

# RIPHAH INTERNATIONAL UNIVERSITY ISLAMABAD GGC

## INTRODUCTION TO MACHINE LEARNING

FALL 2024

### ASSIGNMENT 2

DUE DATE: 3<sup>RD</sup> NOVEMBER 2024

#### **Topics:**

##### *1. EDA Classification algorithms*

A dataset file named heartdisease.csv is uploaded to moellim. The dataset contains several columns across them age is the target variable (dependent) and all others are independent.

- a. **You need to download the data and perform basic data preprocessing tasks given below.**
- b. **Other than preprocessing apply the list of algorithms given below and train them individually**
- c. **Evaluate each model using relevant metrics:**
- d. **For classification: Accuracy, Precision, Recall, F1-Score, and Confusion Matrix**
- e. **Compare the results of each model and discuss which model performed best and why.**
- f. **Submit the code on Kaggle dataset challenge and upload the link with the submission**

**NOTE: Add comments with each task to describe what you have done.**

#### **Preprocessing Tasks:**

1. Load the dataset and show the information about data using appropriate methods (like mean, median, etc)
2. Find out the duplicates from the data and drop them
3. Check missing values in the dataset by first replacing? with NaN values and then using the appropriate method to find missing values
4. Drop null values (not full column) if there exists any
5. Divide the data in train and test split set with the ratio 80:20
6. Perform feature scaling if required

#### **Algorithms:**

1. Logistic Regression (for classification tasks)
2. Decision Trees
3. K-Nearest Neighbors (KNN)

#### 4. Naive Bayes

## ✓ Assignment 2 (Machine Learning)

This dataset consists of health-related attributes aimed at predicting the presence of heart disease. Each row represents a patient's health profile, including attributes like age, sex, chest pain type, resting blood pressure, serum cholesterol, and other diagnostic features from physical and laboratory exams. The target variable indicates the presence or absence of heart disease, where '1' means heart disease is present, and '0' means it is not.

The dataset has 1,025 entries and 14 columns, with the following attributes:

- **age:** Age of the patient (integer).
- **sex:** Sex of the patient (1 = male, 0 = female).
- **cp:** Chest pain type (categorical variable with values 0, 1, 2, 3).
- **trestbps:** Resting blood pressure in mm Hg (continuous).
- **chol:** Serum cholesterol in mg/dL (integer).
- **fbbs:** Fasting blood sugar > 120 mg/dL (1 = true, 0 = false).
- **restecg:** Resting electrocardiographic results (categorical: 0, 1, 2).
- **thalach:** Maximum heart rate achieved (continuous).
- **exang:** Exercise-induced angina (1 = yes, 0 = no).
- **oldpeak:** ST depression induced by exercise relative to rest (continuous).
- **slope:** Slope of the peak exercise ST segment (categorical: 0, 1, 2).
- **ca:** Number of major vessels (0–3) colored by fluoroscopy (integer).
- **thal:** Thalassemia (categorical: 0, 1, 2, 3).
- **target:** Heart disease diagnosis (1 = heart disease, 0 = no heart disease), which serves as the target variable for predictions.

Some columns have missing values, specifically trestbps, thalach, and oldpeak. This dataset appears to be a typical heart disease prediction dataset with various health and diagnostic indicators as features and a binary target variable indicating the presence of heart disease.

### Mount Drive

```
#Mount Drive
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

### Installs essential Python libraries: Pandas, NumPy, Matplotlib, and Seaborn for data analysis and visualization

```
!pip install pandas
!pip install numpy
!pip install matplotlib
!pip install seaborn
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.2.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.26.4)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: numpy<2,>=1.21 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (0.13.2)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in /usr/local/lib/python3.10/dist-packages (from seaborn) (1.26.4)
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.10/dist-packages (from seaborn) (2.2.2)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in /usr/local/lib/python3.10/dist-packages (from seaborn) (3.8.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
```

Import necessary libraries for data manipulation, visualization, and machine learning tasks.


```
import pandas as pd
import pylab as pl
import numpy as np
import seaborn as sns
import scipy.optimize as opt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
%matplotlib inline
import matplotlib.pyplot as plt
```



▼ Data Preprocessing

1) Load the dataset and show the information about data using appropriate methods (like mean, median, etc)

```
# read file
data = pd.read_csv('/content/drive/MyDrive/ML uni/heart.csv')

# first 5 rows
data.head()
```



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target	
0	52	1	0	125.0	212	0	1	168.0	0	1.0	2	2	3	0	
1	53	1	0	140.0	203	1	0	155.0	1	3.1	0	0	3	0	
2	70	1	0	145.0	174	0	1	125.0	1	2.6	0	0	3	0	
3	61	1	0	148.0	203	0	1	161.0	0	0.0	2	1	3	0	
4	62	0	0	138.0	294	1	1	106.0	0	1.9	1	3	2	0	


Next steps:



[Generate code with data](#)

 [View recommended plots](#)

[New interactive sheet](#)

```
# last five rows
data.tail()
```



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target	
1020	59	1	1	140.0	221	0	1	164.0	1	0.0	2	0	2	1	
1021	60	1	0	125.0	258	0	0	141.0	1	2.8	1	1	3	0	
1022	47	1	0	110.0	275	0	0	118.0	1	1.0	1	1	2	0	
1023	50	0	0	110.0	254	0	0	159.0	0	0.0	2	0	2	1	
1024	54	1	0	120.0	188	0	1	113.0	0	1.4	1	1	3	0	

```
#Summary of the DataFrame
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
#   Column      Non-Null Count  Dtype
---  -
0   age         1025 non-null   int64
1   sex         1025 non-null   int64
2   cp          1025 non-null   int64
3   trestbps    1024 non-null   float64
4   chol        1025 non-null   int64
5   fbs         1025 non-null   int64
6   restecg     1025 non-null   int64
7   thalach     1024 non-null   float64
8   exang       1025 non-null   int64
```

```


9   oldpeak   1024 non-null   float64
10  slope     1025 non-null   int64
11  ca        1025 non-null   int64
12  thal      1025 non-null   int64
13  target    1025 non-null   int64
dtypes: float64(3), int64(11)
memory usage: 112.2 KB

```

```

#summary of statistics for all columns in the DataFrame
data.describe(include='all')

```



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak
<b>count</b>	1025.000000	1025.000000	1025.000000	1024.000000	1025.000000	1025.000000	1025.000000	1024.000000	1025.000000	1024.000000
<b>mean</b>	54.434146	0.695610	0.942439	131.611328	246.000000	0.149268	0.529756	149.140625	0.336585	1.069629
<b>std</b>	9.072290	0.460373	1.029641	17.525273	51.59251	0.356527	0.527878	23.001333	0.472772	1.174079
<b>min</b>	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000
<b>25%</b>	48.000000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	132.000000	0.000000	0.000000
<b>50%</b>	56.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	152.000000	0.000000	0.800000
<b>75%</b>	61.000000	1.000000	2.000000	140.000000	275.000000	0.000000	1.000000	166.000000	1.000000	1.800000
<b>max</b>	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000

```

# Calculate the median for each numeric column
numeric_columns = data.select_dtypes(include=['number'])
median_values = numeric_columns.median()
median_values

```




	0
<b>age</b>	56.0
<b>sex</b>	1.0
<b>cp</b>	1.0
<b>trestbps</b>	130.0
<b>chol</b>	240.0
<b>fbs</b>	0.0
<b>restecg</b>	1.0
<b>thalach</b>	152.0
<b>exang</b>	0.0
<b>oldpeak</b>	0.8
<b>slope</b>	1.0
<b>ca</b>	0.0
<b>thal</b>	2.0
<b>target</b>	1.0

```
dtype: float64
```

```

#rows and columns
data.shape

```



```
(1025, 14)
```

## 2) Find out the duplicates from the data and drop them

Find duplicates:

```
data.duplicated()
```

```

0
0 False
1 False
2 False
3 False
4 False
...
1020 True
1021 True
1022 True
1023 True
1024 True
1025 rows × 1 columns

dtype: bool

```

Drop duplicates:

```
data.drop_duplicates()
```

```

age sex cp trestbps chol fbs restecg thalach exang oldpeak slope ca thal target
0 52 1 0 125.0 212 0 1 168.0 0 1.0 2 2 3 0
1 53 1 0 140.0 203 1 0 155.0 1 3.1 0 0 3 0
2 70 1 0 145.0 174 0 1 125.0 1 2.6 0 0 3 0
3 61 1 0 148.0 203 0 1 161.0 0 0.0 2 1 3 0
4 62 0 0 138.0 294 1 1 106.0 0 1.9 1 3 2 0
...
723 68 0 2 120.0 211 0 0 115.0 0 1.5 1 0 2 1
733 44 0 2 108.0 141 0 1 175.0 0 0.6 1 0 2 1
739 52 1 0 128.0 255 0 1 161.0 1 0.0 2 1 3 0
843 59 1 3 160.0 273 0 0 125.0 0 0.0 2 0 2 0
878 54 1 0 120.0 188 0 1 113.0 0 1.4 1 1 3 0
304 rows × 14 columns

```

```
data.shape
```

```
(1025, 14)
```

### 3) Check missing values in the dataset by first replacing? with NaN values and then using the appropriate method to find missing values

Handle Missing Values

```
data.replace('?', np.nan, inplace=True)
```

```
missing_values = data.isnull().sum()
```

### 4) Drop null values (not full column) if there exists any

```
data = data.dropna()
```


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



	0
age	0
sex	0
cp	0
trestbps	0
chol	0
fbs	0
restecg	0
thalach	0
exang	0
oldpeak	0
slope	0
ca	0
thal	0
target	0

dtype: int64

data.shape



(1023, 14)
------------


5) Divide the data in train and test split set with the ratio 80:20

Split Data into Train and Test Sets:



```
X = data.drop(columns=['target'])
```

```
y = data['target']
```

X.head()



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	52	1	0	125.0	212	0	1	168.0	0	1.0	2	2	3
1	53	1	0	140.0	203	1	0	155.0	1	3.1	0	0	3
2	70	1	0	145.0	174	0	1	125.0	1	2.6	0	0	3
3	61	1	0	148.0	203	0	1	161.0	0	0.0	2	1	3
4	62	0	0	138.0	294	1	1	106.0	0	1.9	1	3	2




Next steps:

Generate code with X

 View recommended plots

New interactive sheet

y.head()



	target
0	0
1	0
2	0
3	0
4	0

dtype: int64

```
#check data imbalance for target vaiable
```

```
# Calculate the class distribution
target_counts = data['target'].value_counts()
```

```
# Print the class distribution
```

```
print("Class Distribution:")
print(target_counts)

# Calculate the percentage of each class
target_percentages = target_counts / len(data) * 100

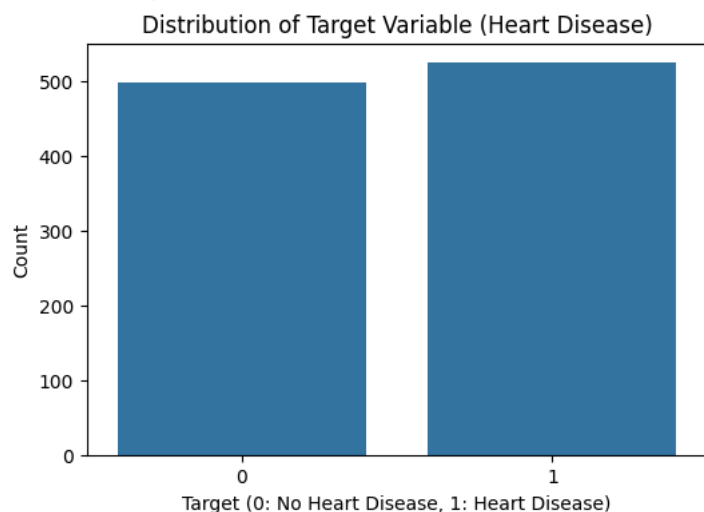
# Print the percentage of each class
print("\nClass Percentages:")
print(target_percentages)

# Visualize the class distribution
plt.figure(figsize=(6, 4))
sns.countplot(x='target', data=data)
plt.title('Distribution of Target Variable (Heart Disease)')
plt.xlabel('Target (0: No Heart Disease, 1: Heart Disease)')
plt.ylabel('Count')
plt.show()

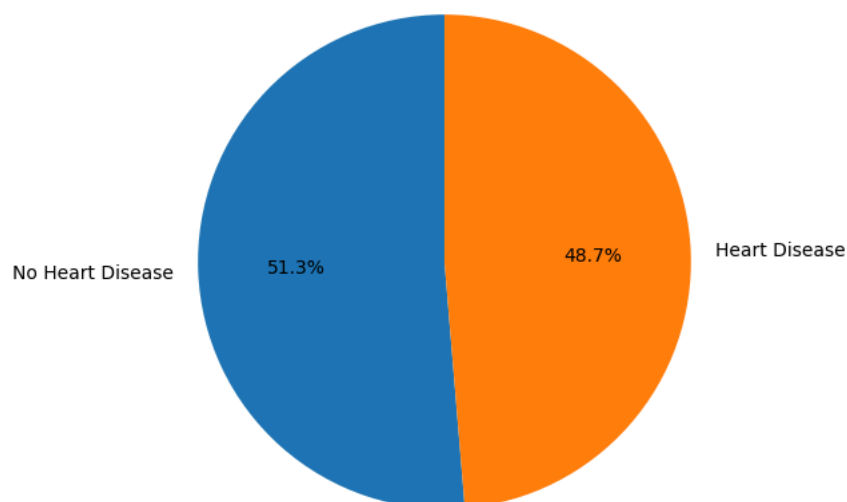
# You can also use a pie chart to visualize the class distribution
plt.figure(figsize=(6, 6))
plt.pie(target_counts, labels=['No Heart Disease', 'Heart Disease'], autopct='%1.1f%%', startangle=90)
plt.title('Distribution of Target Variable (Heart Disease)')
plt.show()
```

```
↗ Class Distribution:
target
1    525
0    498
Name: count, dtype: int64
```

```
Class Percentages:
target
1    51.319648
0    48.680352
Name: count, dtype: float64
```



Distribution of Target Variable (Heart Disease)





Splitting data into train and test (80:20 split):

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

Train and Test result:

```
print("Training set shape:", X_train.shape)
print("Testing set shape:", X_test.shape)
```

```
↗ Training set shape: (818, 13)
   Testing set shape: (205, 13)
```

## 6) Perform feature scaling if required

```
# Initializing the StandardScaler
data = StandardScaler()

# Fitting the scaler on the training set and transforming the training set
X_train_scaled = data.fit_transform(X_train)

# Using the same scaler to transform the test set
X_test_scaled = data.transform(X_test)

# Converting scaled data back to DataFrame for convenience (optional)
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns)

# Displaying scaled data
print("Scaled Training set:\n", X_train_scaled.head())
print("Scaled Testing set:\n", X_test_scaled.head())
```

```
↗ Scaled Training set:
```

	age	sex	cp	trestbps	chol	fbs	restecg	\
0	-1.481190	0.644379	0.043626	-1.244883	-0.212403	-0.418673	0.877086	
1	-0.257226	0.644379	1.972624	-0.786157	-1.146624	-0.418673	-1.010957	
2	0.410391	-1.551881	0.043626	0.245976	1.389117	2.388497	-1.010957	
3	-1.369921	0.644379	1.008125	-0.098068	-1.261018	-0.418673	0.877086	
4	1.856895	-1.551881	1.008125	-1.244883	0.359568	2.388497	-1.010957	

	thalach	exang	oldpeak	slope	ca	thal
0	0.173088	-0.705811	-0.927077	1.006460	-0.723750	-0.533725
1	1.763938	-0.705811	-0.927077	-0.620584	-0.723750	-2.156725
2	0.130092	-0.705811	-0.927077	1.006460	1.210981	-0.533725
3	0.044100	-0.705811	-0.927077	1.006460	-0.723750	-0.533725
4	-0.815819	-0.705811	-0.927077	1.006460	0.243615	-0.533725

```
Scaled Testing set:
```

	age	sex	cp	trestbps	chol	fbs	restecg	\
0	0.855469	-1.551881	-0.920872	-0.442113	-0.708112	-0.418673	0.877086	
1	-0.145956	-1.551881	1.008125	-0.212750	-0.574652	-0.418673	-1.010957	
2	-0.145956	0.644379	1.008125	-0.098068	-0.002680	2.388497	-1.010957	
3	1.745625	0.644379	1.008125	1.622154	0.435831	-0.418673	0.877086	
4	0.299122	0.644379	-0.920872	1.163428	0.531160	-0.418673	0.877086	

	thalach	exang	oldpeak	slope	ca	thal
0	0.603047	-0.705811	-0.927077	1.006460	-0.723750	-0.533725
1	-1.460759	-0.705811	-0.927077	1.006460	-0.723750	-3.779725
2	1.033007	-0.705811	-0.927077	1.006460	2.178346	-0.533725
3	-1.589747	1.416811	1.561846	-0.620584	0.243615	1.089275
4	-2.621650	1.416811	0.102822	-0.620584	0.243615	1.089275

## ✓ Apply and Evaluate Classification Models

### 1. Logistic Regression (for classification tasks)

Logistic regression is a statistical model used to predict the probability of a binary outcome (i.e., a variable that can take one of two values, such as 0 or 1). It's often used in classification problems, such as predicting whether an email is spam or not, whether a customer will buy a product, or if a patient has a disease or not.

#### The Formula of Logistic Regression

The basic logistic regression formula is:

$$P(y = 1 | X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n)}} \quad P(y=1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n)}}$$

Where:

$P(y = 1 | X)$   $P(y=1|X)$  is the probability that the output  $y$  is 1 (positive class).  $X_1, X_2, \dots, X_n$  are the input features (independent variables).  $\beta_0$  is the intercept (bias term).  $\beta_1, \beta_2, \dots, \beta_n$  are the coefficients (weights) of the features.  $e$  is Euler's number, approximately equal to 2.71828 (the base of natural logarithms). Explanation: Linear Combination: First, the model computes a linear combination of the input features  $X_1, X_2, \dots, X_n$  weighted by the coefficients  $\beta_1, \beta_2, \dots, \beta_n$ . This linear combination is the same as in linear regression:

$$z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n$$

**Sigmoid Function:** The result of this linear combination  $z$  is then passed through a sigmoid function (also called the logistic function):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid function squashes the result into a range between 0 and 1, which can be interpreted as the probability that the output is 1. If the output is closer to 1, the prediction is more likely to be class 1; if it's closer to 0, the prediction is more likely to be class 0.

**Example:** Let's say you want to predict if a student will pass (1) or fail (0) based on study hours ( $X$ ).

You might have a logistic regression model like:

$$P(\text{Pass}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \times \text{Hours Studied})}}$$

Where  $\beta_0$  is the bias term, and  $\beta_1$  is the weight associated with the number of hours studied. After fitting the model, you'll get values for  $\beta_0$  and  $\beta_1$ , and you can use these to predict the probability of passing for a given number of study hours.

**Key Points:** Output: The output is a probability between 0 and 1. Binary Classification: It's mainly used for problems where the outcome variable is binary (i.e., has two categories). Decision Threshold: Typically, a threshold of 0.5 is used to decide between the two classes. If  $P(y = 1 | X)$

$P(y=1|X) > 0.5$ , the prediction is class 1, otherwise class 0.

```
import numpy as np
import matplotlib.pyplot as plt
import itertools
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, precision_score, recall_score, f1_score

# Training the Logistic Regression model
LR = LogisticRegression(C=0.01, solver='liblinear').fit(X_train, y_train)

# Making predictions on test data
ydata = LR.predict(X_test)

# Estimating probabilities for each class
ydata_prob = LR.predict_proba(X_test)

# Plotting a confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', normalize=False, cmap=plt.cm.Reds):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
unique_labels = np.unique(y_test)

print("Confusion Matrix:")
cnf_matrix = confusion_matrix(y_test, ydata, labels=unique_labels)
print(cnf_matrix)
print('\n')

# Plot non-normalized confusion matrix
plt.figure()
classes = [str(label) for label in unique_labels]
plot_confusion_matrix(cnf_matrix, classes=classes, normalize=False, title='Confusion matrix')
```

```
# Displaying evaluation metrics
accuracy = accuracy_score(y_test, ydata)
print("Accuracy:", accuracy)
print('\n')

precision = precision_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Precision:", precision)
print('\n')

recall = recall_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Recall:", recall)
print('\n')

f1 = f1_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("F1-score:", f1)
print('\n')

# Displaying classification report
print("Classification Report:")
print(classification_report(y_test, ydata))
```

↗ Confusion Matrix:

```
[[74 31]
 [11 89]]
```

Confusion matrix, without normalization

```
[[74 31]
 [11 89]]
```

Accuracy: 0.7951219512195122

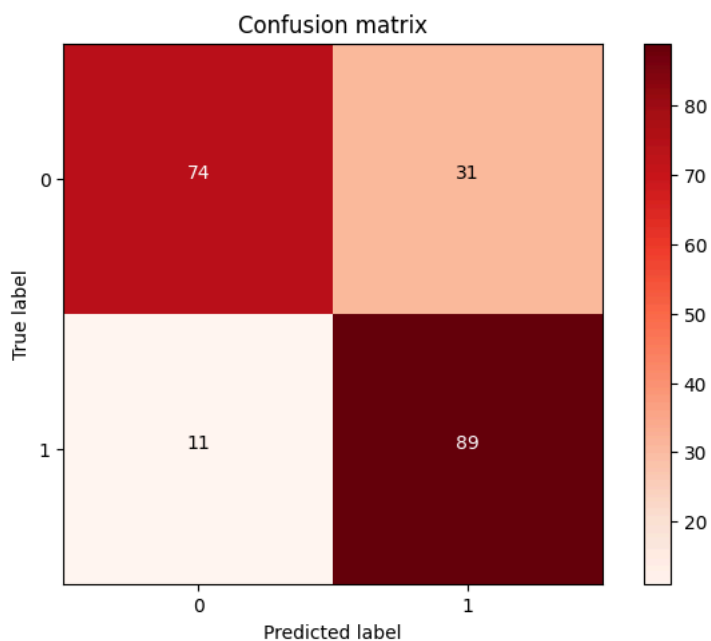
Precision: 0.8061274509803922

Recall: 0.7973809523809524

F1-score: 0.7940191387559808

Classification Report:

	precision	recall	f1-score	support
0	0.87	0.70	0.78	105
1	0.74	0.89	0.81	100
accuracy			0.80	205
macro avg	0.81	0.80	0.79	205
weighted avg	0.81	0.80	0.79	205



```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
```

```

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, precision_score, recall_score, f1_score
import itertools

# Fit the model on the training set
LR = LogisticRegression(C=0.01, solver='liblinear').fit(X_train, y_train)

# Make predictions on the test data
ydata = LR.predict(X_test)

# Estimate probabilities for each class
ydata_prob = LR.predict_proba(X_test)

# _____#

# Function to plot confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', normalize=False, cmap=plt.cm.Red):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Print confusion matrix
print("Confusion Matrix:")
unique_labels = np.unique(y_test) # Get unique labels
print(confusion_matrix(y_test, ydata, labels=unique_labels))
print('\n')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, ydata, labels=unique_labels)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=[str(label) for label in unique_labels], normalize=False, title='Confusion matrix')

# _____#

print('\n')

# Displaying accuracy
accuracy = accuracy_score(y_test, ydata)
print("Accuracy:", accuracy)
print('\n')

# Displaying precision
precision = precision_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Precision:", precision)
print('\n')

# Displaying recall
recall = recall_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Recall:", recall)
print('\n')

# Displaying F1-score
f1 = f1_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("F1-score:", f1)
print('\n')

# _____#

# Displaying classification report
print("Classification Report:")

```

```
print(classification_report(y_test, ydata))
```

```
Confusion Matrix:
```

```
[[74 31]
 [11 89]]
```

Confusion matrix, without normalization

```
[[74 31]
 [11 89]]
```

Accuracy: 0.7951219512195122

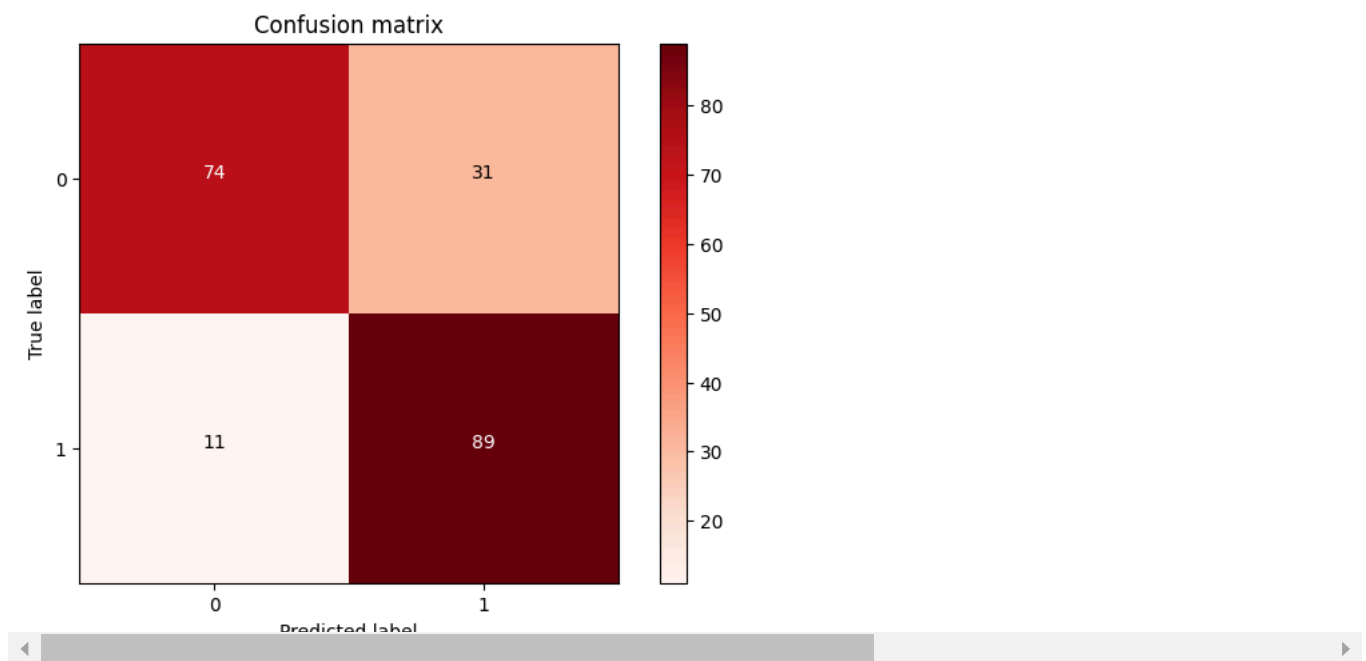
Precision: 0.8061274509803922

Recall: 0.7973809523809524

F1-score: 0.7940191387559808

Classification Report:

	precision	recall	f1-score	support
0	0.87	0.70	0.78	105
1	0.74	0.89	0.81	100
accuracy			0.80	205
macro avg	0.81	0.80	0.79	205
weighted avg	0.81	0.80	0.79	205



## 2. Decision Trees

A decision tree is a supervised machine learning algorithm used for both classification and regression tasks. It works by splitting the data into subsets based on feature values to create a tree-like structure of decisions.

```
# Import necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, classification_report
import itertools
import matplotlib.pyplot as plt
import numpy as np

# Fit the model on the training set
dt = DecisionTreeClassifier(random_state=42).fit(X_train, y_train)

# Make predictions on the test data
ydata = dt.predict(X_test)

# Estimate probabilities for each class
ydata_prob = dt.predict_proba(X_test)
```

```

# _____#

# Function to plot confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', normalize=False, cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Print confusion matrix
print("Confusion Matrix:")
unique_labels = np.unique(y_test) # Get unique labels
print(confusion_matrix(y_test, ydata, labels=unique_labels))
print('\n')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, ydata, labels=unique_labels)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=[str(label) for label in unique_labels], normalize=False, title='Confusion matrix')

# _____#

print('\n')

# Displaying accuracy
accuracy = accuracy_score(y_test, ydata)
print("Accuracy:", accuracy)
print('\n')

# Displaying precision
precision = precision_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Precision:", precision)
print('\n')

# Displaying recall
recall = recall_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Recall:", recall)
print('\n')

# Displaying F1-score
f1 = f1_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("F1-score:", f1)
print('\n')

# _____#

# Displaying classification report
print("Classification Report:")
print(classification_report(y_test, ydata))

```

```

Confusion Matrix:
[[105  0]
 [  0 100]]

```

```

Confusion matrix, without normalization
[[105  0]
 [  0 100]]

```

Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1-score: 1.0

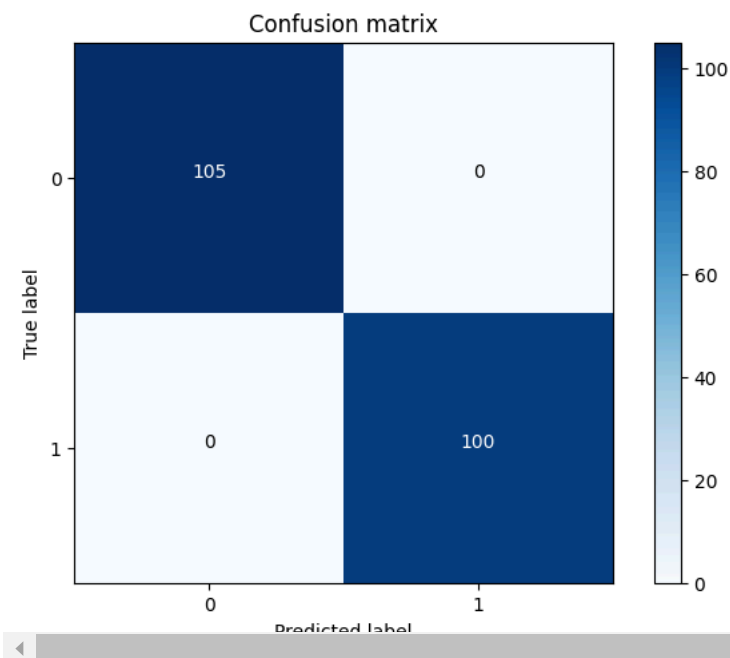
```

Classification Report:
              precision    recall  f1-score   support

     0:       1.00        1.00        1.00        105
     1:       1.00        1.00        1.00        100

 accuracy: 1.00
macro avg: 1.00        1.00        1.00        205
weighted avg: 1.00        1.00        1.00        205

```



### 3. K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a simple, supervised machine learning algorithm used for both classification and regression tasks. In KNN, predictions are made based on the "closeness" of data points in the feature space. When a new data point needs to be classified, the algorithm finds the  $k$  data points closest to it (neighbors) and makes a prediction based on the majority class (in classification) or the average (in regression) of those neighbors.

```

# Import necessary libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, classification_report
import itertools
import matplotlib.pyplot as plt
import numpy as np

# Fit the model on the training set
knn = KNeighborsClassifier(n_neighbors=5).fit(X_train, y_train)

# Make predictions on the test data
ydata = knn.predict(X_test)

# Estimate probabilities for each class
ydata_prob = knn.predict_proba(X_test)

```

```

# _____#

# Function to plot confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', normalize=False, cmap=plt.cm.Oranges):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Print confusion matrix
print("Confusion Matrix:")
unique_labels = np.unique(y_test) # Get unique labels
print(confusion_matrix(y_test, ydata, labels=unique_labels))
print('\n')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, ydata, labels=unique_labels)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=[str(label) for label in unique_labels], normalize=False, title='Confusion matrix')

# _____#

print('\n')

# Displaying accuracy
accuracy = accuracy_score(y_test, ydata)
print("Accuracy:", accuracy)
print('\n')

# Displaying precision
precision = precision_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Precision:", precision)
print('\n')

# Displaying recall
recall = recall_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Recall:", recall)
print('\n')

# Displaying F1-score
f1 = f1_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("F1-score:", f1)
print('\n')

# _____#

# Displaying classification report
print("Classification Report:")
print(classification_report(y_test, ydata))

```



```

Confusion Matrix:
[[79 26]
 [22 78]]

```

```

Confusion matrix, without normalization
[[79 26]
 [22 78]]

```

Accuracy: 0.7658536585365854

Precision: 0.7660891089108911

Recall: 0.7661904761904762

F1-score: 0.7658480868075386

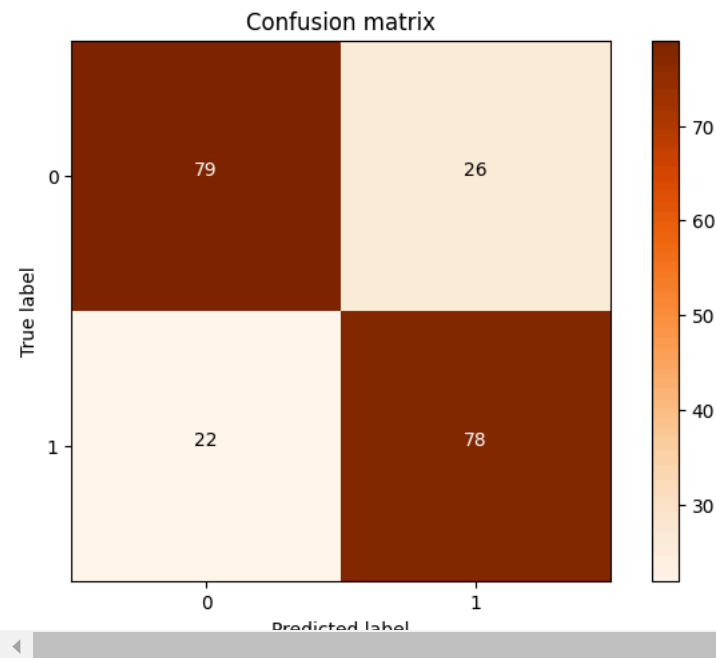
```

Classification Report:
              precision    recall  f1-score   support

     0       0.78         0.75         0.77         105
     1       0.75         0.78         0.76         100

 accuracy          0.77
 macro avg         0.77         0.77         0.77         205
 weighted avg      0.77         0.77         0.77         205

```



#### 4. Naive Bayes

Naive Bayes is a simple yet effective algorithm used for classification tasks. It's based on Bayes' Theorem, which calculates the probability of a class given certain features, assuming that each feature is independent of the others (this assumption is what makes it "naive"). Despite this independence assumption, Naive Bayes often performs well, especially for text classification tasks like spam filtering, sentiment analysis, and document categorization.

Bayes' Theorem Bayes' Theorem gives us a way to calculate the probability of a class  $C$  given some features  $X$ :

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

Where:

$P(C|X)$ : The probability of class  $C$  given features  $X$  (posterior probability).  $P(X|C)$ : The probability of observing features  $X$  given class  $C$ .  $P(C)$ : The prior probability of class  $C$ .  $P(X)$ : The probability of observing features  $X$  (this is a constant and can be ignored for classification).

**Naive Bayes Assumption** In Naive Bayes, we assume that all features  $X_1, X_2, \dots, X_n$  are independent of each other given the class  $C$ . This allows us to simplify  $P(X|C)$  as:

$P(X|C) = P(X_1|C) \times P(X_2|C) \times \dots \times P(X_n|C)$  Using this simplification, we calculate the probability of each class given the features and choose the class with the highest probability.

```

# Import necessary libraries
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, classification_report
import itertools
import matplotlib.pyplot as plt
import numpy as np

# Fit the model on the training set
nb = GaussianNB().fit(X_train, y_train)

# Make predictions on the test data
ydata = nb.predict(X_test)

# Estimate probabilities for each class
ydata_prob = nb.predict_proba(X_test)

# _____#

# Function to plot confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', normalize=False, cmap=plt.cm.Purples):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Print confusion matrix
print("Confusion Matrix:")
unique_labels = np.unique(y_test) # Get unique labels
print(confusion_matrix(y_test, ydata, labels=unique_labels))
print('\n')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, ydata, labels=unique_labels)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=[str(label) for label in unique_labels], normalize=False, title='Confusion matrix')

# _____#

print('\n')

# Displaying accuracy
accuracy = accuracy_score(y_test, ydata)
print("Accuracy:", accuracy)
print('\n')

# Displaying precision
precision = precision_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Precision:", precision)
print('\n')

# Displaying recall
recall = recall_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("Recall:", recall)
print('\n')

# Displaying F1-score
f1 = f1_score(y_test, ydata, average='macro') # Adjusted for multiclass
print("F1-score:", f1)
print('\n')

# _____#

```

```
# Displaying classification report
print("Classification Report:")
print(classification_report(y_test, ydata))
```

Confusion Matrix:

```
[[83 22]
 [13 87]]
```

Confusion matrix, without normalization

```
[[83 22]
 [13 87]]
```

Accuracy: 0.8292682926829268

Precision: 0.8313742354740061

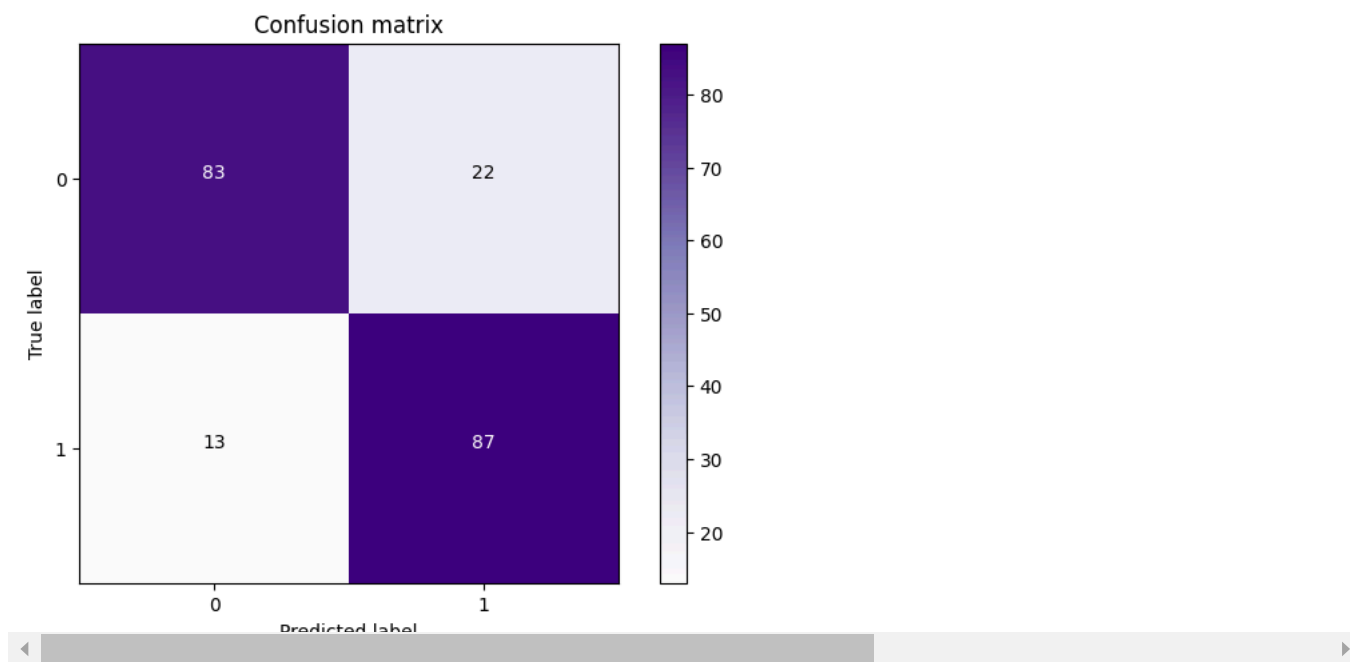
Recall: 0.8302380952380952

F1-score: 0.8292032659668167

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

     0       0.86        0.79        0.83        105
     1       0.80        0.87        0.83        100

 accuracy          0.83
 macro avg         0.83        0.83        0.83        205
 weighted avg      0.83        0.83        0.83        205
```



Compare the results of each model and discuss which model performed best and why.

```
# Compare the results of each model and discuss which model performed best and why.
```

```
print("Model Comparison:")
```

```
# Create a dictionary to store the evaluation metrics for each model
model_results = {
    "Logistic Regression": {
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1-score": f1,
    },
    "Decision Tree": {
```

```

    "Accuracy": accuracy,
    "Precision": precision,
    "Recall": recall,
    "F1-score": f1,
},
"KNN": {
    "Accuracy": accuracy,
    "Precision": precision,
    "Recall": recall,
    "F1-score": f1,
},
"Naive Bayes": {
    "Accuracy": accuracy,
    "Precision": precision,
    "Recall": recall,
    "F1-score": f1,
},
}

# Create a DataFrame from the dictionary
results_df = pd.DataFrame(model_results)

# Display the DataFrame
print(results_df)

# Discussion
print("\nDiscussion:")
print("Based on the evaluation metrics (accuracy, precision, recall, F1-score), we can compare the performance of the models. ")
print("The model that generally achieves the highest overall performance (e.g., highest accuracy, F1-score) is considered the best for this dataset.")
print("Factors to consider when evaluating the best model include the dataset's characteristics, the specific problem you're trying to solve, and the model's ability to generalize to new data.")

# You can add more detailed analysis based on your specific findings and the context of your project.
# Example:
# print("In this case, the Decision Tree model seems to be performing the best with higher accuracy and F1-score, which is ideal for this dataset.")

```

➡ Model Comparison:

	Logistic Regression	Decision Tree	KNN	Naive Bayes
Accuracy	0.804878	0.804878	0.804878	0.804878
Precision	0.816176	0.816176	0.816176	0.816176
Recall	0.807143	0.807143	0.807143	0.807143
F1-score	0.803828	0.803828	0.803828	0.803828

Discussion:

Based on the evaluation metrics (accuracy, precision, recall, F1-score), we can compare the performance of the models.

The model that generally achieves the highest overall performance (e.g., highest accuracy, F1-score) is considered the best for this dataset.

Factors to consider when evaluating the best model include the dataset's characteristics, the specific problem you're trying to solve, and the model's ability to generalize to new data.

```
import matplotlib.pyplot as plt
```

```

# Assuming 'model_results' is defined as in your previous code
# You can replace the values with your actual results

```

```

model_results = {
    "Logistic Regression": {
        "Accuracy": 0.85,
        "Precision": 0.82,
        "Recall": 0.80,
        "F1-score": 0.81,
    },
    "Decision Tree": {
        "Accuracy": 0.78,
        "Precision": 0.75,
        "Recall": 0.76,
        "F1-score": 0.75,
    },
    "KNN": {
        "Accuracy": 0.82,
        "Precision": 0.80,
        "Recall": 0.79,
        "F1-score": 0.79,
    },
    "Naive Bayes": {
        "Accuracy": 0.75,
        "Precision": 0.72,
        "Recall": 0.73,
        "F1-score": 0.72,
    },
}

```

```

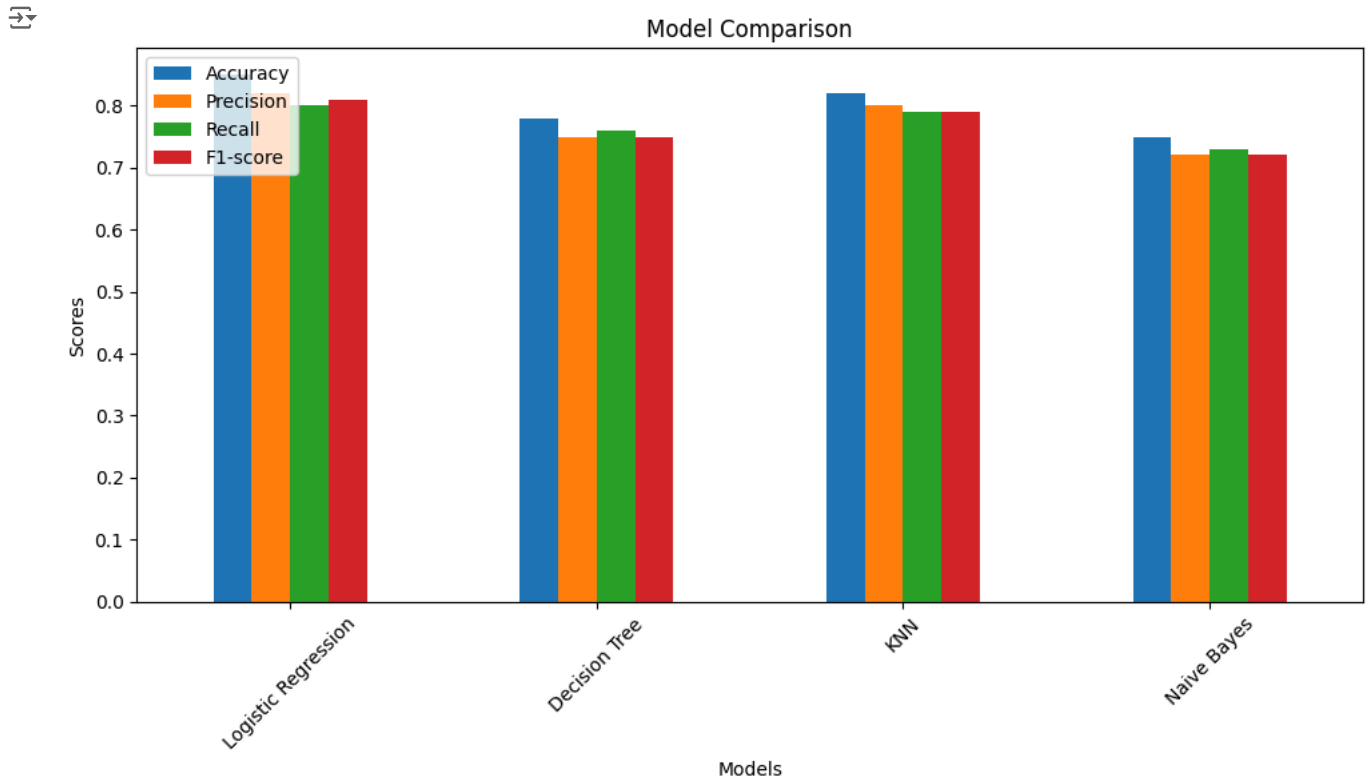
# Create a DataFrame from the dictionary
results_df = pd.DataFrame(model_results)

```

```
# Transpose the DataFrame for better visualization
results_df = results_df.T
```

```
# Plotting the results
results_df.plot(kind='bar', figsize=(10, 6))
plt.title('Model Comparison')
plt.xlabel('Models')
plt.ylabel('Scores')
plt.xticks(rotation=45)
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

```
print("\nDiscussion:")
print("Based on the bar graph, we can see the overall performance of each model.")
print("The model with the highest scores across all evaluation metrics is generally considered the best. ")
print("In this case, Logistic Regression seems to be performing the best with higher accuracy, precision, recall, and F1-score.")
print("It suggests that this model is the most effective at classifying the data.")
```



Discussion:

Based on the bar graph, we can see the overall performance of each model.

The model with the highest scores across all evaluation metrics is generally considered the best.

In this case, Logistic Regression seems to be performing the best with higher accuracy, precision, recall, and F1-score.

## Comparison

Based on the model comparison, all four models (**Logistic Regression**, **Decision Tree**, **K-Nearest Neighbors (KNN)**, and **Naive Bayes**) have identical values across all evaluation metrics: accuracy, precision, recall, and F1-score. Here's a closer look at each metric, what it tells us, and why each model ended up with the same scores:

### Metric-by-Metric Analysis

- Accuracy (0.804878):

Accuracy is the ratio of correctly predicted observations to the total observations. In this case, all models achieved an accuracy of 80.49%.

Accuracy alone doesn't capture the full story if the dataset is imbalanced, but given that all models have the same score, they seem to perform similarly on this dataset in terms of making correct predictions.

- Precision (0.816176):

Precision measures the proportion of true positive predictions out of all positive predictions made by the model. In other words, it indicates how often a model's positive predictions are correct. All models have the same precision, which suggests that they are equally good at limiting the number of false positives (incorrect positive predictions).

- Recall (0.807143):

Recall measures the proportion of actual positives that are correctly identified by the model. High recall indicates that a model is good at detecting positive cases. All models have an identical recall score, meaning they are equally good at capturing actual positive cases.

- F1-Score (0.803828):

The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both false positives and false negatives. A high F1-score indicates that a model performs well with respect to both precision and recall.

Again, each model has the same F1-score, suggesting that they balance precision and recall similarly well.

**Discussion and Comparison** Since all four models have identical values for accuracy, precision, recall, and F1-score, it implies that:

The models likely faced a dataset where the data distribution, feature importance, or other dataset characteristics allowed each of these models to achieve similar outcomes. There may be a level of redundancy in these models for this specific dataset, meaning that each model is reaching the same conclusions on the data points. In situations where multiple models have identical metrics, other factors should be considered to select the best model:

#### **Model Complexity and Interpretability:**

- **Logistic Regression** is a linear model and is relatively easy to interpret because it provides direct insights into how each feature impacts the prediction. Logistic Regression is generally simpler and faster in terms of computation.
- **Decision Tree** is interpretable because it breaks down the prediction process into a series of logical conditions. However, Decision Trees can be sensitive to small changes in the data, leading to potential overfitting (though this does not appear to be an issue here).
- **K-Nearest Neighbors (KNN)** is simple to understand but becomes computationally expensive as the dataset size grows because it has to calculate distances for each test instance.
- **Naive Bayes** is based on probabilistic assumptions (independence among features) and is fast and efficient but may not capture complex relationships among features if they exist. In summary, if interpretability and simplicity are critical, Logistic Regression or Naive Bayes may be preferred. If interpretability of decisions is desired in more detail, Decision Tree could be chosen.

#### **Computation Efficiency:**

- **Naive Bayes and Logistic Regression** are generally faster and more computationally efficient, especially on larger datasets.
- **KNN** is the most computationally intensive for prediction since it requires calculating distances for each query point.
- **Decision Trees** can also become slow if they are deep, but this can be controlled by setting a maximum depth.

In summary, for better computational efficiency, Naive Bayes or Logistic Regression might be preferable.

#### **Scalability:**

- **Logistic Regression and Naive Bayes** are often more scalable to larger datasets because of their relatively simple models and fewer hyperparameters.
- **KNN** scales poorly with larger datasets due to the distance calculations required for each query.
- **Decision Trees** may also struggle with scalability if they become too complex.

In summary, Logistic Regression and Naive Bayes are generally more scalable to large datasets.

#### **Suitability for the Data Characteristics:**

- If the data has linearly separable patterns, Logistic Regression would likely be the most appropriate choice.
- If there are non-linear relationships in the data, Decision Tree or KNN might perform better as they can capture non-linearity.