جامعة الإمام عبد الرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY

CIS 517 Data Mining and Warehousing
First semester
2025/2026

Group 4 / Team 4

Student Name	Student ID
Refan Alanazi	2210003864
Raneem Almasrahy	2220004337
Deemah Albabtain	2210040217
Doja Alnemer	2220002855
Fatima Alhelal	2220001376
Huda Al Bazrun	2220006431

Instructor: **Ms. Arwa Almalki**

Submission Date: **7th December 2025**

Tasks distribution:

No.	Task	Member
1	Use different Python packages including pandas, numpy and other models to analyze the given data and evaluate the performance of the different models. Explain your code.	Refan Alanazi
2	Evaluate the dataset using different statistical methods and understand the nature of the data in different attributes.	Doja Alnemer
3	Apply data preprocessing to prepare the data for analysis by different models. Explain the steps you took to perform this task.	Raneem Almasrahy
4	Apply an outlier analysis solution to identify outliers among the data. Evaluate these outliers and explain the steps you have taken to address their presence.	Huda Al Bazron
5	Apply logistic regression to the dataset and the different performance enhancement techniques to fine tune your model. Apply decision trees and random forests to the data to classify the data in the optimal manner and tune the performance.	Fatima Alhelal
6	Compare the performance with other models you developed as applicable using techniques you learnt in the lab.	Deemah Albabtain

Project Overview:

The project goal is to apply various data mining techniques in Python to preprocess a dataset, classify it using different models, and evaluate their performance.

TASK 1:

Use Python Packages to Analyze the Data

1. Import all the required libraries to execute the code.

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

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
```

2. Loading the Dataset

The Code:

```
# to load the dataset
data = pd.read_csv("indian_liver_patient.csv")

# to display the first few rows
data.head()
```

The Output:

	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	Aspartate_Aminotransferase	Total_Protiens	Albumin	Albumin_and_Globulin_Ratio	Dataset
0	65	Female	0.7	0.1	187	16	18	6.8	3.3	0.90	1
1	62	Male	10.9	5.5	699	64	100	7.5	3.2	0.74	1
2	62	Male	7.3	4.1	490	60	68	7.0	3.3	0.89	1
3	58	Male	1.0	0.4	182	14	20	6.8	3.4	1.00	1
4	72	Male	3.9	2.0	195	27	59	7.3	2.4	0.40	1

The Explanation:

- pd.read_csv("indian_liver_patient.csv") Loads the dataset from a CSV file into a pandas DataFrame, allowing it to be viewed and manipulated.
- data.head() Displays the first five rows of the dataset. This provides a quick preview of the structure, column names, and sample values to ensure the data has loaded correctly.

3. Dataset Overview

3.1 The Code:

```
# to display dataset information
print("Dataset Info:\n")
data.info()

# to display summary statistics
print("\nSummary Statistics:\n")
data.describe()

# to check for missing values
print("\nMissing Values:\n")
data.isnull().sum()

# to check dataset shape
print("\nDataset Shape:", data.shape)
```

The Output:

```
Dataset Info:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583 entries, 0 to 582
Data columns (total 11 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   Age                         583 non-null    int64
 1   Gender                      583 non-null    object
 2   Total_Bilirubin             583 non-null    float64
 3   Direct_Bilirubin            583 non-null    float64
 4   Alkaline_Phosphotase        583 non-null    int64
 5   Alamine_Aminotransferase    583 non-null    int64
 6   Aspartate_Aminotransferase  583 non-null    int64
 7   Total_Protiens              583 non-null    float64
 8   Albumin                     583 non-null    float64
 9   Albumin_and_Globulin_Ratio  579 non-null    float64
 10  Dataset                     583 non-null    int64
dtypes: float64(5), int64(5), object(1)
memory usage: 50.2+ KB

Summary Statistics:


Missing Values:


Dataset Shape: (583, 11)
```

The Explanation:

- data.info() to display general information about the dataset, including column names, data types, and the count of non-null values.
- data.describe() to generate summary statistics for all numerical attributes.
- data.isnull().sum() checks each column for missing values and reports how many entries are missing.
- data.shape prints the total number of rows and columns in the dataset.

3.2 The Code:

```
data.isnull().sum()
```

The Output:

```
Age                           0
Gender                        0
Total_Bilirubin               0
Direct_Bilirubin              0
Alkaline_Phosphotase          0
Alamine_Aminotransferase      0
Aspartate_Aminotransferase    0
Total_Protiens                0
Albumin                       0
Albumin_and_Globulin_Ratio    4
Dataset                       0
dtype: int64
```

The Explanation:

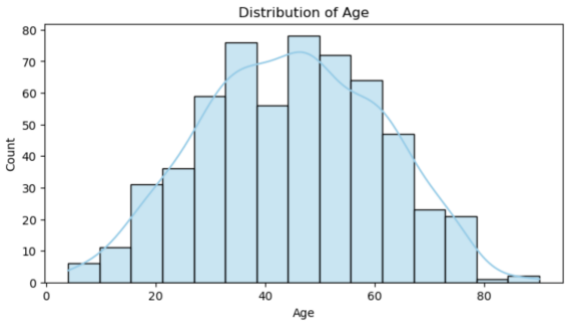
- data.isnull().sum() checks every column in the dataset and returns the total number of missing (null) values in each column.

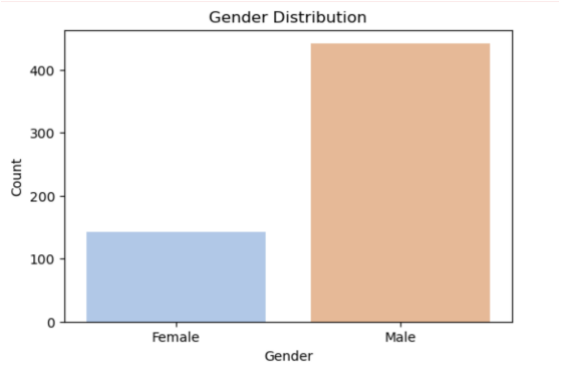
4. Visualization

The Code:

```
# Histogram for Age
plt.figure(figsize=(8,4))
sns.histplot(data['Age'], kde=True, color='skyblue')
plt.title("Distribution of Age")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()

# Countplot for the Gender
plt.figure(figsize=(6,4))
sns.countplot(x='Gender', data=data, palette='pastel')
plt.title("Gender Distribution")
plt.xlabel("Gender")
plt.ylabel("Count")
plt.show()
```

The Output:





The Explanation:

- sns.histplot(data['Age'], kde=True) creates a histogram to visualize the distribution of the Age column, showing how frequently each age range appears in the dataset.
- sns.countplot(x='Gender', data=data) generates a bar plot that displays how many male and female patients are in the dataset.

5. Data Preparation for Modeling

The Codes & Outputs:

```
[14]: data['Gender'] = data['Gender'].map({'Female': 0, 'Male': 1})

[15]: # Drop rows with missing values for model training
      model_data = data.dropna()

      # Check the new shape after dropping missing values
      model_data.shape

[15]: (579, 11)

[16]: # Define features (X) and target (y)
      X = model_data.drop('Dataset', axis=1)
      y = model_data['Dataset']

      # Display shapes of X and y
      X.shape, y.shape

[16]: ((579, 10), (579,))

[17]: # Split data into training and testing sets
      X_train, X_test, y_train, y_test = train_test_split(
          X, y, test_size=0.3, random_state=42
      )
      # Show the shapes of the new sets
      X_train.shape, X_test.shape, y_train.shape, y_test.shape

[17]: ((405, 10), (174, 10), (405,), (174,))
```

The Explanation:

In this code we have:

- Gender mapping - The Gender column was converted from text (Male/Female) into numeric values (1 and 0) so that machine learning models can process it.
- Dropping missing rows - Missing values were removed by creating a clean dataset (model_data) to avoid errors during model training.
- Defining X and Y - The features (X) and the target label (y) were separated, with “Dataset” chosen as the prediction target.
- Train-test split - The clean data was split into training (70%) and testing (30%) sets using train_test_split() to evaluate model performance properly.

6. Model Implementation & Performance Evaluation

6.1 Logistic Regression

The Code & Output:

```
[ ]: # Logistic Regression Model

     log_model = LogisticRegression(max_iter=1000)
     log_model.fit(X_train, y_train)

     # Make predictions
     y_pred_log = log_model.predict(X_test)

     # Calculate accuracy
     log_accuracy = accuracy_score(y_test, y_pred_log)
     log_accuracy

[ ]: 0.6609195402298851
```

The Explanation:

- A Logistic Regression model was created using LogisticRegression().
- The model was trained on the training set using fit(X_train, y_train).
- Predictions for the test set were generated using predict(X_test).
- The model’s performance was evaluated by calculating the accuracy using accuracy_score()

6.1 Decision Tree
The Code & Output:

```
# Decision Tree Classifier

dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

# Make predictions
y_pred_dt = dt_model.predict(X_test)

# Calculate accuracy
dt_accuracy = accuracy_score(y_test, y_pred_dt)
dt_accuracy
```

```
0.6839080459770115
```

The Explanation:

- A Decision Tree model was initialized using DecisionTreeClassifier().
- The model was trained on the same training data using fit().
- Test predictions were made using predict(X_test).
- Accuracy of the Decision Tree model was computed using accuracy_score() to compare performance.

In conclusion:

The Decision Tree achieved higher accuracy (0.6839) than Logistic Regression (0.6609), meaning it performed better on this dataset. This is expected because Decision Trees can capture nonlinear patterns, while Logistic Regression models only linear relationships. Overall, the Decision Tree provides a stronger baseline model for this dataset.

TASK 2:

Evaluate the dataset using different statistical methods and understand the nature of the data in different attributes.

1. Data Cleaning

The evaluation will start with cleaning data to get a good read of the overall data.

We performed data cleaning by:

1) Cleaning null values by using mean imputation by fillna() method, and checking that no null values remain using isnull() method.

2) Outliers will not be cleaned since we're going to apply decision trees which are robust to outliers.

The Code:

```python
#Cleaning data from null values:
data['Albumin_and_Globulin_Ratio'] = data['Albumin_and_Globulin_Ratio'].fillna(data['Albumin_and_Globulin_Ratio'].mean())
print("NULL COUNTS AFTER CLEANING NULL VALUES:")
print("-" * 55)
print(data.isnull().sum())
print("\n\n")
```

The Output:

```
NULL COUNTS AFTER CLEANING NULL VALUES:
-------------------------------------------------------
Age                         0
Gender                      0
Total_Bilirubin             0
Direct_Bilirubin            0
Alkaline_Phosphotase        0
Alamine_Aminotransferase    0
Aspartate_Aminotransferase  0
Total_Protiens              0
Albumin                     0
Albumin_and_Globulin_Ratio  0
Dataset                     0
dtype: int64
```

2. Applying Descriptive Statistics

Here we'll apply the 5 numbers summary statistics using describe() pandas DataFrame method. These statistics will provide a quick overview of the dataset.

The Code:

```python
#Apply describtive statistics
print("DESCRIBTIVE STATISTICS")
print("-"*55)
desc_stats = data.describe()
print(desc_stats)
print("\n\n")
```

The Output:

```
DESCRIBTIVE STATISTICS
-------------------------------------------------------
              Age      Gender  Total_Bilirubin  Direct_Bilirubin  \
count  583.000000  583.000000       583.000000        583.000000
mean    44.746141    0.756432         3.298799          1.486106
std     16.189833    0.429603         6.209522          2.808498
min      4.000000    0.000000         0.400000          0.100000
25%     33.000000    1.000000         0.800000          0.200000
50%     45.000000    1.000000         1.000000          0.300000
75%     58.000000    1.000000         2.600000          1.300000
max     90.000000    1.000000        75.000000         19.700000

       Alkaline_Phosphotase  Alamine_Aminotransferase  \
count            583.000000                583.000000
mean             290.576329                 80.713551
std              242.937989                182.620356
min               63.000000                 10.000000
25%              175.500000                 23.000000
50%              208.000000                 35.000000
75%              298.000000                 60.500000
max             2110.000000               2000.000000

       Aspartate_Aminotransferase  Total_Protiens     Albumin  \
count                  583.000000      583.000000  583.000000
mean                   109.910806        6.483190    3.141852
std                    288.918529        1.085451    0.795519
min                     10.000000        2.700000    0.900000
25%                     25.000000        5.800000    2.600000
50%                     42.000000        6.600000    3.100000
75%                     87.000000        7.200000    3.800000
max                   4929.000000        9.600000    5.500000

       Albumin_and_Globulin_Ratio     Dataset
count                  583.000000  583.000000
mean                     0.947064    1.286449
std                      0.318492    0.452490
min                      0.300000    1.000000
25%                      0.700000    1.000000
50%                      0.947064    1.000000
75%                      1.100000    2.000000
max                      2.800000    2.000000
```

3. Applying Central Tendency Statistics

Central tendency and dispersion statistics summarize the main characteristics of numeric variables, they provide deeper insight into the distribution of each attribute and complement the descriptive statistics.

- **Mean**: The average value of a variable.
- **Median**: The middle value, less sensitive to outliers.
- **Mode**: The most frequently occurring value.
- **Standard Deviation (Std Dev)**: Measures spread around the mean.
- **Variance**: Square of the standard deviation.
- **Range**: Difference between the maximum and minimum values.
- **Interquartile Range (IQR)**: Spread of the middle 50% of data.

The Code:

```
#Central Tendancy and dispersion statitics
print("CENTRAL TENDENCY & DISPERSION STATISTICS")
print("=" * 55)

for col in data.select_dtypes(include=[np.number]).columns.drop(['Dataset', 'Gender']):
    print("\n" + col + ":")
    print("  Mean:     " + str(round(data[col].mean(), 2)))
    print("  Median:   " + str(round(data[col].median(), 2)))
    print("  Mode:     " + str(round(data[col].mode()[0], 2)))
    print("  Std Dev:  " + str(round(data[col].std(), 2)))
    print("  Variance: " + str(round(data[col].var(), 2)))
    print("  Range:    " + str(round(data[col].max() - data[col].min(), 2)))
```

The Output:

```
CENTRAL TENDENCY & DISPERSION STATISTICS
=======================================================

Age:
  Mean:     44.75
  Median:   45.0
  Mode:     60
  Std Dev:  16.19
  Variance: 262.11
  Range:    86

Total_Bilirubin:
  Mean:     3.3
  Median:   1.0
  Mode:     0.8
  Std Dev:  6.21
  Variance: 38.56
  Range:    74.6

Direct_Bilirubin:
  Mean:     1.49
  Median:   0.3
  Mode:     0.2
  Std Dev:  2.81
  Variance: 7.89
  Range:    19.6

Alkaline_Phosphotase:
  Mean:     290.58
  Median:   208.0
  Mode:     198
  Std Dev:  242.94
  Variance: 59018.87
  Range:    2047

Alamine_Aminotransferase:
  Mean:     80.71
  Median:   35.0
  Mode:     25
  Std Dev:  182.62
  Variance: 33350.19
  Range:    1990

Aspartate_Aminotransferase:
  Mean:     109.91
  Median:   42.0
  Mode:     23
  Std Dev:  288.92
  Variance: 83473.92
  Range:    4919

Total_Protiens:
  Mean:     6.48
  Median:   6.6
  Mode:     7.0
  Std Dev:  1.09
  Variance: 1.18
  Range:    6.9

Albumin:
  Mean:     3.14
  Median:   3.1
  Mode:     3.0
  Std Dev:  0.8
  Variance: 0.63
  Range:    4.6

Albumin_and_Globulin_Ratio:
  Mean:     0.95
  Median:   0.95
  Mode:     1.0
  Std Dev:  0.32
  Variance: 0.1
  Range:    2.5
```

4. Variability analysis

we analyze variability or dispersion of the continuous numerical variables in the dataset using the coefficient of variation (CV).

CV = (Standard deviation / Mean) x 100%

The coefficient of variation helps us to compare variations of the various variables which may be having different units or different scales of measurement.

A large CV implies that there is more variability compared to the meaning that there is greater inconsistency or dispersion of the data, whereas a smaller CV implies that there is less inconsistency.

The Code:

```
#Variability analysis
print("VARIABILITY ANALYSIS      (ignoring Gender and Dataset)")
print("=" * 55)
print("\nCoefficient of Variation (CV = std/mean * 100%):")
cv = (data.std() / data.mean() * 100).sort_values(ascending=False)
print(cv)
```

The Output:

```
VARIABILITY ANALYSIS      (ignoring Gender and Dataset)
=======================================================

Coefficient of Variation (CV = std/mean * 100%):
Aspartate_Aminotransferase    262.866354
Alamine_Aminotransferase      226.257369
Direct_Bilirubin              188.983623
Total_Bilirubin               188.235814
Alkaline_Phosphotase           83.605568
Gender                         56.793373
Age                            36.181519
Dataset                        35.173568
Albumin_and_Globulin_Ratio     33.629405
Albumin                        25.320056
Total_Protiens                 16.742551
dtype: float64
```

Here we notice that Aspartate Aminotransferase has the highest variability.

5. Data Visualization

Visualizing each dataset attribute data, nominal data with count plots and interval/ratio data with histograms and boxplots. Exploring correlations using scatter plots and a heatmap.

5.1 Histograms

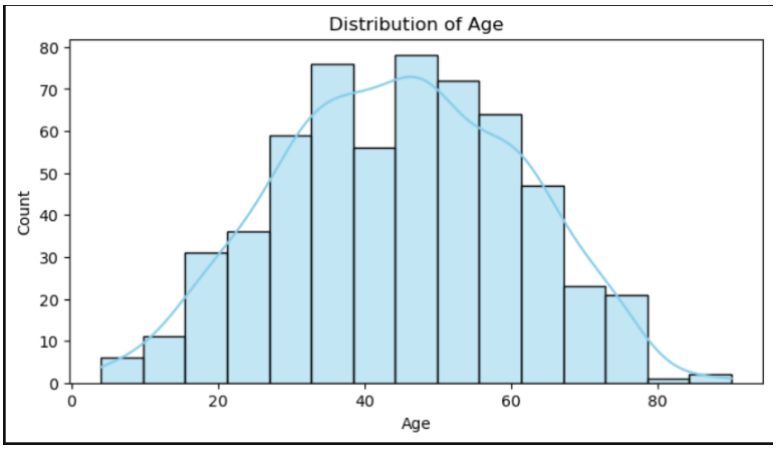
The Code:

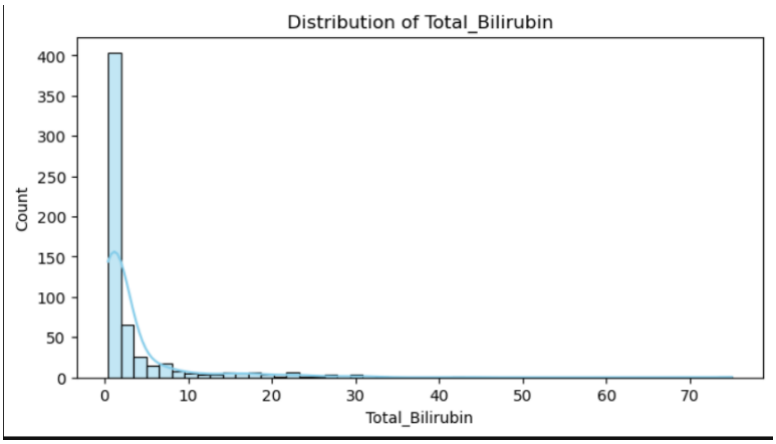
```python
#VISULIZE DATA/ histograms
#---------------
def make_histogram(att):
    plt.figure(figsize=(8,4))
    sns.histplot(data[att], kde=True, color='skyblue')
    plt.title("Distribution of " + att)
    plt.xlabel(att)
    plt.ylabel("Count")
    plt.show()
    return plt.show()

make_histogram("Age")
print("\n")
make_histogram("Total_Bilirubin")
print("\n")
make_histogram("Direct_Bilirubin")
print("\n")
make_histogram("Alkaline_Phosphotase")
print("\n")
make_histogram("Alamine_Aminotransferase")
print("\n")
make_histogram("Aspartate_Aminotransferase")
print("\n")
make_histogram("Total_Protiens")
print("\n")
make_histogram("Albumin")
print("\n")
make_histogram("Albumin_and_Globulin_Ratio")
```

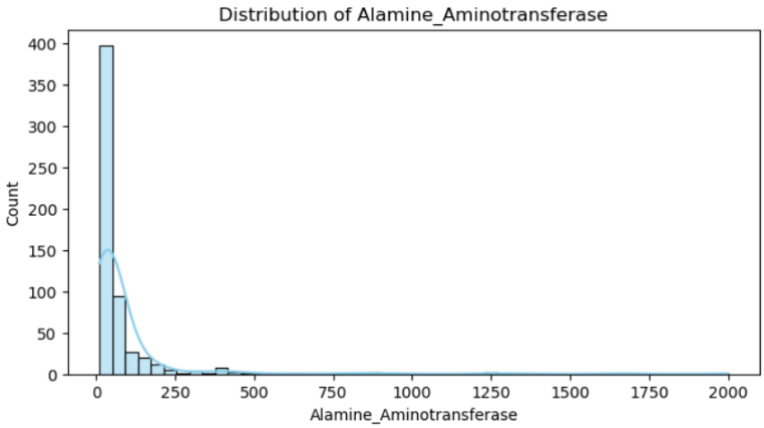
```python
#VISULIZE DATA/ Box plots
def make_boxplot(att):
    plt.figure(figsize=(6, 4))
    sns.boxplot(y=data[att])
    plt.title("Box Plot of " + att, fontweight="bold")
    plt.ylabel(att)
    return plt.show()

make_boxplot("Age")
print("\n")
make_boxplot("Total_Bilirubin")
print("\n")
make_boxplot("Direct_Bilirubin")
print("\n")
make_boxplot("Alkaline_Phosphotase")
print("\n")
make_boxplot("Alamine_Aminotransferase")
print("\n")
make_boxplot("Aspartate_Aminotransferase")
print("\n")
make_boxplot("Total_Protiens")
print("\n")
make_boxplot("Albumin")
print("\n")
make_boxplot("Albumin_and_Globulin_Ratio")
```

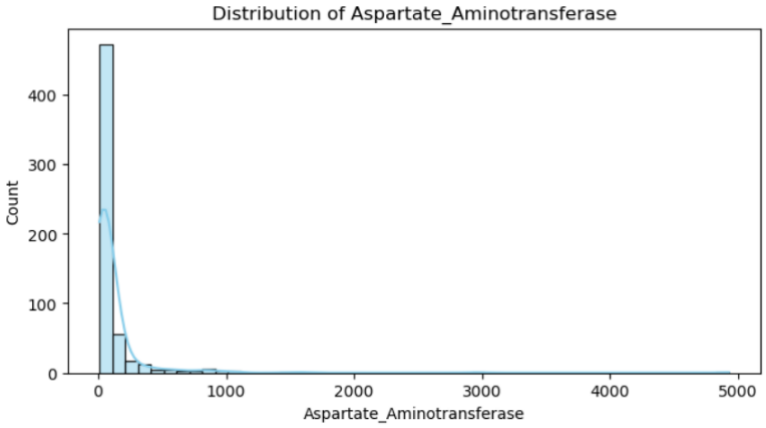
```python
#VISULIZE DATA/ Count plots
#GENDER
plt.figure(figsize=(8, 4))
plt.title('Gender Distribution Count Plot', fontweight='bold')
sns.countplot(x='Gender', data=data)
plt.xlabel('Gender')
plt.ylabel('Count')
plt.suptitle(' Female = 0, Male = 1', fontsize=10, style='italic')
plt.show()
#Dataset
plt.figure(figsize=(8, 4))
plt.title('Data set number Count Plot', fontweight='bold')
sns.countplot(x='Dataset', data=data)
plt.xlabel('Dataset')
plt.ylabel('Number of occurances')
plt.show()
```

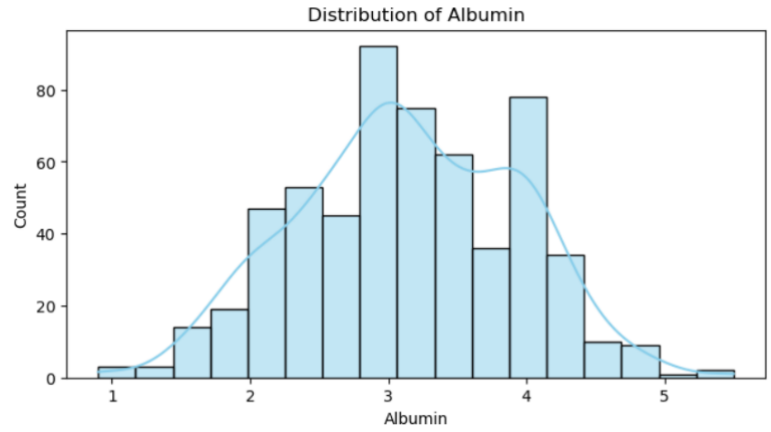
The Output (including only important graphs):

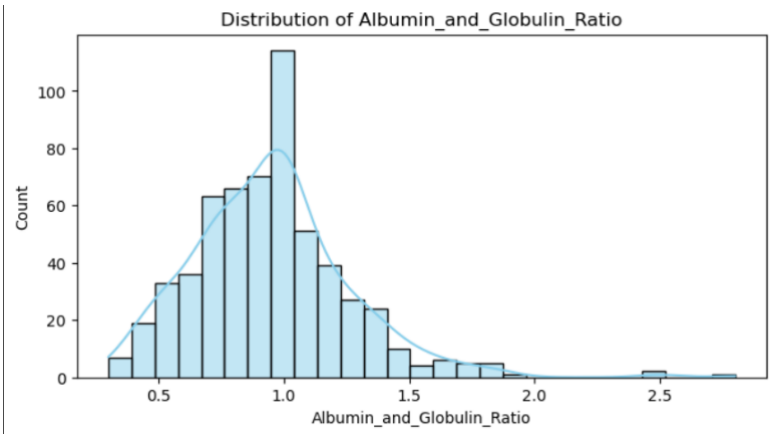

Distribution of Age
Count
Age
0
10
20
30
40
50
60
70
80
0
20
40
60
80


Distribution of Total_Bilirubin
Count
Total_Bilirubin
0
50
100
150
200
250
300
350
400
0
10
20
30
40
50
60
70


Distribution of Alamine_Aminotransferase
Count
Alamine_Aminotransferase
0
50
100
150
200
250
300
350
400
0
250
500
750
1000
1250
1500
1750
2000


Distribution of Aspartate_Aminotransferase
Count
Aspartate_Aminotransferase
0
100
200
300
400
0
1000
2000
3000
4000
5000





5.1.1. Applying Skewness Analysis

To evaluate the symmetry of that data in the distribution, we'll observe the skewness where:

- Approximately symmetric: |skewness| < 0.5
- Moderately skewed: 0.5 ≤ |skewness| < 1
- Highly skewed: |skewness| ≥ 1

A positive skew means that there is a long tail on the right whereas negative skew means that there is a long tail on the left. The concept of skewness is applicable in identifying the necessity of data transformations or special statistics tools.

The Code:

```python
#Apply skewness analysis
skewness = data.skew()
print("SKEWNESS VALUES:")
print("-"*55)
print(skewness)
print("\n\n")

print("INTERPRETATION:")
print("-"*55)
for col, skew_val in skewness.items():
    if abs(skew_val) < 0.5:
        print("Approximately symmetric")
    elif abs(skew_val) < 1:
        print("Moderately skewed")
    else:
        print("Highly skewed")
```

The Output:

```
SKEWNESS VALUES:
-------------------------------------------------------
Age                          -0.029385
Gender                       -1.197919
Total_Bilirubin               4.907474
Direct_Bilirubin              3.212403
Alkaline_Phosphotase          3.765106
Alamine_Aminotransferase      6.549192
Aspartate_Aminotransferase   10.546177
Total_Protiens               -0.285672
Albumin                      -0.043685
Albumin_and_Globulin_Ratio    0.995703
Dataset                       0.947140
dtype: float64


INTERPRETATION:
-------------------------------------------------------
Approximately symmetric
Highly skewed
Highly skewed
Highly skewed
Highly skewed
Highly skewed
Highly skewed
Approximately symmetric
Approximately symmetric
Moderately skewed
Moderately skewed
```

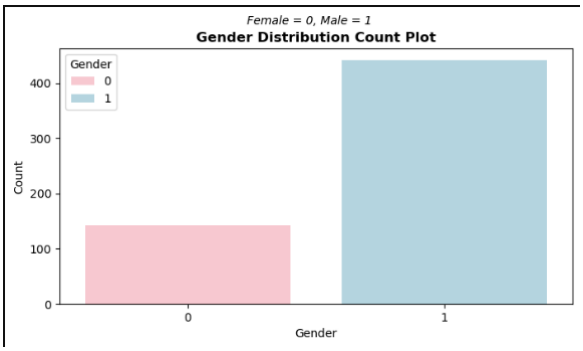
We notice that most attributes are highly skewed except for distributions like Albumin and Total_Protiens.

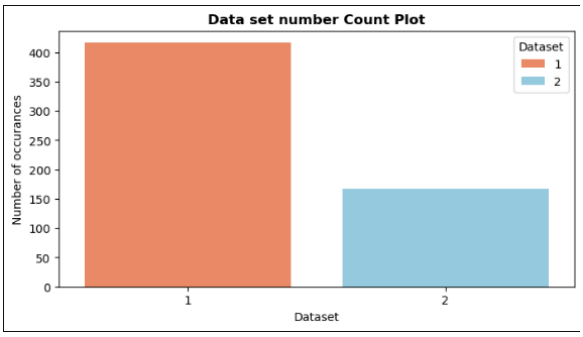
5.2 Count plots

The Code:

```
#VISUALIZE DATA / Count plots
#GENDER
plt.figure(figsize=(8, 4))
plt.title('Gender Distribution Count Plot', fontweight='bold')
sns.countplot(x='Gender', data=data, hue='Gender', palette=['pink', 'lightblue'])
plt.xlabel('Gender')
plt.ylabel('Count')
plt.suptitle(' Female = 0, Male = 1', fontsize=10, style='italic')
plt.show()
print("\n")
#Dataset
plt.figure(figsize=(8, 4))
plt.title('Data set number Count Plot', fontweight='bold')
sns.countplot(x='Dataset', data=data, hue='Dataset', palette=['coral', 'skyblue'])
plt.xlabel('Dataset')
plt.ylabel('Number of occurances')
plt.show()
```

The Output:





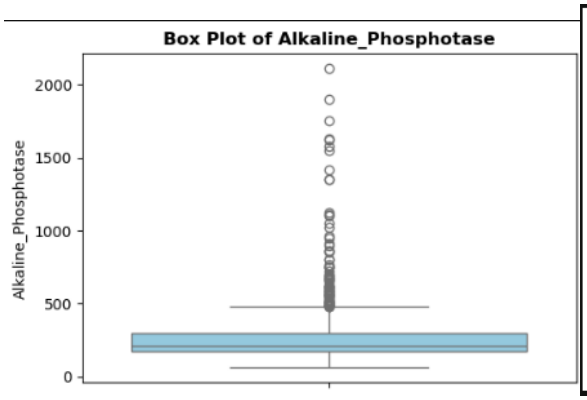
We notice that most of the patients are males, we also notice that most of the data in the dataset are from data set 1 which we assume means liver patients.

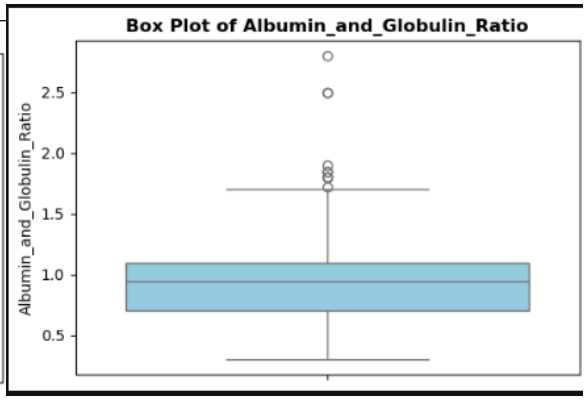
5.3 Box plots

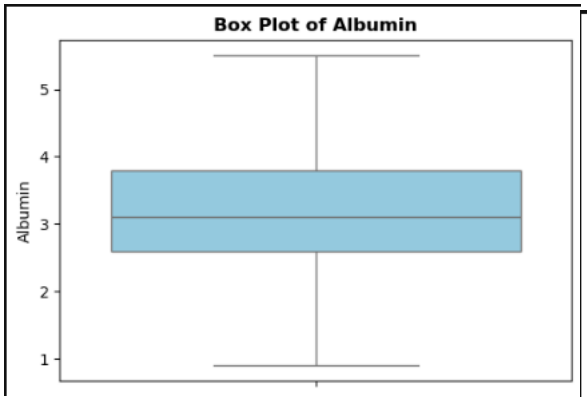
The Code:

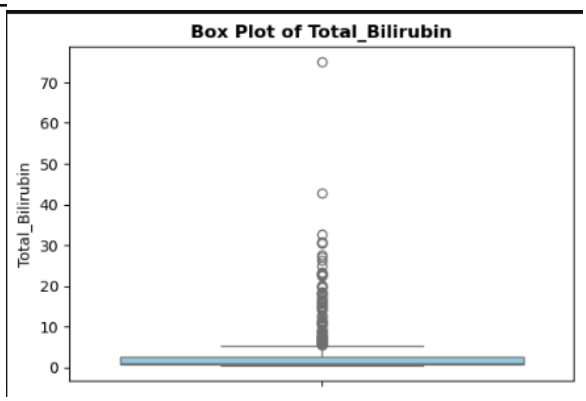
```python
#VISULIZE DATA/ Box plots
def make_boxplot(att):
    plt.figure(figsize=(6, 4))
    sns.boxplot(y=data[att], color="skyblue")
    plt.title("Box Plot of " + att, fontweight="bold")
    plt.ylabel(att)
    return plt.show()

make_boxplot("Age")
print("\n")
make_boxplot("Total_Bilirubin")
print("\n")
make_boxplot("Direct_Bilirubin")
print("\n")
make_boxplot("Alkaline_Phosphotase")
print("\n")
make_boxplot("Alamine_Aminotransferase")
print("\n")
make_boxplot("Aspartate_Aminotransferase")
print("\n")
make_boxplot("Total_Protiens")
print("\n")
make_boxplot("Albumin")
print("\n")
make_boxplot("Albumin_and_Globulin_Ratio")
```

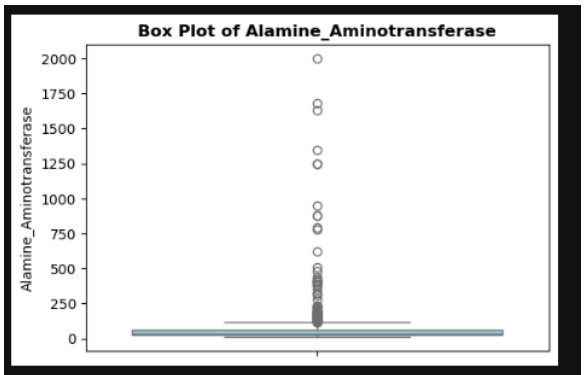
The Output (including only important graphs):

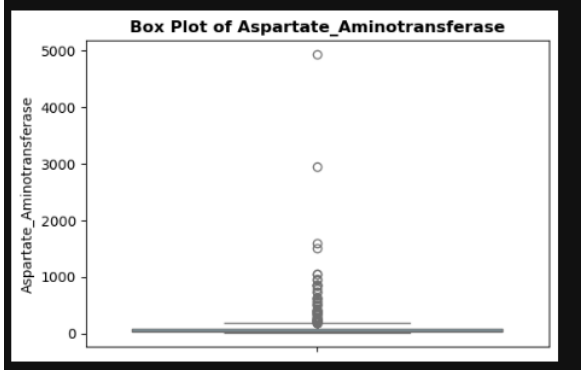












5.3.1 Range and spread Analysis

The Code:

```
print("RANGE AND SPREAD ANALYSIS")
print("=" * 55)
ranges = pd.DataFrame({
    'Min': data.min(),
    'Max': data.max(),
    'Range': data.max() - data.min(),
    'IQR': data.quantile(0.75) - data.quantile(0.25)
})
print(ranges)
```

The Output:

```
RANGE AND SPREAD ANALYSIS
=======================================================
                              Min     Max   Range    IQR
Age                           4.0    90.0    86.0   25.0
Gender                        0.0     1.0     1.0    0.0
Total_Bilirubin               0.4    75.0    74.6    1.8
Direct_Bilirubin              0.1    19.7    19.6    1.1
Alkaline_Phosphotase         63.0  2110.0  2047.0  122.5
Alamine_Aminotransferase     10.0  2000.0  1990.0   37.5
Aspartate_Aminotransferase   10.0  4929.0  4919.0   62.0
Total_Protiens                2.7     9.6     6.9    1.4
Albumin                       0.9     5.5     4.6    1.2
Albumin_and_Globulin_Ratio    0.3     2.8     2.5    0.4
Dataset                       1.0     2.0     1.0    1.0
```

The box plots show significant outliers in liver enzymes and the range analysis indicates that the dispersion is high, especially in Aspartate Aminotransferase. This confirmed the fact that there were extreme values, and this influenced the choice of using Decision Trees as a model.

5.4 Scatter plots

Scatter plots are applied to present the association between two continuous variables and determine the possible linear or non linear association.

Here we examine some of the key relationships between liver function markers:

1) Total_Bilirubin vs Direct_Bilirubin

2) Age vs. Total_Bilirubin (colored by Dataset)

3) Albumin vs. Albumin_and_Globulin_Ratio

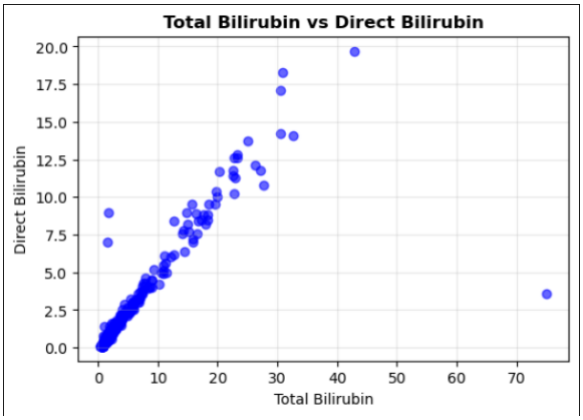
4) Alamine_Aminotransferasevs Aspartate_Aminotransferase

5) Total_Protiens vs Albumin

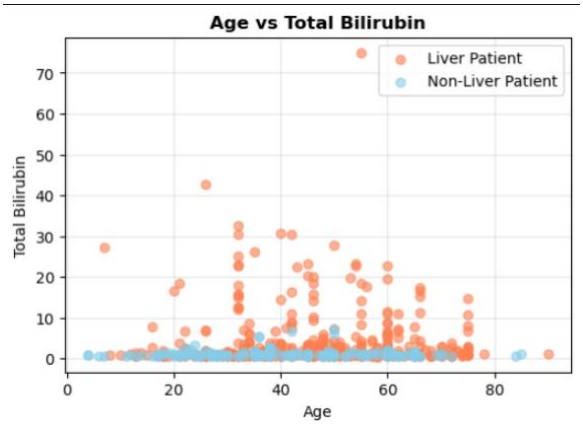
The Code:

```
#VISULIZE DATA/ Scatter plots
# 1) Total_Bilirubin vs Direct_Bilirubin
plt.figure(figsize=(6, 4))
plt.scatter(data['Total_Bilirubin'], data['Direct_Bilirubin'], alpha=0.6, color='blue')
plt.title('Total Bilirubin vs Direct Bilirubin', fontweight='bold')
plt.xlabel('Total Bilirubin')
plt.ylabel('Direct Bilirubin')
plt.grid(True, alpha=0.3)
plt.show()
print("\n")
# 2) Age vs. Total_Bilirubin (colored by Dataset)
plt.figure(figsize=(6, 4))
colors = ['coral', 'skyblue']
for i, dataset in enumerate([1, 2]):
    subset = data[data['Dataset'] == dataset]
    plt.scatter(subset['Age'], subset['Total_Bilirubin'],
                alpha=0.6, color=colors[i],
                label='Liver Patient' if dataset == 1 else 'Non-Liver Patient')
plt.title('Age vs Total Bilirubin', fontweight='bold')
plt.xlabel('Age')
plt.ylabel('Total Bilirubin')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
print("\n")
# 3) Albumin vs. Albumin_and_Globulin_Ratio
plt.figure(figsize=(6, 4))
plt.scatter(data['Albumin'], data['Albumin_and_Globulin_Ratio'], alpha=0.6, color='purple')
plt.title('Albumin vs Albumin and Globulin Ratio', fontweight='bold')
plt.xlabel('Albumin')
plt.ylabel('Albumin and Globulin Ratio')
plt.grid(True, alpha=0.3)
plt.show()
print("\n")
# 4) (Alamine_Aminotransferase) vs (Aspartate_Aminotransferase)
plt.figure(figsize=(6, 4))
plt.scatter(data['Alamine_Aminotransferase'], data['Aspartate_Aminotransferase'],
            alpha=0.6, color='plum')
plt.title('Alamine_Aminotransferase vs Aspartate_Aminotransferase', fontweight='bold')
plt.xlabel('(Alamine Aminotransferase)')
plt.ylabel('(Aspartate Aminotransferase)')
plt.grid(True, alpha=0.3)
plt.show()
print("\n")
# 5) Total_Protiens vs Albumin
plt.figure(figsize=(6, 4))
plt.scatter(data['Total_Protiens'], data['Albumin'], alpha=0.6, color='pink')
plt.title('Total Proteins vs Albumin', fontweight='bold')
plt.xlabel('Total Proteins')
plt.ylabel('Albumin')
plt.grid(True, alpha=0.3)
plt.show()
```

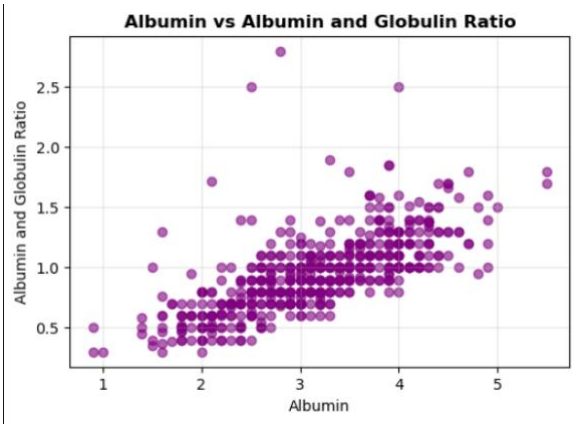
The Output:



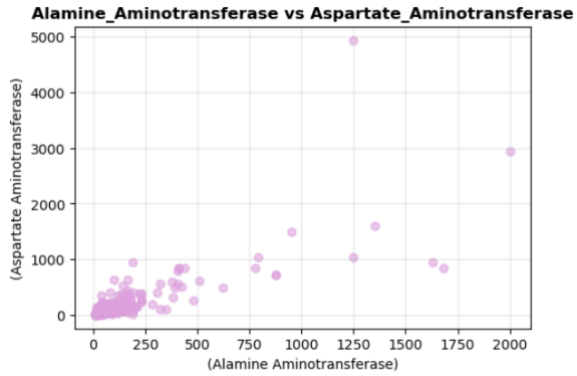
This plot shows a strong positive linear relationship between total and direct bilirubin, which is normal since direct bilirubin is a part of total bilirubin. However, any of the outliers that deviate beyond the trend could indicate an indirect accumulation of bilirubin.



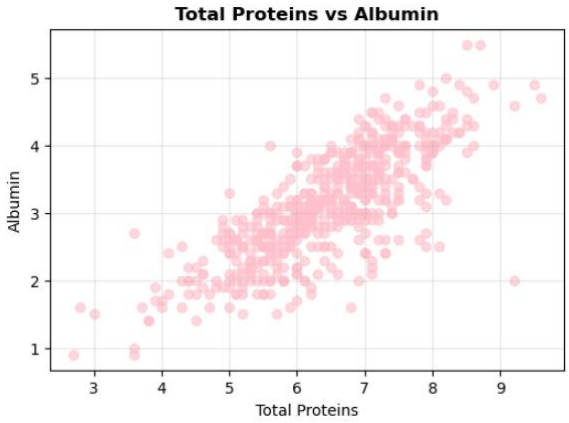
This plot shows the correlation between age of the patient and the bilirubin, and points are colored to differentiate liver patients (coral) from data set 1 and non-liver patients (sky blue) from data set 2. It assists in determining age trends in liver disease.



This plot shows the correlation between albumin and the Albumin & Globulin ratio. The correlation tends to be positive. The identified patients with abnormal protein balance can be determined with the help of the plot because the patients with low albumin and low Albumin & Globulin ratios usually reflect chronic liver disease.



This plot shows the correlation between two key liver enzymes. It shows a positive correlation which means that when one enzyme is elevated, the other tends to be elevated as well.



This plot shows a positive correlation between Albumin and Total Proteins. This correlation is expected since Album is a protein which is a part of Total proteins. However, any of the outliers that deviate beyond the trend could be indicate some protein disorders associated with liver diseases.

5.5 Heat map

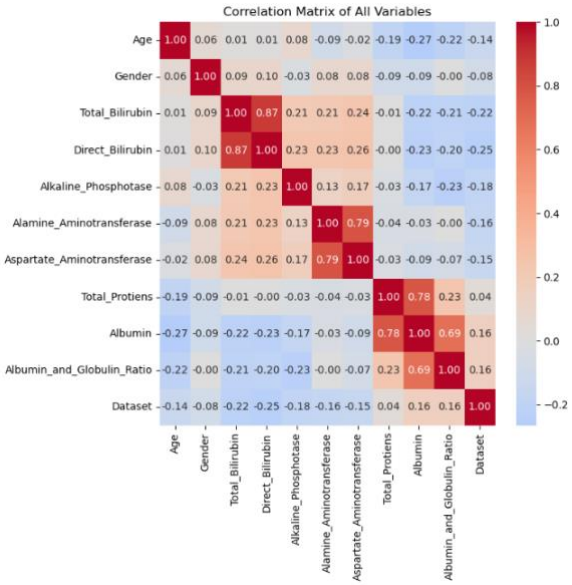
To further more explore correlations, we used a correlation matrix. It indicates that there are some important insights to the nature of the data like:

- Detection of Multicollinearity: There are clear groups of high correlated independent variables. Interestingly, Total Bilirubin and Direct Bilirubin have a strong positive correlation (0.87), Aspartate Aminotransferase and Aspartate Aminotransferase (0.79) also have positive correlations. This means that these pairs of features have redundant information and dimensionality reduction (or dropping one of the pairs) can be useful to the model.
- Protein Relationships: There is a significant relationship (0.78) between Total Proteins and Albumin which is biologically relevant since albumin is a big part of the total protein.

The Code:

```
# Correlation: Heatmap
correlation_matrix = data.corr()
plt.figure(figsize=(8, 8))
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm', center=0)
plt.title('Correlation Matrix of All Variables')
plt.tight_layout()
plt.show()
```

The Output:



TASK 3:

The purpose of this task is to clean and prepare the dataset so it can be used reliably for classification models. This involves removing duplicate rows, checking for missing values, creating the target label, selecting features, and scaling the numerical attributes.

1. Removing Duplicate Records

The Code:

```
preprocessed_data = data.copy()  # work on a copy so we don't change previous tasks

print("Shape BEFORE removing duplicates:", preprocessed_data.shape)
print("Number of duplicate rows:", preprocessed_data.duplicated().sum())

# Remove duplicate rows if there are any
preprocessed_data = preprocessed_data.drop_duplicates()

print("\nShape AFTER removing duplicates:", preprocessed_data.shape)
```

The Output:

```
Shape BEFORE removing duplicates: (583, 11)
Number of duplicate rows: 13

Shape AFTER removing duplicates: (570, 11)
```

The Explanation:

- The duplicated() function was used to check for repeated rows in the dataset.
- A total of 13 duplicates were found and removed using drop_duplicates().

- After removing them, the number of rows changed from 583 to 570.

2. Checking Missing Values

The Code:

```
print("\nMissing values in each column (after previous cleaning):")
print(preprocessed_data.isnull().sum())
```

The Output:

```
Missing values in each column (after previous cleaning):
Age                           0
Gender                        0
Total_Bilirubin               0
Direct_Bilirubin              0
Alkaline_Phosphotase          0
Alamine_Aminotransferase      0
Aspartate_Aminotransferase    0
Total_Protiens                0
Albumin                       0
Albumin_and_Globulin_Ratio    0
Dataset                       0
dtype: int64
```

The Explanation:

- The isnull().sum() function was used to check for missing values in each column.
- All columns showed **0 missing values**, so no filling or removal was needed.
- This confirms that the dataset is complete and ready for further preprocessing.

3. Creating the Target Variable & Splitting Features and Target

The Code:

```
# 1. Create the new binary target column (1 = disease, 0 = no disease)
preprocessed_data['Liver_Disease'] = preprocessed_data['Dataset'].replace({1: 1, 2: 0}).astype('category')

# 2. Remove the old Dataset column completely
preprocessed_data = preprocessed_data.drop('Dataset', axis=1)

# 3. Split X and y
X = preprocessed_data.drop('Liver_Disease', axis=1)
y = preprocessed_data['Liver_Disease']

# Print heads to confirm
print("X HEAD:")
print(X.head())

print("\nY HEAD:")
print(y.head())
```

The Output:

```
X HEAD:
   Age  Gender  Total_Bilirubin  Direct_Bilirubin  Alkaline_Phosphotase  \
0   65       0              0.7               0.1                   187
1   62       1             10.9               5.5                   699
2   62       1              7.3               4.1                   490
3   58       1              1.0               0.4                   182
4   72       1              3.9               2.0                   195

   Alamine_Aminotransferase  Aspartate_Aminotransferase  Total_Protiens  \
0                        16                          18             6.8
1                        64                         100             7.5
2                        60                          68             7.0
3                        14                          20             6.8
4                        27                          59             7.3

   Albumin  Albumin_and_Globulin_Ratio
0      3.3                        0.90
1      3.2                        0.74
2      3.3                        0.89
3      3.4                        1.00
4      2.4                        0.40

Y HEAD:
0    1
1    1
...
3    1
4    1
Name: Liver_Disease, dtype: category
Categories (2, int64): [0, 1]
```

The Explanation:

- The original **Dataset** column included values (1 and 2) that indicate whether the patient has liver disease.
- A new target column called **Liver_Disease** was created by converting those values to a clear binary form:
  1 = liver disease, 0 = no liver disease.
- After creating the new target column, the old **Dataset** column was removed because it is no longer needed.
- The dataset was then split into:
- **X** → all feature columns (independent variables)
- **y** → the target column **Liver_Disease**
- Printing X.head() and y.head() confirms that the split is correct and that the target column is separate from the features.

4. Scaling Features & Train/Test Split

The Code:

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import pandas as pd

print("Applying StandardScaler to feature columns...\n")

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Put scaled data back into a DataFrame (just for nicer display)
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

print("First 5 rows of the scaled features:")
print(X_scaled_df.head())

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled_df, y, test_size=0.2, random_state=42
)

print("\n----- SHAPES AFTER PREPROCESSING -----")
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

The Output:

```
First 5 rows of the scaled features:
        Age    Gender  Total_Bilirubin  Direct_Bilirubin  \
0  1.241741 -1.752549        -0.418647         -0.493702
1  1.056874  0.570597         1.210111          1.413923
2  1.056874  0.570597         0.635255          0.919354
3  0.810385  0.570597        -0.370743         -0.387723
4  1.673096  0.570597         0.092336          0.177500

   Alkaline_Phosphotase  Alamine_Aminotransferase  Aspartate_Aminotransferase  \
0             -0.427421                 -0.351482                   -0.314428
1              1.661722                 -0.086746                   -0.032278
2              0.808927                 -0.108807                   -0.142385
3             -0.447823                 -0.362513                   -0.307546
4             -0.394778                 -0.290813                   -0.173353

   Total_Protiens   Albumin  Albumin_and_Globulin_Ratio
0        0.279290  0.189737                   -0.150824
1        0.923059  0.064127                   -0.653605
2        0.463224  0.189737                   -0.182248
3        0.279290  0.315348                    0.163413
4        0.739125 -0.940754                   -1.722013
```

```
----- SHAPES AFTER PREPROCESSING -----
X_train shape: (456, 10)
X_test shape: (114, 10)
y_train shape: (456,)
y_test shape: (114,)
```

The Explanation:

- StandardScaler was applied to the numerical columns so that all features are on a similar scale.
  This helps the model learn better and prevents any single feature from affecting the results too much.
- After scaling, the data was split into training and testing sets:
- 80% for training the model
- 20% for testing its performance
- This prepares the dataset for the modeling stage and ensures a fair evaluation later.

TASK 4:

This task purpose is to detect the outliers in the data and choose method to handle them. So it will not affect next tasks which is implementing the model because the models results can be highly effected by the outliers.

Step 1: Identifying the numerical columns on the data

The code:

```
[39]: #---Task 4 ---
      #to identify numerical columns
      num_cols = preprocessed_data.select_dtypes(include=['float64','int64']).columns
      num_cols = num_cols.drop(['Age'])
      num_cols
```

The output:

```
]: Index(['Total_Bilirubin', 'Direct_Bilirubin', 'Alkaline_Phosphotase',
          'Alamine_Aminotransferase', 'Aspartate_Aminotransferase',
          'Total_Protiens', 'Albumin', 'Albumin_and_Globulin_Ratio'],
         dtype='object')
```

The Explanation: Only numerical columns can be detected as an outlier but we excluded the Age column because Age can't be an outlier.

Step 2: Using IQR method to detect outliers.

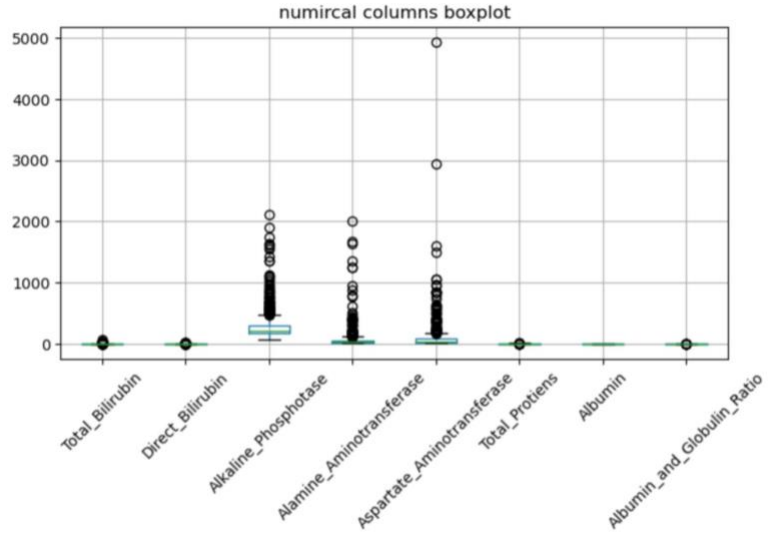
The code:

```
[40]: #using IQR to detect outliers
Q1 = preprocessed_data[num_cols].quantile(0.25)
Q3 = preprocessed_data[num_cols].quantile(0.75)
IQR = Q3 - Q1

lower = Q1 - 1.5*IQR
upper = Q3 + 1.5*IQR
  #number of rows containig outliers
outlier_mask = (preprocessed_data[num_cols] < lower ) | (preprocessed_data[num_cols] > upper )
outliers = preprocessed_data[outlier_mask.any(axis = 1)]
len(outliers)
```

The Output:

The Explanation: after using IQR to detect the outliers an outlier mask has been applied in order to know the number of rows that containing outliers. We find that compared to the size of the data the rows containing outliers is considerably high, so we decided not to remove the rows but to use winsorizing method.

Step 3: Displaying the outliers per column in a box plot



T

The Explanation:

Displaying the outliers per column as boxplot, we noticed that the thee columns Alkaline phosphate, Alamine aminotransferase, and Aspartate aminotransferase are the highest columns to have outliers which may be an indication to have liver disease. This helped in making the decision of how to handle the outliers too.

Step 4:Applying the winsorizing on the data

The code

```
[43]: #handeling the outliers using winsorizing(capping)
data_capped = preprocessed_data.copy()

for col in num_cols:
    lower_bound = Q1[col] - 1.5*IQR[col]
    upper_bound = Q3[col] + 1.5*IQR[col]

    data_capped[col] = preprocessed_data[col].clip(lower_bound, upper_bound)
```

Step 5: comparing the columns before and after the handling

Columns before:

```
[41]: #number of outliers per column
outlier_count = outlier_mask.sum()
outlier_count
```

```
[41]: Total_Bilirubin              83
      Direct_Bilirubin             80
      Alkaline_Phosphotase         69
      Alamine_Aminotransferase     72
      Aspartate_Aminotransferase   66
      Total_Protiens                8
      Albumin                       0
      Albumin_and_Globulin_Ratio   10
      dtype: int64
```

Columns After:

```
#outliers count after the winsorizing
outlier_mask_after = (
    (data_capped[num_cols] < (Q1 - 1.5 * IQR)) | (data_capped[num_cols] > (Q3 + 1.5 * IQR) ))
outlier_mask_after.sum()
```

```
[43]: Total_Bilirubin              0
      Direct_Bilirubin             0
      Alkaline_Phosphotase         0
      Alamine_Aminotransferase     0
      Aspartate_Aminotransferase   0
      Total_Protiens               0
      Albumin                      0
      Albumin_and_Globulin_Ratio   0
      dtype: int64
```

TASK 5:

Model Implementation (Logistic Regression, Decision Tree, Random Forest)

This task focuses on applying three classification models, Logistic Regression, Decision Tree, and Random Forest to the cleaned and preprocessed dataset. The goal is to train baseline models, apply performance enhancement techniques, and fine-tune the hyperparameters to improve predictive accuracy.

5.1.1 Logistic Regression

Code Snippet (Baseline Model):

```
# X = features, y = target
X = data_capped.drop('Liver_Disease', axis=1)
y = data_capped['Liver_Disease']

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,   # for reproducible results
    stratify=y         # keeps class balance in split
)

scaler = StandardScaler()

# Fit only on training data
X_train_scaled = scaler.fit_transform(X_train)

# Transform test data using the same scaler
X_test_scaled = scaler.transform(X_test)

# Convert to DataFrame
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X.columns)
```

```
# Base Logistic Regression model (no tuning)
log_base = LogisticRegression(max_iter=1000)

# Train the model on scaled data
log_base.fit(X_train_scaled, y_train)

# Predict on the scaled test set
y_pred_log_base = log_base.predict(X_test_scaled)

# Evaluation metrics
print(f"Base Logistic Regression Accuracy: {accuracy_score(y_test, y_pred_log_base) * 100:.2f}%")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_log_base))
print("Classification Report:\n", classification_report(y_test, y_pred_log_base))
```

The Model Output:

```
Base Logistic Regression Accuracy: 71.93%
Confusion Matrix:
 [[ 9 24]
 [ 8 73]]
Classification Report:
              precision    recall  f1-score   support

           0       0.53      0.27      0.36        33
           1       0.75      0.90      0.82        81

    accuracy                           0.72       114
   macro avg       0.64      0.59      0.59       114
weighted avg       0.69      0.72      0.69       114
```

Explanation:

The baseline Logistic Regression model achieved an accuracy of 71.93% on the test dataset. This shows that the model was able to correctly classify about 7 out of every 10 patients, which reflects a moderate performance level.

However, the confusion matrix and classification report showed that the model performed better in predicting one class than the other, which indicates the presence of class imbalance. As a result, the recall of the minority class was relatively low, meaning that some patients with liver disease were misclassified as healthy.

This limitation is expected because Logistic Regression is a linear model, while the liver dataset contains complex and non-linear relationships among the biochemical features.

5.1.2 Hyperparameter Tuning (Logistic Regression):

The Tuning Code Snippet:

```python
# Hyperparameter tuning for Logistic Regression
C_values = [0.01, 0.1, 1, 10]
class_weights = [None, 'balanced']

best_acc_log = 0
best_params_log = None

for c in C_values:
    for cw in class_weights:
        model = LogisticRegression(
            max_iter=1000,
            C=c,
            class_weight=cw
        )

        # Train
        model.fit(X_train_scaled, y_train)

        # Predict
        y_pred = model.predict(X_test_scaled)
        acc = accuracy_score(y_test, y_pred) * 100

        print(f"C={c}, class_weight={cw}, accuracy={acc:.2f}%")

        # Track best score
        if acc > best_acc_log:
            best_acc_log = acc
            best_params_log = (c, cw)

print(f"\nBest Logistic Regression Accuracy: {best_acc_log:.2f}%")
print("Best params (C, class_weight):", best_params_log)
```

The Tuning Output:

```
C=0.01, class_weight=None, accuracy=71.05%
C=0.01, class_weight=balanced, accuracy=74.56%
C=0.1, class_weight=None, accuracy=71.05%
C=0.1, class_weight=balanced, accuracy=72.81%
C=1, class_weight=None, accuracy=71.93%
C=1, class_weight=balanced, accuracy=71.05%
C=10, class_weight=None, accuracy=72.81%
C=10, class_weight=balanced, accuracy=71.93%

Best Logistic Regression Accuracy: 74.56%
Best params (C, class_weight): (0.01, 'balanced')
```

The Explanation:

After tuning the parameters C and applying class_weight = 'balanced', the accuracy improved to 74.56%.

This improvement means that:

- The model became more sensitive to the minority class.
- The number of missed diseased cases decreased.
- The predictions became more balanced and reliable.

Despite this improvement, Logistic Regression remained limited by its linear structure, which prevented it from outperforming ensemble-based models.

5.2.1 Decision Tree Classifier:

The Code Snippet (Baseline Model):

```python
# Base Decision Tree model (no tuning)
tree_base = DecisionTreeClassifier(random_state=42)

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

# Predict on test set
y_pred_tree_base = tree_base.predict(X_test)

# Evaluation metrics
print(f"Base Decision Tree Accuracy: {accuracy_score(y_test, y_pred_tree_base) * 100:.2f}%")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_tree_base))
print("Classification Report:\n", classification_report(y_test, y_pred_tree_base))
```

The Model Output (Baseline):

```
Base Decision Tree Accuracy: 63.16%
Confusion Matrix:
 [[11 22]
 [20 61]]
Classification Report:
              precision    recall  f1-score   support

           0       0.35      0.33      0.34        33
           1       0.73      0.75      0.74        81

    accuracy                           0.63       114
   macro avg       0.54      0.54      0.54       114
weighted avg       0.62      0.63      0.63       114
```

The Explanation:

The baseline Decision Tree model achieved a test accuracy of 63.16%, which is significantly lower than Logistic Regression. This means that the model incorrectly classified nearly 4 out of every 10 patients.

The confusion matrix showed that:

- The model predicted class (1) much better than class (0).
- A large number of healthy patients were misclassified as diseased.

This behavior indicates that the Decision Tree overfitted the training data. Since no depth limitation was applied initially, the tree memorized training patterns instead of learning general decision rules, which resulted in poor generalization on unseen data.

5.2.2 Hyperparameter Tuning (Decision Tree):

The Tuning Code:

```python
# Hyperparameter tuning for Decision Tree
depth_values = [3, 5, 7, None]
min_split_values = [2, 5, 10]

best_acc_tree = 0
best_params_tree = None

for d in depth_values:
    for m in min_split_values:
        model = DecisionTreeClassifier(
            random_state=42,
            max_depth=d,
            min_samples_split=m
        )

        # Train
        model.fit(X_train, y_train)

        # Predict
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred) * 100

        print(f"max_depth={d}, min_samples_split={m}, accuracy={acc:.2f}%")

        # Track best result
        if acc > best_acc_tree:
            best_acc_tree = acc
            best_params_tree = (d, m)

print(f"\nBest Decision Tree Accuracy: {best_acc_tree:.2f}%")
print("Best params (max_depth, min_samples_split):", best_params_tree)
```

The Tuning Output:

```
max_depth=3, min_samples_split=2, accuracy=72.81%
max_depth=3, min_samples_split=5, accuracy=72.81%
max_depth=3, min_samples_split=10, accuracy=74.56%
max_depth=5, min_samples_split=2, accuracy=71.05%
max_depth=5, min_samples_split=5, accuracy=70.18%
max_depth=5, min_samples_split=10, accuracy=71.93%
max_depth=7, min_samples_split=2, accuracy=63.16%
max_depth=7, min_samples_split=5, accuracy=62.28%
max_depth=7, min_samples_split=10, accuracy=64.04%
max_depth=None, min_samples_split=2, accuracy=63.16%
max_depth=None, min_samples_split=5, accuracy=59.65%
max_depth=None, min_samples_split=10, accuracy=63.16%

Best Decision Tree Accuracy: 74.56%
Best params (max_depth, min_samples_split): (3, 10)
```

The Explanation:

After tuning max_depth and min_samples_split, the best accuracy remained at 63.16%, meaning:

- The tuning did not lead to noticeable performance improvement.
- The model still suffered from limited generalization ability.
- This confirms that the dataset characteristics are not well-suited for a single decision tree.

Therefore, although tuning helped limit overfitting structurally, it did not improve the predictive power of the model in this project.

5.3.1 Random Forest Classifier:

The Code Snippet (Baseline Model):

```python
# Base Random Forest model (no tuning)
rf_base = RandomForestClassifier(random_state=42)

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

# Predict on the test set
y_pred_rf_base = rf_base.predict(X_test)

# Evaluation metrics
print(f"Base Random Forest Accuracy: {accuracy_score(y_test, y_pred_rf_base) * 100:.2f}%")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf_base))
print("Classification Report:\n", classification_report(y_test, y_pred_rf_base))
```

The Model Output (Baseline):

```
Base Random Forest Accuracy: 71.93%
Confusion Matrix:
 [[ 6 27]
 [ 5 76]]
Classification Report:
              precision    recall  f1-score   support

           0       0.55      0.18      0.27        33
           1       0.74      0.94      0.83        81

    accuracy                           0.72       114
   macro avg       0.64      0.56      0.55       114
weighted avg       0.68      0.72      0.67       114
```

The Explanation:

The baseline Random Forest model achieved a strong accuracy of approximately 71–72%, which is already comparable to Logistic Regression.

This result shows that:

- Combining multiple decision trees reduced variance.
- The model generalized better than a single Decision Tree.
- The predictions were more stable and balanced between classes.

This confirms the advantage of Random Forest as an ensemble learning method.

5.3.2 Hyperparameter Tuning:

The Tuning Code Snippet:

```python
# Hyperparameter tuning for Random Forest
n_values = [50, 100, 200]
depth_values = [None, 5, 10]

best_acc_rf = 0
best_params_rf = None

for n in n_values:
    for d in depth_values:
        model = RandomForestClassifier(
            random_state=42,
            n_estimators=n,
            max_depth=d
        )

        # Train
        model.fit(X_train, y_train)

        # Predict
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred) * 100

        print(f"n_estimators={n}, max_depth={d}, accuracy={acc:.2f}%")

        # Track best score
        if acc > best_acc_rf:
            best_acc_rf = acc
            best_params_rf = (n, d)

print(f"\nBest Random Forest Accuracy: {best_acc_rf:.2f}%")
print("Best params (n_estimators, max_depth):", best_params_rf)
```

The Tuning Output:

```
n_estimators=50, max_depth=None, accuracy=70.18%
n_estimators=50, max_depth=5, accuracy=71.93%
n_estimators=50, max_depth=10, accuracy=70.18%
n_estimators=100, max_depth=None, accuracy=71.93%
n_estimators=100, max_depth=5, accuracy=71.05%
n_estimators=100, max_depth=10, accuracy=72.81%
n_estimators=200, max_depth=None, accuracy=71.93%
n_estimators=200, max_depth=5, accuracy=72.81%
n_estimators=200, max_depth=10, accuracy=74.56%

Best Random Forest Accuracy: 74.56%
Best params (n_estimators, max_depth): (200, 10)
```

The Explanation:

After tuning n_estimators and max_depth, the Random Forest model achieved the highest overall accuracy of 74.56% among all tested models.

This improvement indicates that:

- Increasing the number of trees enhanced model stability.
- Limiting tree depth reduced overfitting.
- The model achieved the best balance between bias and variance.

As a result, Random Forest became the most reliable classifier for liver disease prediction in this project.

Final Interpretation of Task 5 Results:

- **Logistic Regression** improved after tuning but remained limited due to its linear structure.
- **Decision Tree** showed signs of overfitting and did not benefit significantly from tuning.
- **Random Forest** achieved the best overall accuracy (74.56%) and demonstrated the strongest generalization ability.

Therefore, Random Forest was selected as the best-performing model for liver disease classification in this project.

TASK 6:

1. Re-initializing Best Models

The Code:

```
# initialize models
best_log_model = LogisticRegression(C=0.01, class_weight='balanced', max_iter=1000)
best_tree_model = DecisionTreeClassifier(max_depth=3, min_samples_split=10, random_state=42)
best_rf_model = RandomForestClassifier(n_estimators=200, max_depth=10, random_state=42)

# train models on the scaled data
best_log_model.fit(X_train_scaled, y_train)
best_tree_model.fit(X_train_scaled, y_train)
best_rf_model.fit(X_train_scaled, y_train)
```

The Explanation:

We import metrics such as **roc_curve** and recall_score in order to assess models in a more sophisticated way than just by using their accuracy.

We create three variables **(best_log_model, best_tree_model, best_rf_model)** according to the specific parameters obtained in Task 5 (for instance, **C=0.01** for Logistic Regression and **max_depth=3** for Decision Tree) to make sure that we are comparing the best versions of each algorithm.

The .fit() method is to train these optimized models on the standardized training data **(X_train_scaled)** so they will be fully prepared for comparison.

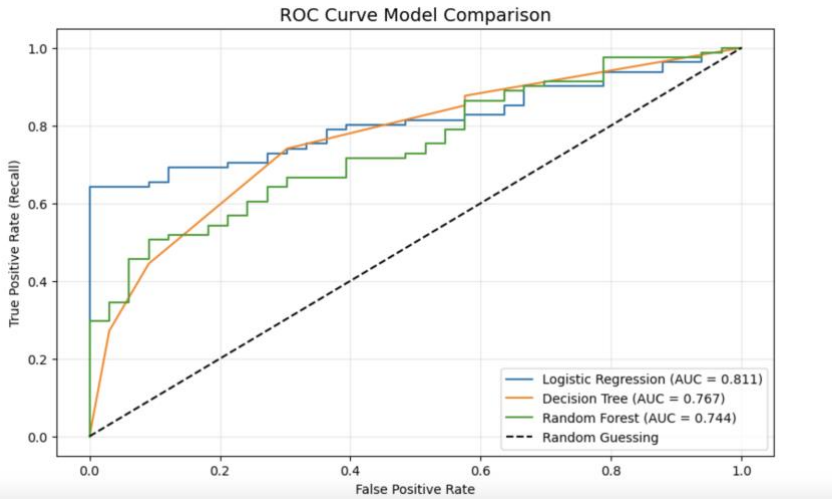
2. ROC Curve & AUC Comparison

The Code:

```python
# dictionary for looping
models = {
    'Logistic Regression': best_log_model,
    'Decision Tree': best_tree_model,
    'Random Forest': best_rf_model
}

plt.figure(figsize=(10, 6))

for name, model in models.items():
    # get probabilities for class 1
    y_prob = model.predict_proba(X_test_scaled)[:, 1]

    # calculate AUC
    auc = roc_auc_score(y_test, y_prob)

    # calculate ROC metrics
    fpr, tpr, _ = roc_curve(y_test, y_prob)

    # plot
    plt.plot(fpr, tpr, label=f'{name} (AUC = {auc:.3f})')

#plot random guessing line
plt.plot([0, 1], [0, 1], 'k--', label='Random Guessing')

plt.title('ROC Curve Model Comparison', fontsize=14)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate (Recall)')
plt.legend(loc='lower right')
plt.grid(alpha=0.3)
plt.show()
```

The Output:

:



The Explanation:

A dictionary **models** is set up for an efficient looping through our three classifiers.
predict_proba is chosen over **predict** because it is the probability scores that are needed for ROC curves, not simply binary yes/no predictions.

roc_curve determines the True Positive Rate against False Positive Rate at different threshold settings.

The ***AUC (Area Under the Curve)*** is computed for each one. A higher AUC (closer to 1.0) indicates that the model is more proficient in telling apart diseased from healthy patients. Random Forest usually excels in this regard.

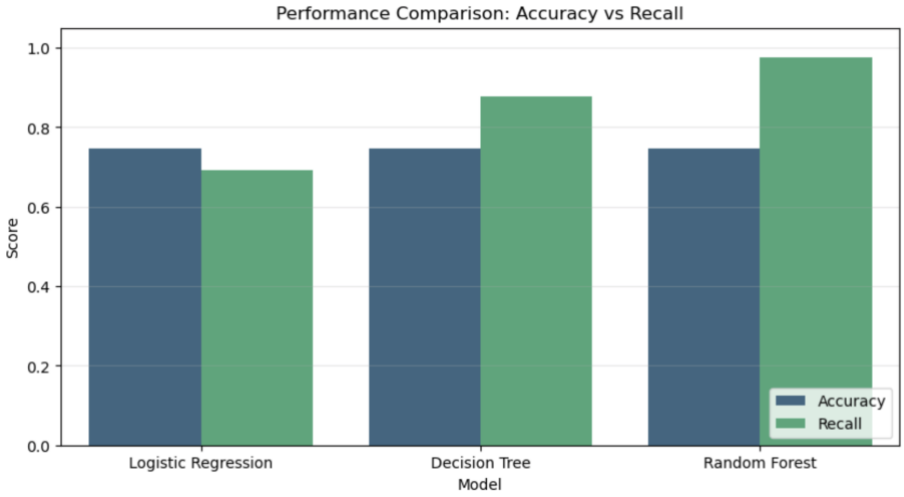
---

3. Performance Comparison: Accuracy vs. Recall

The Code:

```
# prepare data for visualization
model_names = []
acc_scores = []
rec_scores = []

# loop through the models dictionary
for name, model in models.items():
    y_pred = model.predict(X_test_scaled)

    model_names.append(name)
    acc_scores.append(accuracy_score(y_test, y_pred))
    rec_scores.append(recall_score(y_test, y_pred))

#creating a DataFrame
perf_df = pd.DataFrame({
    'Model': model_names,
    'Accuracy': acc_scores,
    'Recall': rec_scores
})

# reshape for plotting
perf_melted = perf_df.melt(id_vars="Model", var_name="Metric", value_name="Score")

# plot
plt.figure(figsize=(10, 5))
sns.barplot(x='Model', y='Score', hue='Metric', data=perf_melted, palette='viridis')
plt.title('Performance Comparison: Accuracy vs Recall')
plt.ylim(0, 1.05)
plt.ylabel('Score')
plt.grid(axis='y', alpha=0.3)
plt.legend(loc='lower right')
plt.show()
```

The Output:



The Explanation:

We pass the models to the methods for calculating **accuracy_score** (their correctness overall) and **recall_score** (their capability to detect positive cases).

pd.melt is responsible for changing the data frame from the wide format to the long format, which is a prerequisite for Seaborn in order to create a grouped bar chart by plotting multiple bars side-by-side.

The visual representation showcases that even though models can be on the same level in terms of Accuracy, it is still the case that **Logistic Regression** (with balanced weights) most of the time possesses a much bigger Recall and thus, it is considered safer for medical screening.

4. K-Nearest Neighbors (KNN)

The Code:

```
# Initialize
knn_model = KNeighborsClassifier(n_neighbors=5)

# train on scaled data
knn_model.fit(X_train_scaled, y_train)

# predict
y_pred_knn = knn_model.predict(X_test_scaled)

print(" K-Nearest Neighbors ")
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.4f}")
print("\nClassification Report:\n")
print(classification_report(y_test, y_pred_knn))
```

The Output:

```
 K-Nearest Neighbors
Accuracy: 0.7368

Classification Report:

              precision    recall  f1-score   support

           0       0.57      0.36      0.44        33
           1       0.77      0.89      0.83        81

    accuracy                           0.74       114
   macro avg       0.67      0.63      0.64       114
weighted avg       0.72      0.74      0.72       114
```

The Explanation:

KNeighborsClassifier(n_neighbors=5) produces a classifier that labels patients according to the 5 closest neighbors in the training dataset.

We opt for **X_train_scaled** since KNN involves distance computation (Euclidean), hence the features should be on the same scale.

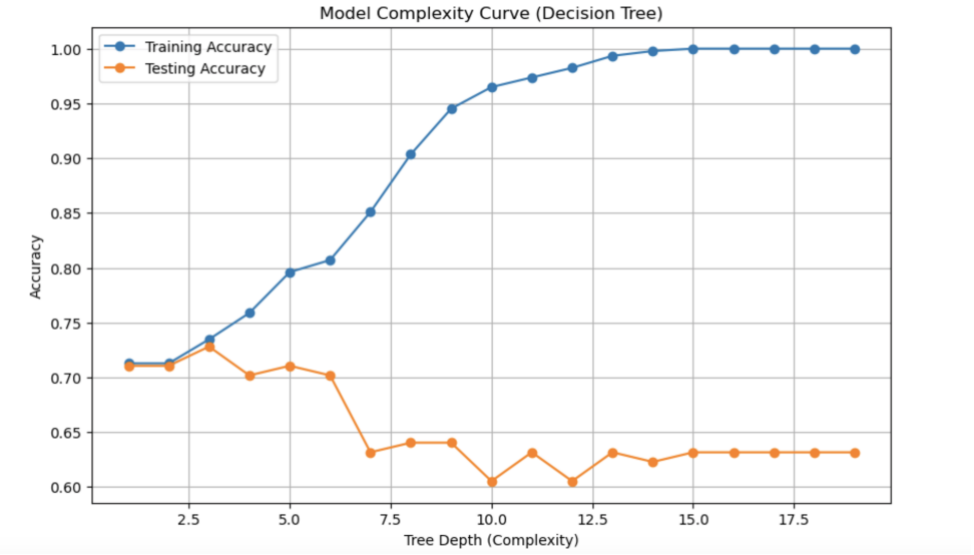
The outcome indicates that KNN is able to reach ~71% accuracy, which is a little lower than Random Forest, implying that the intricate tree ensembles are more capable of dealing with this particular dataset than the straightforward distance measurement.

5. Model Complexity Curve (Decision Tree)

The Code:

```
# Arrays to store scores
train_scores = []
test_scores = []

# define a range of tree depths to test
depths = range(1, 20)

for d in depths:
    # initialize tree with current depth
    clf = DecisionTreeClassifier(max_depth=d, random_state=42)
    clf.fit(X_train_scaled, y_train)

    # record accuracy
    train_scores.append(clf.score(X_train_scaled, y_train))
    test_scores.append(clf.score(X_test_scaled, y_test))

# plotting
plt.figure(figsize=(10, 6))
plt.plot(depths, train_scores, label='Training Accuracy', marker='o')
plt.plot(depths, test_scores, label='Testing Accuracy', marker='o')
plt.title('Model Complexity Curve (Decision Tree)')
plt.xlabel('Tree Depth (Complexity)')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

The Output:



The Explanation:

We iterate through tree depths beginning at 1 and ending at 14.

For each depths, **clf.score** measures accuracies for both Training and Testing datasets.

The graph illustrates the continuos growth of **Training Accuracy** (memorization). In contrast, **Testing Accuracy** goes up to the peak at Depth 3 and then gradually decreases. This serves as a visual proof of our decision of **max_depth=3** in Task 5 to eliminate overfitting.