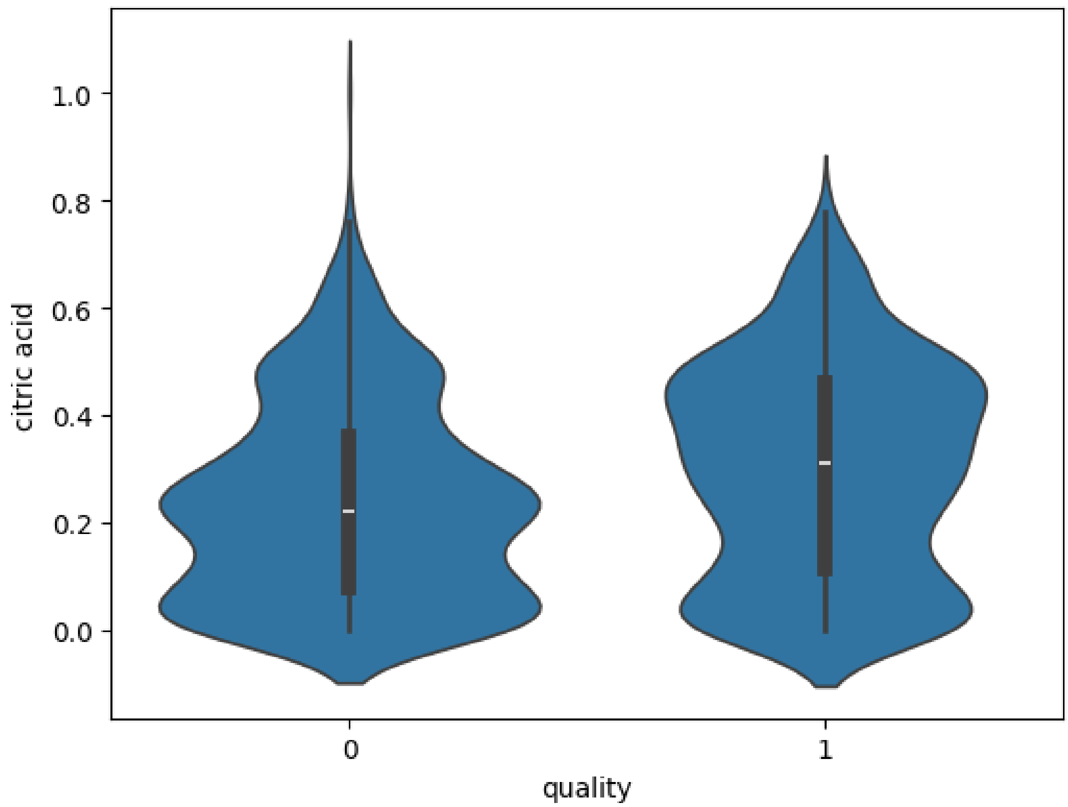
[ ]:
```
sns.violinplot(x='quality', y='volatile acidity', data=wine_data)
plt.show()
```



It seems that there is a downward trend between the `Volatile Acidity` and the `Quality` of the wine, in the sense that good wines have lower `Volatile Acidity` values in general, but this is a pretty small difference.

Analyzing `Quality` and `Citric Acid`:

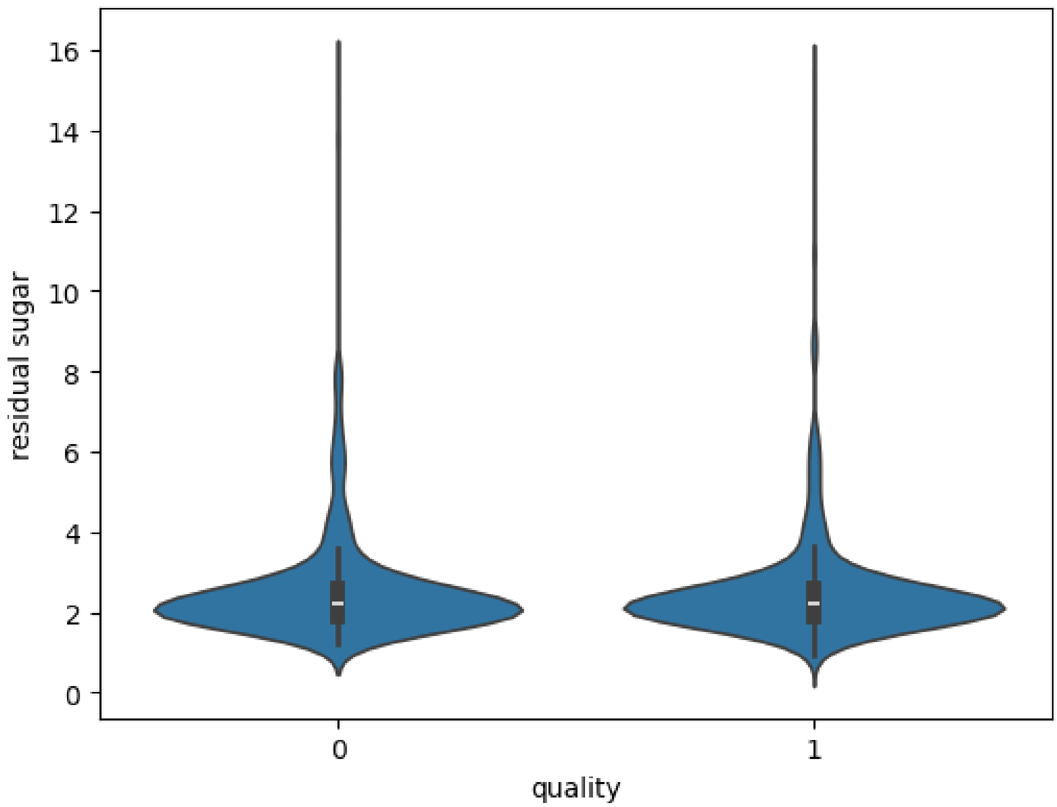
[ ]:
```
sns.violinplot(x='quality', y='citric acid', data=wine_data)
plt.show()
```



Higher `quality` wines tend to have a bit of a higher `citric acid` value.

Analyzing `Quality` and `Residual Sugar`:

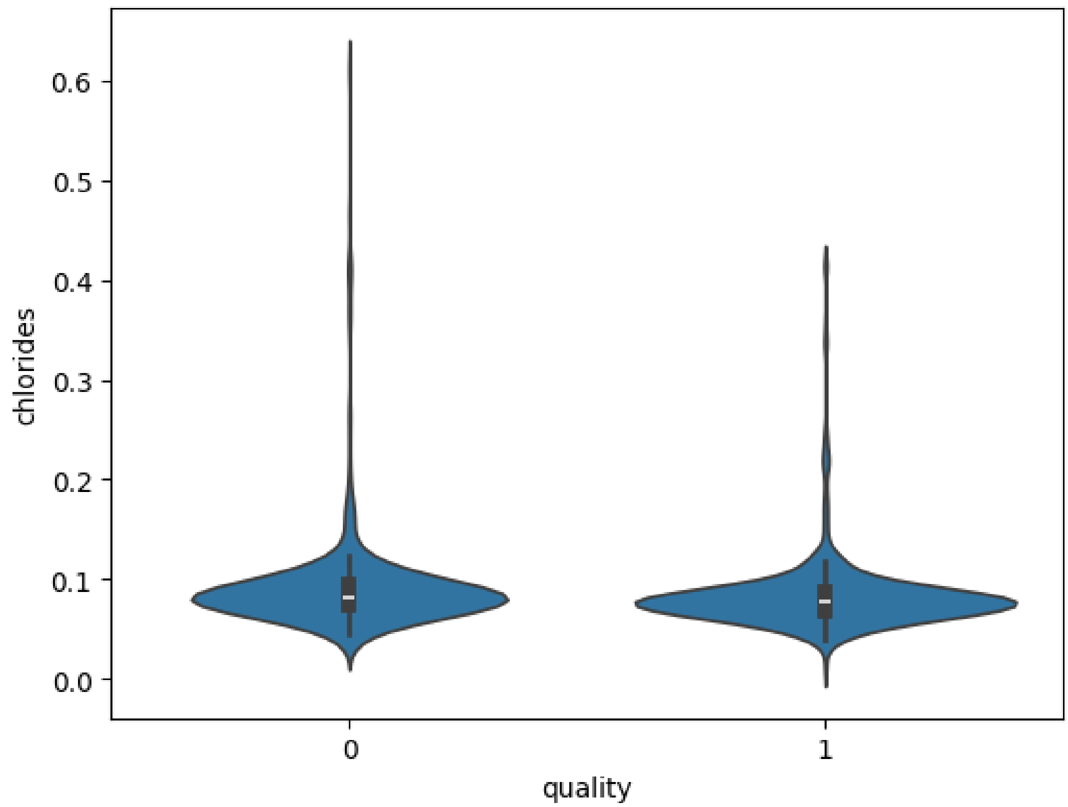
[ ]:
```
sns.violinplot(x='quality', y='residual sugar', data=wine_data)
plt.show()
```



There doesn't seem to exist a relationship between `residual sugar` and `quality` neither.

Comparing `quality` and `chlorides`:

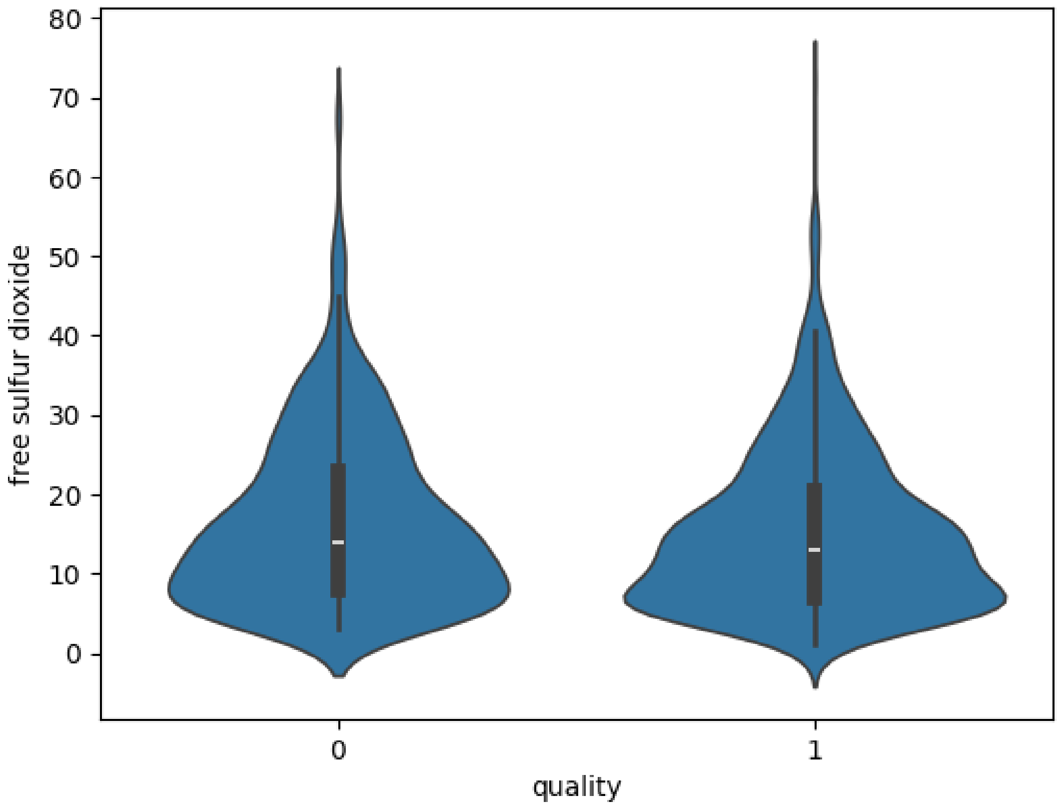
[ ]:
```
sns.violinplot(x='quality', y='chlorides', data=wine_data)
plt.show()
```



There’s also no relationship between `chlorides` and the `quality` of wine.

Comparing `quality` and `free sulfur dioxide`:

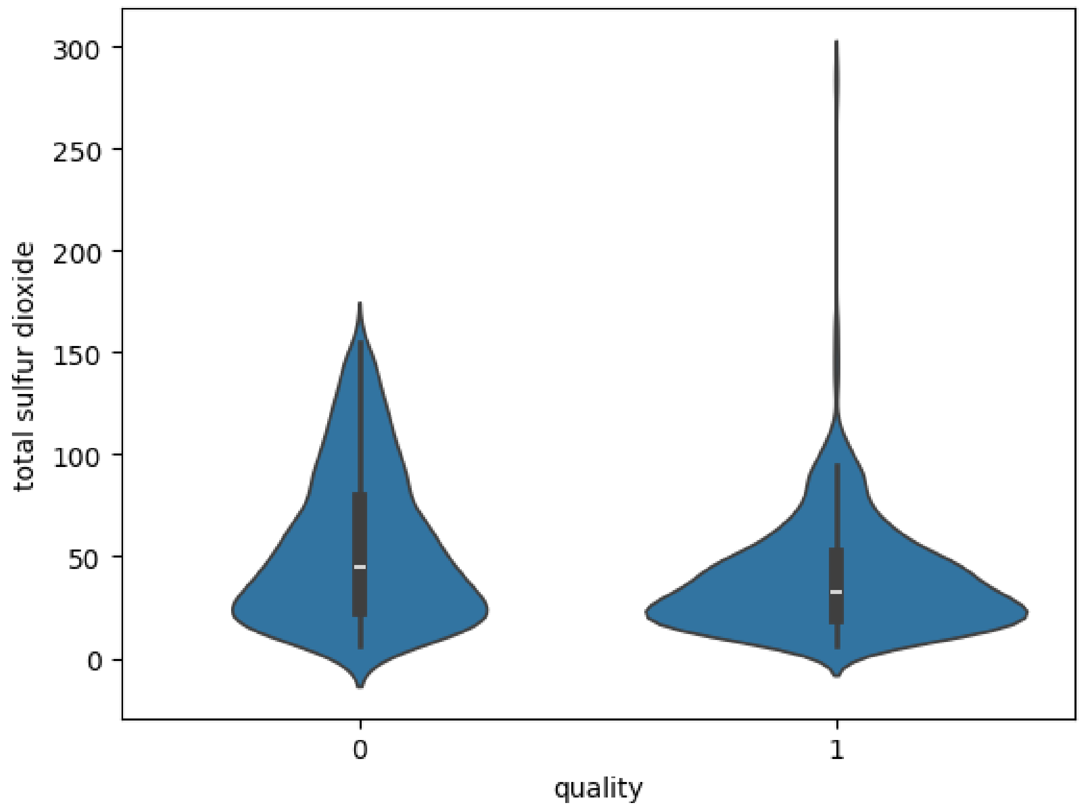
[ ]:
```
sns.violinplot(x='quality', y='free sulfur dioxide', data=wine_data)
plt.show()
```



No clear relationship can be seen between the `free sulfur dioxide` variable and the `quality` variable.

Comparing `quality` and `total sulfur dioxide`:

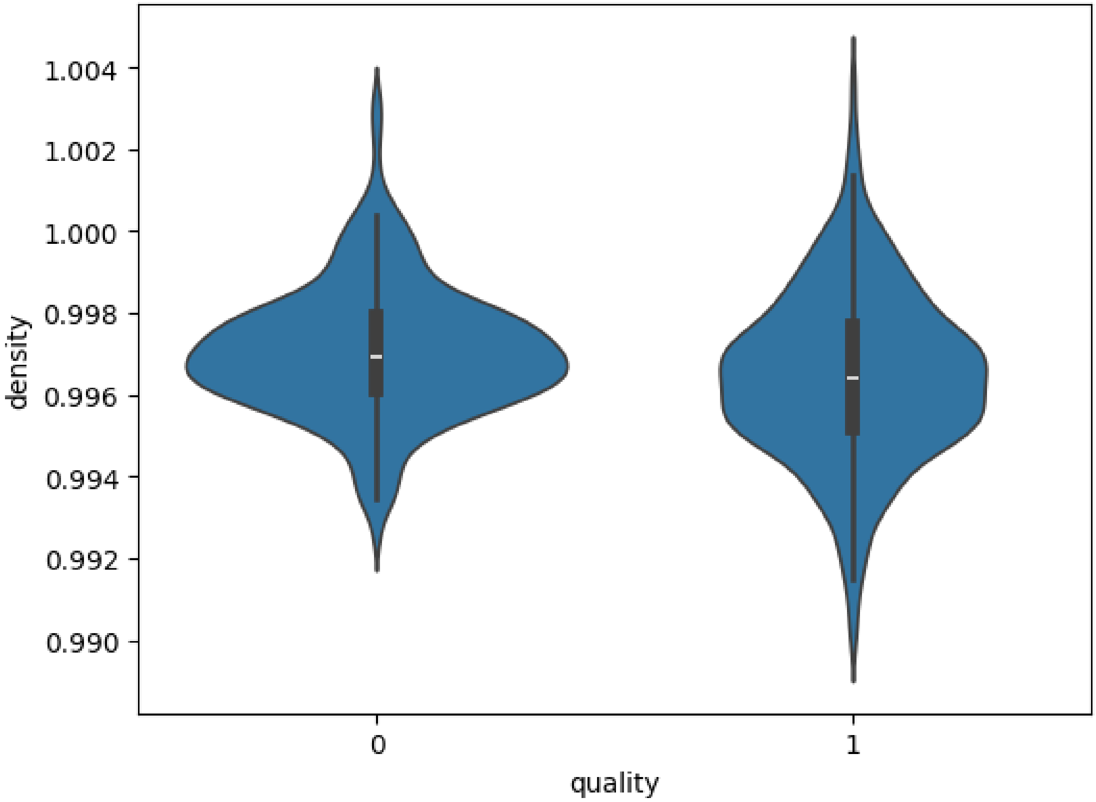
[ ]:
```
sns.violinplot(x='quality', y='total sulfur dioxide', data=wine_data)
plt.show()
```



Higher `quality` wines tend to have smaller `total sulfur dioxide` values. Generating an extremely weak downward trend in the graph.

Comparing `quality` and `density`:

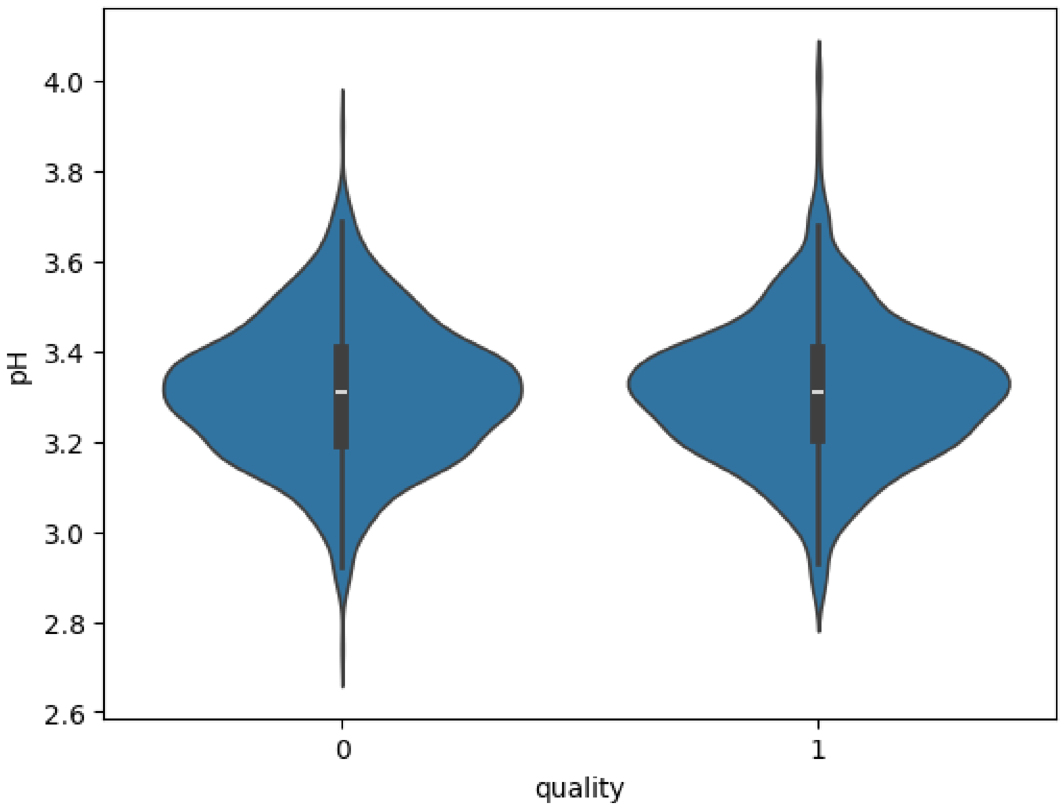
[ ]:
```
sns.violinplot(x='quality', y='density', data=wine_data)
plt.show()
```



There isn't a clear trend between the `density` and the `quality` values.

Comparing `quality` and `pH`:

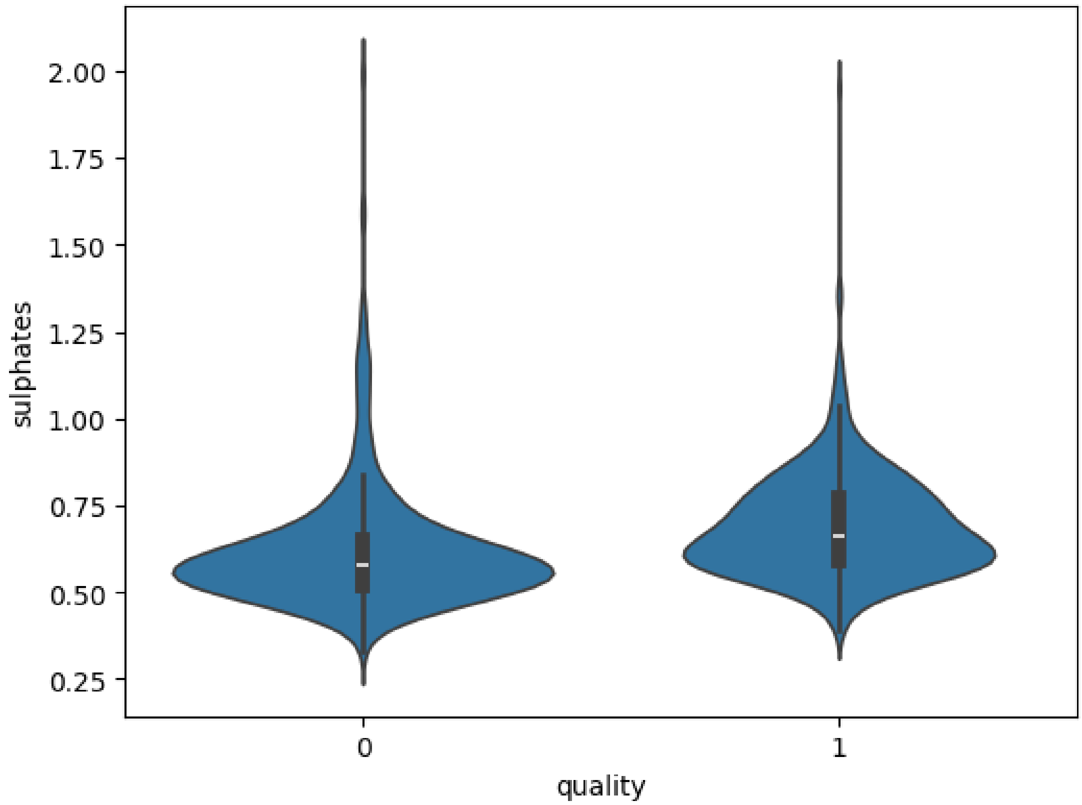
[ ]:
```
sns.violinplot(x='quality', y='pH', data=wine_data)
plt.show()
```



No clear trend is found between `pH` and `quality`.

Comparing `quality` and `sulphates`:

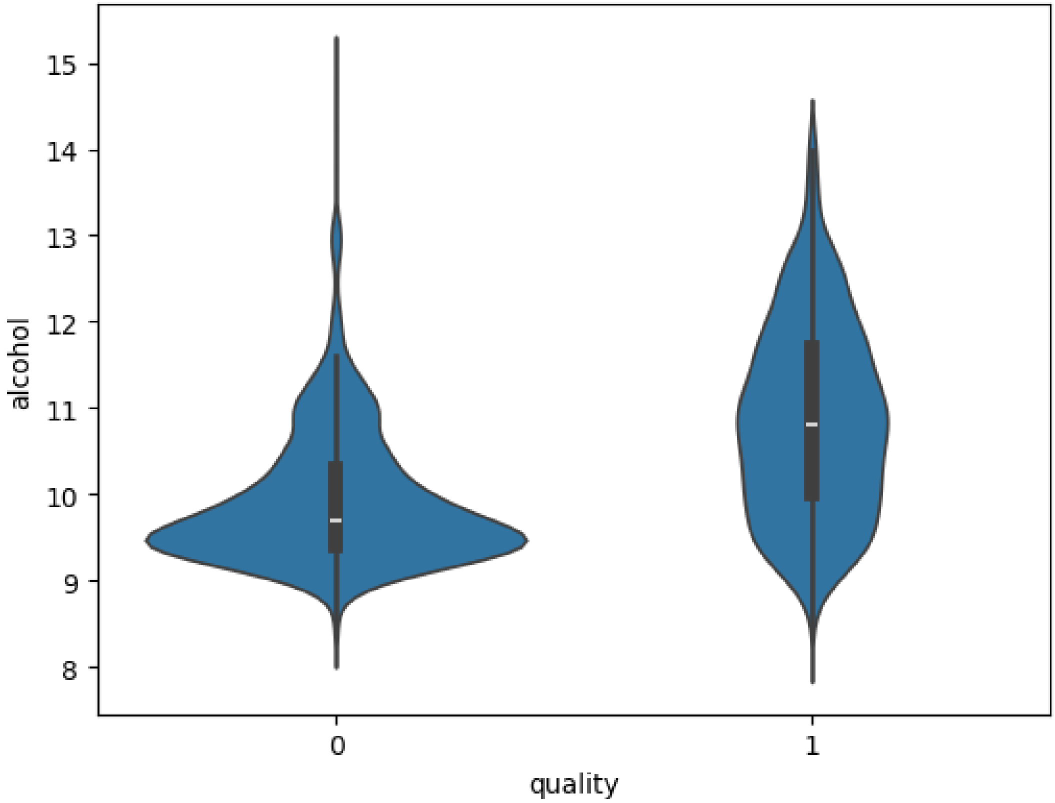
[ ]:
```
sns.violinplot(x='quality', y='sulphates', data=wine_data)
plt.show()
```



Higher `quality` wines tend to have higher `sulphates` values.

Comparing `quality` and `alcohol`:

[ ]:
```
sns.violinplot(x='quality', y='alcohol', data=wine_data)
plt.show()
```

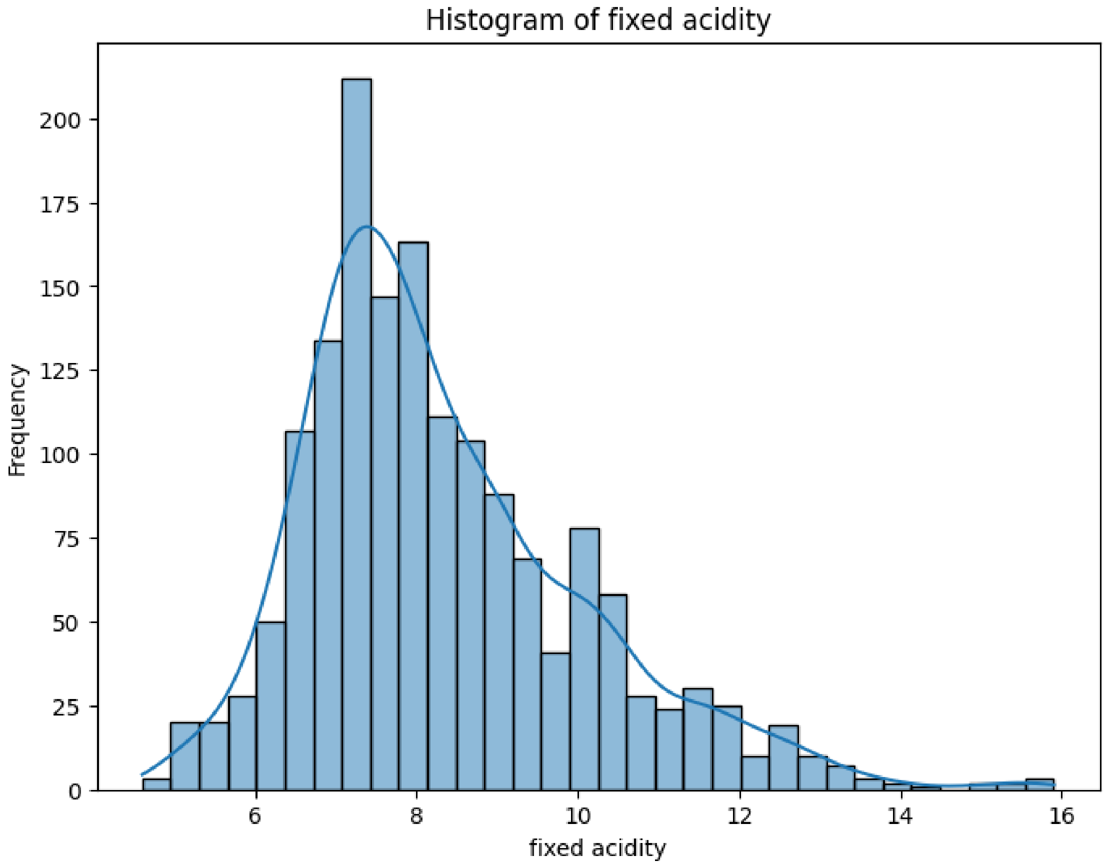


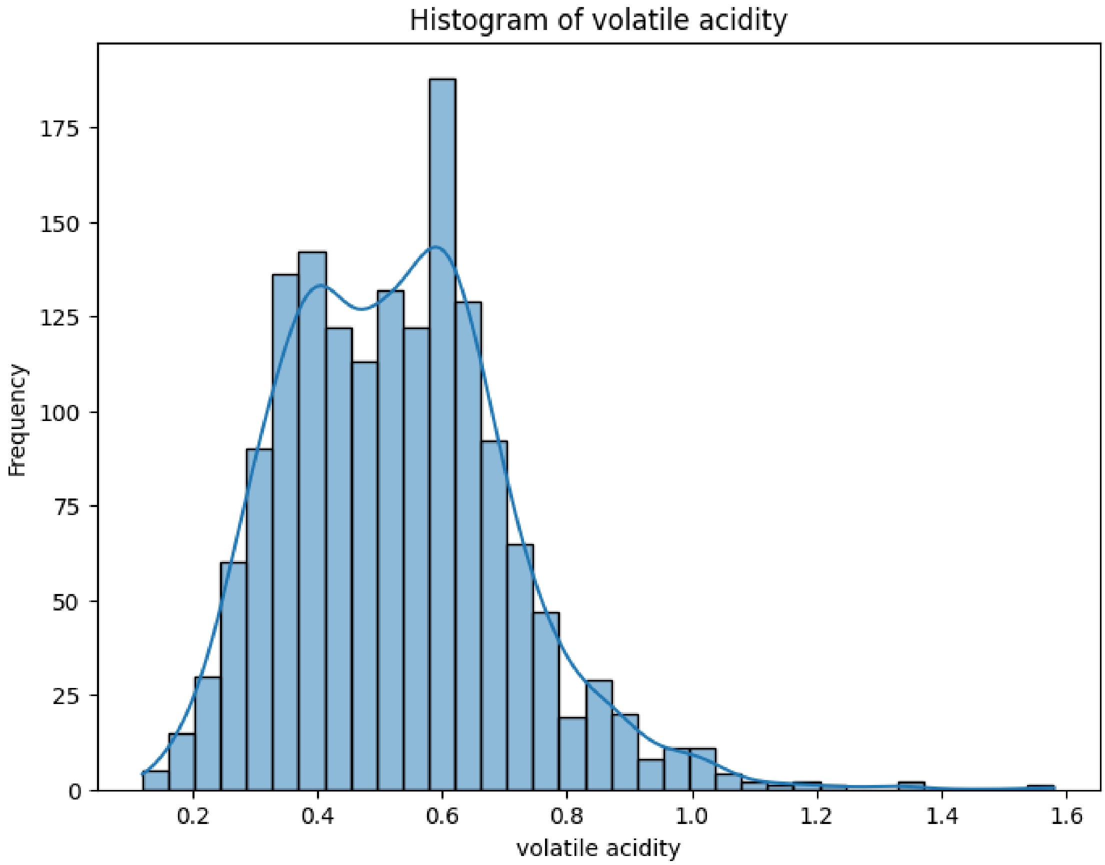
There's a clear upward trend in the sense that higher `quality` wines have higher `alcohol` amounts.

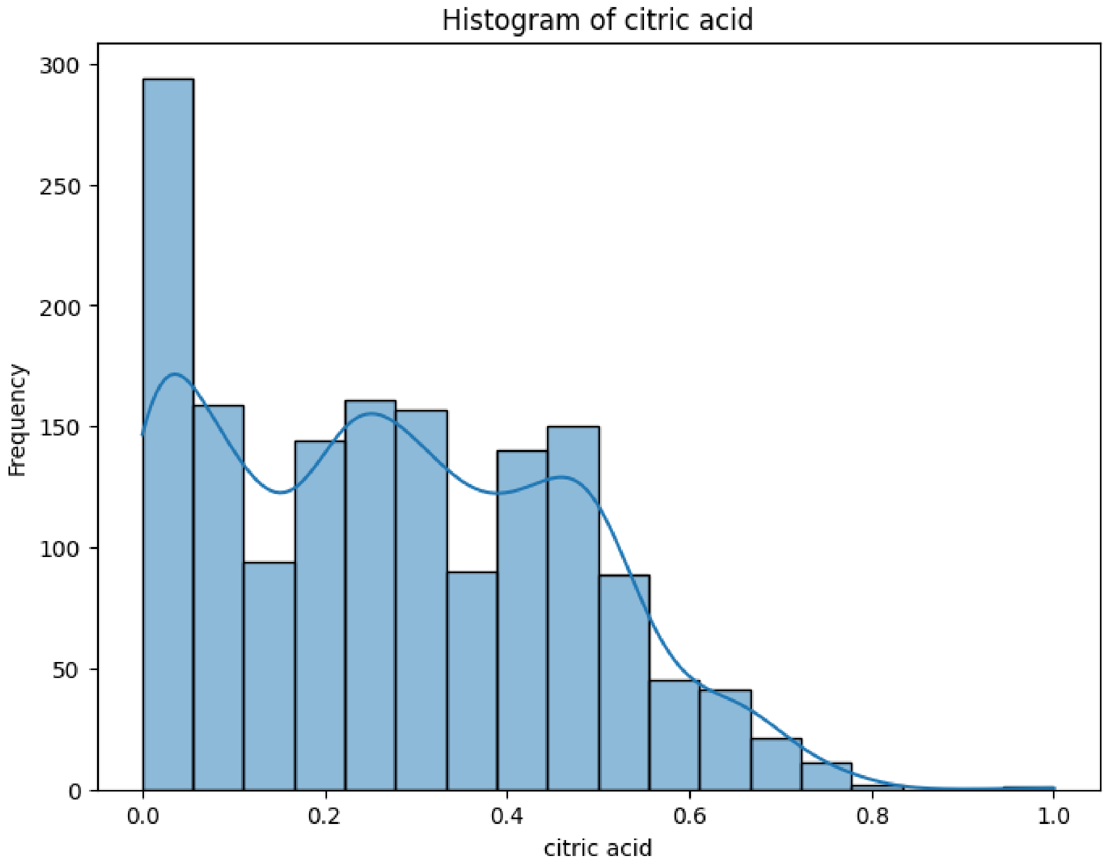
4.5.2 Variable Distribution

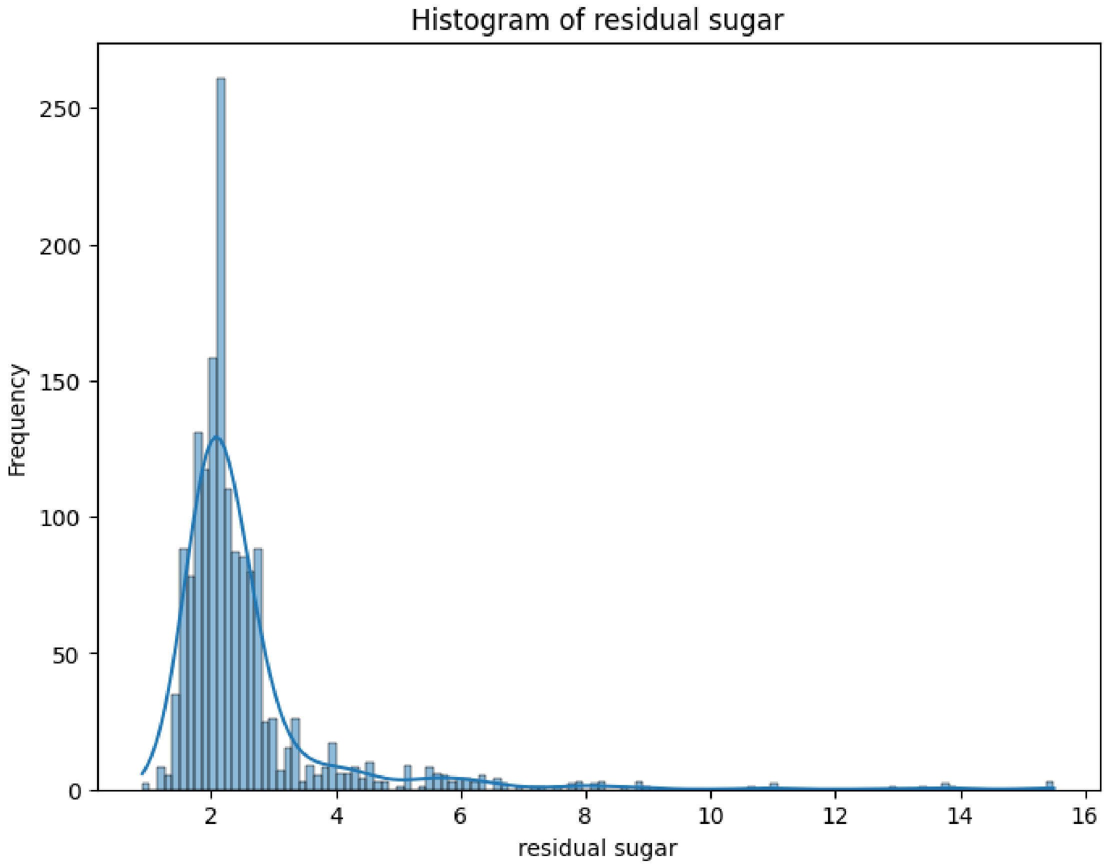
Now we will look at how these individual variables are distributed in our dataset:

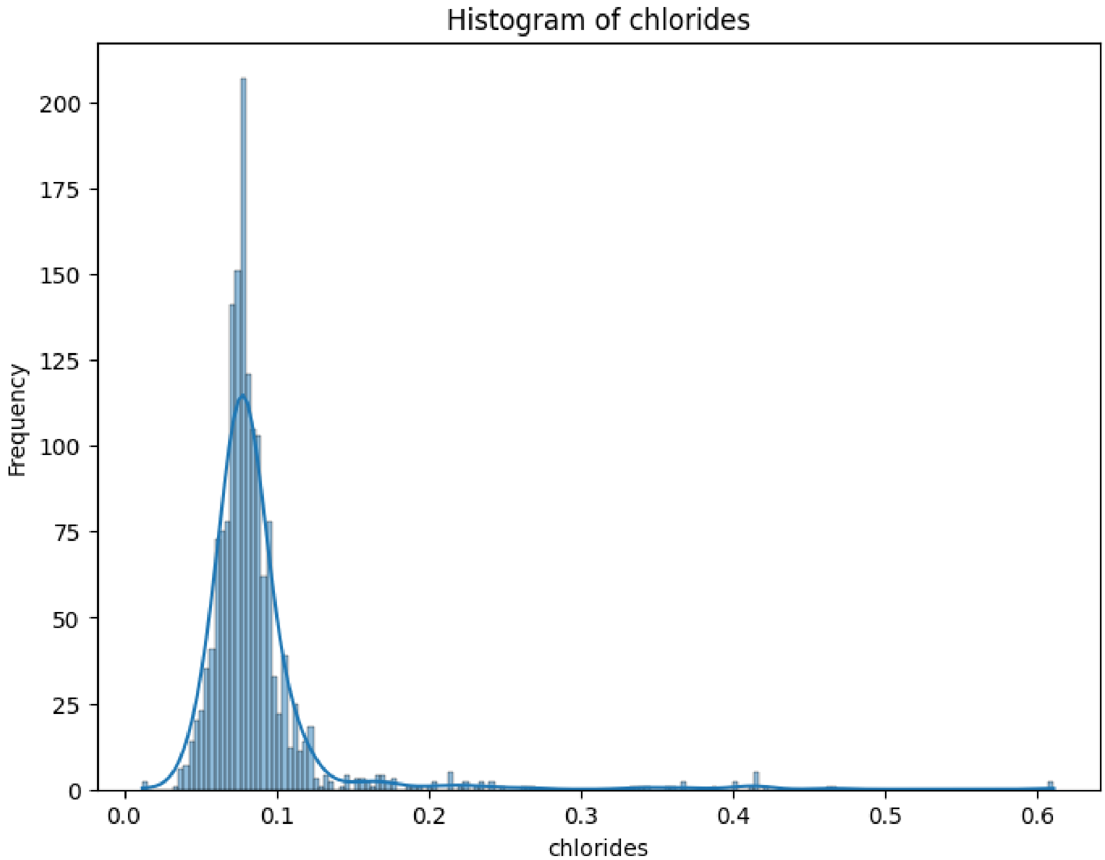
[ ]:
```
for column in wine_data.columns:
  if column != 'quality':
    plt.figure(figsize=(8, 6))
    sns.histplot(wine_data[column], kde=True)
    plt.title(f'Histogram of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.show()
```


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

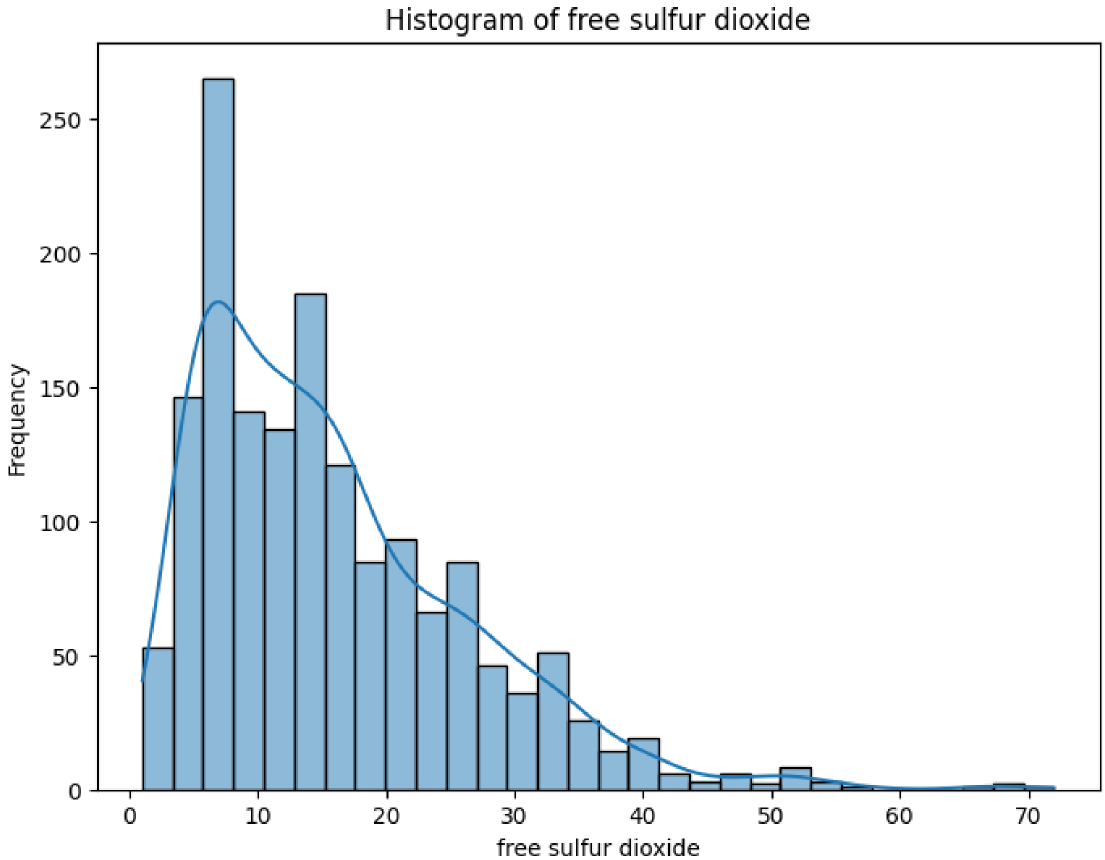

Histogram of volatile acidity
Frequency
0
25
50
75
100
125
150
175
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
volatile acidity

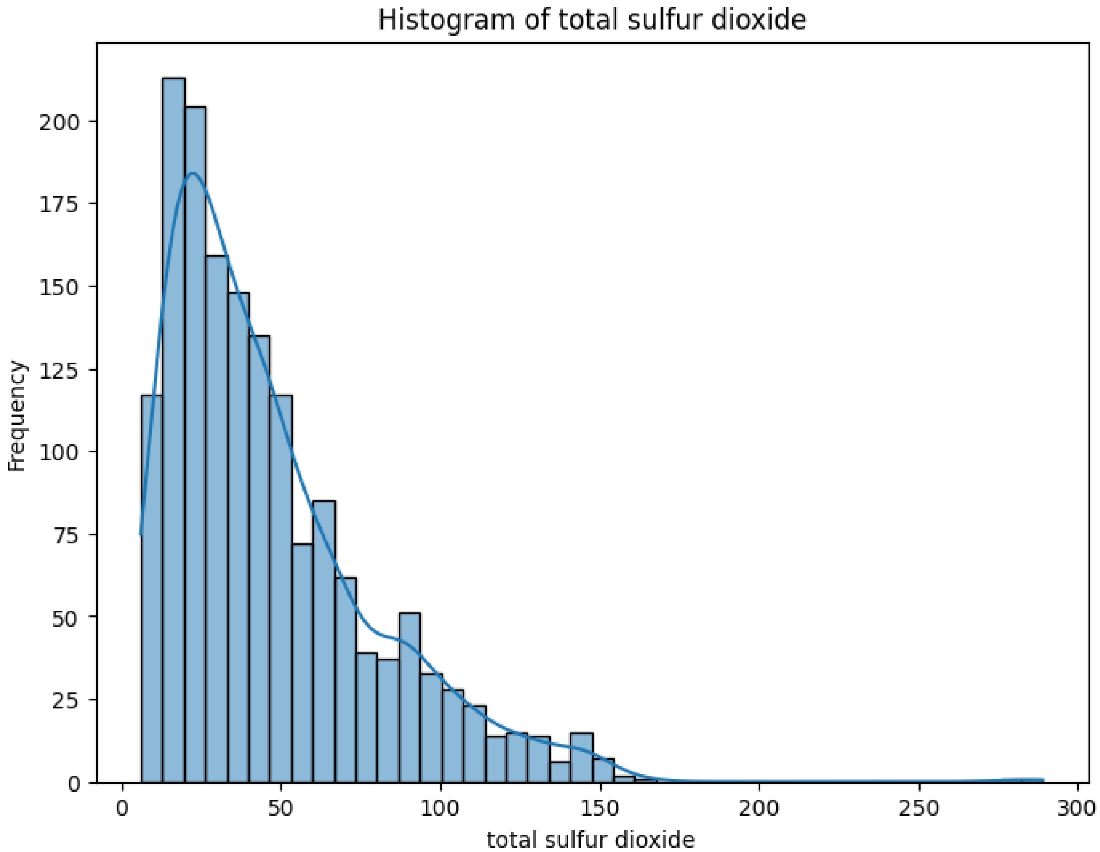

Histogram of citric acid
Frequency
300
250
200
150
100
50
0
0.0
0.2
0.4
0.6
0.8
1.0
citric acid

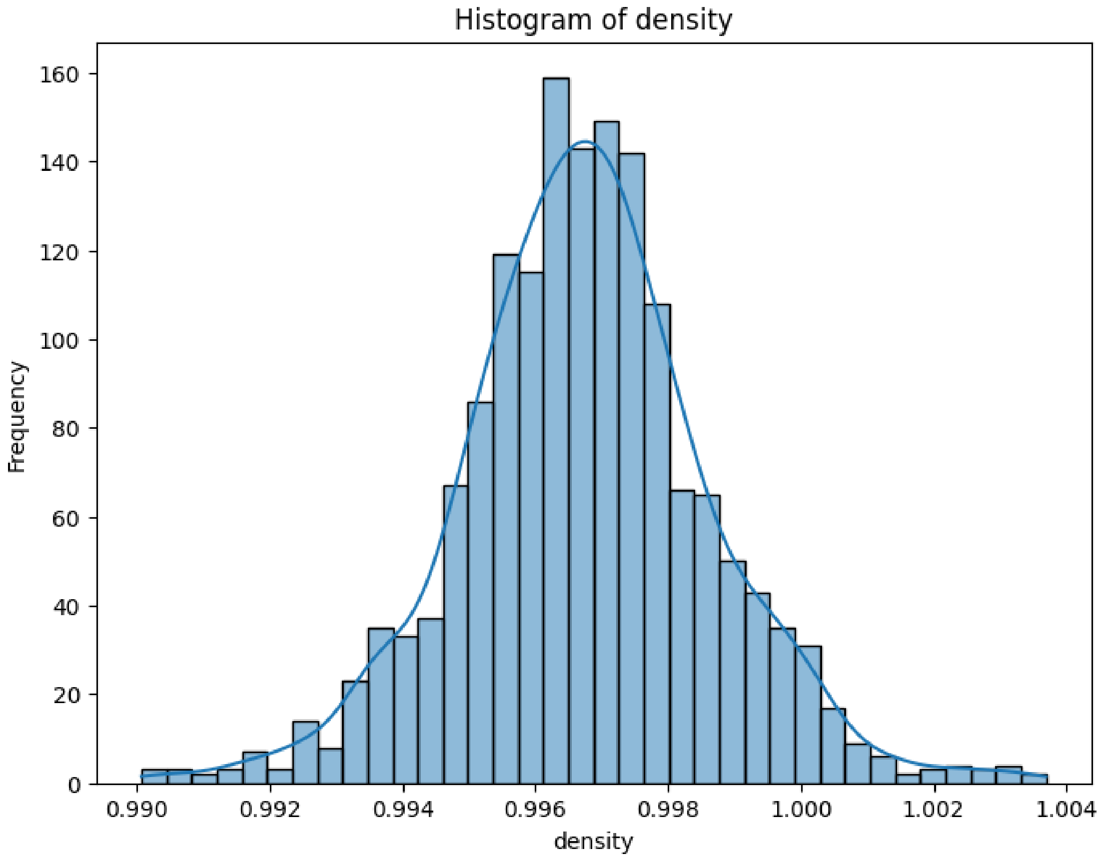

Histogram of residual sugar
Frequency
0
50
100
150
200
250
2
4
6
8
10
12
14
16
residual sugar

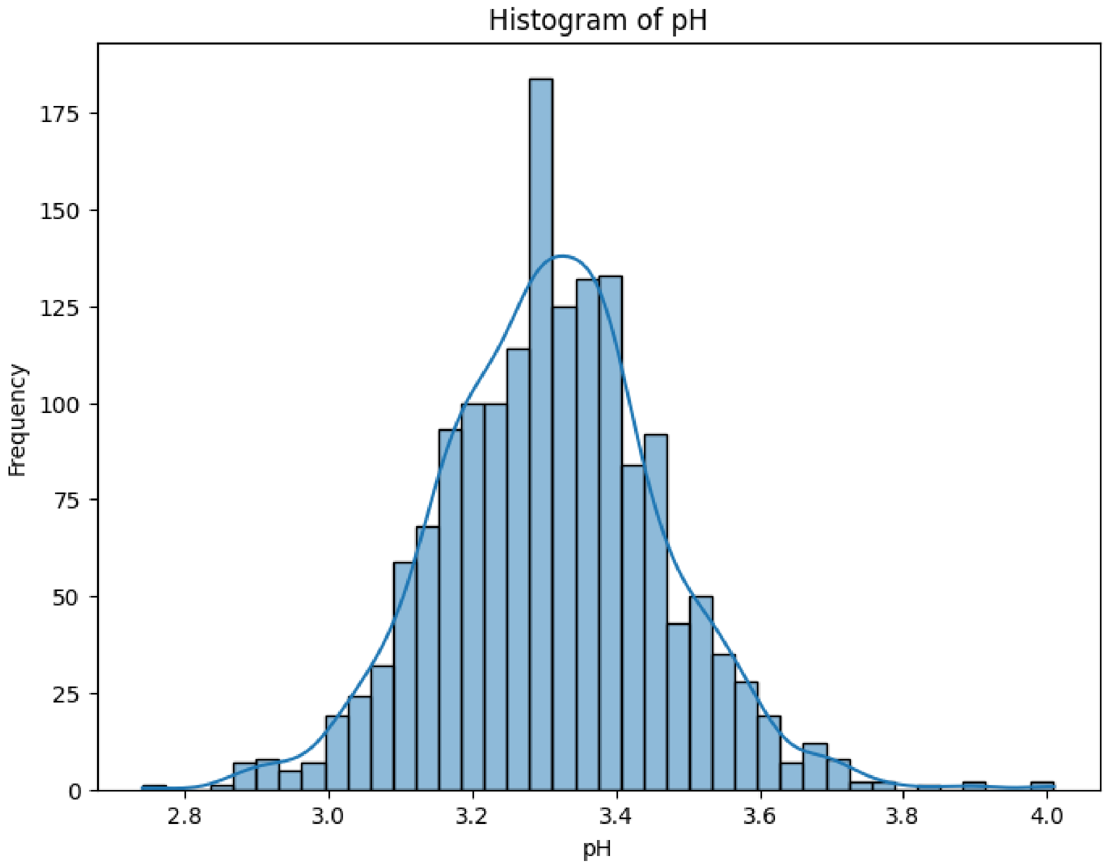


Histogram of chlorides
Frequency
200
175
150
125
100
75
50
25
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
chlorides

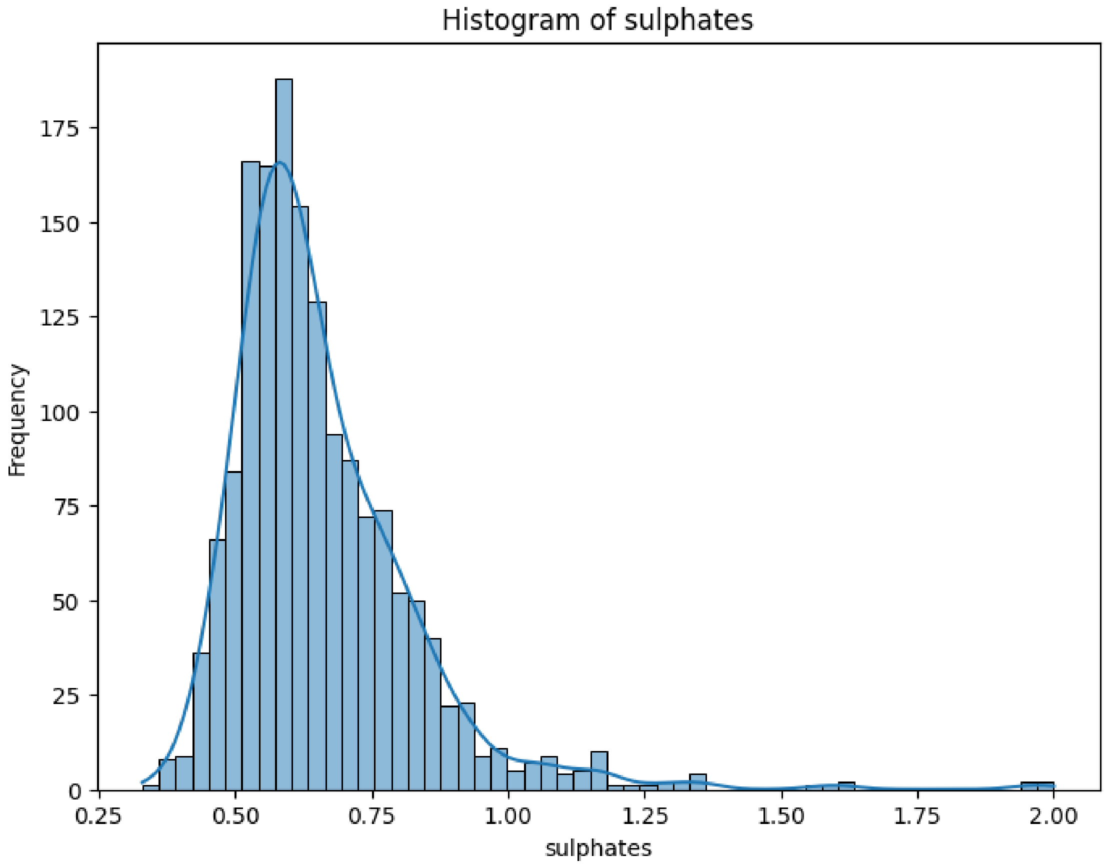

Histogram of free sulfur dioxide
Frequency
free sulfur dioxide
0
50
100
150
200
250
0
10
20
30
40
50
60
70


Histogram of total sulfur dioxide
Frequency
0
25
50
75
100
125
150
175
200
0
50
100
150
200
250
300
total sulfur dioxide

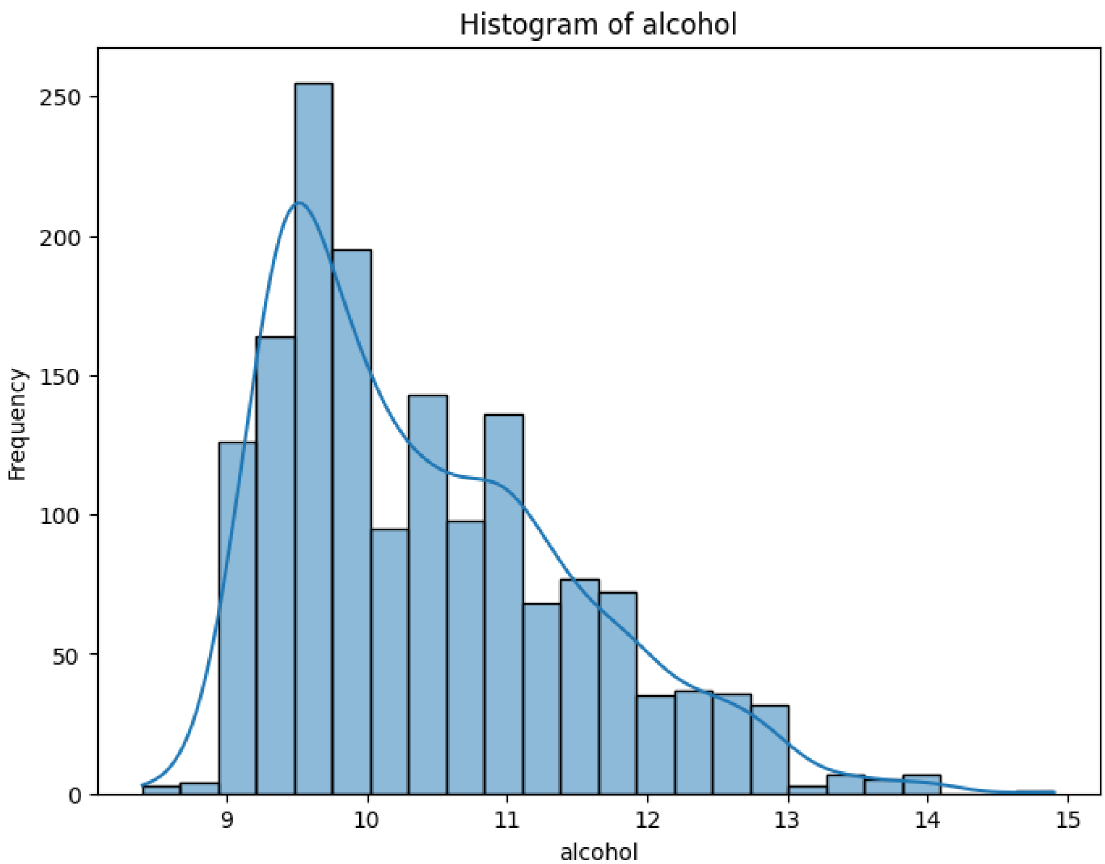

Histogram of density
Frequency
0
20
40
60
80
100
120
140
160
0.990
0.992
0.994
0.996
0.998
1.000
1.002
1.004
density



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



Histogram of sulphates
Frequency
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

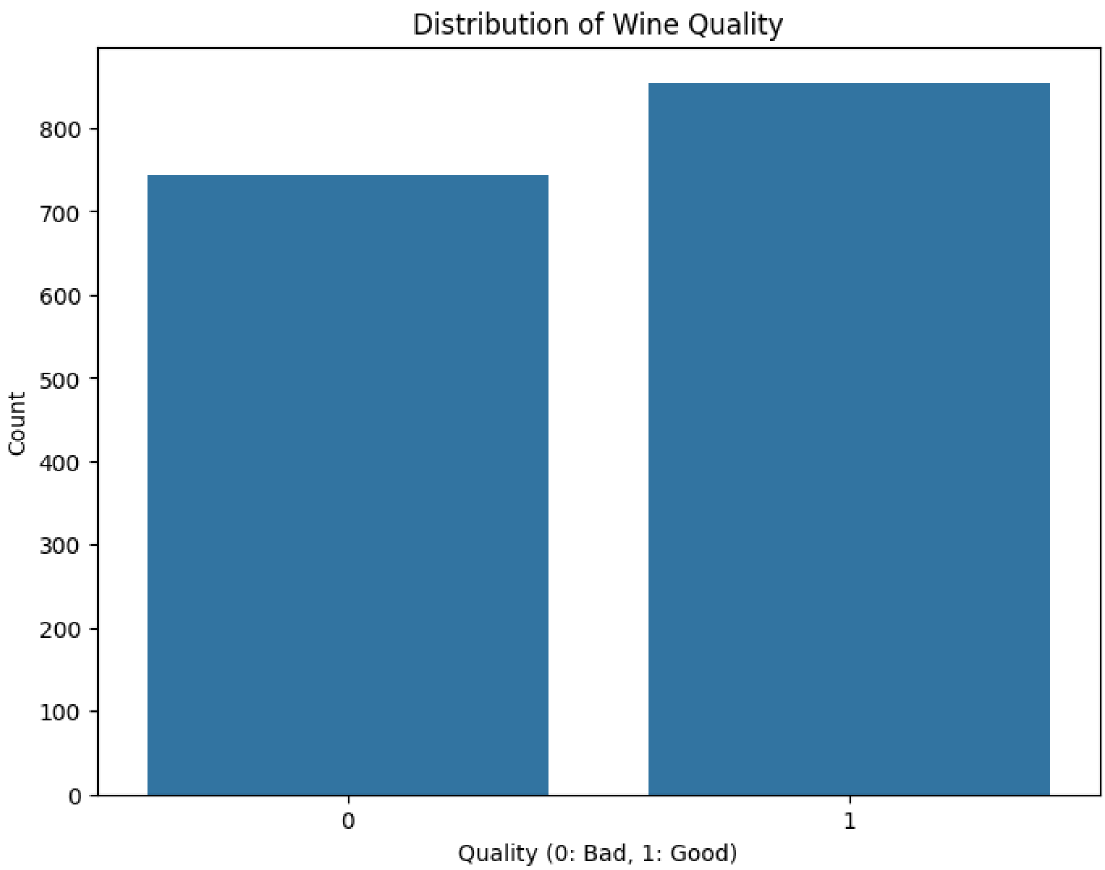


As it can be seen, a lot of our variables follow a somewhat normal distribution, with the following ones being the least similar to the bell-shaped curve:

- `volatile acidity`
- `citric acid`
- `free sulfur dioxide`
- `total sulfur dioxide`
- `alcohol`

Target Variable Distribution Looking at the actual target variable, we will analyze its distribution of values (since in this case the variable is categorical, then the histogram returns the amount of values present in each class).

[ ]:
```
plt.figure(figsize=(8, 6))
sns.countplot(x='quality', data=wine_data)
plt.title('Distribution of Wine Quality')
plt.xlabel('Quality (0: Bad, 1: Good)')
plt.ylabel('Count')
plt.show()
```



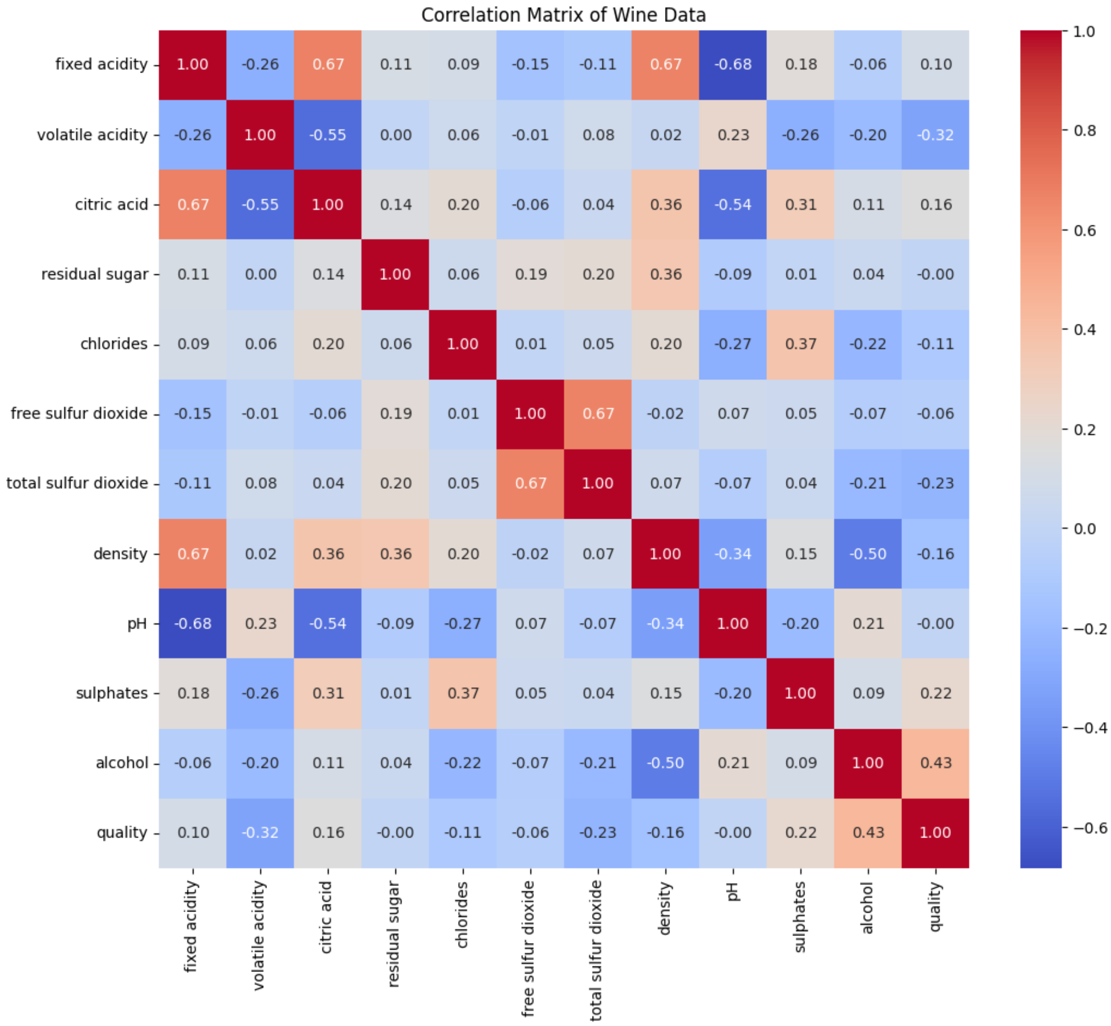
The above graph shows that the dataset has more good quality wine than bad quality wine, noting that the difference is not that big.

This shows that maybe some upsampling of the bad quality wine class will be needed after the data split.

4.6 Correlation Between Variables

In order to build our model, we need to analyze which variables cause the biggest effect on the wine quality. For this we will create a correlation heatmap that reveals this in a visual manner:

[ ]:
```
plt.figure(figsize=(12, 10))
correlation_matrix = wine_data.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of Wine Data')
plt.show()
```



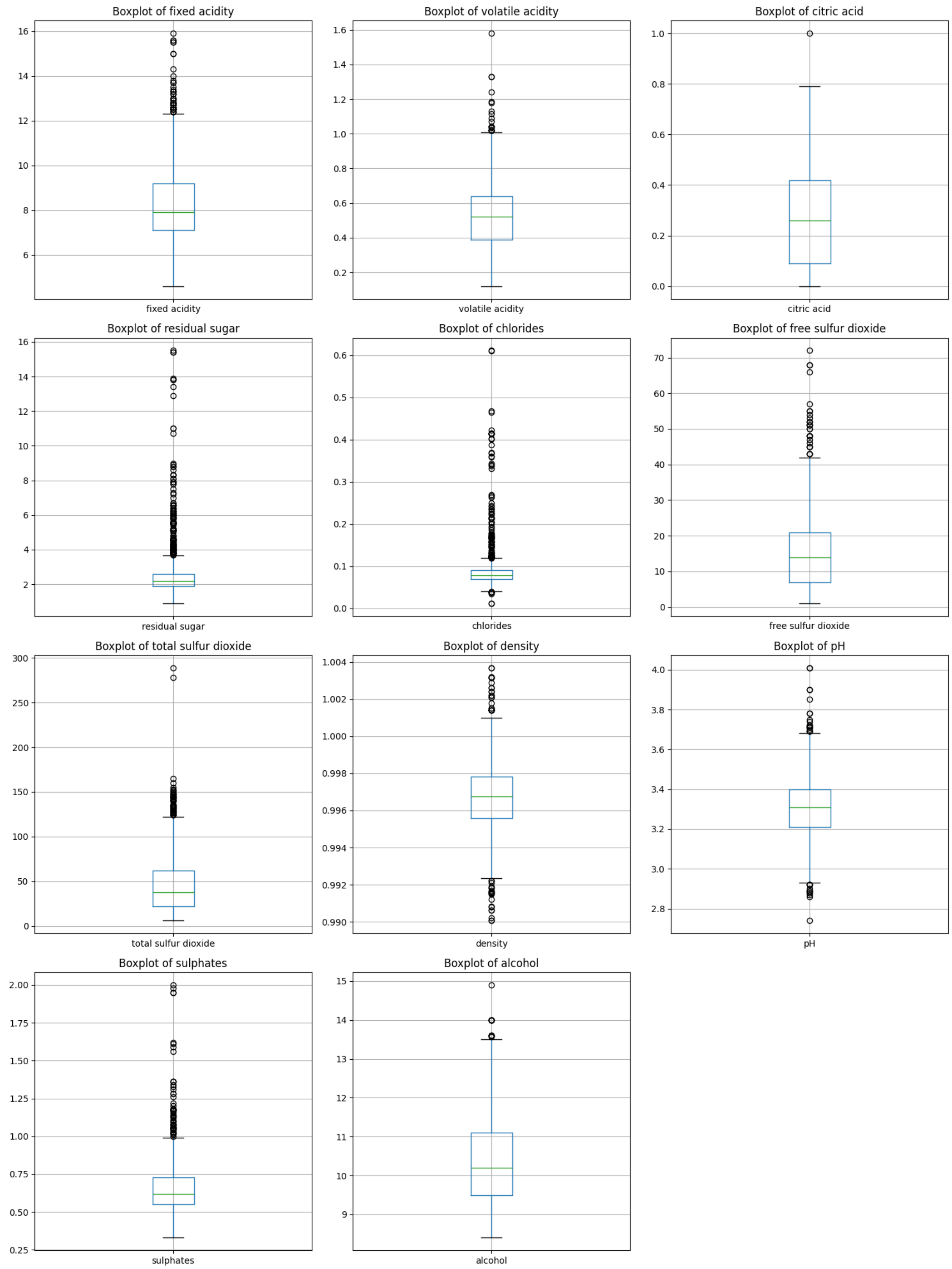
No strong correlations are found to determine wine quality, except for `volatile acidity` and `alcohol`. Since those are very few variables and their correlation is not very strong, all features will be used to train the models.

4.7 Outlier Detection

Now we will check to see if there are any outliers present in our data:

[ ]:
```
# Get prediction factors
num_columns = wine_data.select_dtypes(include=['float64']).columns

# Prepare the plot to load a boxplot for each factor
num_plots = len(num_columns)
num_cols = 3
num_rows = (num_plots // num_cols) + (num_plots % num_cols > 0)
```

```python
fig, axes = plt.subplots(num_rows, num_cols, figsize=(num_cols * 5, num_rows *
    5))
axes = axes.flatten()

# Create a boxplot for every factor
for i, column in enumerate(num_columns):
    wine_data.boxplot(column=[column], ax=axes[i])
    axes[i].set_title(f'Boxplot of {column}')

for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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



There is a considerable amount of outliers in our dataset, with them being strongly present in almost each variable except `citric acid`, since it just has one circle (outlier) in its boxplot.

4.8 Outlier Deletion

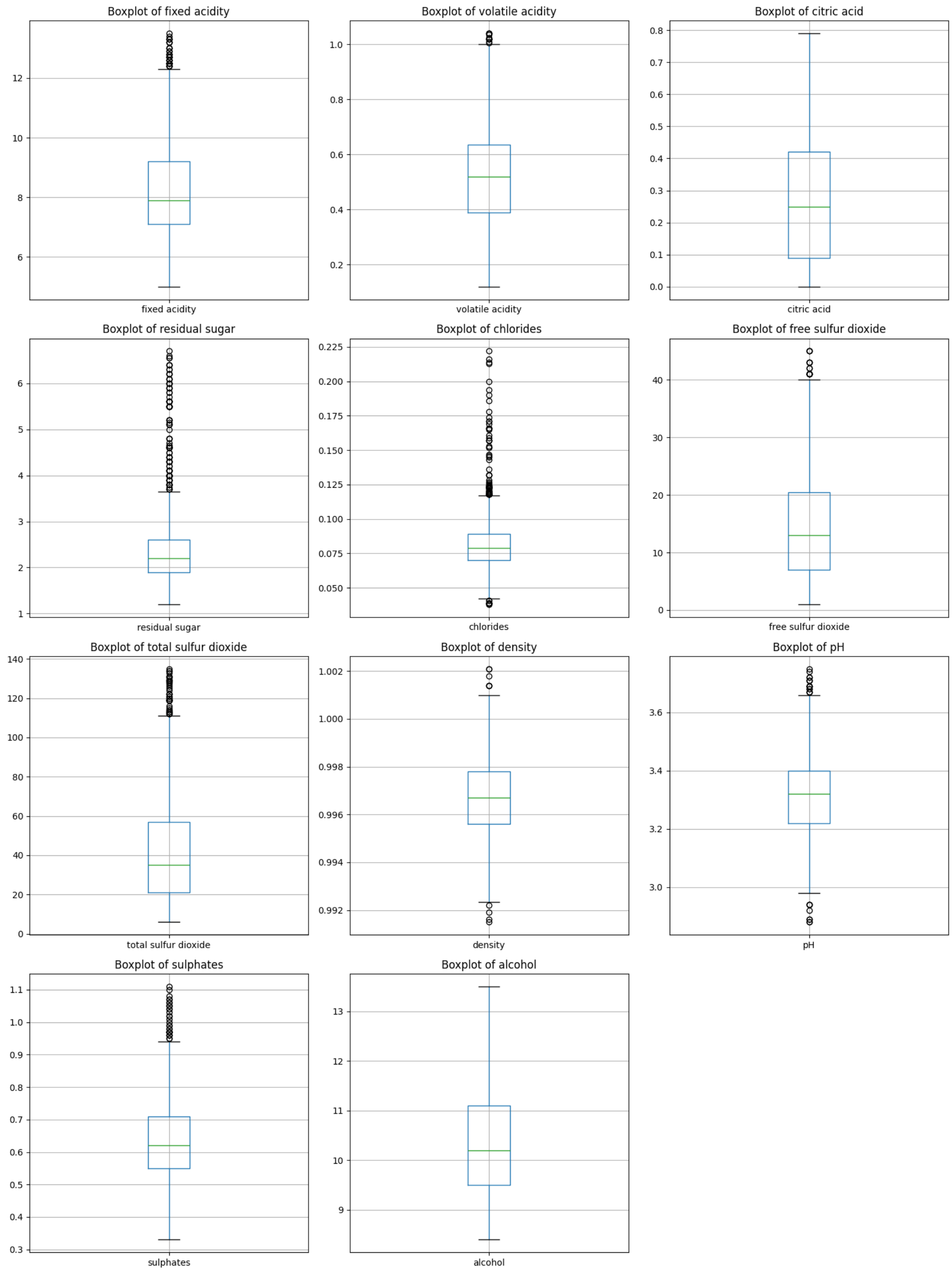
Since a lot of our variables are somewhat normally distributed, then the z-score method will be used to delete outliers.

The z-score method consists on deleting those values that fall more than 3 standard deviations away from the mean value.

[ ]:
```python
import numpy as np

for column in wine_data.select_dtypes(include=np.number).columns:
    # Calculate the mean and standard deviation
    mean = wine_data[column].mean()
    std = wine_data[column].std()

    # Calculate the z-score for each value
    z_scores = (wine_data[column] - mean) / std

    # Remove values with z-scores greater than 3 or less than -3
    wine_data = wine_data[(z_scores <= 3) & (z_scores >= -3)]
```

Now, taking a look again at the barplots to check on how the data looks after outlier deletion:

[ ]:
```python
# Get prediction factors
num_columns = wine_data.select_dtypes(include=['float64']).columns

# Prepare the plot to load a boxplot for each factor
num_plots = len(num_columns)
num_cols = 3
num_rows = (num_plots // num_cols) + (num_plots % num_cols > 0)

fig, axes = plt.subplots(num_rows, num_cols, figsize=(num_cols * 5, num_rows *␣
↪5))
axes = axes.flatten()

# Create a boxplot for every factor
for i, column in enumerate(num_columns):
    wine_data.boxplot(column=[column], ax=axes[i])
    axes[i].set_title(f'Boxplot of {column}')

for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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

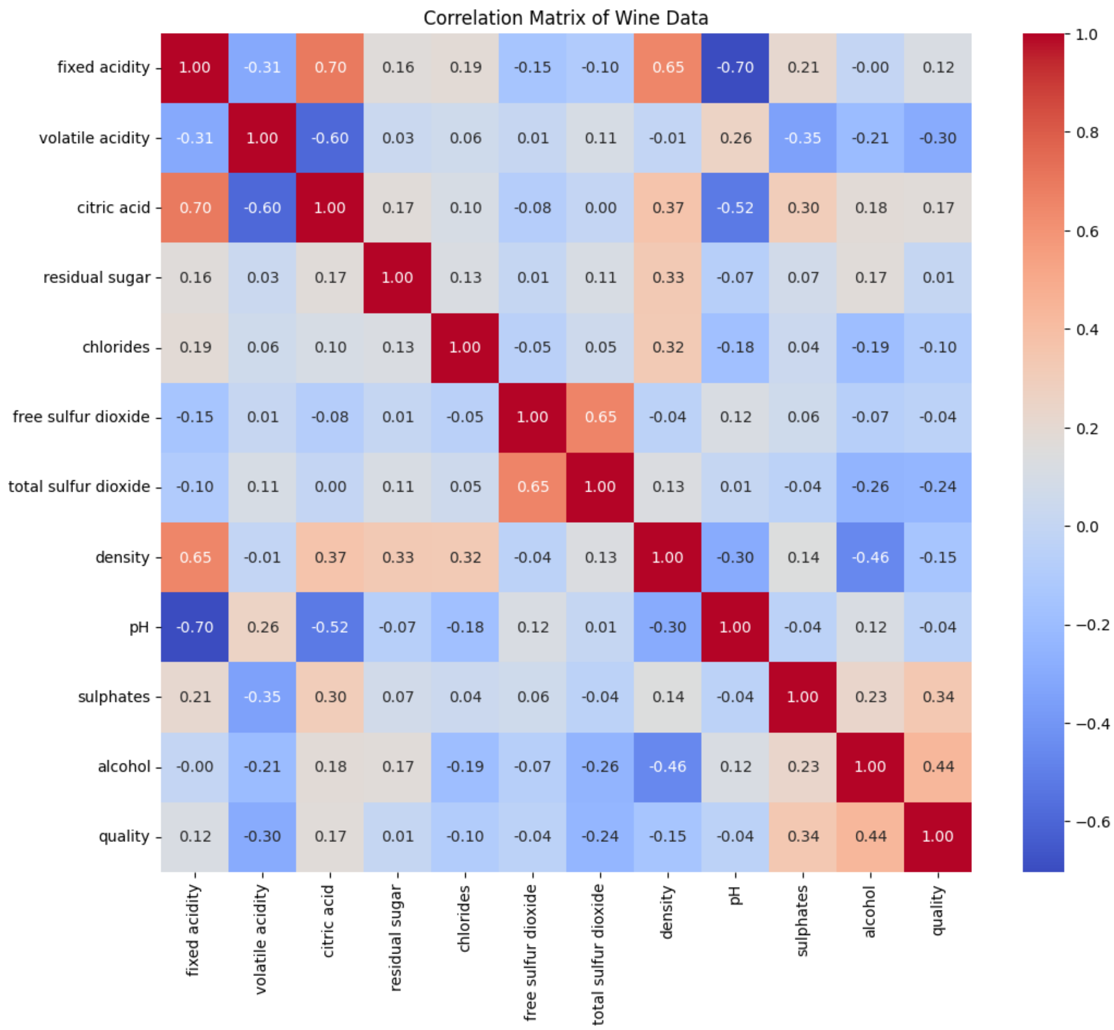


Looking at the boxplots at plain sight, there doesn't seem to be much of a change after applying outlier deletion. However, when closely-inspected, the newly created boxplots have different scales than the ones that were created before outlier deletion, meaning that some actual values were deleted from the dataset.

This becomes even more evident when observing variables like `citric acid` and `alcohol`.

Observing the correlation heatmap once again, with the goal of watching how these changes possibly affect the correlation between variables:

[ ]:
```
plt.figure(figsize=(12, 10))
correlation_matrix = wine_data.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of Wine Data')
plt.show()
```



There doesn't seem to be a strong difference between the original heatmap and this one. Meaning we can continue with our outliers being deleted.

4.9 Data Split

Before we continue to manipulate our data, it is necessary to split it between a training set and a testing set. This is due to data leakage, since we want to avoid mixing any information between the two sets.

We will use a 80% training, 20% testing proportion, with stratification on the `wine_quality` column to preserve the category proportion between the sets.

```
[ ]: from sklearn.model_selection import train_test_split

y = wine_data['quality']
X = wine_data.drop(['quality'], axis=1)

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

4.10 Data Imbalance

Given the fact that the amount of good quality wine is not the same as the amount of bad quality wine, our data is imbalanced. This imbalance may cause a negative effect on the models' performance, we will use upsampling to correct this.

First we need to know which category is the one that needs upsampling.

```
[ ]: y_train.value_counts()
```

```
[ ]: quality
1    620
0    524
Name: count, dtype: int64
```

Seeing that the bad quality wine class is the one that needs upsampling, we will apply random over sampling to achieve our goal:

```
[ ]: from imblearn.over_sampling import RandomOverSampler

# Initialize the RandomOverSampler
ros = RandomOverSampler(random_state=42)

# Perform random upsampling
X_train, y_train = ros.fit_resample(X_train, y_train)

# Check the distribution of the target variable
print(y_train.value_counts())
```

```
quality
1    620
0    620
Name: count, dtype: int64
```

As per the output, now our wine `quality` variables have the same count of observations.

Since we can't modify the `y_test` variable, we can just look at how these variables are distributed:

[ ]:
```
y_test.value_counts()
```

[ ]:
```
quality
1    156
0    131
Name: count, dtype: int64
```

Seeing that both classes are similarly distributed, with the good quality class having a bit more observations.

4.11 Data Scaling

Since neural networks have a non-linear nature, we will use normalization. On the other hand, standardization is better for linear models when a Gaussian distribution is present (which in our dataset it is, in most columns).

Normalizing for neural networks becomes key since this affects their convergence time towards a solution.

[ ]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
```

We'll only scale the feature values:

[ ]:
```
scaler = MinMaxScaler()
scaler.fit(X_train)

X_train_normalized = scaler.transform(X_train)
X_test_normalized = scaler.transform(X_test)

X_train_normalized = pd.DataFrame(X_train_normalized, columns=X_train.columns)
X_test_normalized = pd.DataFrame(X_test_normalized, columns=X_test.columns)
```

[ ]:
```
scaler = StandardScaler()
scaler.fit(X_train)

X_train_standardized = scaler.transform(X_train)
X_test_standardized = scaler.transform(X_test)

X_train_standardized = pd.DataFrame(X_train_standardized, columns=X_train.
 ↪columns)
X_test_standardized = pd.DataFrame(X_test_standardized, columns=X_test.columns)
```

[ ]:
```
X_train_normalized.describe()
```

[ ]:
```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1240.000000       1240.000000  1240.000000     1240.000000
```

```
mean       0.390712          0.438867     0.333218        0.219837
std        0.195248          0.180492     0.241914        0.157489
min        0.000000          0.000000     0.000000        0.000000
25%        0.247059          0.304348     0.113924        0.130841
50%        0.341176          0.434783     0.316456        0.186916
75%        0.485294          0.554348     0.531646        0.261682
max        1.000000          1.000000     1.000000        1.000000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  1240.000000          1240.000000           1240.000000  1240.000000
mean      0.236251             0.324102              0.291592     0.495563
std       0.111529             0.209650              0.221287     0.158933
min       0.000000             0.000000              0.000000     0.000000
25%       0.173913             0.136364              0.116279     0.388679
50%       0.222826             0.272727              0.240310     0.496226
75%       0.271739             0.454545              0.403101     0.594340
max       1.000000             1.000000              1.000000     1.000000

                pH    sulphates      alcohol
count  1240.000000  1240.000000  1240.000000
mean      0.503365     0.395606     0.386991
std       0.162623     0.163972     0.194954
min       0.000000     0.000000     0.000000
25%       0.390805     0.282051     0.215686
50%       0.505747     0.371795     0.333333
75%       0.609195     0.487179     0.509804
max       1.000000     1.000000     1.000000
```

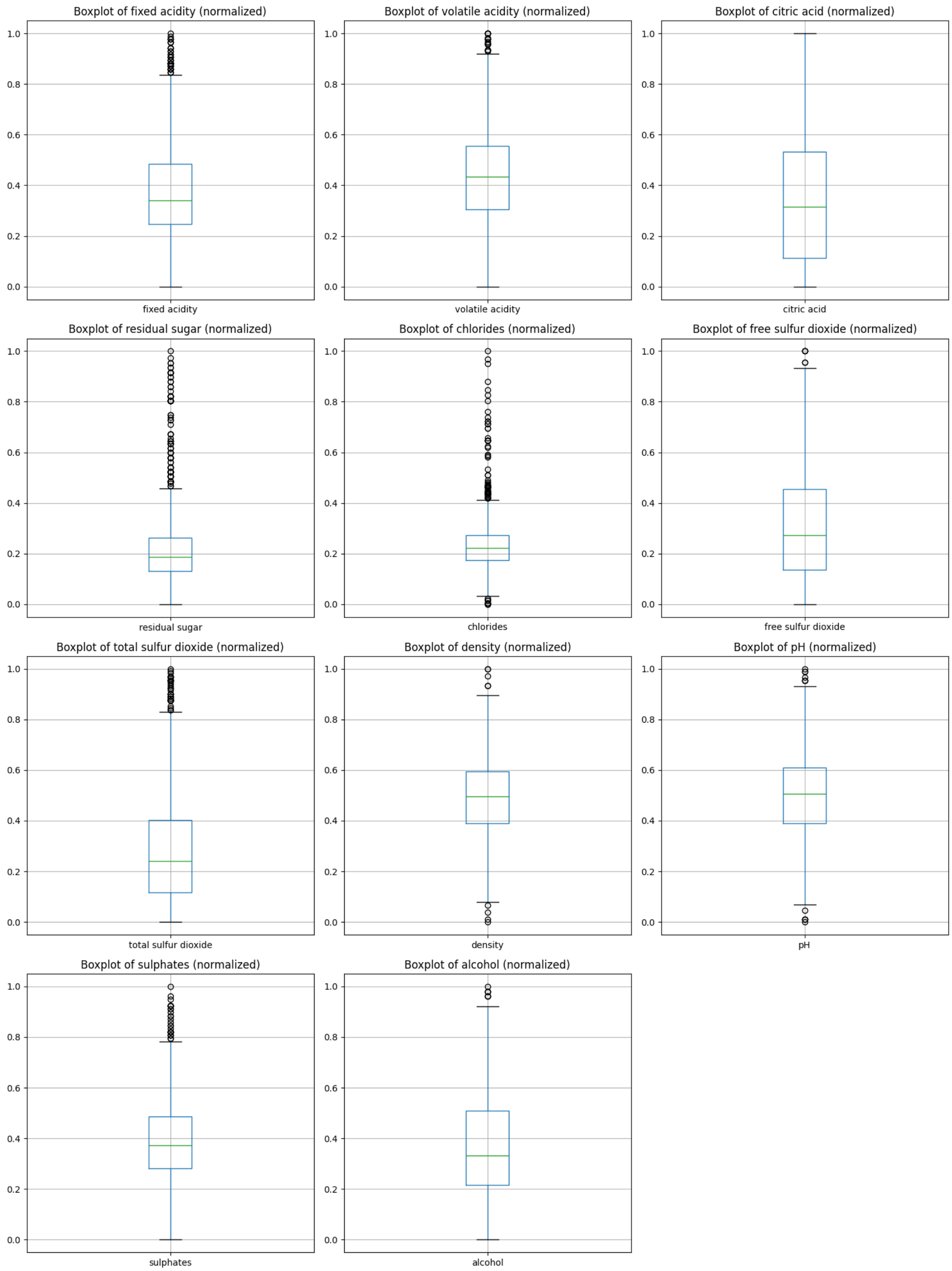
[ ]:
```
X_train_standardized.head()
```

[ ]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0      -0.856600         -0.685348     0.349454       -0.209123  -0.022920
1      -0.916880          0.881051    -1.377975       -0.565322  -0.071669
2       0.047592         -1.408302     0.715878       -0.802788  -0.705412
3      -0.675762          0.579820    -1.063897        0.147076  -0.169168
4       1.253181         -1.709532     1.029957       -0.684055  -0.315416

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0             0.730896             -0.266882 -1.100328  0.651033   0.871570
1             0.730896             -0.161746  0.152631  0.580325   0.793351
2            -0.787395             -0.827607 -0.429312  0.509616  -0.301709
3             0.188649              1.415295  0.419849  1.358121  -0.849239
4            -1.004293             -0.792562  0.502984 -0.480306   2.983471

    alcohol
0  0.730827
1 -0.677814
```

```
2  0.127124
3 -0.677814
4  0.428976
```

We can appreciate the change in scales with the boxplots from before.

```
[ ]: # Prepare the plot to load a boxplot for each factor
num_plots = len(num_columns)
num_cols = 3
num_rows = (num_plots // num_cols) + (num_plots % num_cols > 0)

fig, axes = plt.subplots(num_rows, num_cols, figsize=(num_cols * 5, num_rows *
 ↪5))
axes = axes.flatten()

# Create a boxplot for every factor
for i, column in enumerate(num_columns):
    pd.DataFrame(X_train_normalized, columns=num_columns).
 ↪boxplot(column=[column], ax=axes[i])
    axes[i].set_title(f'Boxplot of {column} (normalized)')

for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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


Boxplot of fixed acidity (normalized)
fixed acidity
Boxplot of volatile acidity (normalized)
volatile acidity
Boxplot of citric acid (normalized)
citric acid
Boxplot of residual sugar (normalized)
residual sugar
Boxplot of chlorides (normalized)
chlorides
Boxplot of free sulfur dioxide (normalized)
free sulfur dioxide
Boxplot of total sulfur dioxide (normalized)
total sulfur dioxide
Boxplot of density (normalized)
density
Boxplot of pH (normalized)
pH
Boxplot of sulphates (normalized)
sulphates
Boxplot of alcohol (normalized)
alcohol

5 Model Implementation

5.1 Metrics

To measure the performance of the models, we will mainly use the accuracy and f1 score values, but we will also have the ROC AUC, precision and recall metrics as supporting values.

The confusion matrices will also be printed.

```
[ ]: import time
import matplotlib.pyplot as plt
from sklearn.metrics import (
    confusion_matrix, classification_report,
    roc_curve, roc_auc_score, accuracy_score
)
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
```

We will also use this function to print the performance results for an individual model:

```
[ ]: def evaluate_best_model(model, X_train_set=X_train_standardized,␣
 ↪y_train_set=y_train, X_test_set=X_test_standardized, y_test_set=y_test):
    start_time = time.time()
    y_pred = model.predict(X_test_set)
    end_time = time.time()
    prediction_time = end_time - start_time

    # Evaluate accuracy on training and test sets
    train_accuracy = model.score(X_train_set, y_train_set)
    test_accuracy = model.score(X_test_set, y_test_set)

    # Print the accuracy
    print(f"Training accuracy: {train_accuracy:.4f}")
    print(f"Test accuracy: {test_accuracy:.4f}")

    # Print the confusion matrix
    print("Confusion Matrix:")
    print(confusion_matrix(y_test_set, y_pred))

    # Print the classification report
    print("Classification Report for the best model:")
    print(classification_report(y_test_set, y_pred))

    # Compute ROC AUC
    y_prob = model.predict_proba(X_test_set)[:, 1]  # Get probabilities for the␣
 ↪positive class
    roc_auc = roc_auc_score(y_test_set, y_prob)
    fpr, tpr, thresholds = roc_curve(y_test_set, y_prob)
```

```python
    print(f"ROC AUC Score: {roc_auc:.4f}")

    # Plot ROC Curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.4f})")
    plt.plot([0, 1], [0, 1], color="gray", linestyle="--")  # Diagonal line
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Receiver Operating Characteristic (ROC) Curve")
    plt.legend(loc="lower right")
    plt.grid()
    plt.show()

    # Print prediction time
    print(f"Prediction time: {prediction_time:.4f} seconds")

    return test_accuracy, prediction_time, roc_auc
```

And this to print out the elapsed training time:

[ ]:
```python
def print_elapsed_time(start_time, end_time):
    elapsed_time = end_time - start_time

    minutes, seconds = divmod(elapsed_time, 60)
    print(f"Total training time: {int(minutes):02d}:{int(seconds):02d} (mm:ss)")
```

5.2 Logistic Regression

We will start with logistic regression, the simplest model to implement.

Importing the LogisticRegression package.

[ ]:
```python
from sklearn.linear_model import LogisticRegression
```

Training the model:

[ ]:
```python
model_lr = LogisticRegression()

start_time = time.time()
model_lr.fit(X_train_normalized, y_train)
end_time = time.time()

print_elapsed_time(start_time, end_time)
```

```
Total training time: 00:00 (mm:ss)
```

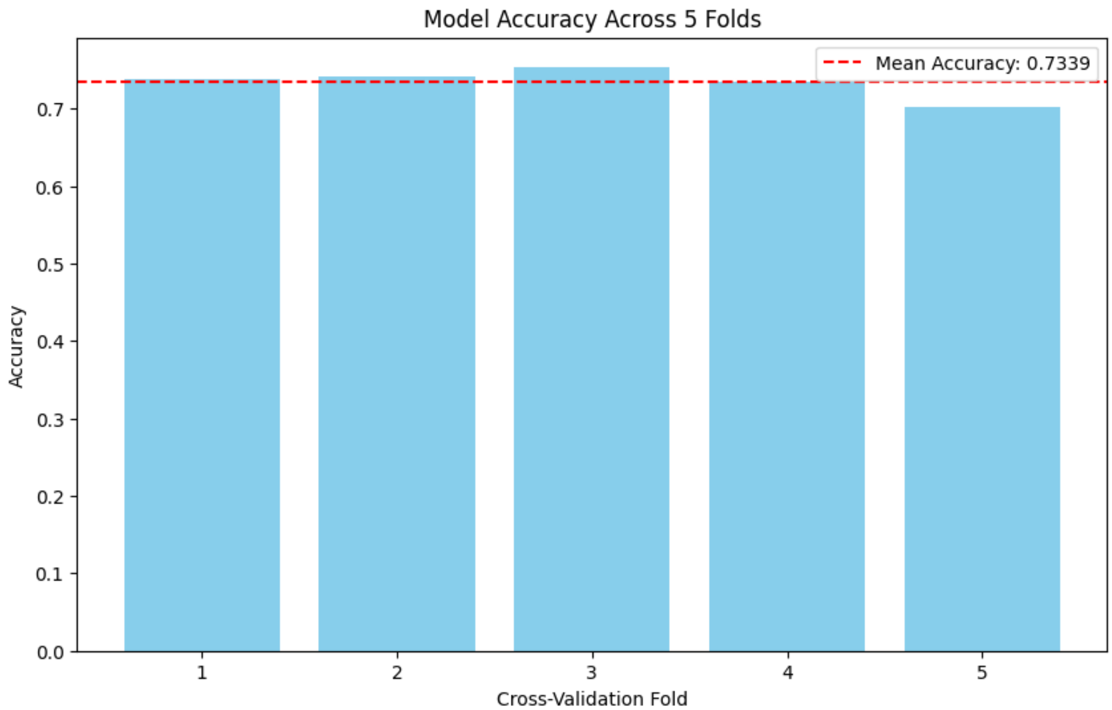
Using our trained model to predict upon the 5 train and test splits:

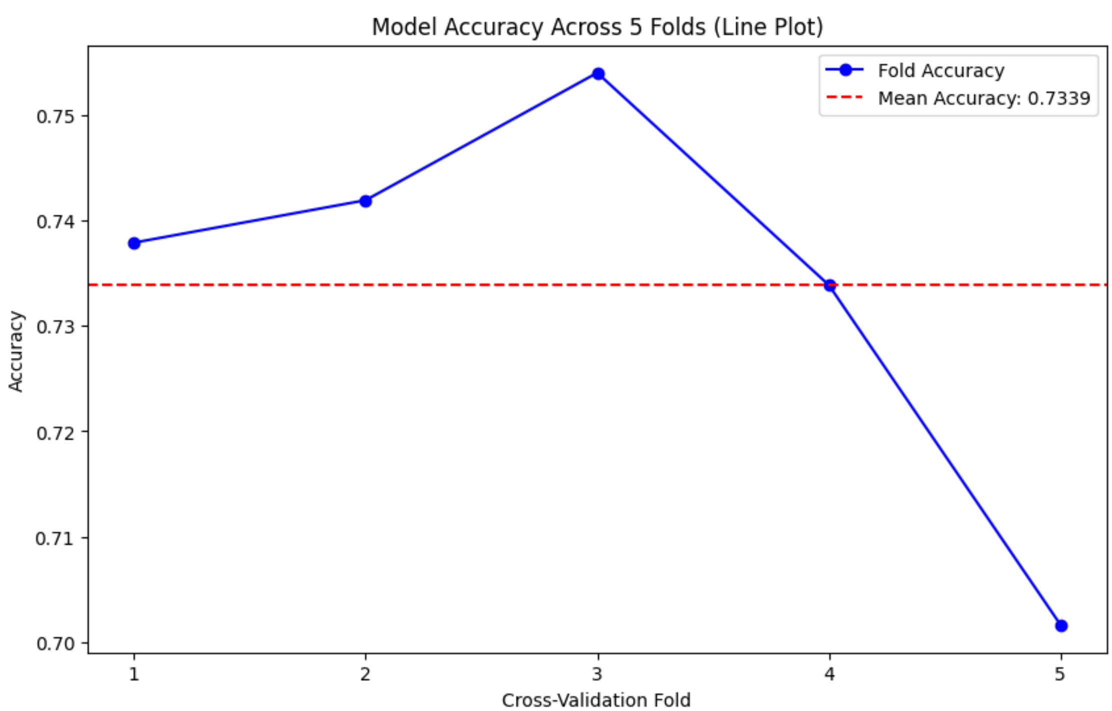
[ ]:
```python
# Ensure the data is in NumPy format
X_train_normalized = np.array(X_train_normalized)
```

```
y_train = np.array(y_train)

# Set up 5 splits with an 80/20 split
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
model_lr = LogisticRegression()

# Lists to store times, accuracy scores, and trained models
fit_times = []
accuracy_scores = []
models = []

# Loop through the splits
for fold, (train_idx, test_idx) in enumerate(kf.split(X_train_normalized,␣
↪y_train)):
    # Split the dataset
    X_train_fold, X_test_fold = X_train_normalized[train_idx],␣
↪X_train_normalized[test_idx]
    y_train_fold, y_test_fold = y_train[train_idx], y_train[test_idx]

    # Train the model and measure time
    start_time = time.time()
    model_lr.fit(X_train_fold, y_train_fold)
    end_time = time.time()

    # Store and print elapsed time
    fit_times.append(end_time - start_time)
    print_elapsed_time(start_time, end_time)

    # Predict on the test fold and calculate accuracy
    y_pred = model_lr.predict(X_test_fold)
    accuracy = accuracy_score(y_test_fold, y_pred)
    accuracy_scores.append(accuracy)
    models.append(model_lr)


# Summary of accuracy
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
print("\nSummary of accuracy per fold:")
for i, acc in enumerate(accuracy_scores, 1):
    print(f"Fold {i}: Accuracy = {acc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Standard Deviation of Accuracy: {std_accuracy:.4f}")

# Bar Plot for Visual Representation of Accuracy
plt.figure(figsize=(10, 6))
plt.bar(range(1, 6), accuracy_scores, color='skyblue')
```

```
plt.axhline(mean_accuracy, color='r', linestyle='--', label=f'Mean Accuracy: {mean_accuracy:.4f}')
plt.xlabel('Cross-Validation Fold')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Across 5 Folds')
plt.xticks(range(1, 6))
plt.legend()
plt.show()

# Line Plot for Accuracy Across Folds
plt.figure(figsize=(10, 6))
plt.plot(range(1, 6), accuracy_scores, marker='o', color='b', linestyle='-', label='Fold Accuracy')
plt.axhline(mean_accuracy, color='r', linestyle='--', label=f'Mean Accuracy: {mean_accuracy:.4f}')
plt.xlabel('Cross-Validation Fold')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Across 5 Folds (Line Plot)')
plt.xticks(range(1, 6))
plt.legend()
plt.show()

# Get the best performing model
best_fold = np.argmax(accuracy_scores)
best_model = models[best_fold]
print(f"\nThe best model was from fold {best_fold + 1} with accuracy: {accuracy_scores[best_fold]:.4f}")
```

```
Total training time: 00:00 (mm:ss)
Total training time: 00:00 (mm:ss)
Total training time: 00:00 (mm:ss)
Total training time: 00:00 (mm:ss)
Total training time: 00:00 (mm:ss)

Summary of accuracy per fold:
Fold 1: Accuracy = 0.7379
Fold 2: Accuracy = 0.7419
Fold 3: Accuracy = 0.7540
Fold 4: Accuracy = 0.7339
Fold 5: Accuracy = 0.7016
Mean Accuracy: 0.7339
Standard Deviation of Accuracy: 0.0175
```


Model Accuracy Across 5 Folds
Mean Accuracy: 0.7339
Accuracy
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
3
4
5
Cross-Validation Fold


Model Accuracy Across 5 Folds (Line Plot)
Fold Accuracy
Mean Accuracy: 0.7339
Accuracy
0.70
0.71
0.72
0.73
0.74
0.75
1
2
3
4
5
Cross-Validation Fold

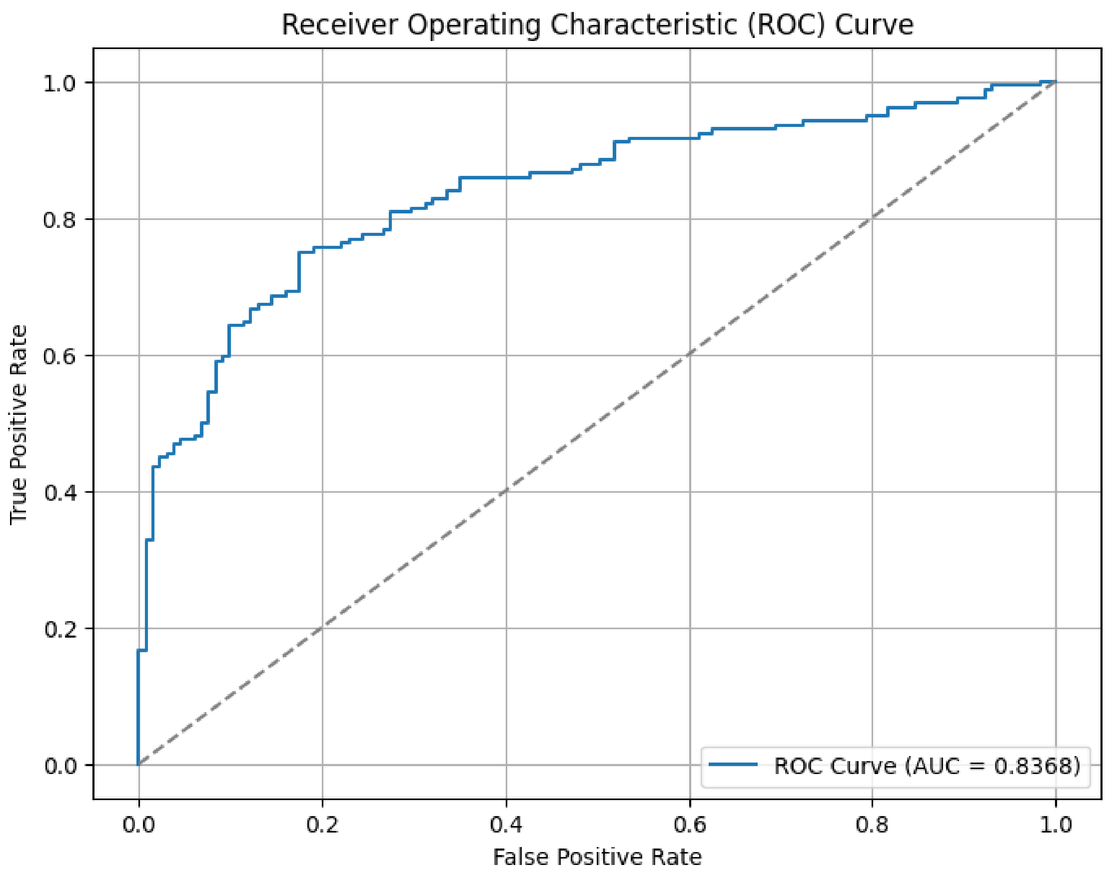
```
The best model was from fold 3 with accuracy: 0.7540
```

Once the model is trained, we will proceed with evaluating our model with the testing data:

```
[ ]: lr_accuracy, lr_prediction_time, lr_roc_auc = evaluate_best_model(best_model,␣
     ↪X_test_set=X_test_normalized)
```

```
Training accuracy: 0.7419
Test accuracy: 0.7596
Confusion Matrix:
[[110  21]
 [ 48 108]]
Classification Report for the best model:
              precision    recall  f1-score   support

           0       0.70      0.84      0.76       131
           1       0.84      0.69      0.76       156

    accuracy                           0.76       287
   macro avg       0.77      0.77      0.76       287
weighted avg       0.77      0.76      0.76       287

ROC AUC Score: 0.8368
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but LogisticRegression was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but LogisticRegression was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but LogisticRegression was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but LogisticRegression was fitted without feature names
  warnings.warn(
```




```
Prediction time: 0.0042 seconds
```

The metrics are as follows:

Metric	Value
Training Accuracy	0.7419
Test Accuracy	0.7596
Precision	0.77
Recall	0.76
F1-Score	0.76
ROC AUC	0.8368

	Predicted 0	Predicted 1
Actual 0	110	21
Actual 1	48	108

Comparing the training and testing accuracy, there is no significant difference between to say there is overfitting. Analyzing the f1-score (and the precision and recall), the model is identifies the wine quality fairly good. Although, there is a lot of room for improvement, as we can see in the confusion matrix.

5.3 KNN

We will continue with a bit more complex model, which is KNN. This model will ensure better adaptability to non linear data, since the classification depends on the sample's neighbors.

We will use a an amount of neighbors that ranges from 5 to 50 in increments of 5. In order to know which hyperparameter is the best, we will use the Grid Search library, that will compute all combinatios and pick the best one.

[ ]:
```python
from sklearn.neighbors import KNeighborsClassifier
```

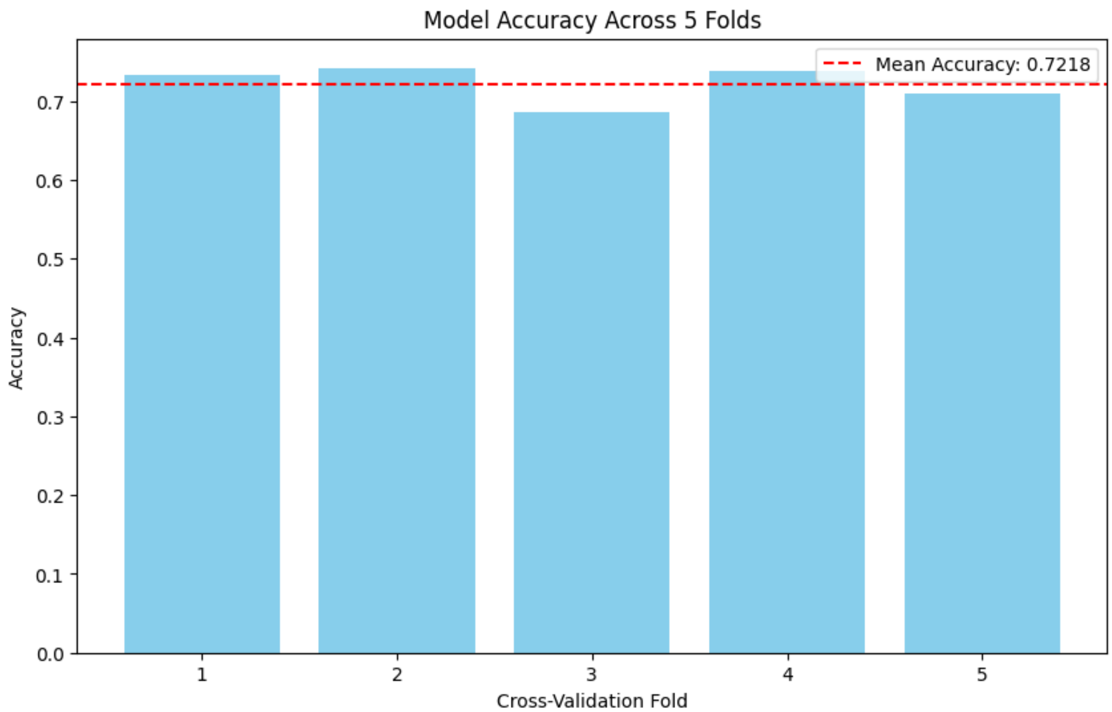
Training the model with the 5 folds:

[ ]:
```python
# Ensure the data is in NumPy format
X_train_standardized = np.array(X_train_standardized)
y_train = np.array(y_train)

# Set up 5 splits with an 80/20 split
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
model_knn = KNeighborsClassifier()

# Define the parameter grid for GridSearchCV
param_grid = {
    'n_neighbors': range(6, 50, 5),
}

# Lists to store times, accuracy scores, and best models
fit_times = []
accuracy_scores = []
best_models = []

# Loop through the splits
for fold, (train_idx, test_idx) in enumerate(kf.split(X_train_standardized,␣
 ↪y_train)):
    print(f"--- Executing fold {fold + 1} ---")

    # Split the dataset
    X_train_fold, X_test_fold = X_train_standardized[train_idx],␣
 ↪X_train_standardized[test_idx]
    y_train_fold, y_test_fold = y_train[train_idx], y_train[test_idx]

    # Set up GridSearchCV for KNN
    grid_search = GridSearchCV(model_knn, param_grid, cv=3, scoring='accuracy',␣
 ↪n_jobs=-1)

    # Train the model using GridSearchCV and measure time
```

```python
    start_time = time.time()
    grid_search.fit(X_train_fold, y_train_fold)
    end_time = time.time()

    # Store and print elapsed time
    fit_times.append(end_time - start_time)
    print_elapsed_time(start_time, end_time)

    # Get the best model from the current fold
    best_model = grid_search.best_estimator_
    best_models.append(best_model)

    # Predict on the test fold using the best model
    y_pred = best_model.predict(X_test_fold)
    accuracy = accuracy_score(y_test_fold, y_pred)
    accuracy_scores.append(accuracy)
    print(f"Best parameters for fold {fold + 1}: {grid_search.best_params_}")
    print(f"Accuracy for fold {fold + 1}: {accuracy:.4f}")

# Summary of accuracy
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Standard Deviation of Accuracy: {std_accuracy:.4f}")

# Bar Plot for Visual Representation of Accuracy
plt.figure(figsize=(10, 6))
plt.bar(range(1, 6), accuracy_scores, color='skyblue')
plt.axhline(mean_accuracy, color='r', linestyle='--', label=f'Mean Accuracy:␣
 ↪{mean_accuracy:.4f}')
plt.xlabel('Cross-Validation Fold')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Across 5 Folds')
plt.xticks(range(1, 6))
plt.legend()
plt.show()

# Get the best performing model overall
best_fold = np.argmax(accuracy_scores)
best_model = best_models[best_fold]
print(f"\nThe best model was from fold {best_fold + 1} with accuracy:␣
 ↪{accuracy_scores[best_fold]:.4f}")
```

```
--- Executing fold 1 ---
Total training time: 00:03 (mm:ss)
Best parameters for fold 1: {'n_neighbors': 16}
Accuracy for fold 1: 0.7339
--- Executing fold 2 ---
```

```
Total training time: 00:01 (mm:ss)
Best parameters for fold 2: {'n_neighbors': 46}
Accuracy for fold 2: 0.7419
--- Executing fold 3 ---
Total training time: 00:01 (mm:ss)
Best parameters for fold 3: {'n_neighbors': 11}
Accuracy for fold 3: 0.6855
--- Executing fold 4 ---
Total training time: 00:00 (mm:ss)
Best parameters for fold 4: {'n_neighbors': 46}
Accuracy for fold 4: 0.7379
--- Executing fold 5 ---
Total training time: 00:00 (mm:ss)
Best parameters for fold 5: {'n_neighbors': 11}
Accuracy for fold 5: 0.7097
Mean Accuracy: 0.7218
Standard Deviation of Accuracy: 0.0213
```




```
The best model was from fold 2 with accuracy: 0.7419
```

Now that the model is trained, we can see which amount of neighbors has the bes accuracy.

[ ]:
```python
best_model = grid_search.best_estimator_
print("Mejores hiperparámetros:", grid_search.best_params_)
```

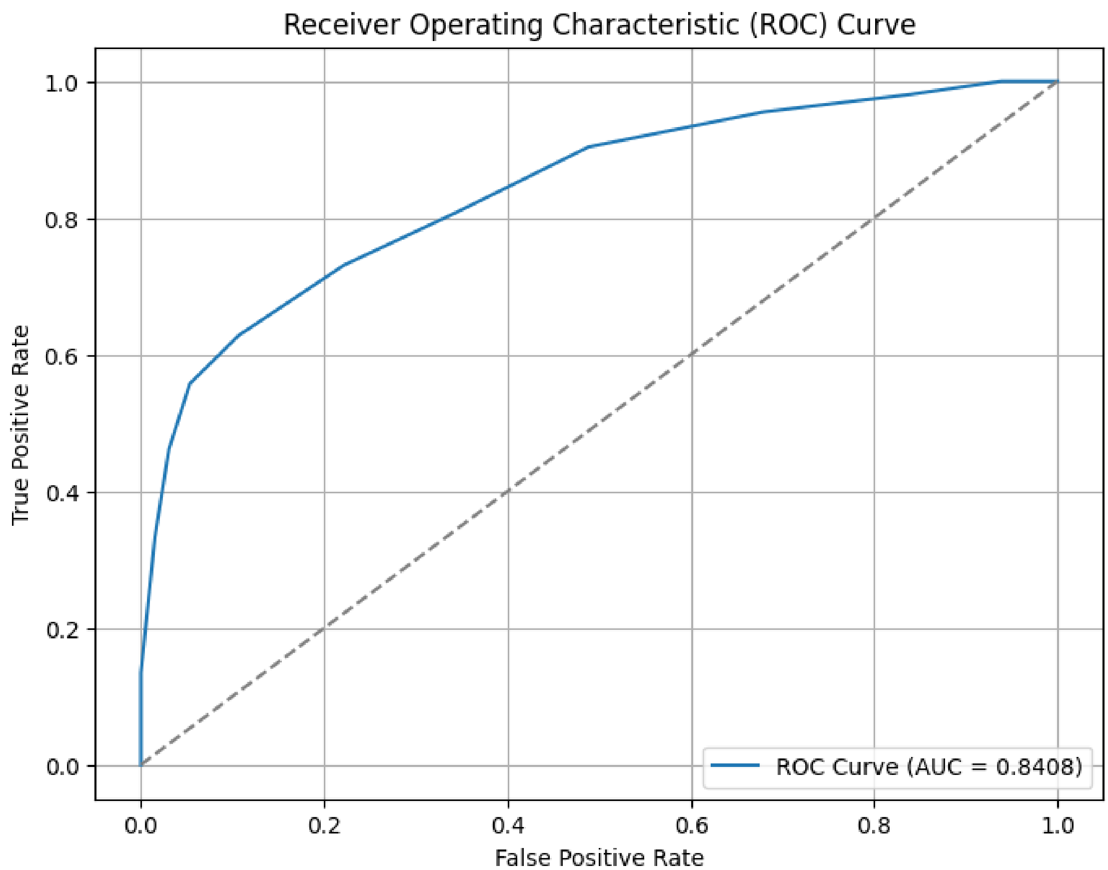
```
Mejores hiperparámetros: {'n_neighbors': 11}
```

It appears that 10 neighbors had the best training accuracy. Now we will proceed to test the model.

```
[ ]: knn_accuracy, knn_prediction_time, knn_roc_auc =␣
      ↪evaluate_best_model(best_model, X_test_set=X_test_standardized)
```

```
Training accuracy: 0.7613
Test accuracy: 0.7526
Confusion Matrix:
[[102  29]
 [ 42 114]]
Classification Report for the best model:
              precision    recall  f1-score   support

           0       0.71      0.78      0.74       131
           1       0.80      0.73      0.76       156

    accuracy                           0.75       287
   macro avg       0.75      0.75      0.75       287
weighted avg       0.76      0.75      0.75       287

ROC AUC Score: 0.8408
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but KNeighborsClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but KNeighborsClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but KNeighborsClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but KNeighborsClassifier was fitted without feature names
  warnings.warn(
```




```
Prediction time: 0.0280 seconds
```

Looking at the metrics:

Metric	Value
Training Accuracy	0.7613
Test Accuracy	0.7526
Precision	0.76
Recall	0.75
F1-Score	0.75
ROC AUC	0.8408

	Predicted 0	Predicted 1
Actual 0	102	29
Actual 1	42	114

Since the Training and Test accuracy are very similar, the model doesn't seem to be overfitted. It also appears to be classifying the wines with lower precision, recall and F1-score values. This model performs worse than Logistic Regression.

5.4 Decision Tree

Now we will implement a higher complexity model, a decision tree. This model will classify the wine quality by constructing a binary tree with the training data.

[ ]:
```
from sklearn.tree import DecisionTreeClassifier
```

Training with the 5 splits:

[ ]:
```
# Set up 5 splits with StratifiedKFold
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
model_dt = DecisionTreeClassifier(random_state=42)

# Define the parameter grid for GridSearchCV
param_grid = {
    'max_depth': [20, 30, 40],
    'min_samples_split': [20, 30, 40],
    'min_samples_leaf': [20, 32, 64],
    'max_features': ['sqrt', 'log2'],
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random']
}

# Lists to store times, accuracy scores, and best models
fit_times = []
accuracy_scores = []
best_models = []

# Loop through the splits
for fold, (train_idx, test_idx) in enumerate(kf.split(X_train_standardized,␣
 ↪y_train)):
    print(f"--- Executing fold {fold + 1} ---")

    # Split the dataset
    X_train_fold, X_test_fold = X_train_standardized[train_idx],␣
 ↪X_train_standardized[test_idx]
    y_train_fold, y_test_fold = y_train[train_idx], y_train[test_idx]

    # Set up GridSearchCV for Decision Tree
    grid_search = GridSearchCV(model_dt, param_grid, cv=3, scoring='accuracy',␣
 ↪n_jobs=-1)

    # Train the model using GridSearchCV and measure time
    start_time = time.time()
    grid_search.fit(X_train_fold, y_train_fold)
    end_time = time.time()
```

```
    # Store and print elapsed time
    fit_times.append(end_time - start_time)
    print_elapsed_time(start_time, end_time)

    # Get the best model from the current fold
    best_model = grid_search.best_estimator_
    best_models.append(best_model)

    # Predict on the test fold using the best model
    y_pred = best_model.predict(X_test_fold)
    accuracy = accuracy_score(y_test_fold, y_pred)
    accuracy_scores.append(accuracy)
    print(f"Best parameters for fold {fold + 1}: {grid_search.best_params_}")
    print(f"Accuracy for fold {fold + 1}: {accuracy:.4f}")

# Summary of accuracy
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Standard Deviation of Accuracy: {std_accuracy:.4f}")

# Bar Plot for Visual Representation of Accuracy
plt.figure(figsize=(10, 6))
plt.bar(range(1, 6), accuracy_scores, color='skyblue')
plt.axhline(mean_accuracy, color='r', linestyle='--', label=f'Mean Accuracy:␣
 ↪{mean_accuracy:.4f}')
plt.xlabel('Cross-Validation Fold')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Across 5 Folds')
plt.xticks(range(1, 6))
plt.legend()
plt.show()

# Get the best performing model overall
best_fold = np.argmax(accuracy_scores)
best_model = best_models[best_fold]
print(f"\nThe best model was from fold {best_fold + 1} with accuracy:␣
 ↪{accuracy_scores[best_fold]:.4f}")
```

```
--- Executing fold 1 ---

/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:
invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,

Total training time: 00:01 (mm:ss)
Best parameters for fold 1: {'criterion': 'gini', 'max_depth': 20,
'max_features': 'sqrt', 'min_samples_leaf': 20, 'min_samples_split': 20,
```

```
'splitter': 'best'}
Accuracy for fold 1: 0.7460
--- Executing fold 2 ---
```

```
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:
invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

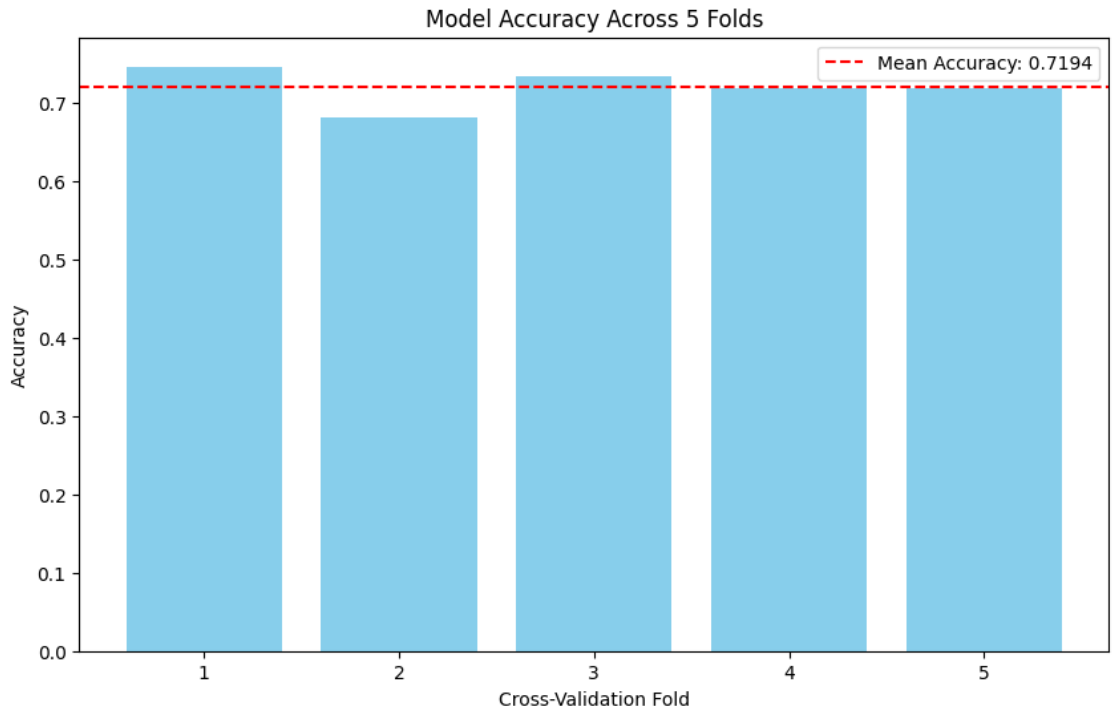
```
Total training time: 00:01 (mm:ss)
Best parameters for fold 2: {'criterion': 'gini', 'max_depth': 20,
'max_features': 'sqrt', 'min_samples_leaf': 64, 'min_samples_split': 20,
'splitter': 'best'}
Accuracy for fold 2: 0.6815
--- Executing fold 3 ---
```

```
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:
invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

```
Total training time: 00:01 (mm:ss)
Best parameters for fold 3: {'criterion': 'entropy', 'max_depth': 20,
'max_features': 'sqrt', 'min_samples_leaf': 32, 'min_samples_split': 20,
'splitter': 'best'}
Accuracy for fold 3: 0.7339
--- Executing fold 4 ---
Total training time: 00:01 (mm:ss)
Best parameters for fold 4: {'criterion': 'gini', 'max_depth': 20,
'max_features': 'sqrt', 'min_samples_leaf': 64, 'min_samples_split': 20,
'splitter': 'best'}
Accuracy for fold 4: 0.7177
--- Executing fold 5 ---
```

```
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:
invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

```
Total training time: 00:01 (mm:ss)
Best parameters for fold 5: {'criterion': 'gini', 'max_depth': 20,
'max_features': 'sqrt', 'min_samples_leaf': 20, 'min_samples_split': 20,
'splitter': 'best'}
Accuracy for fold 5: 0.7177
Mean Accuracy: 0.7194
Standard Deviation of Accuracy: 0.0217
```




```
The best model was from fold 1 with accuracy: 0.7460
```

We used different values for the hyper parameters `max_depth`, `min_samples_split`, `min_samples_leaf` and `max_features`. But in order to understand the meaning of the values, we first need to undersand what they do. All of these attributes are hyperparameters useful for prepruning, this is, attributes used to limit the growth of the tree in order to avoid overfitting.

`max_depth` means the maximum depth that the tree can have, it means how large can the tree be. `min_samples_split` refers to the amount of samples necessary to split a node. `min_samples_leaf` this is minimum amount to split a node into a leaf. And finally `max_features` which correspond to the amount of features that are considered when a node needs to be splitted.

Let's check which are the best values for this specific problem.

```
[ ]: best_model = grid_search.best_estimator_
     print("Mejores hiperparámetros:", grid_search.best_params_)
```

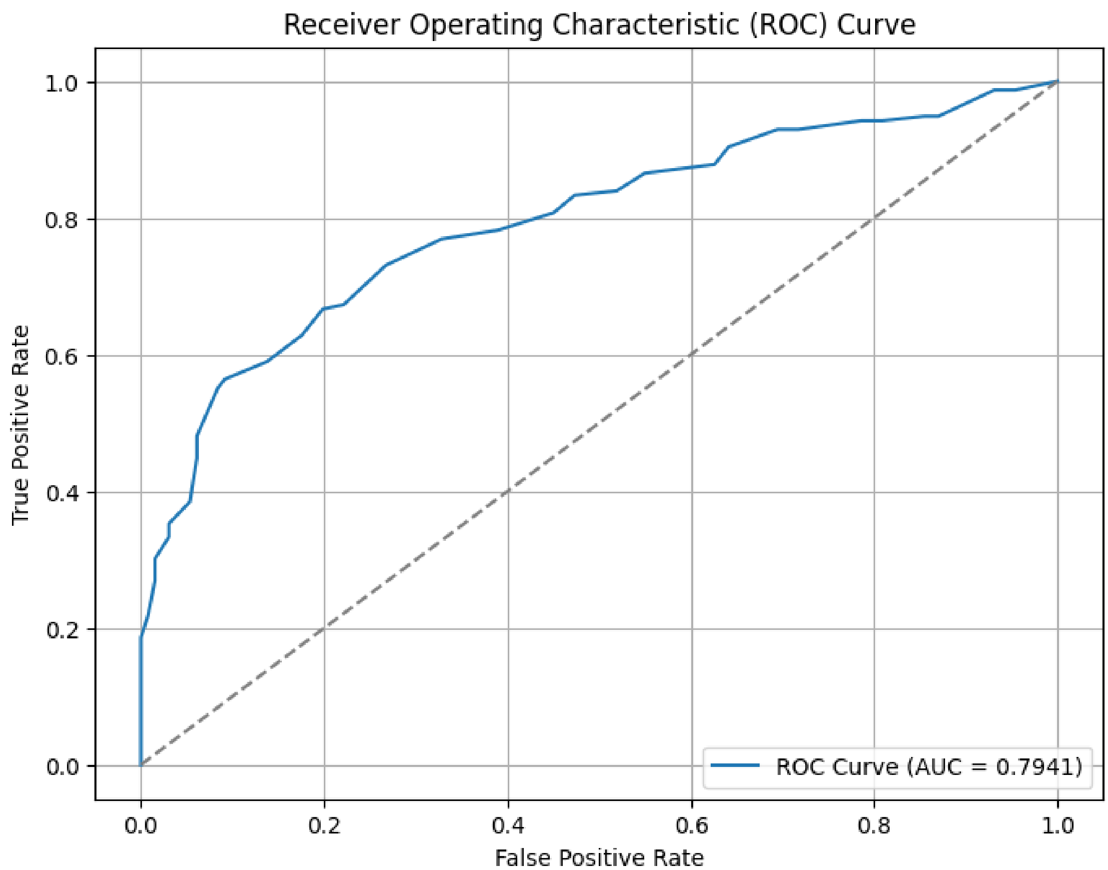
```
Mejores hiperparámetros: {'criterion': 'gini', 'max_depth': 20, 'max_features':
'sqrt', 'min_samples_leaf': 20, 'min_samples_split': 20, 'splitter': 'best'}
```

The best parameters are: * 'max_depth' = 10: This means that the decision tree will only be 10 levels deep. * 'max_features' = None: There is no limit to the amount of features considered to split a node, this is, all features will be used. * 'min_samples_leaf' = 64: Every leaf will have at least 64 samples, each. * 'min_samples_split' = 10: Every node will need at least 10 samples in order to be splitted, otherwise it will stay as a leaf.

[ ]:
```python
dt_accuracy, dt_prediction_time, dt_roc_auc = evaluate_best_model(best_model)
```

```
Training accuracy: 0.7685
Test accuracy: 0.7213
Confusion Matrix:
[[107  24]
 [ 56 100]]
Classification Report for the best model:
              precision    recall  f1-score   support

           0       0.66      0.82      0.73       131
           1       0.81      0.64      0.71       156

    accuracy                           0.72       287
   macro avg       0.73      0.73      0.72       287
weighted avg       0.74      0.72      0.72       287

ROC AUC Score: 0.7941
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but DecisionTreeClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but DecisionTreeClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but DecisionTreeClassifier was fitted without feature names
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:486: UserWarning: X has
feature names, but DecisionTreeClassifier was fitted without feature names
  warnings.warn(
```




```
Prediction time: 0.0065 seconds
```

Showing the metrics:

Metric	Value
Training Accuracy	0.7685
Test Accuracy	0.7213
Precision	0.74
Recall	0.72
F1-Score	0.72
ROC AUC Score	0.7941

	Predicted 0	Predicted 1
Actual 0	107	24
Actual 1	56	100

Following the trending of the last models, the decision tree doesn't seem to be overfitting to the training data. With this being said, the decision tree model is outperformed by KNN, making this the worst model so far.

5.5 MLP

Now we jump to a very complex model: a Multiple Layer Perceptron model. The Neural Networks use a set of "neurons" positioned in many layers and interconected with one another.

We will use keras from TensorFlow to implement the neural network. First we will import all the necessary libraries.

```
[ ]: import tensorflow as tf
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense, Input, Dropout
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras.layers import BatchNormalization
```

Now, lets create a simple architecture, just to understand its parts.

```
[ ]: def create_mlp_model():
       mlp_model = Sequential(
           [
               Input(shape=(X_train.shape[1],)),   # Input layer, has as many␣
       ↪perceptrons as there are features
               Dense(32, activation='relu'),        # Hidden layer 1
               Dropout(0.35),
               Dense(32, activation='relu'),        # Hidden layer 2
               Dropout(0.35),
               Dense(32, activation='relu'),        # Hidden layer 3
               Dropout(0.3),
               Dense(32, activation='leaky_relu'),        # Hidden layer 4
               Dropout(0.3),
               Dense(32, activation='leaky_relu'),        # Hidden layer 5
               Dropout(0.3),
               Dense(16, activation='leaky_relu'),         # Hidden layer 6
               Dropout(0.2),
               Dense(1, activation='sigmoid')        # Output layer
           ]
       )
       mlp_model.compile(optimizer='adam', loss='binary_crossentropy',␣
       ↪metrics=['accuracy'])
       return mlp_model
```

Here we are using 5 hidden layers of 32 and a hidden layer of 16 perceptrons plus the output layer. The output layer consists of a single perceptron. The hidden layers use the activation function ReLU (and even some others use leaky ReLU), while the output layer uses the Sigmoid activation function. This way we are able to make a binary classification, since the Sigmoid function outputs a probability that can be rounded to 1 or 0.

Showing a summary of the model:

[ ]:
```
mlp_model.summary()
```

```
Model: "sequential"


 Layer (type)                         Output Shape                   ␣
↪Param #

 dense (Dense)                        (None, 32)                      ␣
↪384

 dropout (Dropout)                    (None, 32)                      ␣
↪  0

 dense_1 (Dense)                      (None, 32)                    ␣
↪1,056

 dropout_1 (Dropout)                  (None, 32)                      ␣
↪  0

 dense_2 (Dense)                      (None, 32)                    ␣
↪1,056

 dropout_2 (Dropout)                  (None, 32)                      ␣
↪  0

 dense_3 (Dense)                      (None, 32)                    ␣
↪1,056

 dropout_3 (Dropout)                  (None, 32)                      ␣
↪  0

 dense_4 (Dense)                      (None, 32)                    ␣
↪1,056

 dropout_4 (Dropout)                  (None, 32)                      ␣
↪  0

 dense_5 (Dense)                      (None, 16)                      ␣
↪528

 dropout_5 (Dropout)                  (None, 16)                      ␣
↪  0

 dense_6 (Dense)                      (None, 1)                       ␣
↪ 17
```

```
Total params: 5,153 (20.13 KB)
```

```
Trainable params: 5,153 (20.13 KB)
```

```
Non-trainable params: 0 (0.00 B)
```

Before the training of the model, we need to compile it. Here we will choose the loss function, that will determine how far the predictions are from the real data and change the weights of each perceptron through back propagation.

We also need to choose an optimizer, an algorithm that will help us know how to advance towards the best and most optimal configuration. We can also add other metrics like accuracy to analyze the model's behavior through the learning process.

In this case we will use binary cross-entropy as the loss function, because this is a binary classification problem, and this is the function that is mostly uses for these cases. We also use Adam as an since it handles the variations in learning rates throughout the training. The learning rate measures how big the steps are made towards achieving the lowest loss. Finally we will add the accuracy to the compilation, since is our main evaluation metric.

[ ]:
```
mlp_model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

One last step before training the model. We need to divide (once again) our data into the training set and the validation set. Since we will train the model through many epochs, we need to have a validation set that will be used to measure the generalization ability of the model.

[ ]:
```
X_train_mlp, X_val, y_train_mlp, y_val = train_test_split(X_train_normalized,␣
 ↪y_train, test_size=0.2, random_state=42)
```

Having done this, we can train the model. To do this we must choose the epoch count and the batch size. The batch size refers to the amount of data we give the model in one forward pass. On the other hand, the epoch refers to the amount of times we give all of these batches to the model (giving it all of the training set per epoch).

Having a big batch size value means that the model will calculate the loss more accurately, changing the model weights accordingly, and thus learning faster. This comes at a higher computational cost.

Having a bigger epoch count means giving the model more chances to improve, but this runs the risk of wasting a lot time training with no significant improvement. After every epoch, the loss function and the accuracy will be calculated for the validation set too.

After testing, we've decided to choose an epoch count of 100 and a batch size of 32.

Now the actual model training will be done:

[ ]:
```

```

[ ]:
```
from sklearn.model_selection import KFold
import numpy as np
import time

# Number of folds
n_folds = 5

# Initialize KFold
kf = KFold(n_splits=n_folds, shuffle=True, random_state=42)

# Store validation metrics and best models
validation_scores = []
best_model = None
best_score = -np.inf

# Perform cross-validation
fold = 1
for train_index, val_index in kf.split(X_train_normalized):
    print(f"Fold {fold}")

    # Split the data
    X_train_fold, X_val_fold = X_train_normalized[train_index],␣
↪X_train_normalized[val_index]
    y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

    # Initialize the MLP model (reinitialize for each fold)
    mlp_model = create_mlp_model()  # Replace with your model creation function

    # Train the model
    start_time = time.time()
    history = mlp_model.fit(
        X_train_fold, y_train_fold,
        epochs=100,
        batch_size=32,
        validation_data=(X_val_fold, y_val_fold),
        verbose=1
    )
    end_time = time.time()

    # Print elapsed time
    print_elapsed_time(start_time, end_time)

    # Evaluate the model on the validation set
    val_score = mlp_model.evaluate(X_val_fold, y_val_fold, verbose=0)
    print(f"Validation Score for Fold {fold}: {val_score}")
    validation_scores.append(val_score)
```

```
    # Evaluate the model on the validation set
    val_score = mlp_model.evaluate(X_val_fold, y_val_fold, verbose=0)

    # Extract the metric
    val_metric = val_score[1]  # Index 1 corresponds to the first metric in the␣
↪'metrics' list

    # Save the best model
    if val_metric > best_score:
        best_model = mlp_model
        best_score = val_metric


    fold += 1

# Report the results
print(f"Validation Scores across folds: {validation_scores}")
print(f"Mean Validation Score: {np.mean(validation_scores):.4f}")
print(f"Best Validation Score: {best_score:.4f}")

# Use the best model to make predictions
final_predictions = best_model.predict(X_test_normalized)
```

```
Fold 1
Epoch 1/100
31/31              5s 11ms/step -
accuracy: 0.5115 - loss: 0.6902 - val_accuracy: 0.5202 - val_loss: 0.6910
Epoch 2/100
31/31              0s 4ms/step -
accuracy: 0.5013 - loss: 0.6961 - val_accuracy: 0.4798 - val_loss: 0.6915
Epoch 3/100
31/31              0s 4ms/step -
accuracy: 0.5011 - loss: 0.6925 - val_accuracy: 0.5524 - val_loss: 0.6900
Epoch 4/100
31/31              0s 5ms/step -
accuracy: 0.5060 - loss: 0.6929 - val_accuracy: 0.6895 - val_loss: 0.6859
Epoch 5/100
31/31              0s 4ms/step -
accuracy: 0.5294 - loss: 0.6882 - val_accuracy: 0.7339 - val_loss: 0.6730
Epoch 6/100
31/31              0s 5ms/step -
accuracy: 0.6113 - loss: 0.6743 - val_accuracy: 0.7460 - val_loss: 0.6295
Epoch 7/100
31/31              0s 4ms/step -
accuracy: 0.5959 - loss: 0.6569 - val_accuracy: 0.7621 - val_loss: 0.5820
Epoch 8/100
31/31              0s 4ms/step -
accuracy: 0.6466 - loss: 0.6374 - val_accuracy: 0.7702 - val_loss: 0.5561
```

```
Epoch 9/100
31/31            0s 5ms/step -
accuracy: 0.6679 - loss: 0.6259 - val_accuracy: 0.7782 - val_loss: 0.5523
Epoch 10/100
31/31            0s 4ms/step -
accuracy: 0.6774 - loss: 0.6005 - val_accuracy: 0.8065 - val_loss: 0.5097
Epoch 11/100
31/31            0s 5ms/step -
accuracy: 0.6486 - loss: 0.6310 - val_accuracy: 0.7823 - val_loss: 0.5364
Epoch 12/100
31/31            0s 4ms/step -
accuracy: 0.6808 - loss: 0.5973 - val_accuracy: 0.7903 - val_loss: 0.5014
Epoch 13/100
31/31            0s 5ms/step -
accuracy: 0.6894 - loss: 0.5893 - val_accuracy: 0.8065 - val_loss: 0.4962
Epoch 14/100
31/31            0s 4ms/step -
accuracy: 0.7159 - loss: 0.5720 - val_accuracy: 0.7944 - val_loss: 0.5001
Epoch 15/100
31/31            0s 5ms/step -
accuracy: 0.6942 - loss: 0.5749 - val_accuracy: 0.8024 - val_loss: 0.5037
Epoch 16/100
31/31            0s 4ms/step -
accuracy: 0.7141 - loss: 0.5815 - val_accuracy: 0.7944 - val_loss: 0.4978
Epoch 17/100
31/31            0s 5ms/step -
accuracy: 0.7152 - loss: 0.5850 - val_accuracy: 0.7863 - val_loss: 0.5006
Epoch 18/100
31/31            0s 4ms/step -
accuracy: 0.7110 - loss: 0.5584 - val_accuracy: 0.7903 - val_loss: 0.5052
Epoch 19/100
31/31            0s 6ms/step -
accuracy: 0.7164 - loss: 0.5887 - val_accuracy: 0.7944 - val_loss: 0.5021
Epoch 20/100
31/31            0s 4ms/step -
accuracy: 0.7105 - loss: 0.5964 - val_accuracy: 0.7863 - val_loss: 0.5058
Epoch 21/100
31/31            0s 4ms/step -
accuracy: 0.7374 - loss: 0.5444 - val_accuracy: 0.7863 - val_loss: 0.4911
Epoch 22/100
31/31            0s 4ms/step -
accuracy: 0.7211 - loss: 0.5720 - val_accuracy: 0.7823 - val_loss: 0.4979
Epoch 23/100
31/31            0s 4ms/step -
accuracy: 0.7039 - loss: 0.5739 - val_accuracy: 0.7742 - val_loss: 0.4896
Epoch 24/100
31/31            0s 4ms/step -
accuracy: 0.7366 - loss: 0.5562 - val_accuracy: 0.7702 - val_loss: 0.5057
```

```
Epoch 25/100
31/31              0s 4ms/step -
accuracy: 0.6823 - loss: 0.5864 - val_accuracy: 0.7903 - val_loss: 0.5031
Epoch 26/100
31/31              0s 7ms/step -
accuracy: 0.7330 - loss: 0.5580 - val_accuracy: 0.7823 - val_loss: 0.4880
Epoch 27/100
31/31              0s 8ms/step -
accuracy: 0.7251 - loss: 0.5489 - val_accuracy: 0.7742 - val_loss: 0.4958
Epoch 28/100
31/31              0s 7ms/step -
accuracy: 0.7421 - loss: 0.5505 - val_accuracy: 0.7742 - val_loss: 0.4842
Epoch 29/100
31/31              0s 8ms/step -
accuracy: 0.7158 - loss: 0.5772 - val_accuracy: 0.7661 - val_loss: 0.4960
Epoch 30/100
31/31              0s 7ms/step -
accuracy: 0.7006 - loss: 0.5725 - val_accuracy: 0.7742 - val_loss: 0.4843
Epoch 31/100
31/31              0s 8ms/step -
accuracy: 0.7422 - loss: 0.5255 - val_accuracy: 0.7742 - val_loss: 0.4940
Epoch 32/100
31/31              0s 6ms/step -
accuracy: 0.7309 - loss: 0.5627 - val_accuracy: 0.7782 - val_loss: 0.4817
Epoch 33/100
31/31              0s 8ms/step -
accuracy: 0.7460 - loss: 0.5239 - val_accuracy: 0.7782 - val_loss: 0.4893
Epoch 34/100
31/31              0s 8ms/step -
accuracy: 0.7092 - loss: 0.5527 - val_accuracy: 0.7581 - val_loss: 0.4933
Epoch 35/100
31/31              1s 6ms/step -
accuracy: 0.7294 - loss: 0.5525 - val_accuracy: 0.7540 - val_loss: 0.4915
Epoch 36/100
31/31              0s 7ms/step -
accuracy: 0.7103 - loss: 0.5602 - val_accuracy: 0.7661 - val_loss: 0.4892
Epoch 37/100
31/31              0s 8ms/step -
accuracy: 0.7200 - loss: 0.5490 - val_accuracy: 0.7702 - val_loss: 0.4863
Epoch 38/100
31/31              0s 5ms/step -
accuracy: 0.7209 - loss: 0.5331 - val_accuracy: 0.7702 - val_loss: 0.4973
Epoch 39/100
31/31              0s 4ms/step -
accuracy: 0.7393 - loss: 0.5436 - val_accuracy: 0.7621 - val_loss: 0.4930
Epoch 40/100
31/31              0s 4ms/step -
accuracy: 0.7223 - loss: 0.5447 - val_accuracy: 0.7702 - val_loss: 0.4878
```

```
Epoch 41/100
31/31            0s 4ms/step -
accuracy: 0.7309 - loss: 0.5405 - val_accuracy: 0.7621 - val_loss: 0.4950
Epoch 42/100
31/31            0s 5ms/step -
accuracy: 0.7329 - loss: 0.5403 - val_accuracy: 0.7702 - val_loss: 0.4879
Epoch 43/100
31/31            0s 4ms/step -
accuracy: 0.7435 - loss: 0.5412 - val_accuracy: 0.7500 - val_loss: 0.4932
Epoch 44/100
31/31            0s 4ms/step -
accuracy: 0.7524 - loss: 0.5405 - val_accuracy: 0.7540 - val_loss: 0.5078
Epoch 45/100
31/31            0s 4ms/step -
accuracy: 0.7390 - loss: 0.5324 - val_accuracy: 0.7581 - val_loss: 0.4980
Epoch 46/100
31/31            0s 4ms/step -
accuracy: 0.7443 - loss: 0.5346 - val_accuracy: 0.7702 - val_loss: 0.4947
Epoch 47/100
31/31            0s 4ms/step -
accuracy: 0.7806 - loss: 0.4860 - val_accuracy: 0.7621 - val_loss: 0.4933
Epoch 48/100
31/31            0s 4ms/step -
accuracy: 0.7450 - loss: 0.5417 - val_accuracy: 0.7782 - val_loss: 0.4962
Epoch 49/100
31/31            0s 4ms/step -
accuracy: 0.7183 - loss: 0.5612 - val_accuracy: 0.7742 - val_loss: 0.5019
Epoch 50/100
31/31            0s 5ms/step -
accuracy: 0.7252 - loss: 0.5568 - val_accuracy: 0.7621 - val_loss: 0.4927
Epoch 51/100
31/31            0s 5ms/step -
accuracy: 0.7565 - loss: 0.5231 - val_accuracy: 0.7621 - val_loss: 0.5010
Epoch 52/100
31/31            0s 6ms/step -
accuracy: 0.7359 - loss: 0.5417 - val_accuracy: 0.7581 - val_loss: 0.4914
Epoch 53/100
31/31            0s 6ms/step -
accuracy: 0.7167 - loss: 0.5570 - val_accuracy: 0.7742 - val_loss: 0.4859
Epoch 54/100
31/31            0s 7ms/step -
accuracy: 0.7635 - loss: 0.5341 - val_accuracy: 0.7540 - val_loss: 0.4920
Epoch 55/100
31/31            0s 5ms/step -
accuracy: 0.7570 - loss: 0.5124 - val_accuracy: 0.7500 - val_loss: 0.5083
Epoch 56/100
31/31            0s 4ms/step -
accuracy: 0.7651 - loss: 0.5176 - val_accuracy: 0.7742 - val_loss: 0.4890
```

```
Epoch 57/100
31/31            0s 4ms/step -
accuracy: 0.7533 - loss: 0.5094 - val_accuracy: 0.7621 - val_loss: 0.4806
Epoch 58/100
31/31            0s 4ms/step -
accuracy: 0.7483 - loss: 0.5332 - val_accuracy: 0.7540 - val_loss: 0.4998
Epoch 59/100
31/31            0s 5ms/step -
accuracy: 0.7352 - loss: 0.5153 - val_accuracy: 0.7621 - val_loss: 0.4952
Epoch 60/100
31/31            0s 4ms/step -
accuracy: 0.7556 - loss: 0.5148 - val_accuracy: 0.7702 - val_loss: 0.4875
Epoch 61/100
31/31            0s 5ms/step -
accuracy: 0.7613 - loss: 0.4910 - val_accuracy: 0.7540 - val_loss: 0.4929
Epoch 62/100
31/31            0s 4ms/step -
accuracy: 0.7227 - loss: 0.5345 - val_accuracy: 0.7500 - val_loss: 0.5032
Epoch 63/100
31/31            0s 4ms/step -
accuracy: 0.7275 - loss: 0.5621 - val_accuracy: 0.7540 - val_loss: 0.4937
Epoch 64/100
31/31            0s 4ms/step -
accuracy: 0.7626 - loss: 0.4855 - val_accuracy: 0.7702 - val_loss: 0.4882
Epoch 65/100
31/31            0s 5ms/step -
accuracy: 0.7276 - loss: 0.5400 - val_accuracy: 0.7581 - val_loss: 0.4893
Epoch 66/100
31/31            0s 4ms/step -
accuracy: 0.7347 - loss: 0.5537 - val_accuracy: 0.7500 - val_loss: 0.4967
Epoch 67/100
31/31            0s 4ms/step -
accuracy: 0.7262 - loss: 0.5507 - val_accuracy: 0.7742 - val_loss: 0.4853
Epoch 68/100
31/31            0s 5ms/step -
accuracy: 0.7386 - loss: 0.5347 - val_accuracy: 0.7500 - val_loss: 0.4960
Epoch 69/100
31/31            0s 4ms/step -
accuracy: 0.7370 - loss: 0.5444 - val_accuracy: 0.7581 - val_loss: 0.4945
Epoch 70/100
31/31            0s 4ms/step -
accuracy: 0.7599 - loss: 0.5120 - val_accuracy: 0.7419 - val_loss: 0.4958
Epoch 71/100
31/31            0s 4ms/step -
accuracy: 0.7525 - loss: 0.5208 - val_accuracy: 0.7540 - val_loss: 0.4970
Epoch 72/100
31/31            0s 4ms/step -
accuracy: 0.7689 - loss: 0.5280 - val_accuracy: 0.7621 - val_loss: 0.5030
```

```
Epoch 73/100
31/31            0s 4ms/step -
accuracy: 0.7509 - loss: 0.5355 - val_accuracy: 0.7500 - val_loss: 0.4964
Epoch 74/100
31/31            0s 5ms/step -
accuracy: 0.7540 - loss: 0.5427 - val_accuracy: 0.7661 - val_loss: 0.4969
Epoch 75/100
31/31            0s 4ms/step -
accuracy: 0.7384 - loss: 0.5391 - val_accuracy: 0.7419 - val_loss: 0.5060
Epoch 76/100
31/31            0s 4ms/step -
accuracy: 0.7676 - loss: 0.4865 - val_accuracy: 0.7500 - val_loss: 0.4928
Epoch 77/100
31/31            0s 4ms/step -
accuracy: 0.7470 - loss: 0.5279 - val_accuracy: 0.7419 - val_loss: 0.4923
Epoch 78/100
31/31            0s 4ms/step -
accuracy: 0.7503 - loss: 0.5162 - val_accuracy: 0.7500 - val_loss: 0.4956
Epoch 79/100
31/31            0s 5ms/step -
accuracy: 0.7838 - loss: 0.4942 - val_accuracy: 0.7460 - val_loss: 0.4947
Epoch 80/100
31/31            0s 6ms/step -
accuracy: 0.7566 - loss: 0.5303 - val_accuracy: 0.7339 - val_loss: 0.4967
Epoch 81/100
31/31            0s 7ms/step -
accuracy: 0.7590 - loss: 0.4977 - val_accuracy: 0.7500 - val_loss: 0.4951
Epoch 82/100
31/31            0s 6ms/step -
accuracy: 0.7237 - loss: 0.5424 - val_accuracy: 0.7419 - val_loss: 0.4986
Epoch 83/100
31/31            0s 9ms/step -
accuracy: 0.7572 - loss: 0.5243 - val_accuracy: 0.7581 - val_loss: 0.4937
Epoch 84/100
31/31            1s 8ms/step -
accuracy: 0.7538 - loss: 0.5187 - val_accuracy: 0.7460 - val_loss: 0.4997
Epoch 85/100
31/31            0s 8ms/step -
accuracy: 0.7742 - loss: 0.4943 - val_accuracy: 0.7500 - val_loss: 0.4999
Epoch 86/100
31/31            0s 8ms/step -
accuracy: 0.7286 - loss: 0.5259 - val_accuracy: 0.7419 - val_loss: 0.5013
Epoch 87/100
31/31            0s 7ms/step -
accuracy: 0.7450 - loss: 0.5109 - val_accuracy: 0.7419 - val_loss: 0.4973
Epoch 88/100
31/31            0s 10ms/step -
accuracy: 0.7567 - loss: 0.5082 - val_accuracy: 0.7581 - val_loss: 0.4919
```

```
Epoch 89/100
31/31            0s 7ms/step -
accuracy: 0.7851 - loss: 0.4823 - val_accuracy: 0.7500 - val_loss: 0.4884
Epoch 90/100
31/31            0s 8ms/step -
accuracy: 0.7382 - loss: 0.5290 - val_accuracy: 0.7661 - val_loss: 0.4852
Epoch 91/100
31/31            0s 10ms/step -
accuracy: 0.7927 - loss: 0.4751 - val_accuracy: 0.7419 - val_loss: 0.4877
Epoch 92/100
31/31            0s 4ms/step -
accuracy: 0.7836 - loss: 0.4873 - val_accuracy: 0.7419 - val_loss: 0.4913
Epoch 93/100
31/31            0s 4ms/step -
accuracy: 0.7435 - loss: 0.5227 - val_accuracy: 0.7379 - val_loss: 0.4958
Epoch 94/100
31/31            0s 5ms/step -
accuracy: 0.7809 - loss: 0.4569 - val_accuracy: 0.7339 - val_loss: 0.4996
Epoch 95/100
31/31            0s 4ms/step -
accuracy: 0.7492 - loss: 0.5101 - val_accuracy: 0.7460 - val_loss: 0.4979
Epoch 96/100
31/31            0s 5ms/step -
accuracy: 0.7644 - loss: 0.4874 - val_accuracy: 0.7460 - val_loss: 0.4929
Epoch 97/100
31/31            0s 5ms/step -
accuracy: 0.7674 - loss: 0.5022 - val_accuracy: 0.7419 - val_loss: 0.4987
Epoch 98/100
31/31            0s 5ms/step -
accuracy: 0.7327 - loss: 0.5323 - val_accuracy: 0.7540 - val_loss: 0.4930
Epoch 99/100
31/31            0s 4ms/step -
accuracy: 0.7781 - loss: 0.4731 - val_accuracy: 0.7500 - val_loss: 0.5001
Epoch 100/100
31/31            0s 4ms/step -
accuracy: 0.7538 - loss: 0.5067 - val_accuracy: 0.7460 - val_loss: 0.4971
Total training time: 00:30 (mm:ss)
Validation Score for Fold 1: [0.49710720777511597, 0.7459677457809448]
Fold 2
Epoch 1/100
31/31            3s 12ms/step -
accuracy: 0.4977 - loss: 0.7044 - val_accuracy: 0.5524 - val_loss: 0.6896
Epoch 2/100
31/31            0s 4ms/step -
accuracy: 0.5284 - loss: 0.6903 - val_accuracy: 0.6089 - val_loss: 0.6844
Epoch 3/100
31/31            0s 4ms/step -
accuracy: 0.5573 - loss: 0.6837 - val_accuracy: 0.6734 - val_loss: 0.6692
```

```
Epoch 4/100
31/31            0s 5ms/step -
accuracy: 0.5744 - loss: 0.6778 - val_accuracy: 0.6774 - val_loss: 0.6460
Epoch 5/100
31/31            0s 4ms/step -
accuracy: 0.5937 - loss: 0.6682 - val_accuracy: 0.6976 - val_loss: 0.6110
Epoch 6/100
31/31            0s 4ms/step -
accuracy: 0.6327 - loss: 0.6454 - val_accuracy: 0.6815 - val_loss: 0.6004
Epoch 7/100
31/31            0s 4ms/step -
accuracy: 0.6388 - loss: 0.6312 - val_accuracy: 0.6976 - val_loss: 0.5831
Epoch 8/100
31/31            0s 5ms/step -
accuracy: 0.6902 - loss: 0.6032 - val_accuracy: 0.6976 - val_loss: 0.5707
Epoch 9/100
31/31            0s 4ms/step -
accuracy: 0.6942 - loss: 0.6093 - val_accuracy: 0.7016 - val_loss: 0.5681
Epoch 10/100
31/31            0s 4ms/step -
accuracy: 0.6898 - loss: 0.6024 - val_accuracy: 0.7097 - val_loss: 0.5736
Epoch 11/100
31/31            0s 6ms/step -
accuracy: 0.7199 - loss: 0.5833 - val_accuracy: 0.6815 - val_loss: 0.5725
Epoch 12/100
31/31            0s 4ms/step -
accuracy: 0.7058 - loss: 0.5722 - val_accuracy: 0.6855 - val_loss: 0.5702
Epoch 13/100
31/31            0s 5ms/step -
accuracy: 0.7202 - loss: 0.5759 - val_accuracy: 0.6815 - val_loss: 0.5702
Epoch 14/100
31/31            0s 4ms/step -
accuracy: 0.7434 - loss: 0.5605 - val_accuracy: 0.6774 - val_loss: 0.5675
Epoch 15/100
31/31            0s 4ms/step -
accuracy: 0.7210 - loss: 0.5685 - val_accuracy: 0.6774 - val_loss: 0.5652
Epoch 16/100
31/31            0s 5ms/step -
accuracy: 0.7374 - loss: 0.5528 - val_accuracy: 0.6935 - val_loss: 0.5654
Epoch 17/100
31/31            0s 5ms/step -
accuracy: 0.7453 - loss: 0.5374 - val_accuracy: 0.6976 - val_loss: 0.5630
Epoch 18/100
31/31            0s 4ms/step -
accuracy: 0.7266 - loss: 0.5578 - val_accuracy: 0.6976 - val_loss: 0.5620
Epoch 19/100
31/31            0s 4ms/step -
accuracy: 0.7159 - loss: 0.5730 - val_accuracy: 0.6935 - val_loss: 0.5635
```

```
Epoch 20/100
31/31            0s 5ms/step -
accuracy: 0.7522 - loss: 0.5457 - val_accuracy: 0.6935 - val_loss: 0.5647
Epoch 21/100
31/31            0s 4ms/step -
accuracy: 0.7328 - loss: 0.5600 - val_accuracy: 0.6895 - val_loss: 0.5636
Epoch 22/100
31/31            0s 9ms/step -
accuracy: 0.7032 - loss: 0.5755 - val_accuracy: 0.6976 - val_loss: 0.5590
Epoch 23/100
31/31            0s 7ms/step -
accuracy: 0.6968 - loss: 0.5781 - val_accuracy: 0.7177 - val_loss: 0.5569
Epoch 24/100
31/31            0s 7ms/step -
accuracy: 0.7251 - loss: 0.5617 - val_accuracy: 0.6935 - val_loss: 0.5589
Epoch 25/100
31/31            0s 8ms/step -
accuracy: 0.7297 - loss: 0.5437 - val_accuracy: 0.7016 - val_loss: 0.5621
Epoch 26/100
31/31            1s 7ms/step -
accuracy: 0.7505 - loss: 0.5389 - val_accuracy: 0.7097 - val_loss: 0.5568
Epoch 27/100
31/31            0s 9ms/step -
accuracy: 0.7478 - loss: 0.5570 - val_accuracy: 0.7056 - val_loss: 0.5604
Epoch 28/100
31/31            0s 7ms/step -
accuracy: 0.7365 - loss: 0.5566 - val_accuracy: 0.6895 - val_loss: 0.5630
Epoch 29/100
31/31            0s 9ms/step -
accuracy: 0.7311 - loss: 0.5756 - val_accuracy: 0.7097 - val_loss: 0.5574
Epoch 30/100
31/31            1s 8ms/step -
accuracy: 0.7676 - loss: 0.5085 - val_accuracy: 0.7056 - val_loss: 0.5609
Epoch 31/100
31/31            0s 8ms/step -
accuracy: 0.7512 - loss: 0.5270 - val_accuracy: 0.7056 - val_loss: 0.5604
Epoch 32/100
31/31            0s 8ms/step -
accuracy: 0.7506 - loss: 0.5294 - val_accuracy: 0.7016 - val_loss: 0.5588
Epoch 33/100
31/31            0s 5ms/step -
accuracy: 0.7566 - loss: 0.5498 - val_accuracy: 0.7097 - val_loss: 0.5636
Epoch 34/100
31/31            0s 5ms/step -
accuracy: 0.7305 - loss: 0.5245 - val_accuracy: 0.6935 - val_loss: 0.5585
Epoch 35/100
31/31            0s 5ms/step -
accuracy: 0.7720 - loss: 0.5227 - val_accuracy: 0.6935 - val_loss: 0.5594
```

```
Epoch 36/100
31/31            0s 6ms/step -
accuracy: 0.7612 - loss: 0.5195 - val_accuracy: 0.6976 - val_loss: 0.5670
Epoch 37/100
31/31            0s 5ms/step -
accuracy: 0.7420 - loss: 0.5449 - val_accuracy: 0.7258 - val_loss: 0.5560
Epoch 38/100
31/31            0s 4ms/step -
accuracy: 0.7542 - loss: 0.5166 - val_accuracy: 0.6976 - val_loss: 0.5567
Epoch 39/100
31/31            0s 4ms/step -
accuracy: 0.7664 - loss: 0.5139 - val_accuracy: 0.6935 - val_loss: 0.5624
Epoch 40/100
31/31            0s 4ms/step -
accuracy: 0.7373 - loss: 0.5311 - val_accuracy: 0.7097 - val_loss: 0.5557
Epoch 41/100
31/31            0s 4ms/step -
accuracy: 0.7580 - loss: 0.5143 - val_accuracy: 0.7016 - val_loss: 0.5580
Epoch 42/100
31/31            0s 6ms/step -
accuracy: 0.7666 - loss: 0.5008 - val_accuracy: 0.6895 - val_loss: 0.5617
Epoch 43/100
31/31            0s 4ms/step -
accuracy: 0.7518 - loss: 0.5329 - val_accuracy: 0.7097 - val_loss: 0.5561
Epoch 44/100
31/31            0s 4ms/step -
accuracy: 0.7564 - loss: 0.5275 - val_accuracy: 0.7056 - val_loss: 0.5546
Epoch 45/100
31/31            0s 4ms/step -
accuracy: 0.7425 - loss: 0.5555 - val_accuracy: 0.6895 - val_loss: 0.5579
Epoch 46/100
31/31            0s 4ms/step -
accuracy: 0.7574 - loss: 0.5242 - val_accuracy: 0.6976 - val_loss: 0.5555
Epoch 47/100
31/31            0s 5ms/step -
accuracy: 0.7574 - loss: 0.5207 - val_accuracy: 0.7056 - val_loss: 0.5552
Epoch 48/100
31/31            0s 4ms/step -
accuracy: 0.7583 - loss: 0.5348 - val_accuracy: 0.7097 - val_loss: 0.5494
Epoch 49/100
31/31            0s 4ms/step -
accuracy: 0.7801 - loss: 0.5012 - val_accuracy: 0.6976 - val_loss: 0.5554
Epoch 50/100
31/31            0s 6ms/step -
accuracy: 0.7607 - loss: 0.5252 - val_accuracy: 0.6895 - val_loss: 0.5530
Epoch 51/100
31/31            0s 5ms/step -
accuracy: 0.7373 - loss: 0.5342 - val_accuracy: 0.7137 - val_loss: 0.5527
```

```
Epoch 52/100
31/31            0s 5ms/step -
accuracy: 0.7825 - loss: 0.4936 - val_accuracy: 0.7097 - val_loss: 0.5551
Epoch 53/100
31/31            0s 4ms/step -
accuracy: 0.7557 - loss: 0.5260 - val_accuracy: 0.7056 - val_loss: 0.5534
Epoch 54/100
31/31            0s 4ms/step -
accuracy: 0.7992 - loss: 0.4686 - val_accuracy: 0.6976 - val_loss: 0.5583
Epoch 55/100
31/31            0s 8ms/step -
accuracy: 0.7500 - loss: 0.5495 - val_accuracy: 0.6855 - val_loss: 0.5582
Epoch 56/100
31/31            0s 4ms/step -
accuracy: 0.7503 - loss: 0.5061 - val_accuracy: 0.6935 - val_loss: 0.5620
Epoch 57/100
31/31            0s 5ms/step -
accuracy: 0.7568 - loss: 0.5200 - val_accuracy: 0.6935 - val_loss: 0.5633
Epoch 58/100
31/31            0s 4ms/step -
accuracy: 0.7591 - loss: 0.5274 - val_accuracy: 0.7016 - val_loss: 0.5569
Epoch 59/100
31/31            0s 4ms/step -
accuracy: 0.7652 - loss: 0.4998 - val_accuracy: 0.6895 - val_loss: 0.5596
Epoch 60/100
31/31            0s 4ms/step -
accuracy: 0.7573 - loss: 0.5142 - val_accuracy: 0.6976 - val_loss: 0.5553
Epoch 61/100
31/31            1s 18ms/step -
accuracy: 0.7564 - loss: 0.5127 - val_accuracy: 0.7137 - val_loss: 0.5524
Epoch 62/100
31/31            0s 4ms/step -
accuracy: 0.7826 - loss: 0.4929 - val_accuracy: 0.7177 - val_loss: 0.5489
Epoch 63/100
31/31            0s 4ms/step -
accuracy: 0.7514 - loss: 0.5070 - val_accuracy: 0.6976 - val_loss: 0.5531
Epoch 64/100
31/31            0s 5ms/step -
accuracy: 0.7712 - loss: 0.4905 - val_accuracy: 0.7097 - val_loss: 0.5486
Epoch 65/100
31/31            0s 5ms/step -
accuracy: 0.7534 - loss: 0.5234 - val_accuracy: 0.6895 - val_loss: 0.5578
Epoch 66/100
31/31            0s 5ms/step -
accuracy: 0.7657 - loss: 0.4976 - val_accuracy: 0.6976 - val_loss: 0.5616
Epoch 67/100
31/31            0s 5ms/step -
accuracy: 0.8005 - loss: 0.4642 - val_accuracy: 0.7016 - val_loss: 0.5534
```

```
Epoch 68/100
31/31            0s 5ms/step -
accuracy: 0.7897 - loss: 0.4866 - val_accuracy: 0.6976 - val_loss: 0.5560
Epoch 69/100
31/31            0s 5ms/step -
accuracy: 0.7612 - loss: 0.4995 - val_accuracy: 0.7177 - val_loss: 0.5522
Epoch 70/100
31/31            0s 5ms/step -
accuracy: 0.7532 - loss: 0.5423 - val_accuracy: 0.7137 - val_loss: 0.5520
Epoch 71/100
31/31            0s 6ms/step -
accuracy: 0.7535 - loss: 0.5289 - val_accuracy: 0.7097 - val_loss: 0.5512
Epoch 72/100
31/31            0s 8ms/step -
accuracy: 0.7798 - loss: 0.5004 - val_accuracy: 0.7016 - val_loss: 0.5604
Epoch 73/100
31/31            1s 8ms/step -
accuracy: 0.7412 - loss: 0.5227 - val_accuracy: 0.7218 - val_loss: 0.5442
Epoch 74/100
31/31            0s 7ms/step -
accuracy: 0.7832 - loss: 0.4932 - val_accuracy: 0.7056 - val_loss: 0.5529
Epoch 75/100
31/31            0s 9ms/step -
accuracy: 0.7685 - loss: 0.5124 - val_accuracy: 0.6935 - val_loss: 0.5603
Epoch 76/100
31/31            1s 10ms/step -
accuracy: 0.7790 - loss: 0.4901 - val_accuracy: 0.6935 - val_loss: 0.5632
Epoch 77/100
31/31            1s 9ms/step -
accuracy: 0.7674 - loss: 0.5056 - val_accuracy: 0.7137 - val_loss: 0.5512
Epoch 78/100
31/31            0s 4ms/step -
accuracy: 0.7666 - loss: 0.5076 - val_accuracy: 0.6935 - val_loss: 0.5544
Epoch 79/100
31/31            0s 5ms/step -
accuracy: 0.7818 - loss: 0.4967 - val_accuracy: 0.6976 - val_loss: 0.5499
Epoch 80/100
31/31            0s 5ms/step -
accuracy: 0.7722 - loss: 0.5102 - val_accuracy: 0.7056 - val_loss: 0.5566
Epoch 81/100
31/31            0s 5ms/step -
accuracy: 0.7735 - loss: 0.4973 - val_accuracy: 0.7258 - val_loss: 0.5372
Epoch 82/100
31/31            0s 6ms/step -
accuracy: 0.8063 - loss: 0.4831 - val_accuracy: 0.6935 - val_loss: 0.5618
Epoch 83/100
31/31            0s 4ms/step -
accuracy: 0.7916 - loss: 0.4669 - val_accuracy: 0.6935 - val_loss: 0.5602
```

```
Epoch 84/100
31/31            0s 5ms/step -
accuracy: 0.7706 - loss: 0.4995 - val_accuracy: 0.7097 - val_loss: 0.5530
Epoch 85/100
31/31            0s 5ms/step -
accuracy: 0.7882 - loss: 0.4851 - val_accuracy: 0.7177 - val_loss: 0.5585
Epoch 86/100
31/31            0s 5ms/step -
accuracy: 0.7740 - loss: 0.4768 - val_accuracy: 0.6935 - val_loss: 0.5572
Epoch 87/100
31/31            0s 5ms/step -
accuracy: 0.7716 - loss: 0.4997 - val_accuracy: 0.7137 - val_loss: 0.5497
Epoch 88/100
31/31            0s 4ms/step -
accuracy: 0.7847 - loss: 0.4831 - val_accuracy: 0.6976 - val_loss: 0.5507
Epoch 89/100
31/31            0s 5ms/step -
accuracy: 0.7848 - loss: 0.4955 - val_accuracy: 0.6895 - val_loss: 0.5533
Epoch 90/100
31/31            0s 5ms/step -
accuracy: 0.7711 - loss: 0.4947 - val_accuracy: 0.7056 - val_loss: 0.5441
Epoch 91/100
31/31            0s 5ms/step -
accuracy: 0.7731 - loss: 0.4873 - val_accuracy: 0.6976 - val_loss: 0.5504
Epoch 92/100
31/31            0s 4ms/step -
accuracy: 0.7967 - loss: 0.4712 - val_accuracy: 0.7097 - val_loss: 0.5439
Epoch 93/100
31/31            0s 5ms/step -
accuracy: 0.7920 - loss: 0.4838 - val_accuracy: 0.7097 - val_loss: 0.5458
Epoch 94/100
31/31            0s 4ms/step -
accuracy: 0.7692 - loss: 0.4858 - val_accuracy: 0.7177 - val_loss: 0.5385
Epoch 95/100
31/31            0s 4ms/step -
accuracy: 0.7965 - loss: 0.4835 - val_accuracy: 0.7218 - val_loss: 0.5425
Epoch 96/100
31/31            0s 5ms/step -
accuracy: 0.7790 - loss: 0.5036 - val_accuracy: 0.7258 - val_loss: 0.5401
Epoch 97/100
31/31            0s 4ms/step -
accuracy: 0.7751 - loss: 0.5032 - val_accuracy: 0.6895 - val_loss: 0.5518
Epoch 98/100
31/31            0s 4ms/step -
accuracy: 0.7709 - loss: 0.4849 - val_accuracy: 0.7056 - val_loss: 0.5517
Epoch 99/100
31/31            0s 5ms/step -
accuracy: 0.7707 - loss: 0.5117 - val_accuracy: 0.7177 - val_loss: 0.5394
```

```
Epoch 100/100
31/31            0s 4ms/step -
accuracy: 0.7775 - loss: 0.5032 - val_accuracy: 0.7177 - val_loss: 0.5443
Total training time: 00:30 (mm:ss)
Validation Score for Fold 2: [0.5443021655082703, 0.7177419066429138]
Fold 3
Epoch 1/100
31/31            3s 11ms/step -
accuracy: 0.5031 - loss: 0.6961 - val_accuracy: 0.4758 - val_loss: 0.6906
Epoch 2/100
31/31            0s 4ms/step -
accuracy: 0.5380 - loss: 0.6887 - val_accuracy: 0.5000 - val_loss: 0.6838
Epoch 3/100
31/31            0s 5ms/step -
accuracy: 0.5333 - loss: 0.6841 - val_accuracy: 0.6210 - val_loss: 0.6692
Epoch 4/100
31/31            0s 4ms/step -
accuracy: 0.6278 - loss: 0.6601 - val_accuracy: 0.7419 - val_loss: 0.6359
Epoch 5/100
31/31            0s 6ms/step -
accuracy: 0.6478 - loss: 0.6403 - val_accuracy: 0.7298 - val_loss: 0.5919
Epoch 6/100
31/31            0s 10ms/step -
accuracy: 0.6562 - loss: 0.6329 - val_accuracy: 0.7298 - val_loss: 0.5680
Epoch 7/100
31/31            0s 7ms/step -
accuracy: 0.6782 - loss: 0.6130 - val_accuracy: 0.7258 - val_loss: 0.5575
Epoch 8/100
31/31            0s 6ms/step -
accuracy: 0.6975 - loss: 0.6059 - val_accuracy: 0.7460 - val_loss: 0.5484
Epoch 9/100
31/31            0s 7ms/step -
accuracy: 0.6926 - loss: 0.6073 - val_accuracy: 0.7379 - val_loss: 0.5418
Epoch 10/100
31/31            0s 7ms/step -
accuracy: 0.7223 - loss: 0.5679 - val_accuracy: 0.7460 - val_loss: 0.5280
Epoch 11/100
31/31            0s 7ms/step -
accuracy: 0.7222 - loss: 0.5875 - val_accuracy: 0.7500 - val_loss: 0.5327
Epoch 12/100
31/31            0s 7ms/step -
accuracy: 0.7154 - loss: 0.6063 - val_accuracy: 0.7702 - val_loss: 0.5351
Epoch 13/100
31/31            0s 8ms/step -
accuracy: 0.7304 - loss: 0.5825 - val_accuracy: 0.7460 - val_loss: 0.5262
Epoch 14/100
31/31            0s 7ms/step -
accuracy: 0.7250 - loss: 0.5821 - val_accuracy: 0.7500 - val_loss: 0.5274
```

```
Epoch 15/100
31/31            0s 7ms/step -
accuracy: 0.7468 - loss: 0.5486 - val_accuracy: 0.7540 - val_loss: 0.5271
Epoch 16/100
31/31            0s 9ms/step -
accuracy: 0.7403 - loss: 0.5398 - val_accuracy: 0.7621 - val_loss: 0.5214
Epoch 17/100
31/31            0s 4ms/step -
accuracy: 0.7208 - loss: 0.5756 - val_accuracy: 0.7460 - val_loss: 0.5317
Epoch 18/100
31/31            0s 5ms/step -
accuracy: 0.7305 - loss: 0.5738 - val_accuracy: 0.7500 - val_loss: 0.5274
Epoch 19/100
31/31            0s 5ms/step -
accuracy: 0.7626 - loss: 0.5582 - val_accuracy: 0.7500 - val_loss: 0.5255
Epoch 20/100
31/31            0s 4ms/step -
accuracy: 0.7366 - loss: 0.5450 - val_accuracy: 0.7500 - val_loss: 0.5239
Epoch 21/100
31/31            0s 4ms/step -
accuracy: 0.7264 - loss: 0.5587 - val_accuracy: 0.7581 - val_loss: 0.5190
Epoch 22/100
31/31            0s 4ms/step -
accuracy: 0.7347 - loss: 0.5518 - val_accuracy: 0.7621 - val_loss: 0.5114
Epoch 23/100
31/31            0s 5ms/step -
accuracy: 0.7333 - loss: 0.5835 - val_accuracy: 0.7581 - val_loss: 0.5154
Epoch 24/100
31/31            0s 4ms/step -
accuracy: 0.7438 - loss: 0.5603 - val_accuracy: 0.7500 - val_loss: 0.5169
Epoch 25/100
31/31            0s 4ms/step -
accuracy: 0.7155 - loss: 0.5737 - val_accuracy: 0.7581 - val_loss: 0.5149
Epoch 26/100
31/31            0s 4ms/step -
accuracy: 0.7327 - loss: 0.5610 - val_accuracy: 0.7500 - val_loss: 0.5148
Epoch 27/100
31/31            0s 5ms/step -
accuracy: 0.7258 - loss: 0.5577 - val_accuracy: 0.7621 - val_loss: 0.5135
Epoch 28/100
31/31            0s 4ms/step -
accuracy: 0.7533 - loss: 0.5297 - val_accuracy: 0.7540 - val_loss: 0.5108
Epoch 29/100
31/31            0s 4ms/step -
accuracy: 0.7438 - loss: 0.5649 - val_accuracy: 0.7661 - val_loss: 0.5052
Epoch 30/100
31/31            0s 4ms/step -
accuracy: 0.7388 - loss: 0.5235 - val_accuracy: 0.7581 - val_loss: 0.5005
```

```
Epoch 31/100
31/31            0s 4ms/step -
accuracy: 0.7426 - loss: 0.5561 - val_accuracy: 0.7782 - val_loss: 0.5038
Epoch 32/100
31/31            0s 6ms/step -
accuracy: 0.7503 - loss: 0.5337 - val_accuracy: 0.7621 - val_loss: 0.5029
Epoch 33/100
31/31            0s 4ms/step -
accuracy: 0.7539 - loss: 0.5286 - val_accuracy: 0.7661 - val_loss: 0.5029
Epoch 34/100
31/31            0s 4ms/step -
accuracy: 0.7502 - loss: 0.5410 - val_accuracy: 0.7379 - val_loss: 0.5131
Epoch 35/100
31/31            0s 4ms/step -
accuracy: 0.7386 - loss: 0.5613 - val_accuracy: 0.7661 - val_loss: 0.5089
Epoch 36/100
31/31            0s 5ms/step -
accuracy: 0.7239 - loss: 0.5414 - val_accuracy: 0.7581 - val_loss: 0.5054
Epoch 37/100
31/31            0s 5ms/step -
accuracy: 0.7399 - loss: 0.5711 - val_accuracy: 0.7500 - val_loss: 0.5032
Epoch 38/100
31/31            0s 4ms/step -
accuracy: 0.7396 - loss: 0.5494 - val_accuracy: 0.7661 - val_loss: 0.4990
Epoch 39/100
31/31            0s 4ms/step -
accuracy: 0.7477 - loss: 0.5550 - val_accuracy: 0.7702 - val_loss: 0.5008
Epoch 40/100
31/31            0s 5ms/step -
accuracy: 0.7167 - loss: 0.5415 - val_accuracy: 0.7581 - val_loss: 0.5022
Epoch 41/100
31/31            0s 6ms/step -
accuracy: 0.7524 - loss: 0.5188 - val_accuracy: 0.7702 - val_loss: 0.4957
Epoch 42/100
31/31            0s 4ms/step -
accuracy: 0.7333 - loss: 0.5461 - val_accuracy: 0.7460 - val_loss: 0.4979
Epoch 43/100
31/31            0s 4ms/step -
accuracy: 0.7581 - loss: 0.5431 - val_accuracy: 0.7581 - val_loss: 0.4996
Epoch 44/100
31/31            0s 5ms/step -
accuracy: 0.7228 - loss: 0.5451 - val_accuracy: 0.7500 - val_loss: 0.4955
Epoch 45/100
31/31            0s 5ms/step -
accuracy: 0.7206 - loss: 0.5660 - val_accuracy: 0.7702 - val_loss: 0.4947
Epoch 46/100
31/31            0s 4ms/step -
accuracy: 0.7362 - loss: 0.5421 - val_accuracy: 0.7460 - val_loss: 0.5097
```

```
Epoch 47/100
31/31            0s 4ms/step -
accuracy: 0.7246 - loss: 0.5538 - val_accuracy: 0.7742 - val_loss: 0.4961
Epoch 48/100
31/31            0s 4ms/step -
accuracy: 0.7235 - loss: 0.5619 - val_accuracy: 0.7621 - val_loss: 0.4885
Epoch 49/100
31/31            0s 4ms/step -
accuracy: 0.7562 - loss: 0.5381 - val_accuracy: 0.7702 - val_loss: 0.4948
Epoch 50/100
31/31            0s 5ms/step -
accuracy: 0.7592 - loss: 0.5306 - val_accuracy: 0.7702 - val_loss: 0.4901
Epoch 51/100
31/31            0s 4ms/step -
accuracy: 0.7479 - loss: 0.5244 - val_accuracy: 0.7581 - val_loss: 0.4882
Epoch 52/100
31/31            0s 4ms/step -
accuracy: 0.7495 - loss: 0.5158 - val_accuracy: 0.7621 - val_loss: 0.5016
Epoch 53/100
31/31            0s 4ms/step -
accuracy: 0.7247 - loss: 0.5570 - val_accuracy: 0.7621 - val_loss: 0.4970
Epoch 54/100
31/31            0s 5ms/step -
accuracy: 0.7502 - loss: 0.5376 - val_accuracy: 0.7823 - val_loss: 0.4839
Epoch 55/100
31/31            0s 5ms/step -
accuracy: 0.7560 - loss: 0.5148 - val_accuracy: 0.7742 - val_loss: 0.4822
Epoch 56/100
31/31            0s 4ms/step -
accuracy: 0.7613 - loss: 0.5239 - val_accuracy: 0.7742 - val_loss: 0.4848
Epoch 57/100
31/31            0s 6ms/step -
accuracy: 0.7439 - loss: 0.5435 - val_accuracy: 0.7702 - val_loss: 0.4836
Epoch 58/100
31/31            0s 5ms/step -
accuracy: 0.7557 - loss: 0.5067 - val_accuracy: 0.7782 - val_loss: 0.4764
Epoch 59/100
31/31            0s 8ms/step -
accuracy: 0.7579 - loss: 0.5308 - val_accuracy: 0.7661 - val_loss: 0.4991
Epoch 60/100
31/31            0s 6ms/step -
accuracy: 0.7643 - loss: 0.5264 - val_accuracy: 0.7661 - val_loss: 0.4872
Epoch 61/100
31/31            0s 7ms/step -
accuracy: 0.7480 - loss: 0.5450 - val_accuracy: 0.7742 - val_loss: 0.4794
Epoch 62/100
31/31            0s 6ms/step -
accuracy: 0.7601 - loss: 0.5267 - val_accuracy: 0.7782 - val_loss: 0.4798
```

```
Epoch 63/100
31/31            0s 6ms/step -
accuracy: 0.7546 - loss: 0.5529 - val_accuracy: 0.7823 - val_loss: 0.4908
Epoch 64/100
31/31            0s 8ms/step -
accuracy: 0.7637 - loss: 0.5258 - val_accuracy: 0.7863 - val_loss: 0.4771
Epoch 65/100
31/31            1s 7ms/step -
accuracy: 0.7464 - loss: 0.5218 - val_accuracy: 0.7742 - val_loss: 0.4856
Epoch 66/100
31/31            0s 8ms/step -
accuracy: 0.7352 - loss: 0.5282 - val_accuracy: 0.7661 - val_loss: 0.4854
Epoch 67/100
31/31            0s 8ms/step -
accuracy: 0.7486 - loss: 0.5384 - val_accuracy: 0.7782 - val_loss: 0.4824
Epoch 68/100
31/31            0s 6ms/step -
accuracy: 0.7416 - loss: 0.5375 - val_accuracy: 0.7742 - val_loss: 0.4858
Epoch 69/100
31/31            0s 5ms/step -
accuracy: 0.7739 - loss: 0.5164 - val_accuracy: 0.7903 - val_loss: 0.4739
Epoch 70/100
31/31            0s 5ms/step -
accuracy: 0.7622 - loss: 0.5074 - val_accuracy: 0.7782 - val_loss: 0.4711
Epoch 71/100
31/31            0s 5ms/step -
accuracy: 0.7537 - loss: 0.5248 - val_accuracy: 0.7823 - val_loss: 0.4846
Epoch 72/100
31/31            0s 4ms/step -
accuracy: 0.7545 - loss: 0.5335 - val_accuracy: 0.7702 - val_loss: 0.4815
Epoch 73/100
31/31            0s 5ms/step -
accuracy: 0.7261 - loss: 0.5608 - val_accuracy: 0.7621 - val_loss: 0.4836
Epoch 74/100
31/31            0s 4ms/step -
accuracy: 0.7610 - loss: 0.5149 - val_accuracy: 0.7742 - val_loss: 0.4669
Epoch 75/100
31/31            0s 5ms/step -
accuracy: 0.7540 - loss: 0.5083 - val_accuracy: 0.7782 - val_loss: 0.4741
Epoch 76/100
31/31            0s 5ms/step -
accuracy: 0.7245 - loss: 0.5514 - val_accuracy: 0.7782 - val_loss: 0.4718
Epoch 77/100
31/31            0s 4ms/step -
accuracy: 0.7537 - loss: 0.5287 - val_accuracy: 0.7782 - val_loss: 0.4846
Epoch 78/100
31/31            0s 5ms/step -
accuracy: 0.7718 - loss: 0.5021 - val_accuracy: 0.7823 - val_loss: 0.4773
```

```
Epoch 79/100
31/31            0s 5ms/step -
accuracy: 0.7683 - loss: 0.5320 - val_accuracy: 0.7823 - val_loss: 0.4698
Epoch 80/100
31/31            0s 5ms/step -
accuracy: 0.7709 - loss: 0.5051 - val_accuracy: 0.7823 - val_loss: 0.4698
Epoch 81/100
31/31            0s 5ms/step -
accuracy: 0.7698 - loss: 0.5104 - val_accuracy: 0.7782 - val_loss: 0.4687
Epoch 82/100
31/31            0s 4ms/step -
accuracy: 0.7603 - loss: 0.5159 - val_accuracy: 0.7823 - val_loss: 0.4689
Epoch 83/100
31/31            0s 5ms/step -
accuracy: 0.7743 - loss: 0.5020 - val_accuracy: 0.7782 - val_loss: 0.4697
Epoch 84/100
31/31            0s 6ms/step -
accuracy: 0.7722 - loss: 0.5035 - val_accuracy: 0.7742 - val_loss: 0.4855
Epoch 85/100
31/31            0s 4ms/step -
accuracy: 0.7523 - loss: 0.5201 - val_accuracy: 0.7823 - val_loss: 0.4723
Epoch 86/100
31/31            0s 5ms/step -
accuracy: 0.7517 - loss: 0.5048 - val_accuracy: 0.7782 - val_loss: 0.4694
Epoch 87/100
31/31            0s 6ms/step -
accuracy: 0.7566 - loss: 0.5220 - val_accuracy: 0.7621 - val_loss: 0.4777
Epoch 88/100
31/31            0s 4ms/step -
accuracy: 0.7549 - loss: 0.5162 - val_accuracy: 0.7863 - val_loss: 0.4744
Epoch 89/100
31/31            0s 5ms/step -
accuracy: 0.7817 - loss: 0.4990 - val_accuracy: 0.7823 - val_loss: 0.4715
Epoch 90/100
31/31            0s 5ms/step -
accuracy: 0.7828 - loss: 0.4878 - val_accuracy: 0.7742 - val_loss: 0.4678
Epoch 91/100
31/31            0s 4ms/step -
accuracy: 0.7372 - loss: 0.5481 - val_accuracy: 0.7863 - val_loss: 0.4709
Epoch 92/100
31/31            0s 5ms/step -
accuracy: 0.7762 - loss: 0.5070 - val_accuracy: 0.7661 - val_loss: 0.4765
Epoch 93/100
31/31            0s 4ms/step -
accuracy: 0.7323 - loss: 0.5329 - val_accuracy: 0.7782 - val_loss: 0.4754
Epoch 94/100
31/31            0s 4ms/step -
accuracy: 0.7675 - loss: 0.5020 - val_accuracy: 0.7742 - val_loss: 0.4766
```

```
Epoch 95/100
31/31            0s 4ms/step -
accuracy: 0.7613 - loss: 0.5094 - val_accuracy: 0.7742 - val_loss: 0.4706
Epoch 96/100
31/31            0s 4ms/step -
accuracy: 0.7617 - loss: 0.4959 - val_accuracy: 0.7702 - val_loss: 0.4811
Epoch 97/100
31/31            0s 6ms/step -
accuracy: 0.7463 - loss: 0.5286 - val_accuracy: 0.7702 - val_loss: 0.4731
Epoch 98/100
31/31            0s 5ms/step -
accuracy: 0.7171 - loss: 0.5563 - val_accuracy: 0.7782 - val_loss: 0.4742
Epoch 99/100
31/31            0s 5ms/step -
accuracy: 0.7626 - loss: 0.5037 - val_accuracy: 0.7742 - val_loss: 0.4760
Epoch 100/100
31/31            0s 4ms/step -
accuracy: 0.7657 - loss: 0.5075 - val_accuracy: 0.7621 - val_loss: 0.4933
Total training time: 00:28 (mm:ss)
Validation Score for Fold 3: [0.4932604134082794, 0.7620967626571655]
Fold 4
Epoch 1/100
31/31            3s 28ms/step -
accuracy: 0.5425 - loss: 0.6887 - val_accuracy: 0.6774 - val_loss: 0.6849
Epoch 2/100
31/31            1s 10ms/step -
accuracy: 0.5028 - loss: 0.6926 - val_accuracy: 0.6613 - val_loss: 0.6746
Epoch 3/100
31/31            1s 26ms/step -
accuracy: 0.5877 - loss: 0.6808 - val_accuracy: 0.6935 - val_loss: 0.6665
Epoch 4/100
31/31            1s 7ms/step -
accuracy: 0.5954 - loss: 0.6797 - val_accuracy: 0.6895 - val_loss: 0.6364
Epoch 5/100
31/31            0s 5ms/step -
accuracy: 0.6432 - loss: 0.6507 - val_accuracy: 0.6935 - val_loss: 0.6084
Epoch 6/100
31/31            0s 5ms/step -
accuracy: 0.6447 - loss: 0.6444 - val_accuracy: 0.7056 - val_loss: 0.5985
Epoch 7/100
31/31            0s 4ms/step -
accuracy: 0.6471 - loss: 0.6175 - val_accuracy: 0.6895 - val_loss: 0.5892
Epoch 8/100
31/31            0s 4ms/step -
accuracy: 0.6482 - loss: 0.6382 - val_accuracy: 0.7218 - val_loss: 0.5795
Epoch 9/100
31/31            0s 5ms/step -
accuracy: 0.6561 - loss: 0.6167 - val_accuracy: 0.7177 - val_loss: 0.5701
```

```
Epoch 10/100
31/31            0s 4ms/step -
accuracy: 0.6704 - loss: 0.5888 - val_accuracy: 0.7339 - val_loss: 0.5609
Epoch 11/100
31/31            0s 5ms/step -
accuracy: 0.7061 - loss: 0.6040 - val_accuracy: 0.7258 - val_loss: 0.5615
Epoch 12/100
31/31            0s 5ms/step -
accuracy: 0.6952 - loss: 0.6003 - val_accuracy: 0.7177 - val_loss: 0.5703
Epoch 13/100
31/31            0s 5ms/step -
accuracy: 0.7053 - loss: 0.5960 - val_accuracy: 0.7419 - val_loss: 0.5558
Epoch 14/100
31/31            0s 4ms/step -
accuracy: 0.6809 - loss: 0.5920 - val_accuracy: 0.7218 - val_loss: 0.5511
Epoch 15/100
31/31            0s 6ms/step -
accuracy: 0.6887 - loss: 0.5868 - val_accuracy: 0.7419 - val_loss: 0.5462
Epoch 16/100
31/31            0s 4ms/step -
accuracy: 0.7115 - loss: 0.5847 - val_accuracy: 0.7460 - val_loss: 0.5495
Epoch 17/100
31/31            0s 5ms/step -
accuracy: 0.7023 - loss: 0.6018 - val_accuracy: 0.7460 - val_loss: 0.5515
Epoch 18/100
31/31            0s 5ms/step -
accuracy: 0.7092 - loss: 0.5794 - val_accuracy: 0.7298 - val_loss: 0.5484
Epoch 19/100
31/31            0s 6ms/step -
accuracy: 0.7285 - loss: 0.5524 - val_accuracy: 0.7218 - val_loss: 0.5483
Epoch 20/100
31/31            0s 5ms/step -
accuracy: 0.7055 - loss: 0.5622 - val_accuracy: 0.7298 - val_loss: 0.5508
Epoch 21/100
31/31            0s 4ms/step -
accuracy: 0.7321 - loss: 0.5649 - val_accuracy: 0.7097 - val_loss: 0.5505
Epoch 22/100
31/31            0s 5ms/step -
accuracy: 0.7182 - loss: 0.5449 - val_accuracy: 0.7419 - val_loss: 0.5484
Epoch 23/100
31/31            0s 4ms/step -
accuracy: 0.7413 - loss: 0.5485 - val_accuracy: 0.7419 - val_loss: 0.5480
Epoch 24/100
31/31            0s 4ms/step -
accuracy: 0.7127 - loss: 0.5523 - val_accuracy: 0.7419 - val_loss: 0.5471
Epoch 25/100
31/31            0s 4ms/step -
accuracy: 0.6883 - loss: 0.5946 - val_accuracy: 0.7177 - val_loss: 0.5535
```

```
Epoch 26/100
31/31            0s 5ms/step -
accuracy: 0.7324 - loss: 0.5576 - val_accuracy: 0.7258 - val_loss: 0.5511
Epoch 27/100
31/31            0s 4ms/step -
accuracy: 0.7286 - loss: 0.5581 - val_accuracy: 0.7258 - val_loss: 0.5489
Epoch 28/100
31/31            0s 5ms/step -
accuracy: 0.7109 - loss: 0.5488 - val_accuracy: 0.7258 - val_loss: 0.5563
Epoch 29/100
31/31            0s 5ms/step -
accuracy: 0.7109 - loss: 0.5728 - val_accuracy: 0.7258 - val_loss: 0.5673
Epoch 30/100
31/31            0s 6ms/step -
accuracy: 0.7466 - loss: 0.5351 - val_accuracy: 0.7218 - val_loss: 0.5529
Epoch 31/100
31/31            0s 5ms/step -
accuracy: 0.7484 - loss: 0.5471 - val_accuracy: 0.7339 - val_loss: 0.5509
Epoch 32/100
31/31            0s 4ms/step -
accuracy: 0.7393 - loss: 0.5382 - val_accuracy: 0.7258 - val_loss: 0.5484
Epoch 33/100
31/31            0s 5ms/step -
accuracy: 0.7494 - loss: 0.5377 - val_accuracy: 0.7258 - val_loss: 0.5478
Epoch 34/100
31/31            0s 6ms/step -
accuracy: 0.7312 - loss: 0.5433 - val_accuracy: 0.7218 - val_loss: 0.5537
Epoch 35/100
31/31            0s 5ms/step -
accuracy: 0.7607 - loss: 0.5240 - val_accuracy: 0.7298 - val_loss: 0.5514
Epoch 36/100
31/31            0s 5ms/step -
accuracy: 0.7353 - loss: 0.5414 - val_accuracy: 0.7218 - val_loss: 0.5520
Epoch 37/100
31/31            0s 4ms/step -
accuracy: 0.7506 - loss: 0.5589 - val_accuracy: 0.7177 - val_loss: 0.5590
Epoch 38/100
31/31            0s 6ms/step -
accuracy: 0.7257 - loss: 0.5537 - val_accuracy: 0.7218 - val_loss: 0.5482
Epoch 39/100
31/31            0s 4ms/step -
accuracy: 0.7586 - loss: 0.5389 - val_accuracy: 0.7298 - val_loss: 0.5510
Epoch 40/100
31/31            0s 4ms/step -
accuracy: 0.7502 - loss: 0.5425 - val_accuracy: 0.7339 - val_loss: 0.5498
Epoch 41/100
31/31            0s 5ms/step -
accuracy: 0.7529 - loss: 0.5366 - val_accuracy: 0.7137 - val_loss: 0.5518
```

```
Epoch 42/100
31/31             0s 6ms/step -
accuracy: 0.7363 - loss: 0.5267 - val_accuracy: 0.7097 - val_loss: 0.5507
Epoch 43/100
31/31             0s 7ms/step -
accuracy: 0.7535 - loss: 0.5126 - val_accuracy: 0.7218 - val_loss: 0.5503
Epoch 44/100
31/31             0s 8ms/step -
accuracy: 0.7440 - loss: 0.5513 - val_accuracy: 0.7258 - val_loss: 0.5473
Epoch 45/100
31/31             0s 8ms/step -
accuracy: 0.7398 - loss: 0.5298 - val_accuracy: 0.7258 - val_loss: 0.5465
Epoch 46/100
31/31             0s 7ms/step -
accuracy: 0.7328 - loss: 0.5591 - val_accuracy: 0.7177 - val_loss: 0.5488
Epoch 47/100
31/31             0s 6ms/step -
accuracy: 0.7619 - loss: 0.5205 - val_accuracy: 0.7298 - val_loss: 0.5538
Epoch 48/100
31/31             0s 9ms/step -
accuracy: 0.7688 - loss: 0.5025 - val_accuracy: 0.7258 - val_loss: 0.5503
Epoch 49/100
31/31             1s 8ms/step -
accuracy: 0.7315 - loss: 0.5341 - val_accuracy: 0.7177 - val_loss: 0.5525
Epoch 50/100
31/31             0s 8ms/step -
accuracy: 0.7539 - loss: 0.5213 - val_accuracy: 0.7258 - val_loss: 0.5528
Epoch 51/100
31/31             0s 9ms/step -
accuracy: 0.7156 - loss: 0.5595 - val_accuracy: 0.7339 - val_loss: 0.5478
Epoch 52/100
31/31             0s 5ms/step -
accuracy: 0.7292 - loss: 0.5518 - val_accuracy: 0.7218 - val_loss: 0.5493
Epoch 53/100
31/31             0s 4ms/step -
accuracy: 0.7306 - loss: 0.5559 - val_accuracy: 0.7379 - val_loss: 0.5467
Epoch 54/100
31/31             0s 5ms/step -
accuracy: 0.7483 - loss: 0.5478 - val_accuracy: 0.7258 - val_loss: 0.5594
Epoch 55/100
31/31             0s 4ms/step -
accuracy: 0.7621 - loss: 0.5147 - val_accuracy: 0.7177 - val_loss: 0.5498
Epoch 56/100
31/31             0s 5ms/step -
accuracy: 0.7593 - loss: 0.5113 - val_accuracy: 0.7177 - val_loss: 0.5558
Epoch 57/100
31/31             0s 5ms/step -
accuracy: 0.7306 - loss: 0.5302 - val_accuracy: 0.7258 - val_loss: 0.5473
```

```
Epoch 58/100
31/31            0s 4ms/step -
accuracy: 0.7701 - loss: 0.5360 - val_accuracy: 0.7218 - val_loss: 0.5492
Epoch 59/100
31/31            0s 5ms/step -
accuracy: 0.7521 - loss: 0.5170 - val_accuracy: 0.7379 - val_loss: 0.5449
Epoch 60/100
31/31            0s 5ms/step -
accuracy: 0.7419 - loss: 0.5253 - val_accuracy: 0.7177 - val_loss: 0.5501
Epoch 61/100
31/31            0s 4ms/step -
accuracy: 0.7735 - loss: 0.5115 - val_accuracy: 0.7258 - val_loss: 0.5416
Epoch 62/100
31/31            0s 4ms/step -
accuracy: 0.7526 - loss: 0.5136 - val_accuracy: 0.7419 - val_loss: 0.5420
Epoch 63/100
31/31            0s 5ms/step -
accuracy: 0.7184 - loss: 0.5634 - val_accuracy: 0.7258 - val_loss: 0.5460
Epoch 64/100
31/31            0s 6ms/step -
accuracy: 0.7568 - loss: 0.5257 - val_accuracy: 0.7339 - val_loss: 0.5426
Epoch 65/100
31/31            0s 4ms/step -
accuracy: 0.7494 - loss: 0.5164 - val_accuracy: 0.7258 - val_loss: 0.5435
Epoch 66/100
31/31            0s 5ms/step -
accuracy: 0.7223 - loss: 0.5305 - val_accuracy: 0.7419 - val_loss: 0.5446
Epoch 67/100
31/31            0s 4ms/step -
accuracy: 0.7496 - loss: 0.5107 - val_accuracy: 0.7298 - val_loss: 0.5521
Epoch 68/100
31/31            0s 6ms/step -
accuracy: 0.7487 - loss: 0.5180 - val_accuracy: 0.7339 - val_loss: 0.5390
Epoch 69/100
31/31            0s 4ms/step -
accuracy: 0.7335 - loss: 0.5218 - val_accuracy: 0.7419 - val_loss: 0.5389
Epoch 70/100
31/31            0s 5ms/step -
accuracy: 0.7614 - loss: 0.4866 - val_accuracy: 0.7419 - val_loss: 0.5462
Epoch 71/100
31/31            0s 5ms/step -
accuracy: 0.7637 - loss: 0.5277 - val_accuracy: 0.7460 - val_loss: 0.5415
Epoch 72/100
31/31            0s 5ms/step -
accuracy: 0.7458 - loss: 0.5293 - val_accuracy: 0.7298 - val_loss: 0.5482
Epoch 73/100
31/31            0s 5ms/step -
accuracy: 0.7721 - loss: 0.5137 - val_accuracy: 0.7419 - val_loss: 0.5431
```

```
Epoch 74/100
31/31            0s 4ms/step -
accuracy: 0.7677 - loss: 0.5225 - val_accuracy: 0.7460 - val_loss: 0.5411
Epoch 75/100
31/31            0s 5ms/step -
accuracy: 0.7661 - loss: 0.5084 - val_accuracy: 0.7339 - val_loss: 0.5439
Epoch 76/100
31/31            0s 5ms/step -
accuracy: 0.7571 - loss: 0.5170 - val_accuracy: 0.7419 - val_loss: 0.5467
Epoch 77/100
31/31            0s 6ms/step -
accuracy: 0.7632 - loss: 0.5255 - val_accuracy: 0.7460 - val_loss: 0.5441
Epoch 78/100
31/31            0s 5ms/step -
accuracy: 0.7492 - loss: 0.5175 - val_accuracy: 0.7419 - val_loss: 0.5428
Epoch 79/100
31/31            0s 5ms/step -
accuracy: 0.7644 - loss: 0.5217 - val_accuracy: 0.7419 - val_loss: 0.5420
Epoch 80/100
31/31            0s 5ms/step -
accuracy: 0.7593 - loss: 0.5139 - val_accuracy: 0.7298 - val_loss: 0.5470
Epoch 81/100
31/31            0s 5ms/step -
accuracy: 0.7631 - loss: 0.5191 - val_accuracy: 0.7339 - val_loss: 0.5405
Epoch 82/100
31/31            0s 5ms/step -
accuracy: 0.7732 - loss: 0.4996 - val_accuracy: 0.7379 - val_loss: 0.5412
Epoch 83/100
31/31            0s 5ms/step -
accuracy: 0.7705 - loss: 0.5045 - val_accuracy: 0.7339 - val_loss: 0.5541
Epoch 84/100
31/31            0s 5ms/step -
accuracy: 0.7980 - loss: 0.4815 - val_accuracy: 0.7500 - val_loss: 0.5382
Epoch 85/100
31/31            0s 5ms/step -
accuracy: 0.7763 - loss: 0.5039 - val_accuracy: 0.7258 - val_loss: 0.5566
Epoch 86/100
31/31            0s 4ms/step -
accuracy: 0.7648 - loss: 0.5265 - val_accuracy: 0.7379 - val_loss: 0.5541
Epoch 87/100
31/31            0s 4ms/step -
accuracy: 0.7545 - loss: 0.5176 - val_accuracy: 0.7460 - val_loss: 0.5409
Epoch 88/100
31/31            0s 4ms/step -
accuracy: 0.7539 - loss: 0.5041 - val_accuracy: 0.7339 - val_loss: 0.5488
Epoch 89/100
31/31            0s 5ms/step -
accuracy: 0.7425 - loss: 0.5302 - val_accuracy: 0.7460 - val_loss: 0.5414
```

```
Epoch 90/100
31/31              0s 5ms/step -
accuracy: 0.7695 - loss: 0.4934 - val_accuracy: 0.7500 - val_loss: 0.5423
Epoch 91/100
31/31              0s 5ms/step -
accuracy: 0.7398 - loss: 0.5368 - val_accuracy: 0.7339 - val_loss: 0.5410
Epoch 92/100
31/31              0s 8ms/step -
accuracy: 0.7433 - loss: 0.5159 - val_accuracy: 0.7339 - val_loss: 0.5419
Epoch 93/100
31/31              1s 6ms/step -
accuracy: 0.7884 - loss: 0.4836 - val_accuracy: 0.7298 - val_loss: 0.5432
Epoch 94/100
31/31              0s 9ms/step -
accuracy: 0.7784 - loss: 0.4941 - val_accuracy: 0.7379 - val_loss: 0.5455
Epoch 95/100
31/31              0s 6ms/step -
accuracy: 0.7663 - loss: 0.4856 - val_accuracy: 0.7379 - val_loss: 0.5417
Epoch 96/100
31/31              0s 8ms/step -
accuracy: 0.7455 - loss: 0.5219 - val_accuracy: 0.7298 - val_loss: 0.5407
Epoch 97/100
31/31              0s 7ms/step -
accuracy: 0.7532 - loss: 0.5171 - val_accuracy: 0.7258 - val_loss: 0.5490
Epoch 98/100
31/31              0s 8ms/step -
accuracy: 0.7523 - loss: 0.5058 - val_accuracy: 0.7339 - val_loss: 0.5467
Epoch 99/100
31/31              0s 8ms/step -
accuracy: 0.7634 - loss: 0.4865 - val_accuracy: 0.7218 - val_loss: 0.5468
Epoch 100/100
31/31              0s 7ms/step -
accuracy: 0.7583 - loss: 0.4954 - val_accuracy: 0.7540 - val_loss: 0.5394
Total training time: 00:32 (mm:ss)
Validation Score for Fold 4: [0.5394293665885925, 0.7540322542190552]
Fold 5
Epoch 1/100
31/31              3s 13ms/step -
accuracy: 0.5148 - loss: 0.6916 - val_accuracy: 0.5847 - val_loss: 0.6921
Epoch 2/100
31/31              0s 5ms/step -
accuracy: 0.4888 - loss: 0.6944 - val_accuracy: 0.5565 - val_loss: 0.6904
Epoch 3/100
31/31              0s 5ms/step -
accuracy: 0.4905 - loss: 0.6903 - val_accuracy: 0.6290 - val_loss: 0.6880
Epoch 4/100
31/31              0s 5ms/step -
accuracy: 0.5533 - loss: 0.6896 - val_accuracy: 0.6573 - val_loss: 0.6827
```

```
Epoch 5/100
31/31            0s 5ms/step -
accuracy: 0.5523 - loss: 0.6869 - val_accuracy: 0.6774 - val_loss: 0.6753
Epoch 6/100
31/31            0s 5ms/step -
accuracy: 0.5649 - loss: 0.6812 - val_accuracy: 0.6815 - val_loss: 0.6648
Epoch 7/100
31/31            0s 5ms/step -
accuracy: 0.6195 - loss: 0.6684 - val_accuracy: 0.6734 - val_loss: 0.6511
Epoch 8/100
31/31            0s 5ms/step -
accuracy: 0.6761 - loss: 0.6382 - val_accuracy: 0.6573 - val_loss: 0.6314
Epoch 9/100
31/31            0s 7ms/step -
accuracy: 0.6603 - loss: 0.6442 - val_accuracy: 0.6734 - val_loss: 0.6204
Epoch 10/100
31/31            0s 5ms/step -
accuracy: 0.6676 - loss: 0.6335 - val_accuracy: 0.6935 - val_loss: 0.6016
Epoch 11/100
31/31            0s 5ms/step -
accuracy: 0.6992 - loss: 0.6090 - val_accuracy: 0.6976 - val_loss: 0.5927
Epoch 12/100
31/31            0s 5ms/step -
accuracy: 0.6922 - loss: 0.5947 - val_accuracy: 0.7016 - val_loss: 0.5840
Epoch 13/100
31/31            0s 6ms/step -
accuracy: 0.6924 - loss: 0.6123 - val_accuracy: 0.6976 - val_loss: 0.5773
Epoch 14/100
31/31            0s 5ms/step -
accuracy: 0.7125 - loss: 0.5850 - val_accuracy: 0.7218 - val_loss: 0.5687
Epoch 15/100
31/31            0s 5ms/step -
accuracy: 0.7361 - loss: 0.5680 - val_accuracy: 0.7016 - val_loss: 0.5732
Epoch 16/100
31/31            0s 6ms/step -
accuracy: 0.7287 - loss: 0.5684 - val_accuracy: 0.7137 - val_loss: 0.5716
Epoch 17/100
31/31            0s 5ms/step -
accuracy: 0.7115 - loss: 0.5995 - val_accuracy: 0.7056 - val_loss: 0.5641
Epoch 18/100
31/31            0s 4ms/step -
accuracy: 0.7114 - loss: 0.5707 - val_accuracy: 0.7218 - val_loss: 0.5639
Epoch 19/100
31/31            0s 5ms/step -
accuracy: 0.7180 - loss: 0.5736 - val_accuracy: 0.7097 - val_loss: 0.5630
Epoch 20/100
31/31            0s 6ms/step -
accuracy: 0.7445 - loss: 0.5457 - val_accuracy: 0.7137 - val_loss: 0.5627
```

```
Epoch 21/100
31/31            0s 4ms/step -
accuracy: 0.7475 - loss: 0.5579 - val_accuracy: 0.7258 - val_loss: 0.5637
Epoch 22/100
31/31            0s 4ms/step -
accuracy: 0.7208 - loss: 0.5584 - val_accuracy: 0.7258 - val_loss: 0.5571
Epoch 23/100
31/31            0s 5ms/step -
accuracy: 0.7122 - loss: 0.5519 - val_accuracy: 0.7379 - val_loss: 0.5594
Epoch 24/100
31/31            0s 6ms/step -
accuracy: 0.7462 - loss: 0.5422 - val_accuracy: 0.7379 - val_loss: 0.5604
Epoch 25/100
31/31            0s 5ms/step -
accuracy: 0.7042 - loss: 0.5873 - val_accuracy: 0.7177 - val_loss: 0.5584
Epoch 26/100
31/31            0s 4ms/step -
accuracy: 0.7500 - loss: 0.5405 - val_accuracy: 0.7298 - val_loss: 0.5537
Epoch 27/100
31/31            0s 6ms/step -
accuracy: 0.7611 - loss: 0.5385 - val_accuracy: 0.7177 - val_loss: 0.5627
Epoch 28/100
31/31            0s 5ms/step -
accuracy: 0.7181 - loss: 0.5648 - val_accuracy: 0.7218 - val_loss: 0.5502
Epoch 29/100
31/31            0s 7ms/step -
accuracy: 0.7610 - loss: 0.5184 - val_accuracy: 0.7056 - val_loss: 0.5562
Epoch 30/100
31/31            0s 9ms/step -
accuracy: 0.7432 - loss: 0.5423 - val_accuracy: 0.7137 - val_loss: 0.5529
Epoch 31/100
31/31            1s 7ms/step -
accuracy: 0.7478 - loss: 0.5319 - val_accuracy: 0.7379 - val_loss: 0.5475
Epoch 32/100
31/31            0s 6ms/step -
accuracy: 0.7532 - loss: 0.5310 - val_accuracy: 0.7460 - val_loss: 0.5433
Epoch 33/100
31/31            0s 8ms/step -
accuracy: 0.7561 - loss: 0.5383 - val_accuracy: 0.7419 - val_loss: 0.5423
Epoch 34/100
31/31            0s 8ms/step -
accuracy: 0.7500 - loss: 0.5231 - val_accuracy: 0.7339 - val_loss: 0.5444
Epoch 35/100
31/31            0s 7ms/step -
accuracy: 0.7443 - loss: 0.5626 - val_accuracy: 0.7339 - val_loss: 0.5446
Epoch 36/100
31/31            0s 8ms/step -
accuracy: 0.7340 - loss: 0.5431 - val_accuracy: 0.7419 - val_loss: 0.5398
```

```
Epoch 37/100
31/31            0s 9ms/step -
accuracy: 0.7257 - loss: 0.5516 - val_accuracy: 0.7339 - val_loss: 0.5404
Epoch 38/100
31/31            1s 6ms/step -
accuracy: 0.7513 - loss: 0.5271 - val_accuracy: 0.7379 - val_loss: 0.5382
Epoch 39/100
31/31            0s 5ms/step -
accuracy: 0.7606 - loss: 0.5103 - val_accuracy: 0.7419 - val_loss: 0.5389
Epoch 40/100
31/31            0s 5ms/step -
accuracy: 0.7599 - loss: 0.5166 - val_accuracy: 0.7379 - val_loss: 0.5406
Epoch 41/100
31/31            0s 5ms/step -
accuracy: 0.7557 - loss: 0.5198 - val_accuracy: 0.7379 - val_loss: 0.5406
Epoch 42/100
31/31            0s 4ms/step -
accuracy: 0.7582 - loss: 0.5260 - val_accuracy: 0.7177 - val_loss: 0.5419
Epoch 43/100
31/31            0s 5ms/step -
accuracy: 0.7431 - loss: 0.5203 - val_accuracy: 0.7258 - val_loss: 0.5426
Epoch 44/100
31/31            0s 5ms/step -
accuracy: 0.7296 - loss: 0.5576 - val_accuracy: 0.7258 - val_loss: 0.5408
Epoch 45/100
31/31            0s 5ms/step -
accuracy: 0.7995 - loss: 0.4960 - val_accuracy: 0.7298 - val_loss: 0.5368
Epoch 46/100
31/31            0s 6ms/step -
accuracy: 0.7530 - loss: 0.5153 - val_accuracy: 0.7419 - val_loss: 0.5381
Epoch 47/100
31/31            0s 5ms/step -
accuracy: 0.7606 - loss: 0.4990 - val_accuracy: 0.7339 - val_loss: 0.5337
Epoch 48/100
31/31            0s 5ms/step -
accuracy: 0.7346 - loss: 0.5354 - val_accuracy: 0.7339 - val_loss: 0.5361
Epoch 49/100
31/31            0s 5ms/step -
accuracy: 0.7478 - loss: 0.5306 - val_accuracy: 0.7339 - val_loss: 0.5318
Epoch 50/100
31/31            0s 6ms/step -
accuracy: 0.7581 - loss: 0.5317 - val_accuracy: 0.7419 - val_loss: 0.5332
Epoch 51/100
31/31            0s 5ms/step -
accuracy: 0.7510 - loss: 0.5093 - val_accuracy: 0.7460 - val_loss: 0.5309
Epoch 52/100
31/31            0s 4ms/step -
accuracy: 0.7434 - loss: 0.5157 - val_accuracy: 0.7339 - val_loss: 0.5288
```

```
Epoch 53/100
31/31            0s 5ms/step -
accuracy: 0.7534 - loss: 0.5188 - val_accuracy: 0.7258 - val_loss: 0.5366
Epoch 54/100
31/31            0s 5ms/step -
accuracy: 0.7656 - loss: 0.5242 - val_accuracy: 0.7298 - val_loss: 0.5292
Epoch 55/100
31/31            0s 5ms/step -
accuracy: 0.7662 - loss: 0.5111 - val_accuracy: 0.7339 - val_loss: 0.5377
Epoch 56/100
31/31            0s 6ms/step -
accuracy: 0.7630 - loss: 0.4898 - val_accuracy: 0.7339 - val_loss: 0.5291
Epoch 57/100
31/31            0s 5ms/step -
accuracy: 0.7560 - loss: 0.5244 - val_accuracy: 0.7460 - val_loss: 0.5316
Epoch 58/100
31/31            0s 7ms/step -
accuracy: 0.7554 - loss: 0.5049 - val_accuracy: 0.7500 - val_loss: 0.5273
Epoch 59/100
31/31            0s 5ms/step -
accuracy: 0.7359 - loss: 0.5083 - val_accuracy: 0.7379 - val_loss: 0.5294
Epoch 60/100
31/31            0s 6ms/step -
accuracy: 0.7600 - loss: 0.5158 - val_accuracy: 0.7419 - val_loss: 0.5261
Epoch 61/100
31/31            0s 5ms/step -
accuracy: 0.7211 - loss: 0.5494 - val_accuracy: 0.7500 - val_loss: 0.5239
Epoch 62/100
31/31            0s 5ms/step -
accuracy: 0.7420 - loss: 0.5230 - val_accuracy: 0.7419 - val_loss: 0.5297
Epoch 63/100
31/31            0s 5ms/step -
accuracy: 0.7694 - loss: 0.4961 - val_accuracy: 0.7419 - val_loss: 0.5237
Epoch 64/100
31/31            0s 5ms/step -
accuracy: 0.7596 - loss: 0.5229 - val_accuracy: 0.7419 - val_loss: 0.5238
Epoch 65/100
31/31            0s 5ms/step -
accuracy: 0.7457 - loss: 0.5394 - val_accuracy: 0.7218 - val_loss: 0.5334
Epoch 66/100
31/31            0s 5ms/step -
accuracy: 0.7498 - loss: 0.5132 - val_accuracy: 0.7661 - val_loss: 0.5278
Epoch 67/100
31/31            0s 5ms/step -
accuracy: 0.7502 - loss: 0.5103 - val_accuracy: 0.7661 - val_loss: 0.5266
Epoch 68/100
31/31            0s 6ms/step -
accuracy: 0.7478 - loss: 0.5319 - val_accuracy: 0.7419 - val_loss: 0.5288
```

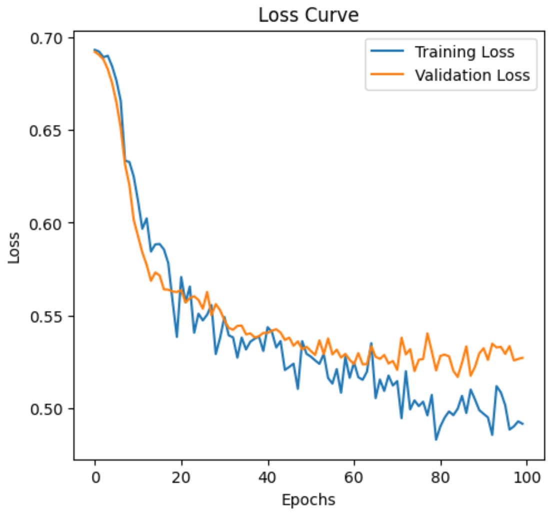
```
Epoch 69/100
31/31            0s 5ms/step -
accuracy: 0.7636 - loss: 0.4947 - val_accuracy: 0.7581 - val_loss: 0.5241
Epoch 70/100
31/31            0s 6ms/step -
accuracy: 0.7877 - loss: 0.4835 - val_accuracy: 0.7702 - val_loss: 0.5255
Epoch 71/100
31/31            0s 5ms/step -
accuracy: 0.7430 - loss: 0.5084 - val_accuracy: 0.7661 - val_loss: 0.5207
Epoch 72/100
31/31            0s 4ms/step -
accuracy: 0.7800 - loss: 0.4821 - val_accuracy: 0.7258 - val_loss: 0.5381
Epoch 73/100
31/31            0s 5ms/step -
accuracy: 0.7469 - loss: 0.5156 - val_accuracy: 0.7460 - val_loss: 0.5291
Epoch 74/100
31/31            0s 6ms/step -
accuracy: 0.7607 - loss: 0.4983 - val_accuracy: 0.7298 - val_loss: 0.5319
Epoch 75/100
31/31            0s 5ms/step -
accuracy: 0.7812 - loss: 0.4821 - val_accuracy: 0.7661 - val_loss: 0.5201
Epoch 76/100
31/31            0s 7ms/step -
accuracy: 0.7445 - loss: 0.4981 - val_accuracy: 0.7379 - val_loss: 0.5262
Epoch 77/100
31/31            0s 8ms/step -
accuracy: 0.7635 - loss: 0.5046 - val_accuracy: 0.7460 - val_loss: 0.5266
Epoch 78/100
31/31            0s 7ms/step -
accuracy: 0.7580 - loss: 0.4962 - val_accuracy: 0.7379 - val_loss: 0.5404
Epoch 79/100
31/31            0s 8ms/step -
accuracy: 0.7713 - loss: 0.5047 - val_accuracy: 0.7419 - val_loss: 0.5307
Epoch 80/100
31/31            0s 6ms/step -
accuracy: 0.7729 - loss: 0.4794 - val_accuracy: 0.7540 - val_loss: 0.5204
Epoch 81/100
31/31            0s 9ms/step -
accuracy: 0.7672 - loss: 0.4978 - val_accuracy: 0.7298 - val_loss: 0.5283
Epoch 82/100
31/31            1s 9ms/step -
accuracy: 0.7742 - loss: 0.5028 - val_accuracy: 0.7460 - val_loss: 0.5289
Epoch 83/100
31/31            1s 8ms/step -
accuracy: 0.7597 - loss: 0.5003 - val_accuracy: 0.7339 - val_loss: 0.5280
Epoch 84/100
31/31            0s 8ms/step -
accuracy: 0.7824 - loss: 0.4868 - val_accuracy: 0.7540 - val_loss: 0.5202
```

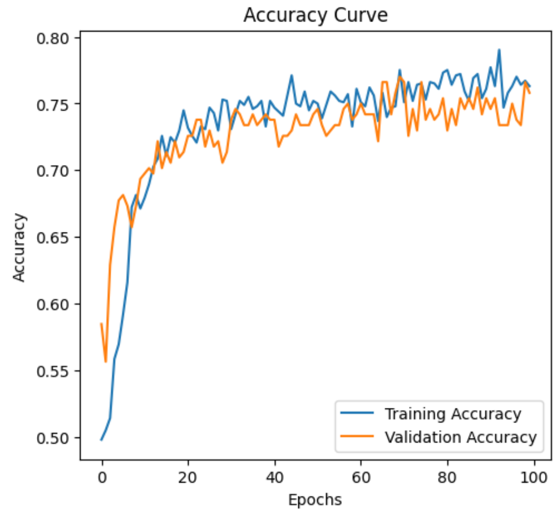
```
Epoch 85/100
31/31            0s 10ms/step -
accuracy: 0.7633 - loss: 0.4912 - val_accuracy: 0.7460 - val_loss: 0.5169
Epoch 86/100
31/31            1s 6ms/step -
accuracy: 0.7633 - loss: 0.4998 - val_accuracy: 0.7540 - val_loss: 0.5246
Epoch 87/100
31/31            0s 9ms/step -
accuracy: 0.7654 - loss: 0.5171 - val_accuracy: 0.7460 - val_loss: 0.5334
Epoch 88/100
31/31            0s 5ms/step -
accuracy: 0.7597 - loss: 0.5346 - val_accuracy: 0.7621 - val_loss: 0.5176
Epoch 89/100
31/31            0s 5ms/step -
accuracy: 0.7538 - loss: 0.4995 - val_accuracy: 0.7419 - val_loss: 0.5223
Epoch 90/100
31/31            0s 5ms/step -
accuracy: 0.7765 - loss: 0.4933 - val_accuracy: 0.7540 - val_loss: 0.5296
Epoch 91/100
31/31            0s 5ms/step -
accuracy: 0.7680 - loss: 0.5154 - val_accuracy: 0.7460 - val_loss: 0.5324
Epoch 92/100
31/31            0s 4ms/step -
accuracy: 0.7718 - loss: 0.4887 - val_accuracy: 0.7540 - val_loss: 0.5261
Epoch 93/100
31/31            0s 6ms/step -
accuracy: 0.7783 - loss: 0.4934 - val_accuracy: 0.7339 - val_loss: 0.5348
Epoch 94/100
31/31            0s 5ms/step -
accuracy: 0.7659 - loss: 0.5082 - val_accuracy: 0.7339 - val_loss: 0.5328
Epoch 95/100
31/31            0s 6ms/step -
accuracy: 0.7741 - loss: 0.4699 - val_accuracy: 0.7339 - val_loss: 0.5331
Epoch 96/100
31/31            0s 6ms/step -
accuracy: 0.7333 - loss: 0.5538 - val_accuracy: 0.7500 - val_loss: 0.5293
Epoch 97/100
31/31            0s 5ms/step -
accuracy: 0.7686 - loss: 0.4925 - val_accuracy: 0.7379 - val_loss: 0.5335
Epoch 98/100
31/31            0s 4ms/step -
accuracy: 0.7523 - loss: 0.5040 - val_accuracy: 0.7339 - val_loss: 0.5258
Epoch 99/100
31/31            0s 5ms/step -
accuracy: 0.7711 - loss: 0.4808 - val_accuracy: 0.7661 - val_loss: 0.5266
Epoch 100/100
31/31            0s 5ms/step -
accuracy: 0.7693 - loss: 0.4813 - val_accuracy: 0.7581 - val_loss: 0.5272
```

```
Total training time: 00:31 (mm:ss)
Validation Score for Fold 5: [0.5272184610366821, 0.7580645084381104]
Validation Scores across folds: [[0.49710720777511597, 0.7459677457809448],
[0.5443021655082703, 0.7177419066429138], [0.4932604134082794,
0.7620967626571655], [0.5394293665885925, 0.7540322542190552],
[0.5272184610366821, 0.7580645084381104]]
Mean Validation Score: 0.6339
Best Validation Score: 0.7621
9/9            0s 11ms/step
```

Notice that we saved the training results in a variable called `history`. This is because we can use this variable to plot the progress of the model across the different epochs and analyze the behaviors and tendencies that would be otherwise ignored.

[ ]:
```
# Graficar las curvas de pérdida y accuracy
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# Curva de pérdida
ax[0].plot(history.history['loss'], label='Training Loss')
ax[0].plot(history.history['val_loss'], label='Validation Loss')
ax[0].set_title('Loss Curve')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Loss')
ax[0].legend()

# Curva de accuracy
ax[1].plot(history.history['accuracy'], label='Training Accuracy')
ax[1].plot(history.history['val_accuracy'], label='Validation Accuracy')
ax[1].set_title('Accuracy Curve')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Accuracy')
ax[1].legend()

plt.show()
```





With these plots, we can notice that the validation loss tends to sit around 0.55 across the last epochs, while training loss keeps lowering. This may technically lead to overfitting, but this is a fact that we will check through the next code blocks. On the other hand, the accuracy of the training and validation set kept improving across epochs.

Now we will actually evaluate the model against our testing set data:

[ ]:
```
y_pred = mlp_model.predict(X_val)
y_pred = (y_pred > 0.5).astype(int)

loss, accuracy = mlp_model.evaluate(X_train_mlp, y_train_mlp)

print("Train Loss:", loss)
print("Train Accuracy:", accuracy)

# Evaluate the model on the test set
loss, accuracy = mlp_model.evaluate(X_val, y_val)

print("Validation Loss:", loss)
print("Validation Accuracy:", accuracy)


# Print the confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_val, y_pred))

# Print the classification report
print("Classification Report")
print(classification_report(y_val, y_pred))
```

```
8/8              0s 11ms/step
```

```
31/31            0s 2ms/step -
accuracy: 0.7888 - loss: 0.4641
Train Loss: 0.4766082167625427
Train Accuracy: 0.7802419066429138
8/8            0s 2ms/step -
accuracy: 0.8160 - loss: 0.4102
Validation Loss: 0.42998310923576355
Validation Accuracy: 0.8024193644523621
Confusion Matrix:
[[107  22]
 [ 27  92]]
Classification Report
              precision    recall  f1-score   support

           0       0.80      0.83      0.81       129
           1       0.81      0.77      0.79       119

    accuracy                           0.80       248
   macro avg       0.80      0.80      0.80       248
weighted avg       0.80      0.80      0.80       248
```

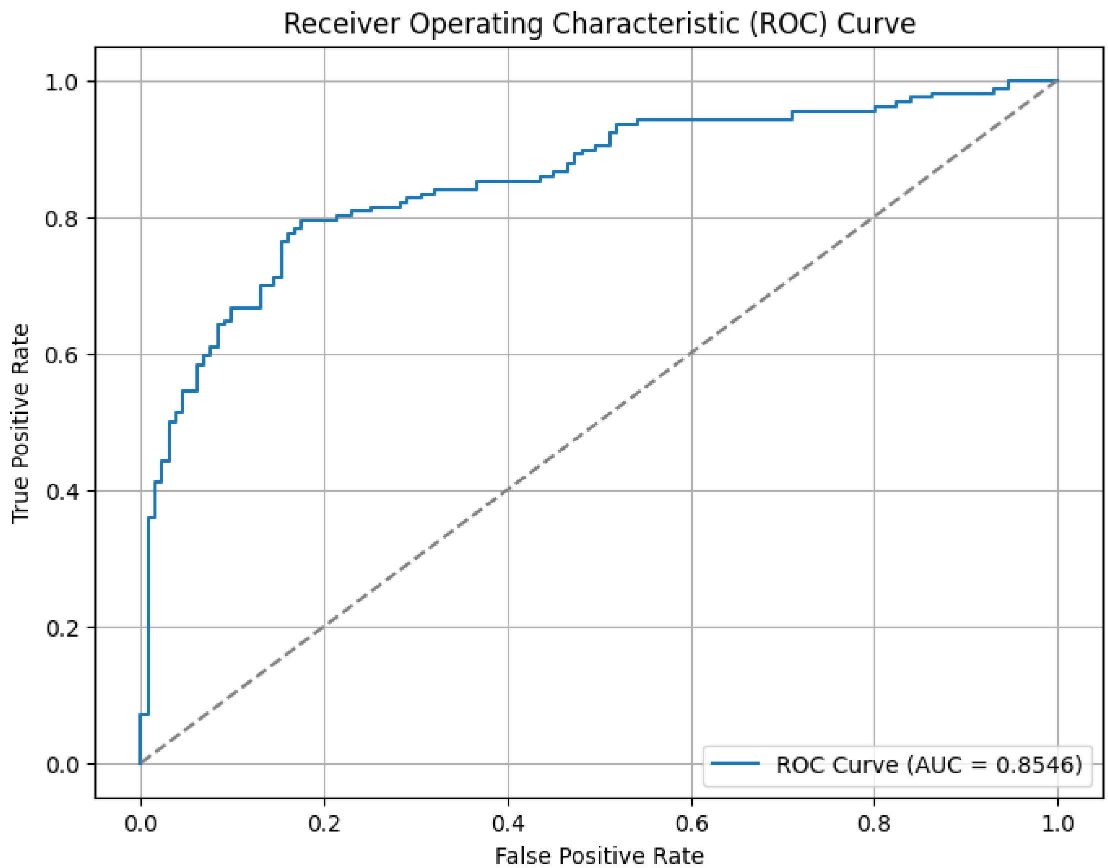
It seems that this model has very good overall perfomance, since it has the the highest accuracy and f1-score punctuation so far. But, we must not forget, this is still training phase, in order to know the real model performance we must use the test data.

[ ]:
```python
# Predict results
start_time = time.time()
y_pred = mlp_model.predict(X_test_normalized)
end_time = time.time()

predictions = (y_pred > 0.5).astype(int)
mlp_prediction_time = end_time - start_time

# Evaluate the model
train_loss, train_accuracy = mlp_model.evaluate(X_train_normalized, y_train)

X_test_normalized = np.array(X_test_normalized)
y_test = np.array(y_test)
test_loss, mlp_accuracy = mlp_model.evaluate(X_test_normalized, y_test)

# Print Test results
print("Train Loss:", train_loss)
print("Train Accuracy:", train_accuracy)

print("Test Loss:", test_loss)
print("Test Accuracy:", mlp_accuracy)
```

```python
# Print the confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, predictions))

# Print the classification report
print("Classification Report")
print(classification_report(y_test, predictions))

# Compute ROC AUC
mlp_roc_auc = roc_auc_score(y_test, y_pred)
fpr, tpr, thresholds = roc_curve(y_test, y_pred)

print(f"ROC AUC Score: {mlp_roc_auc:.4f}")

# Plot ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {mlp_roc_auc:.4f})")
plt.plot([0, 1], [0, 1], color="gray", linestyle="--")  # Diagonal line
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic (ROC) Curve")
plt.legend(loc="lower right")
plt.grid()
plt.show()

# Print prediction time
print(f"Prediction time: {mlp_prediction_time:.4f} seconds")
```

```
9/9               0s 2ms/step
39/39               0s 2ms/step -
accuracy: 0.7666 - loss: 0.4812
9/9               0s 3ms/step -
accuracy: 0.8022 - loss: 0.4890
Train Loss: 0.4672832489013672
Train Accuracy: 0.7846774458885193
Test Loss: 0.4767032861709595
Test Accuracy: 0.8083623647689819
Confusion Matrix:
[[108  23]
 [ 32 124]]
Classification Report
              precision    recall  f1-score   support

           0       0.77      0.82      0.80       131
           1       0.84      0.79      0.82       156

    accuracy                           0.81       287
   macro avg       0.81      0.81      0.81       287
```

```
weighted avg       0.81      0.81      0.81       287

ROC AUC Score: 0.8546
```




```
Prediction time: 0.1327 seconds
```

Metric	Value
Train Loss	0.4673
Train Accuracy	0.7847
Test Loss	0.4767
Test Accuracy	0.8084
Precision	0.81
Recall	0.81
F1-Score	0.81
ROC AUC	0.8546

	Predicted 0	Predicted 1
Actual 0	108	23
Actual 1	32	124

Interestingly enough, the testing accuracy resulted better than the training accuracy by a small margin. This is the best model yet since it has the best precision, recall, f1 score and ROC AUC score of all of the models trained previously.

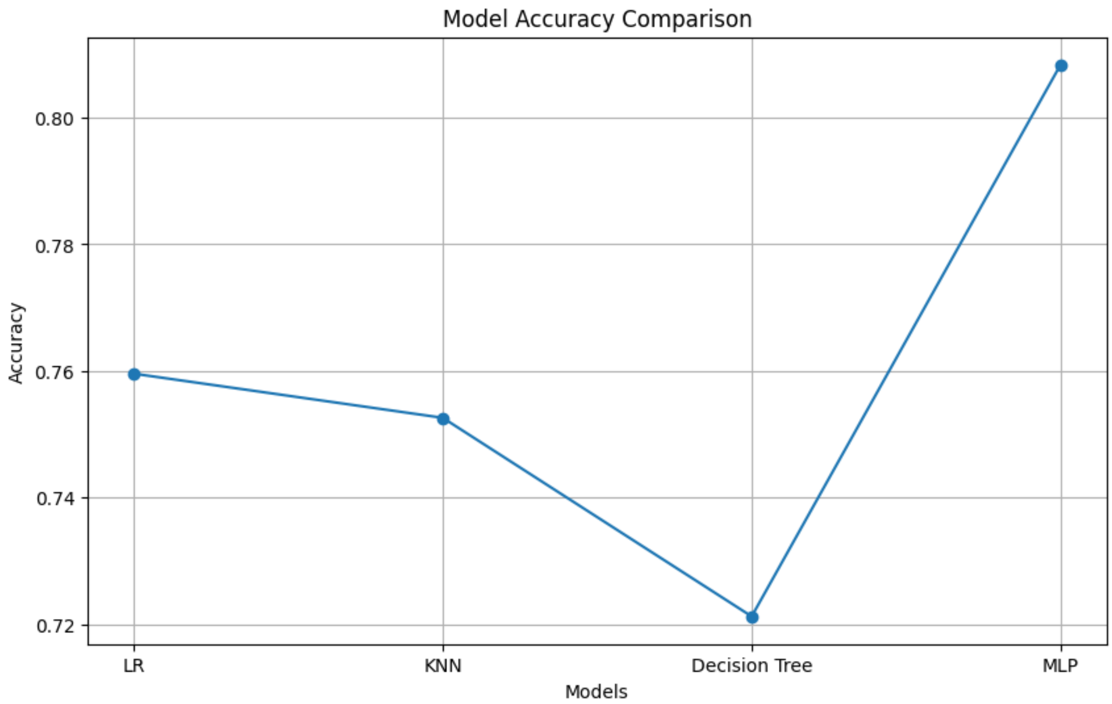
6 Discussion

6.1 Model Comparison

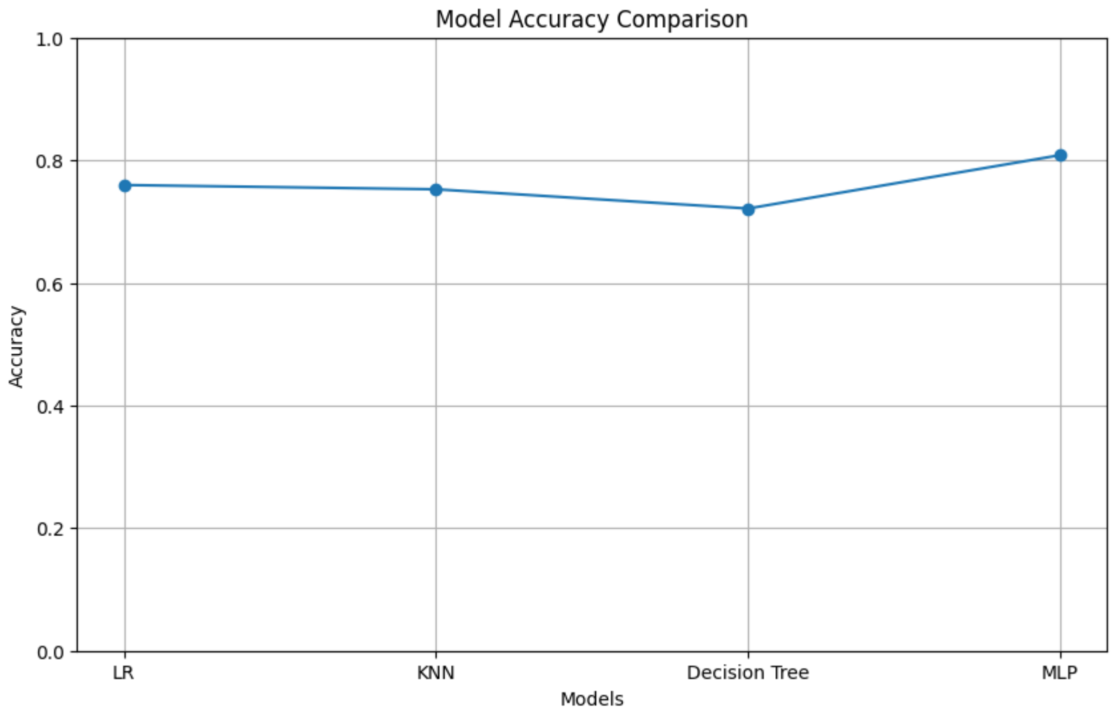
In order to visually compare the models' perfomance, a line plot will be shown to contrast the time and accuracy differences between models.

First up, the accuracy-comparing graph will be created:

[ ]:
```
models = ['LR', 'KNN', 'Decision Tree', 'MLP']
accuracies = [lr_accuracy, knn_accuracy, dt_accuracy, mlp_accuracy]

plt.figure(figsize=(10, 6))
plt.plot(models, accuracies, marker='o')
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.grid(True)
plt.show()
```

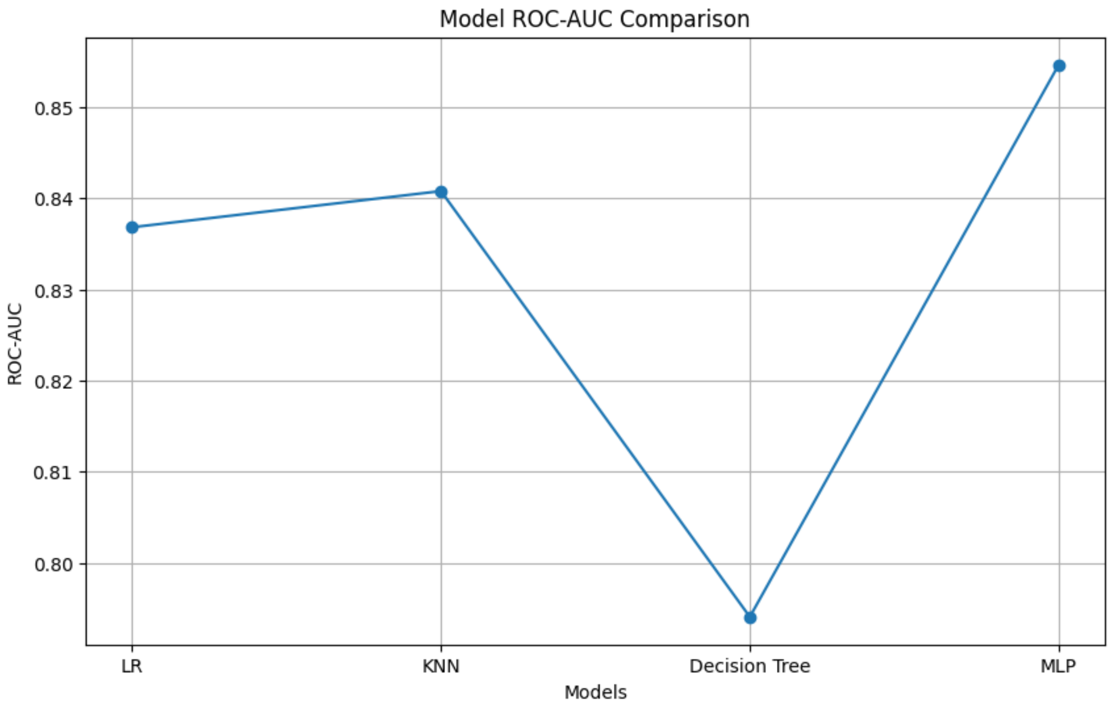


[ ]:
```
models = ['LR', 'KNN', 'Decision Tree', 'MLP']
accuracies = [lr_accuracy, knn_accuracy, dt_accuracy, mlp_accuracy]

plt.figure(figsize=(10, 6))
plt.plot(models, accuracies, marker='o')
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```



As the plot shows, the MLP has the highest accuracy with almost 80% on testing. Although accuracy can tell the overall performance, it is also useful to compare other mesurments, like ROC-AUC.

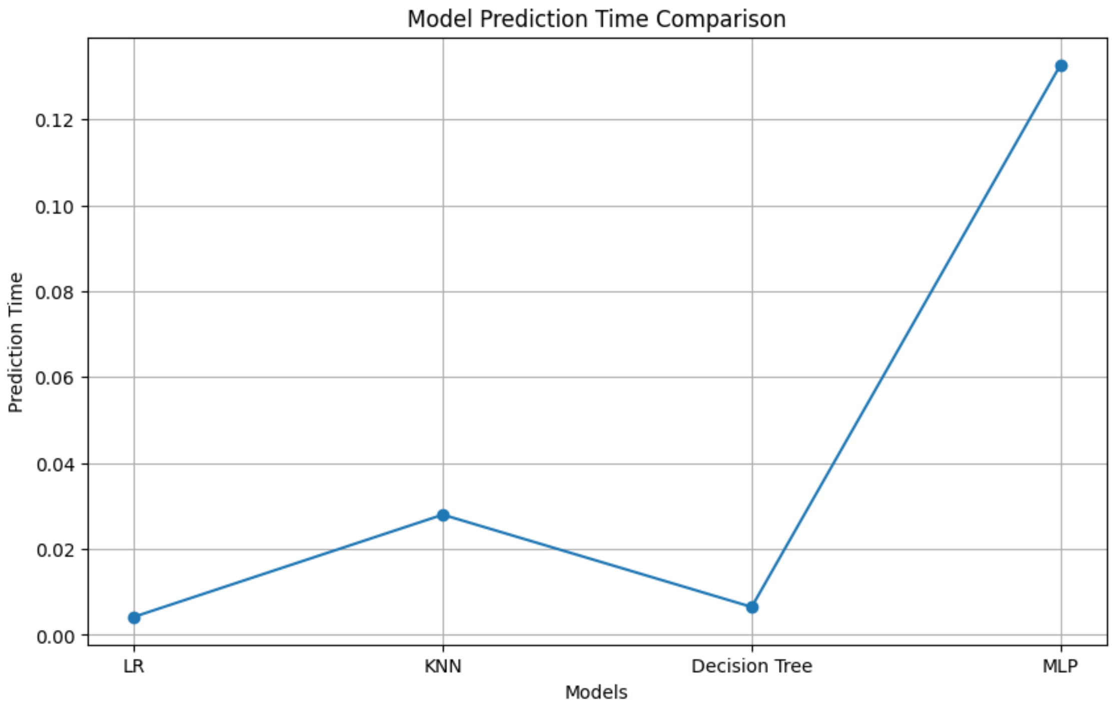
[ ]:
```
models = ['LR', 'KNN', 'Decision Tree', 'MLP']
accuracies = [lr_roc_auc, knn_roc_auc, dt_roc_auc, mlp_roc_auc]

plt.figure(figsize=(10, 6))
plt.plot(models, accuracies, marker='o')
plt.xlabel('Models')
plt.ylabel('ROC-AUC')
plt.title('Model ROC-AUC Comparison')
plt.grid(True)
plt.show()
```



As observed, in this case, the ROC-AUC values differ from the accuracy. This is especially noticeable with KNN, as its accuracy is lower than that of the Linear Regression, while its ROC-AUC score is higher. This indicates that, overall, the Linear Regression is a better classifier than KNN, though KNN produces fewer false positives.

Training the model takes time and in some cases this time cannot be reduced and some specific user might see this as a cost measurement. In practice, the user will not not how much time it took to train a specific model, but it will know how much it will take to use it. For this reason a new plot will be made in order to compare the prediction time of each model.

[ ]:
```
models = ['LR', 'KNN', 'Decision Tree', 'MLP']
accuracies = [lr_prediction_time, knn_prediction_time, dt_prediction_time,␣
↪mlp_prediction_time]

plt.figure(figsize=(10, 6))
plt.plot(models, accuracies, marker='o')
plt.xlabel('Models')
plt.ylabel('Prediction Time')
plt.title('Model Prediction Time Comparison')
plt.grid(True)
plt.show()
```



This plot shows that the model the took the most time to predict is MLP. This doesn't raise a particular concern, since the MLP prediction time value is around 0.12, an extremely low value for normal usage.

6.2 Model performance

Overall, MLP is the model that had the best performance in every metric that was used.

Remarkably, Logistic Regression is the best-performing model of the rest, meaning that it had the highest accuracy value (around 76%) apart from MLP, of course.

It is also imoprtant to note that while KNN is the second worst-performing model of them all, it still had the highest AUC-ROC value apart from MLP.

With this being said, it's still clear that the **best-performing model is the Multi-Layered Perceptron model**. This is because it outperforms every other model in every metric:

- Accuracy (0.0488 better than its nearest rival LR)
- Precision (0.04 better than its nearest rival LR)
- Recall (0.05 better than its nearest rival LR)
- F1-Score (0.05 better than its nearest rival LR)

6.3 Future improvements

Although our Decision Tree model didn't seem to overfit, the Random Forest model (model based on creating multiple decision trees) is usually considered an improvement over the simple Decision Tree model. It is important to keep in mind that implementing a Random Forest model implies losing interpretability of the model, since reading 50 or 100 decision trees isn't even an option to consider.

Another topic for improvement could be done in the MLP model. Since there are ways to join the performance of multiple models at once. This, with the possibility of generating an even better-performing model. The joining of multiple models into a single model is often called making an *ensemble model*, Random Forest is a clear example of an ensemble model.

7 Conclusion

In order to classify the wine quality in a binary manner, the Multi-Layered Perceptron model is the preferred over Logistic Regression, KNN and Decision Trees. The Multi-Layered Perceptron model managed to outperform the other three in every metric used in this study.

However, it is imporant to mention that all of the other models still outperformed a 0.7% threshold, which might be acceptable depending of the problem specification.

If using a neural network isn't a viable option for a specific implementation, then as a second recommendation the Logistic Regression model is endorsed.