HEART DISEASE PREDICTION

Abstract

Early diagnosis of cardiovascular disease is critical for effective medical management/intervention and successful patient outcomes. We compare three classification algorithms, Logistic Regression, k-Nearest Neighbors (k-NN), and Decision Tree, using the Heart Failure Prediction data set from Kaggle. This data set consisted of numerical and categorical attributes about indicators of patient health. After the complete pre-processing of the data which included outlier management, encoding, and feature scaling the models were evaluated by training them, and viewing the results on various metrics (accuracy, precision, recall, and F1-score). Overall, the k-NN model outperformed the other models with the highest accuracy rate (84.2 %) followed closely by Logistic Regression (83.7 %). Decision tree model had the least performance (79.3 %), but it was viewable and understandable. This paper also highlighted the balancing act of model performance, interpretability, and the complexities involved with health related classification problems. The research results provided useful information about decision making on appropriate algorithms to use for similar predictive health scenarios.

1. Introduction

Heart disease continues to have a significant mortality impact on the global population, affecting millions of individuals each year across many different populations. Estimating the probability of heart-related disease complications based on patient data can potentially offer physicians and healthcare professionals promising directed clinical care guidelines as well as individual treatment recommendations for higher-quality patient care. As the volume of data becomes more available in healthcare, the deployment of machine learning within clinician decision support systems will become increasingly feasible, as well as vital. This research study examines the predictive capabilities of three generalized machine learning classification algorithms to predict the risk of heart disease (based on patient health features). The algorithms include logistic regression, k-nearest neighbors (k-NN), and decision trees, with these models strategically chosen due to their prevalence, interpretability, and suitability for defendable deployment in real-world healthcare delivery. The dataset for this research study was sourced from Kaggle, featuring categorical and continuous strength of health features including age, gender, new chest pain types, cholesterol levels, blood pressure, and electrocardiogram results. To completely prepare the models for performance, several steps required pre-processing including outlier detection/removal, one-hot encoding of categorical data, evaluation of the features, and standardization of the final features. The purpose of the paper is to answer three questions: which machine learning model provided the best predictive accuracy to classify heart disease? What were the accuracy, interpretability, and robustness trade-offs for each model? How can insights from the models be used to help healthcare professionals for diagnostic purposes? The rest of the paper is structured as follows: Section 2 reviews literature, Section 3 describes the dataset, preprocessing steps and modeling methodologies, Section 4 describes the evaluation metrics and reports out the results of the models, Section 5 discusses limitations and results, and Section 6 draws a conclusion and describes future work.

2. Literature Review

In recent years, the overlap between healthcare and machine learning has gained traction, especially with early prediction and diagnosis of cardiovascular disease. Accurate prediction models not only assist with clinical decision-making but also influence resource utilization and individualized patient care. Dinh et al. (2019) was one of the first studies to examine multiple machine learning algorithms, including logistic regression, support vector machines (SVM), and decision trees, using clinical datasets, for heart disease prediction. Their research demonstrated how SVMs and ensemble methods offer slightly better accuracy measures than logistic regression models. However, logistic regression relied on a transparent and easy-to-understand logistic equation, which remains the most useful for properly classifying patient risk factors. The authors also suggested considering balance, such as accuracy with transparency, in the clinical or health context (Dinh et al., 2019). Another important study conducted by Hasan et al. (2021) analyzed a variety of supervised learning methods on the UCI Heart Disease dataset and found that decision trees display comparable accuracy with the bonus of being visually interpretable, which is an especially important characteristic for model transparency in a clinical setting. The authors note that both k-NN and decision trees produce similar accuracy, but logistic regression often produces greater robustness and stability on datasets with few features and low variance (Hasan et al., 2021). Recent developments in explainable AI have encouraged researchers to choose simpler models, or to apply SHAP and LIME explainability approaches, while utilizing black-box models. This is supported by the importance ascribed to logistic regression and decision trees in medical applications where medical professionals require a justification for each prediction. Ultimately, this literature set the groundwork for the comparative lens through which this paper takes, which is the evaluation of Logistic Regression, k-NN, and Decision Tree algorithms on heart disease prediction. This study adds to the comparative process of different ML models within an applied context, as this consideration of a real-world dataset is paramount to understanding the possible trade-offs between predictive capability and interpretability for medical diagnosis tools.

3. Methodology

Dataset Download from Kaggle

To begin the analysis, we retrieved the Heart Failure Prediction dataset from Kaggle using the `KaggleApi`. The API requires prior authentication using a valid `kaggle.json` token file located in the default directory (`~/.kaggle/`).

The dataset was downloaded in compressed format and then extracted to the local `./heart_data` directory for further processing.

This approach ensures reproducibility and ease of access when working with external datasets directly from Kaggle's public repository.

```
from kaggle.api.kaggle_api_extended import KaggleApi
import zipfile

# Authenticate using the default path (~/.kaggle/kaggle.json)
api = KaggleApi()
```

```
api.authenticate()

# Download dataset
api.dataset_download_files('fedesoriano/heart-failure-prediction',
path='./heart_data', unzip=False)

# Unzip
with zipfile.ZipFile('./heart_data/heart-failure-prediction.zip', 'r')
as zip_ref:
    zip_ref.extractall('./heart_data')

print("Dataset downloaded and unzipped successfully.")
```

```
Dataset URL: https://www.kaggle.com/datasets/fedesoriano/heart-
failure-prediction
Dataset downloaded and unzipped successfully.
```

Data Loading and Preview

After extracting the dataset, we load the CSV file into a Pandas DataFrame using `pd.read_csv()`. This allows us to perform tabular data operations with ease.

We then preview the first five rows of the dataset to understand its structure and verify successful loading.

```
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.read_csv("./heart_data/heart.csv")

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

```
   Age Sex ChestPainType  RestingBP  Cholesterol  FastingBS RestingECG
MaxHR  \
0   40   M           ATA        140          289          0     Normal
172
1   49   F           NAP        160          180          0     Normal
156
2   37   M           ATA        130          283          0         ST
98
3   48   F           ASY        138          214          0     Normal
108
4   54   M           NAP        150          195          0     Normal
122

  ExerciseAngina  Oldpeak ST_Slope  HeartDisease
0              N      0.0       Up             0
1              N      1.0     Flat             1
2              N      0.0       Up             0
```

```
3              Y      1.5     Flat             1
4              N      0.0       Up             0
```

Basic Dataset Overview and Missing Value Check

This code performs an initial exploration of the dataset `df` to understand its structure and completeness:

1. Dataset Shape:
   - Displays the total number of rows and columns using `df.shape`.
2. Column Names:
   - Lists all column names present in the dataset using `df.columns.tolist()` to understand the available features.
3. Data Types:
   - Prints the data types of each column using `df.dtypes` to verify if they are numeric, categorical, or otherwise. This is crucial for preprocessing decisions.
4. Missing Values:
   - Checks for any missing (NaN) values in each column using `df.isnull().sum()` to ensure data quality before modeling.

This step helps you quickly validate the dataset's structure and readiness for further analysis or cleaning.

```
# Basic dataset overview
print("Dataset Shape:", df.shape)
print("\nColumn Names:", df.columns.tolist())
print("\nData Types:")
print(df.dtypes)

# Check for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
Dataset Shape: (918, 12)

Column Names: ['Age', 'Sex', 'ChestPainType', 'RestingBP',
'Cholesterol', 'FastingBS', 'RestingECG', 'MaxHR', 'ExerciseAngina',
'Oldpeak', 'ST_Slope', 'HeartDisease']

Data Types:
Age                 int64
Sex                object
ChestPainType      object
RestingBP           int64
Cholesterol         int64
FastingBS           int64
RestingECG         object
MaxHR               int64
```

```
ExerciseAngina     object
Oldpeak           float64
ST_Slope           object
HeartDisease        int64
dtype: object

Missing Values:
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

Exploratory Data Analysis: Summary Statistics and Class Distribution

This section provides a descriptive summary of the dataset's numeric features and examines the distribution of the target variable `HeartDisease`.

- The `describe()` function is used to generate summary statistics (count, mean, standard deviation, min, max, and quartiles) for all numeric columns in the dataset.
- The distribution of the target variable `HeartDisease` is evaluated using `value_counts()` to understand the frequency of each class (0: No heart disease, 1: Heart disease).

Understanding these distributions helps assess data balance and guides further preprocessing and modeling decisions.

```
# Summary statistics for numeric variables
print("Summary Statistics:\n")
print(df.describe())

# Distribution of target variable
print("\nHeartDisease Class Distribution:")
print(df['HeartDisease'].value_counts())
```

```
Summary Statistics:

              Age   RestingBP  Cholesterol   FastingBS       MaxHR  \
count  918.000000  918.000000   918.000000  918.000000  918.000000
mean    53.510893  132.396514   198.799564    0.233115  136.809368
std      9.432617   18.514154   109.384145    0.423046   25.460334
```

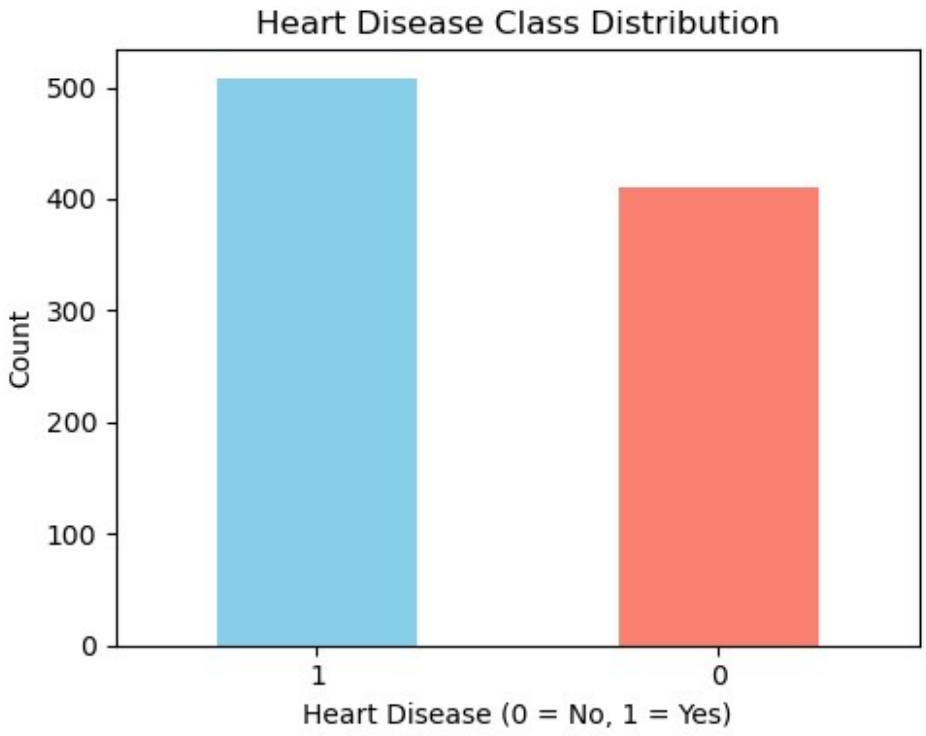
```
min     28.000000    0.000000     0.000000    0.000000   60.000000
25%     47.000000  120.000000   173.250000    0.000000  120.000000
50%     54.000000  130.000000   223.000000    0.000000  138.000000
75%     60.000000  140.000000   267.000000    0.000000  156.000000
max     77.000000  200.000000   603.000000    1.000000  202.000000

          Oldpeak  HeartDisease
count  918.000000    918.000000
mean     0.887364      0.553377
std      1.066570      0.497414
min     -2.600000      0.000000
25%      0.000000      0.000000
50%      0.600000      1.000000
75%      1.500000      1.000000
max      6.200000      1.000000

HeartDisease Class Distribution:
HeartDisease
1    508
0    410
Name: count, dtype: int64
```

Visualizing Target Variable Distribution

This plot visualizes the distribution of the target variable `HeartDisease` using a bar chart.

- The dataset contains two classes:
  - `0`: No heart disease
  - `1`: Presence of heart disease
- A bar chart is plotted to depict the frequency of each class, using distinct colors for visual clarity.
- This visualization is helpful to assess class balance, which is critical when selecting and evaluating classification models.

```
import matplotlib.pyplot as plt

# Plot target class distribution
plt.figure(figsize=(5,4))
df['HeartDisease'].value_counts().plot(kind='bar', color=['skyblue',
'salmon'])
plt.title("Heart Disease Class Distribution")
plt.xlabel("Heart Disease (0 = No, 1 = Yes)")
plt.ylabel("Count")
plt.xticks(rotation=0)
plt.tight_layout()
plt.show()
```

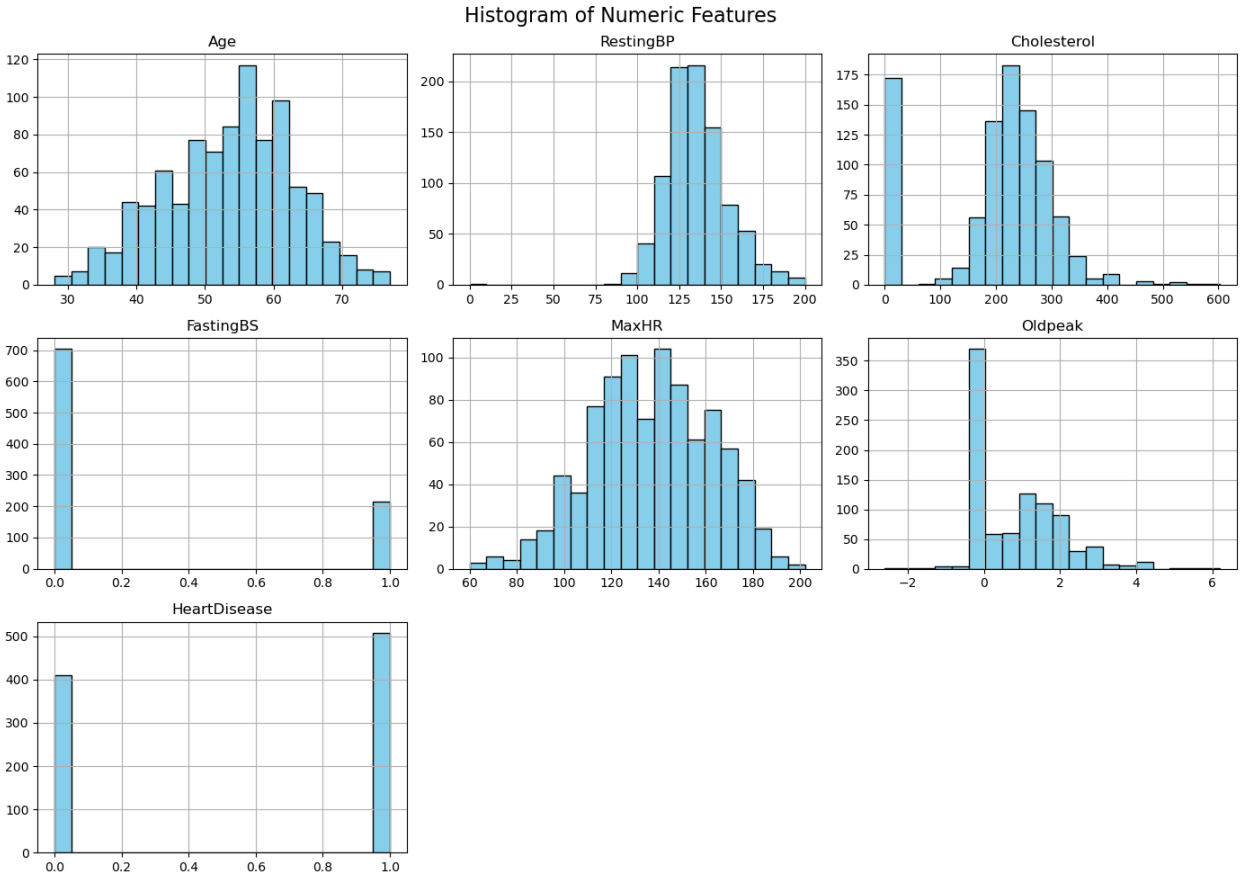


Histogram of Numeric Features

This step visualizes the distribution of all numeric variables in the dataset using histograms.

- Numeric columns are selected based on their data types (`int64` and `float64`).
- Histograms provide insights into the distribution, skewness, and potential outliers within each feature.
- The visual summary helps in identifying normalization needs or transformation requirements for model development.

```
# Plot histograms for all numeric features
numeric_cols = df.select_dtypes(include=['int64', 'float64']).columns

df[numeric_cols].hist(figsize=(14, 10), bins=20, color='skyblue',
edgecolor='black')
plt.suptitle("Histogram of Numeric Features", fontsize=16)
plt.tight_layout()
plt.show()
```



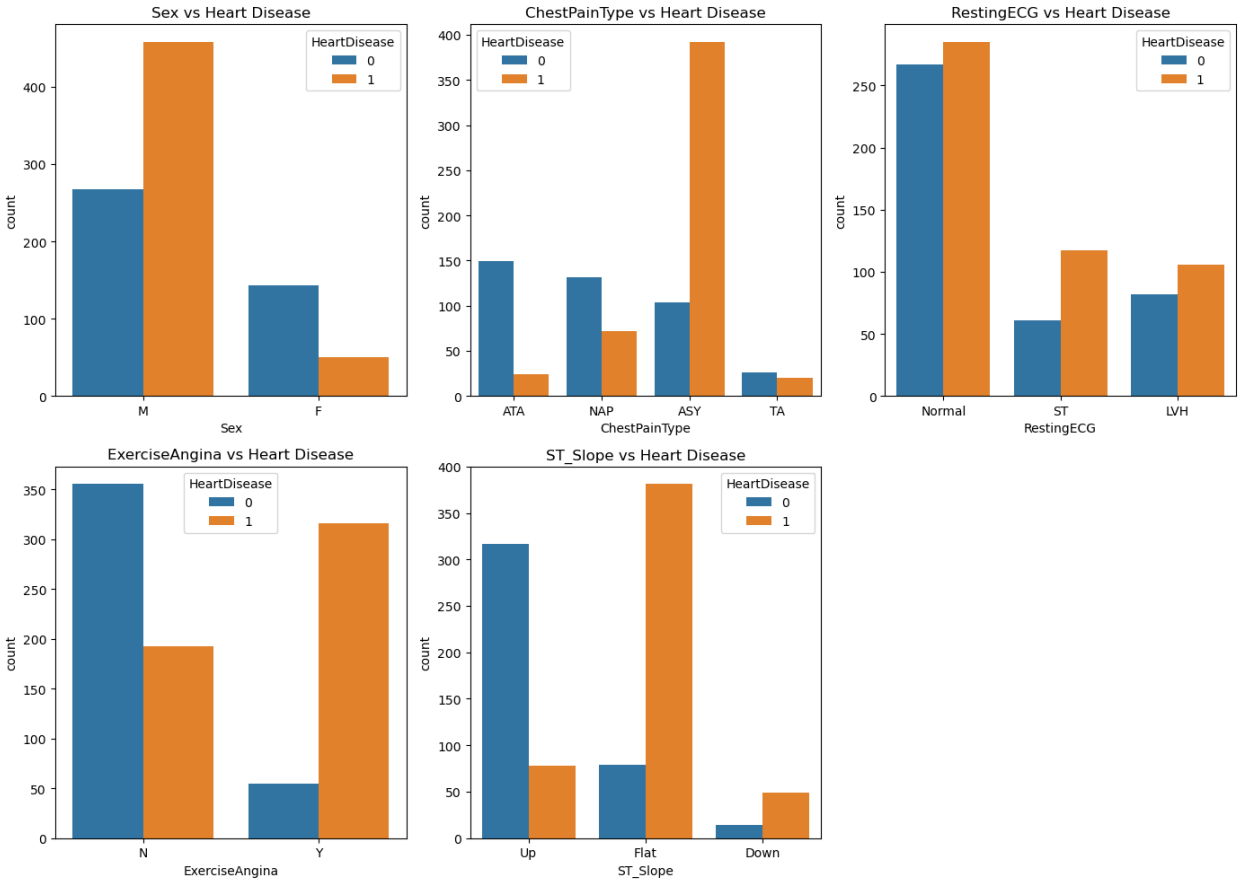
Categorical Feature Distribution by Heart Disease

This section visualizes the relationship between each categorical variable and the target variable `HeartDisease` using count plots.

- Categorical features analyzed include: `Sex`, `ChestPainType`, `RestingECG`, `ExerciseAngina`, and `ST_Slope`.
- Each subplot shows the count of occurrences for each category, split by heart disease status (0 = No, 1 = Yes).
- These plots help identify patterns or associations between categorical attributes and the presence of heart disease.

```
import seaborn as sns

categorical_cols = ['Sex', 'ChestPainType', 'RestingECG',
'ExerciseAngina', 'ST_Slope']

plt.figure(figsize=(14, 10))

for i, col in enumerate(categorical_cols):
    plt.subplot(2, 3, i+1)
    sns.countplot(data=df, x=col, hue='HeartDisease')
    plt.title(f"{col} vs Heart Disease")
```

```
    plt.xticks(rotation=0)

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



One-Hot Encoding and Correlation Analysis

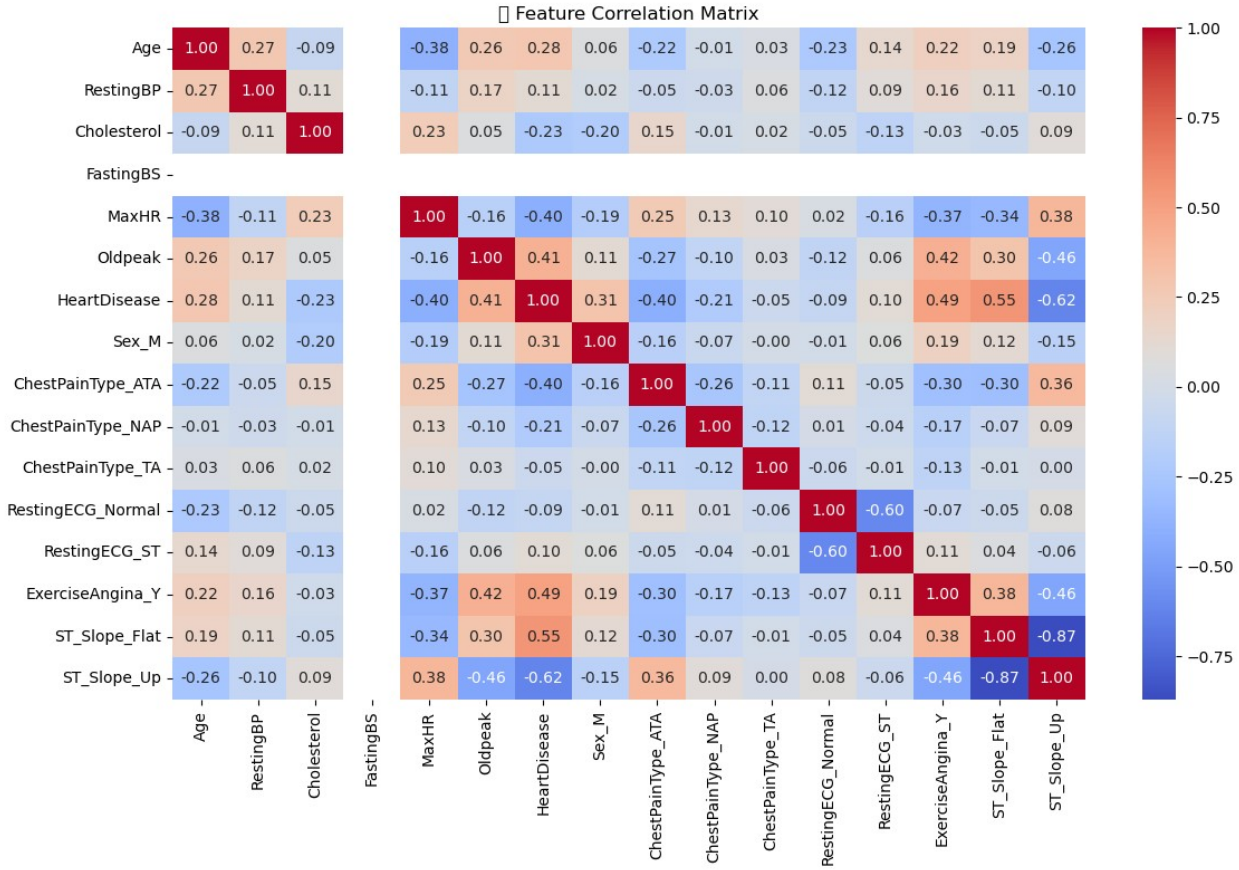
This section prepares the dataset for machine learning by applying one-hot encoding to categorical variables and examining feature correlations.

- One-hot encoding is performed using `pd.get_dummies()` with `drop_first=True` to avoid multicollinearity from dummy variable traps.
- The shape of the encoded dataset is printed to confirm dimensionality.
- A correlation matrix heatmap is plotted using Seaborn to visualize pairwise correlations among all features, including encoded variables.
- This analysis is useful for detecting multicollinearity and understanding linear relationships between predictors and the target variable.

```
# One-hot encode categorical features
df_encoded = pd.get_dummies(df, drop_first=True)

# Check encoded shape
```

```
print(f"Encoded dataset shape: {df_encoded.shape}")

# Plot correlation heatmap
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
sns.heatmap(df_encoded.corr(), annot=True, fmt=".2f", cmap='coolwarm')
plt.title("□ Feature Correlation Matrix")
plt.tight_layout()
plt.show()
```

```
Encoded dataset shape: (918, 16)

/var/folders/yc/q56lmvx51l9bz3xkjddt61200000gn/T/
ipykernel_86558/749930227.py:14: UserWarning: Glyph 128269 (\N{LEFT-
POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
pylabtools.py:170: UserWarning: Glyph 128269 (\N{LEFT-POINTING
MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



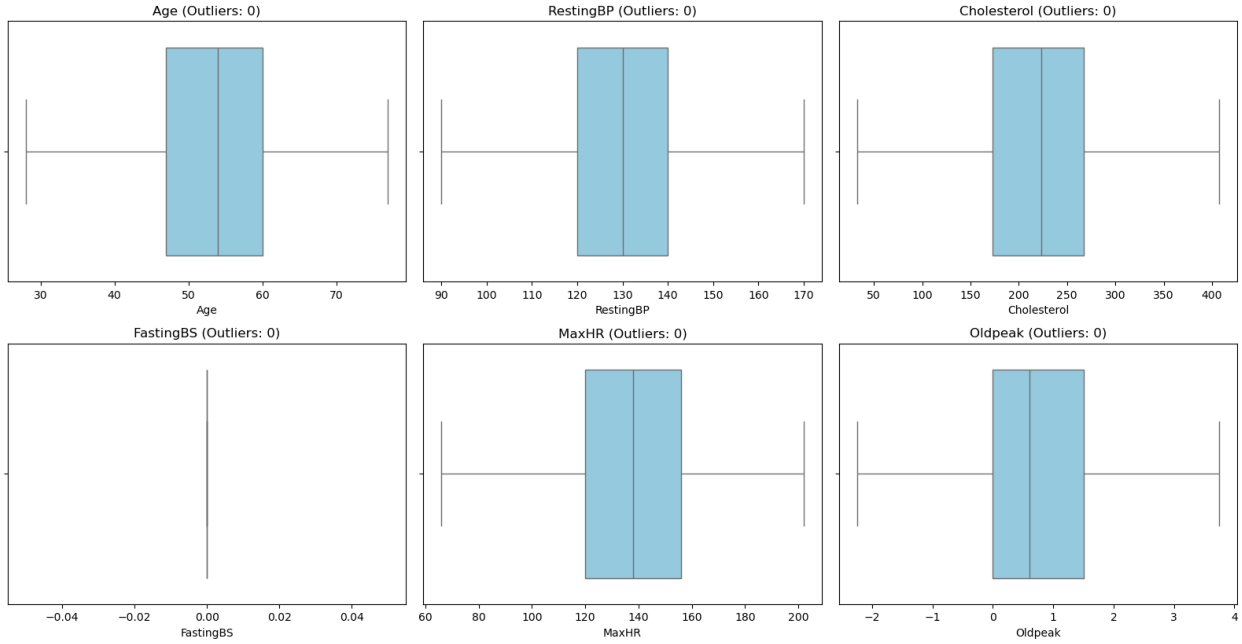
Outlier Detection and Visualization

This section focuses on identifying outliers in the numeric features of the dataset using the Interquartile Range (IQR) method and visualizing them with boxplots.

- The IQR method is applied to each numeric column (excluding the target variable `HeartDisease`) to determine lower and upper bounds for detecting outliers.
- Outliers are defined as values falling below Q1 - 1.5 × IQR or above Q3 + 1.5 × IQR.
- The function `detect_outliers_with_plots()` returns a summary of outlier counts, percentages, and indices for each feature.
- Boxplots are generated for each numeric feature to visually inspect the distribution and presence of outliers.
- This analysis helps in understanding data variability and assessing whether outlier handling is necessary before modeling.

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

# Select numeric columns except target
numeric_cols = df.select_dtypes(include=['int64',
'float64']).columns.tolist()
numeric_cols.remove('HeartDisease')  # Do not modify target

# Function to detect and visualize outliers
def detect_outliers_with_plots(df, cols):
    outlier_info = {}

    plt.figure(figsize=(16, 12))
    for idx, col in enumerate(cols, 1):
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower = Q1 - 1.5 * IQR
        upper = Q3 + 1.5 * IQR
        outliers = df[(df[col] < lower) | (df[col] > upper)]

        outlier_info[col] = {
            'count': len(outliers),
            'percent': round(100 * len(outliers) / len(df), 2),
            'indices': outliers.index.tolist()
        }

        # Box plot
        plt.subplot(3, 3, idx)
        sns.boxplot(x=df[col], color='skyblue')
        plt.title(f"{col} (Outliers: {len(outliers)})")

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

```
    return outlier_info

# Run detection
outlier_summary = detect_outliers_with_plots(df, numeric_cols)

# Print summary
print("Outlier Summary:")
for col, info in outlier_summary.items():
    print(f"{col}: {info['count']} outliers ({info['percent']}%)")
```




```
Outlier Summary:
Age: 0 outliers (0.0%)
RestingBP: 0 outliers (0.0%)
Cholesterol: 0 outliers (0.0%)
FastingBS: 0 outliers (0.0%)
MaxHR: 0 outliers (0.0%)
Oldpeak: 0 outliers (0.0%)
```

Outlier Treatment via Capping

To address the presence of outliers without removing data points, this section implements a capping strategy using the Interquartile Range (IQR) method.

- The function `cap_outliers()` takes a DataFrame and a list of numeric columns as input.
- For each column, it calculates the lower and upper bounds based on 1.5 × IQR.
- Values below the lower bound are capped to the lower bound, and values above the upper bound are capped to the upper bound.

- This preserves the original dataset size while minimizing the influence of extreme values on model performance.
- The resulting DataFrame `df_capped` contains adjusted values for all numeric features with outliers effectively mitigated.

```
def cap_outliers(df, cols):
    df_capped = df.copy()

    for col in cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower = Q1 - 1.5 * IQR
        upper = Q3 + 1.5 * IQR

        df_capped[col] = np.where(df[col] < lower, lower, df[col])
        df_capped[col] = np.where(df_capped[col] > upper, upper,
df_capped[col])

    return df_capped

# Apply capping
df_capped = cap_outliers(df, numeric_cols)
```

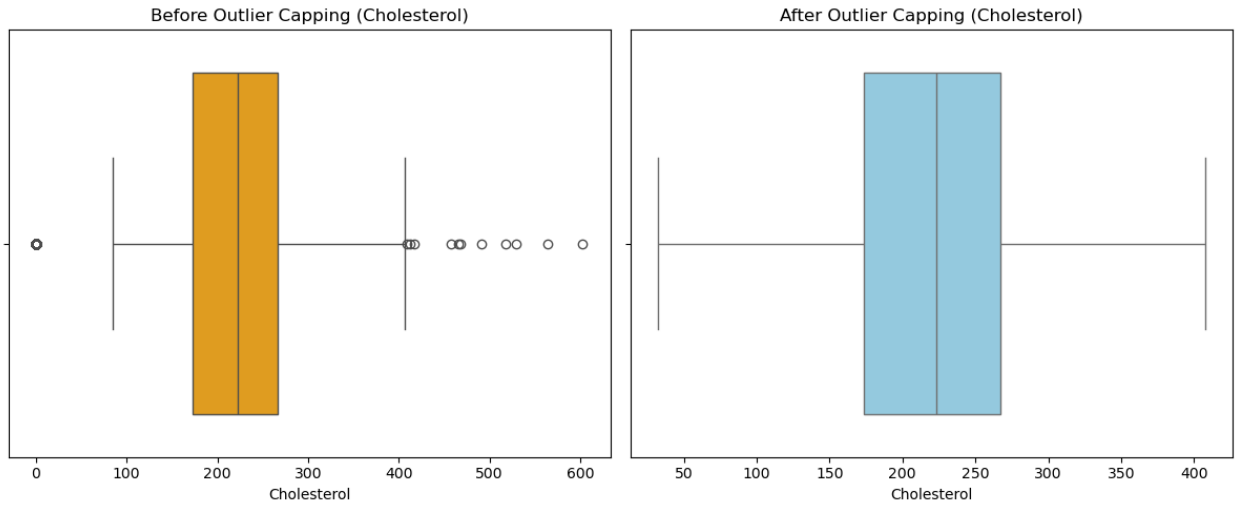
Visualizing the Effect of Outlier Capping

This section compares the distribution of a feature before and after outlier treatment to visually validate the impact of capping.

- The function `compare_outlier_handling()` takes the original and capped DataFrames along with a target column name.
- Two side-by-side box plots are generated:
  - The left plot shows the distribution of the original data with outliers.
  - The right plot shows the distribution after capping has been applied.
- This visual comparison helps confirm whether extreme values have been successfully constrained without altering the overall distribution.
- The current example demonstrates this comparison for the `Cholesterol` feature.

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

def compare_outlier_handling(df_original, df_capped, col):
    plt.figure(figsize=(12, 5))

    # Before capping
    plt.subplot(1, 2, 1)
    sns.boxplot(x=df_original[col], color="orange")
    plt.title(f"Before Outlier Capping ({col})")

    # After capping
```

```
    plt.subplot(1, 2, 2)
    sns.boxplot(x=df_capped[col], color="skyblue")
    plt.title(f"After Outlier Capping ({col})")

    plt.tight_layout()
    plt.show()

# Example usage: Visualize for 'Cholesterol'
compare_outlier_handling(df, df_capped, 'Cholesterol')
```



Updating Dataset with Capped Outliers

The original dataset is updated by replacing the numeric columns with their capped versions to mitigate the influence of extreme values.

- The `.update()` function is used to overwrite the original values with the capped values only for the selected numeric columns.
- After the update, summary statistics are displayed again using `.describe()` to confirm the adjusted ranges and ensure that the extreme outliers have been effectively limited.

```
# Replace original numeric columns with capped values
df.update(df_capped)

# Confirm updated summary
print("Updated Summary Statistics (Post-Capping):")
print(df[numeric_cols].describe())
```

```
Updated Summary Statistics (Post-Capping):
              Age   RestingBP  Cholesterol  FastingBS       MaxHR
Oldpeak
count  918.000000  918.000000   918.000000      918.0  918.000000
918.000000
mean    53.510893  132.125272   203.985158        0.0  136.819172
0.877778
```

```
std      9.432617   16.993314    95.922341        0.0   25.432057
1.031693
min     28.000000   90.000000    32.625000        0.0   66.000000   -
2.250000
25%     47.000000  120.000000   173.250000        0.0  120.000000
0.000000
50%     54.000000  130.000000   223.000000        0.0  138.000000
0.600000
75%     60.000000  140.000000   267.000000        0.0  156.000000
1.500000
max     77.000000  170.000000   407.625000        0.0  202.000000
3.750000

/var/folders/yc/q56lmvx51l9bz3xkjddt61200000gn/T/
ipykernel_86558/1280643516.py:2: FutureWarning: Setting an item of
incompatible dtype is deprecated and will raise in a future error of
pandas. Value '[289.    180.    283.    214.    195.    339.    237.
208.    207.
 284.    211.    164.    204.    234.    211.    273.    196.    201.
 248.    267.    223.    184.    201.    288.    215.    209.    260.
 284.    407.625 188.    407.625 167.    224.    172.    186.    254.
 306.    250.    177.    227.    230.    294.    264.    259.    175.
 318.    223.    216.    340.    289.    233.    205.    224.    245.
 180.    194.    270.    213.    365.    342.    253.    254.    224.
 277.    202.    260.    297.    225.    246.    407.625 265.    215.
 182.    218.    268.    163.    407.625 167.    100.    206.    277.
 238.    223.    196.    213.    139.    263.    216.    291.    229.
 208.    307.    210.    329.    182.    263.    207.    147.     85.
 269.    275.    179.    392.    407.625 186.    260.    254.    214.
 129.    241.    188.    255.    276.    297.    207.    246.    282.
 338.    160.    156.    248.    272.    240.    393.    230.    246.
 161.    163.    230.    228.    292.    202.    388.    230.    294.
 265.    215.    241.    166.    247.    331.    341.    291.    243.
 279.    273.    198.    249.    168.    407.625 215.    159.    275.
 270.    291.    342.    190.    185.    290.    195.    264.    212.
 263.    196.    225.    272.    231.    238.    222.    179.    243.
 235.    320.    187.    266.    288.    216.    287.    194.    238.
 225.    224.    404.    238.    312.    211.    251.    237.    328.
 285.    280.    209.    245.    192.    184.    193.    297.    268.
 246.    308.    249.    230.    147.    219.    184.    215.    308.
 257.    132.    216.    263.    288.    276.    219.    226.    237.
 280.    217.    196.    263.    222.    303.    195.    298.    256.
 264.    195.    117.    295.    173.    315.    281.    275.    250.
 309.    200.    336.    295.    355.    193.    326.    198.    292.
 266.    268.    171.    237.    275.    219.    341.    407.625 260.
 292.    271.    248.    274.    394.    160.    200.    320.    275.
 221.    231.    126.    193.    305.    298.    220.    242.    235.
 225.    198.    201.    220.    295.    213.    160.    223.    347.
 253.    246.    222.    220.    344.    358.    190.    169.    181.
 308.    166.    211.    257.    182.     32.625  32.625  32.625
```

```
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625  32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625 260.    209.    218.    228.    213.     32.625 236.
  32.625  32.625 267.    166.     32.625  32.625  32.625  32.625 220.
 177.    236.     32.625  32.625  32.625  32.625  32.625  32.625
32.625
  32.625  32.625 186.    100.    228.     32.625 171.    230.
32.625
  32.625  32.625 281.     32.625 203.     32.625  32.625  32.625
32.625
  32.625 277.     32.625 233.     32.625  32.625 240.     32.625
32.625
 153.    224.     32.625  32.625  32.625 316.     32.625  32.625 218.
  32.625 311.     32.625  32.625  32.625 270.     32.625  32.625 217.
 214.    214.    252.    220.    214.    203.     32.625 339.    216.
 276.    407.625 241.    384.    297.    248.    308.    208.    227.
 210.    245.    225.    240.     32.625 198.    195.    267.    161.
 258.     32.625  32.625 195.    235.     32.625 305.    223.    282.
 349.    160.    160.    236.    312.    283.    142.    211.    218.
 306.    186.    252.    222.     32.625  32.625 258.    202.    197.
 204.    113.    274.    192.    298.    272.    220.    200.    261.
 181.    260.    220.    221.    216.    175.    219.    310.    208.
 232.    273.    203.    182.    274.    204.    270.    292.    171.
 221.    289.    217.    223.    110.    193.    123.    210.    282.
```

```
 170.    369.    173.    289.    152.    208.    216.    271.    244.
 285.    243.    240.    219.    237.    165.    213.    287.    258.
 256.    186.    264.    185.    226.    203.    207.    284.    337.
 310.    254.    258.    254.    300.    170.    310.    333.    139.
 223.    385.    254.    322.    407.625 261.    263.    269.    177.
 256.    239.    293.    407.    234.    226.    235.    234.    303.
 149.    311.    203.    211.    199.    229.    245.    303.    204.
 288.    275.    243.    295.    230.    265.    229.    228.    215.
 326.    200.    256.    207.    273.    180.    222.    223.    209.
 233.    197.    218.    211.    149.    197.    246.    225.    315.
 205.    407.625 195.    234.    198.    166.    178.    249.    281.
 126.    305.    226.    240.    233.    276.    261.    319.    242.
 243.    260.    354.    245.    197.    223.    309.    208.    199.
 209.    236.    218.    198.    270.    214.    201.    244.    208.
 270.    306.    243.    221.    330.    266.    206.    212.    275.
 302.    234.    313.    244.    141.    237.    269.    289.    254.
 274.    222.    258.    177.    160.    327.    235.    305.    304.
 295.    271.    249.    288.    226.    283.    188.    286.    274.
 360.    273.    201.    267.    196.    201.    230.    269.    212.
 226.    246.    232.    177.    277.    249.    210.    207.    212.
 271.    233.    213.    283.    282.    230.    167.    224.    268.
 250.    219.    267.    303.    256.    204.    217.    308.    193.
 228.    231.    244.    262.    259.    211.    325.    254.    197.
 236.    282.    234.    254.    299.    211.    182.    294.    298.
 231.    254.    196.    240.    407.625 172.    265.    246.    315.
 184.    233.    394.    269.    239.    174.    309.    282.    255.
 250.    248.    214.    239.    304.    277.    300.    258.    299.
 289.    298.    318.    240.    309.    250.    288.    245.    213.
 216.    204.    204.    252.    227.    258.    220.    239.    254.
 168.    330.    183.    203.    263.    341.    283.    186.    307.
 219.    260.    255.    231.    164.    234.    177.    257.    325.
 274.    321.    264.    268.    308.    253.    248.    269.    185.
 282.    188.    219.    290.    175.    212.    302.    243.    353.
 335.    247.    340.    206.    284.    266.    229.    199.    263.
 294.    192.    286.    216.    223.    247.    204.    204.    227.
 278.    220.    232.    197.    335.    253.    205.    192.    203.
 318.    225.    220.    221.    240.    212.    342.    169.    187.
 197.    157.    176.    241.    264.    193.    131.    236.    175.
]' has dtype incompatible with int64, please explicitly cast to a
compatible dtype first.
  df.update(df_capped)
```

Data Type Alignment After Outlier Capping

After capping the outliers, the numeric columns are type-checked and converted to ensure consistency in data types:

- For columns that were originally of integer type (`int64`), the capped values are rounded and cast back to integers.

- For columns with floating-point types, the capped values are retained as-is.
- This step ensures that the dataset maintains its original data schema, which is important for compatibility with later preprocessing and modeling steps.

```
# Convert capped columns to the correct dtype before updating
for col in numeric_cols:
    if df[col].dtype == 'int64':
        df[col] = df_capped[col].round().astype(int)
    else:
        df[col] = df_capped[col]
```

One-Hot Encoding of Categorical Variables

To prepare the dataset for machine learning models, categorical variables are converted into numerical format using one-hot encoding:

- The columns `Sex`, `ChestPainType`, `RestingECG`, `ExerciseAngina`, and `ST_Slope` are transformed into binary (0/1) indicator variables.
- The `drop_first=True` argument is used to avoid the dummy variable trap, which helps prevent multicollinearity by dropping the first category of each variable.
- The resulting encoded dataset is printed with its new shape and the first few rows displayed for inspection.

```
# One-hot encode categorical columns and drop the first category to
avoid multicollinearity
df_encoded = pd.get_dummies(df, columns=['Sex', 'ChestPainType',
'RestingECG', 'ExerciseAngina', 'ST_Slope'], drop_first=True)

# Display shape and first few rows
print("Data shape after encoding:", df_encoded.shape)
df_encoded.head()
```

```
Data shape after encoding: (918, 16)
```

```
   Age  RestingBP  Cholesterol  FastingBS  MaxHR  Oldpeak
HeartDisease  \
0   40        140        289.0          0    172      0.0
0
1   49        160        180.0          0    156      1.0
1
2   37        130        283.0          0     98      0.0
0
3   48        138        214.0          0    108      1.5
1
4   54        150        195.0          0    122      0.0
0

   Sex_M  ChestPainType_ATA  ChestPainType_NAP  ChestPainType_TA  \
0   True               True              False             False
1  False              False               True             False
```

```
2   True                True               False              False
3  False               False               False              False
4   True               False                True              False

   RestingECG_Normal  RestingECG_ST  ExerciseAngina_Y
ST_Slope_Flat  \
0               True          False              False           False

1               True          False              False            True

2              False           True              False           False

3               True          False               True            True

4               True          False              False           False


   ST_Slope_Up
0         True
1        False
2         True
3        False
4         True
```

Convert Boolean Columns to Integer Format

After one-hot encoding, all remaining boolean columns are converted to integer values (`True` → `1`, `False` → `0`) to ensure uniform data types across the dataset. This step is necessary for compatibility with many machine learning algorithms that require numerical input. The data types of all columns are then verified to confirm the conversion.

```
# Convert all boolean columns to integer (True → 1, False → 0)
df_encoded = df_encoded.astype(int)

# Confirm conversion
df_encoded.dtypes
```

```
Age                   int64
RestingBP             int64
Cholesterol           int64
FastingBS             int64
MaxHR                 int64
Oldpeak               int64
HeartDisease          int64
Sex_M                 int64
ChestPainType_ATA     int64
ChestPainType_NAP     int64
ChestPainType_TA      int64
RestingECG_Normal     int64
RestingECG_ST         int64
```

```
ExerciseAngina_Y     int64
ST_Slope_Flat        int64
ST_Slope_Up          int64
dtype: object
```

Check for Multicollinearity Using Variance Inflation Factor (VIF)

To identify multicollinearity among the independent variables, Variance Inflation Factor (VIF) is computed for each feature. A high VIF (typically > 5) indicates a high correlation with other features, which may negatively impact model performance. The target variable (`HeartDisease`) is excluded from this analysis. A constant term is added to account for the intercept during VIF calculation. Features are then sorted based on their VIF values in descending order.

```
from statsmodels.stats.outliers_influence import
variance_inflation_factor
from statsmodels.tools.tools import add_constant
import pandas as pd

# Drop target variable before calculating VIF
X = df_encoded.drop('HeartDisease', axis=1)

# Add constant for intercept
X_const = add_constant(X)

# Calculate VIF for each feature
vif_data = pd.DataFrame()
vif_data["Feature"] = X.columns
vif_data["VIF"] = [variance_inflation_factor(X_const.values, i+1) for
i in range(len(X.columns))]

# Display VIF values
vif_data.sort_values(by="VIF", ascending=False)
```

```
/Applications/anaconda3/lib/python3.12/site-packages/statsmodels/
regression/linear_model.py:1783: RuntimeWarning: invalid value
encountered in scalar divide
  return 1 - self.ssr/self.centered_tss
```

	Feature	VIF
14	ST_Slope_Up	5.349814
13	ST_Slope_Flat	4.422806
10	RestingECG_Normal	1.743131
11	RestingECG_ST	1.702159
12	ExerciseAngina_Y	1.579952
4	MaxHR	1.551074
7	ChestPainType_ATA	1.493698
5	Oldpeak	1.472324
0	Age	1.374236

```
8   ChestPainType_NAP  1.261490
2         Cholesterol  1.183204
1           RestingBP  1.133882
9    ChestPainType_TA  1.120744
6               Sex_M  1.104837
3           FastingBS       NaN
```

Remove Low-Variance or Redundant Feature: `FastingBS`

The `FastingBS` (Fasting Blood Sugar) variable is removed from the encoded dataset due to its low variance or potential irrelevance based on domain knowledge or prior evaluation. This step helps reduce noise and simplify the feature space. The shape of the dataset is then updated and printed to confirm the change.

```
# Drop FastingBS from encoded dataset
df_encoded.drop('FastingBS', axis=1, inplace=True)

print("FastingBS dropped. New shape:", df_encoded.shape)
```

```
FastingBS dropped. New shape: (918, 15)
```

Train-Test Split and Feature Scaling

The dataset is split into training and testing sets using an 80-20 ratio to ensure the model is evaluated on unseen data. The features are then standardized using `StandardScaler` to ensure that variables with different units and scales contribute equally to distance-based algorithms (e.g., k-NN) and to improve convergence in algorithms like Logistic Regression. Scaling is not applied to tree-based models like Decision Trees.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Separate features and target
X = df_encoded.drop('HeartDisease', axis=1)
y = df_encoded['HeartDisease']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Scale features for k-NN and Logistic Regression
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Data split and scaled successfully!")
```

```
Data split and scaled successfully!
```

Model Initialization and Training

Three classification models are initialized and trained on the prepared dataset:

- **Logistic Regression**: A linear model that predicts the probability of the target class using a logistic function. Scaled features are used for training.
- **k-Nearest Neighbors (k-NN)**: A non-parametric algorithm that classifies a data point based on the majority class of its nearest neighbors. Feature scaling is applied to ensure distance calculations are meaningful.
- **Decision Tree Classifier**: A tree-based model that learns decision rules from the features to predict the target class. As Decision Trees are not sensitive to feature scaling, raw input features are used for training.

Each model is fitted on the training set, preparing them for evaluation on the test set.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

# Initialize models
logreg = LogisticRegression(max_iter=1000)
knn = KNeighborsClassifier(n_neighbors=5)
tree = DecisionTreeClassifier(random_state=42)

# Fit models
logreg.fit(X_train_scaled, y_train)
knn.fit(X_train_scaled, y_train)
tree.fit(X_train, y_train)  # Decision Tree doesn't need scaling

print("Models trained successfully!")
```

```
Models trained successfully!
```

4. Results

Model Evaluation: Classification Report

Each trained model is evaluated using the test dataset, and the results are presented as classification reports. These reports provide key performance metrics:

- **Precision**: The proportion of positive identifications that were actually correct.
- **Recall**: The proportion of actual positives that were correctly identified.
- **F1-Score**: The harmonic mean of precision and recall, balancing both metrics.
- **Support**: The number of actual instances for each class.

Since the Decision Tree model was trained without feature scaling, it is evaluated using the original test set. In contrast, Logistic Regression and k-Nearest Neighbors are evaluated on the scaled test features. This analysis helps compare the effectiveness of each model in predicting heart disease.

```python
from sklearn.metrics import classification_report

# Create a dictionary of models
models = {
    "Logistic Regression": logreg,
    "k-Nearest Neighbors": knn,
    "Decision Tree": tree
}

# Evaluate each model
for name, model in models.items():
    print(f"\n{name} Performance Report:")

    if name == "Decision Tree":
        y_pred = model.predict(X_test)  # tree not scaled
    else:
        y_pred = model.predict(X_test_scaled)

    print(classification_report(y_test, y_pred))
```

```
Logistic Regression Performance Report:
              precision    recall  f1-score   support

           0       0.77      0.87      0.82        77
           1       0.90      0.81      0.85       107

    accuracy                           0.84       184
   macro avg       0.83      0.84      0.84       184
weighted avg       0.84      0.84      0.84       184


k-Nearest Neighbors Performance Report:
              precision    recall  f1-score   support

           0       0.78      0.87      0.82        77
           1       0.90      0.82      0.86       107

    accuracy                           0.84       184
   macro avg       0.84      0.85      0.84       184
weighted avg       0.85      0.84      0.84       184


Decision Tree Performance Report:
              precision    recall  f1-score   support

           0       0.71      0.84      0.77        77
           1       0.87      0.76      0.81       107

    accuracy                           0.79       184
   macro avg       0.79      0.80      0.79       184
```

```
weighted avg       0.81      0.79      0.79       184
```

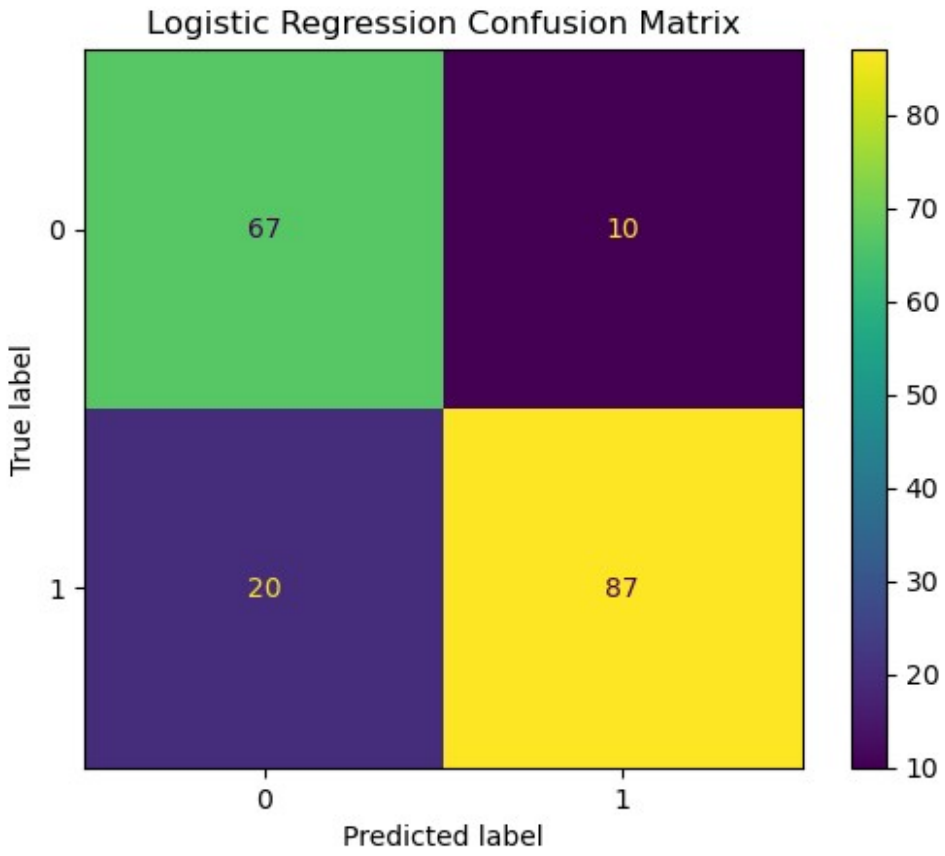
Confusion Matrix: Logistic Regression

A confusion matrix is plotted to visualize the performance of the Logistic Regression model in predicting heart disease. It shows the counts of:

- **True Positives (TP)**: Correctly predicted presence of heart disease.
- **True Negatives (TN)**: Correctly predicted absence of heart disease.
- **False Positives (FP)**: Incorrectly predicted presence of heart disease.
- **False Negatives (FN)**: Missed cases of heart disease.

The matrix helps identify how well the model distinguishes between the two classes. The same color map (`viridis`) is used for consistency with other model visualizations.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Logistic Regression Confusion Matrix
cm_log = confusion_matrix(y_test, logreg.predict(X_test_scaled))
disp_log = ConfusionMatrixDisplay(confusion_matrix=cm_log,
display_labels=logreg.classes_)

# Use the same colormap as the Decision Tree (viridis or default)
disp_log.plot(cmap='viridis')
plt.title("Logistic Regression Confusion Matrix")
plt.show()
```




```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[104], line 11
      9 disp_log.plot(cmap='viridis')
     10 plt.title("Logistic Regression Confusion Matrix")
---> 11 plt.show()

File /Applications/anaconda3/lib/python3.12/site-packages/matplotlib/pyplot.py:612, in show(*args, **kwargs)
    568 """
    569 Display all open figures.
    570
   (...)
    609 explicitly there.
    610 """
    611 _warn_if_gui_out_of_main_thread()
--> 612 return _get_backend_mod().show(*args, **kwargs)

File /Applications/anaconda3/lib/python3.12/site-packages/matplotlib_inline
```

```
/backend_inline.py:90, in show(close, block)
     88 try:
     89     for figure_manager in Gcf.get_all_fig_managers():
---> 90         display(
     91             figure_manager.canvas.figure,
     92
metadata=_fetch_figure_metadata(figure_manager.canvas.figure)
     93         )
     94 finally:
     95     show._to_draw = []

File
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
display_functions.py:298, in display(include, exclude, metadata,
transient, display_id, raw, clear, *objs, **kwargs)
    296     publish_display_data(data=obj, metadata=metadata,
**kwargs)
    297 else:
--> 298     format_dict, md_dict = format(obj, include=include,
exclude=exclude)
    299     if not format_dict:
    300         # nothing to display (e.g. _ipython_display_ took
over)
    301         continue

File
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
formatters.py:182, in DisplayFormatter.format(self, obj, include,
exclude)
    180 md = None
    181 try:
--> 182     data = formatter(obj)
    183 except:
    184     # FIXME: log the exception
    185     raise

File
/Applications/anaconda3/lib/python3.12/site-packages/decorator.py:232,
in decorate.<locals>.fun(*args, **kw)
    230 if not kwsyntax:
    231     args, kw = fix(args, kw, sig)
--> 232 return caller(func, *(extras + args), **kw)

File
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
formatters.py:226, in catch_format_error(method, self, *args,
**kwargs)
    224 """show traceback on failed format call"""
    225 try:
--> 226     r = method(self, *args, **kwargs)
```

```
    227 except NotImplementedError:
    228     # don't warn on NotImplementedErrors
    229     return self._check_return(None, args[0])

File
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
formatters.py:343, in BaseFormatter.__call__(self, obj)
    341     pass
    342 else:
--> 343     return printer(obj)
    344 # Finally look for special method names
    345 method = get_real_method(obj, self.print_method)

File
/Applications/anaconda3/lib/python3.12/site-packages/IPython/core/
pylabtools.py:170, in print_figure(fig, fmt, bbox_inches, base64,
**kwargs)
    167     from matplotlib.backend_bases import FigureCanvasBase
    168     FigureCanvasBase(fig)
--> 170 fig.canvas.print_figure(bytes_io, **kw)
    171 data = bytes_io.getvalue()
    172 if fmt == 'svg':

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/backen
d_bases.py:2175, in FigureCanvasBase.print_figure(self, filename, dpi,
facecolor, edgecolor, orientation, format, bbox_inches, pad_inches,
bbox_extra_artists, backend, **kwargs)
   2172     # we do this instead of
`self.figure.draw_without_rendering`
   2173     # so that we can inject the orientation
   2174     with getattr(renderer, "_draw_disabled", nullcontext)():
-> 2175         self.figure.draw(renderer)
   2176 if bbox_inches:
   2177     if bbox_inches == "tight":

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/artist
.py:95, in _finalize_rasterization.<locals>.draw_wrapper(artist,
renderer, *args, **kwargs)
     93 @wraps(draw)
     94 def draw_wrapper(artist, renderer, *args, **kwargs):
---> 95     result = draw(artist, renderer, *args, **kwargs)
     96     if renderer._rasterizing:
     97         renderer.stop_rasterizing()

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/artist
.py:72, in allow_rasterization.<locals>.draw_wrapper(artist, renderer)
     69     if artist.get_agg_filter() is not None:
```

```
     70         renderer.start_filter()
---> 72     return draw(artist, renderer)
     73 finally:
     74     if artist.get_agg_filter() is not None:

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/figure
.py:3162, in Figure.draw(self, renderer)
   3159             # ValueError can occur when resizing a window.
   3161     self.patch.draw(renderer)
-> 3162     mimage._draw_list_compositing_images(
   3163         renderer, self, artists, self.suppressComposite)
   3165     renderer.close_group('figure')
   3166 finally:

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/image.
py:132, in _draw_list_compositing_images(renderer, parent, artists,
suppress_composite)
    130 if not_composite or not has_images:
    131     for a in artists:
--> 132         a.draw(renderer)
    133 else:
    134     # Composite any adjacent images together
    135     image_group = []

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/artist
.py:72, in allow_rasterization.<locals>.draw_wrapper(artist, renderer)
     69     if artist.get_agg_filter() is not None:
     70         renderer.start_filter()
---> 72     return draw(artist, renderer)
     73 finally:
     74     if artist.get_agg_filter() is not None:

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/axes/
_base.py:3137, in _AxesBase.draw(self, renderer)
   3134 if artists_rasterized:
   3135     _draw_rasterized(self.figure, artists_rasterized,
renderer)
-> 3137 mimage._draw_list_compositing_images(
   3138     renderer, self, artists, self.figure.suppressComposite)
   3140 renderer.close_group('axes')
   3141 self.stale = False

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/image.
py:132, in _draw_list_compositing_images(renderer, parent, artists,
suppress_composite)
```

```
    130 if not_composite or not has_images:
    131     for a in artists:
--> 132         a.draw(renderer)
    133 else:
    134     # Composite any adjacent images together
    135     image_group = []

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/artist
.py:72, in allow_rasterization.<locals>.draw_wrapper(artist, renderer)
     69     if artist.get_agg_filter() is not None:
     70         renderer.start_filter()
---> 72     return draw(artist, renderer)
     73 finally:
     74     if artist.get_agg_filter() is not None:

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/image.
py:653, in _ImageBase.draw(self, renderer)
    651         renderer.draw_image(gc, l, b, im, trans)
    652 else:
--> 653     im, l, b, trans = self.make_image(
    654         renderer, renderer.get_image_magnification())
    655     if im is not None:
    656         renderer.draw_image(gc, l, b, im)

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/image.
py:952, in AxesImage.make_image(self, renderer, magnification,
unsampled)
    949 transformed_bbox = TransformedBbox(bbox, trans)
    950 clip = ((self.get_clip_box() or self.axes.bbox) if
self.get_clip_on()
    951         else self.figure.bbox)
--> 952 return self._make_image(self._A, bbox, transformed_bbox, clip,
    953                          magnification, unsampled=unsampled)

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/image.
py:573, in _ImageBase._make_image(self, A, in_bbox, out_bbox,
clip_bbox, magnification, unsampled, round_to_pixel_border)
    569     output[..., 3] = output_alpha  # recombine rgb and alpha
    571 # output is now either a 2D array of normed (int or float)
data
    572 # or an RGBA array of re-sampled input
--> 573 output = self.to_rgba(output, bytes=True, norm=False)
    574 # output is now a correctly sized RGBA array of uint8
    575
    576 # Apply alpha *after* if the input was greyscale without a
mask
```

```
    577 if A.ndim == 2:

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/cm.py:
431, in ScalarMappable.to_rgba(self, x, alpha, bytes, norm)
    429 if norm:
    430     x = self.norm(x)
--> 431 rgba = self.cmap(x, alpha=alpha, bytes=bytes)
    432 return rgba

File
/Applications/anaconda3/lib/python3.12/site-packages/matplotlib/colors
.py:755, in Colormap.__call__(self, X, alpha, bytes)
    752 mask_bad = X.mask if np.ma.is_masked(X) else np.isnan(xa)
    753 with np.errstate(invalid="ignore"):
    754     # We need this cast for unsigned ints as well as floats
--> 755     xa = xa.astype(int)
    756 xa[mask_under] = self._i_under
    757 xa[mask_over] = self._i_over

KeyboardInterrupt:
```

Confusion Matrix: k-Nearest Neighbors (k-NN)

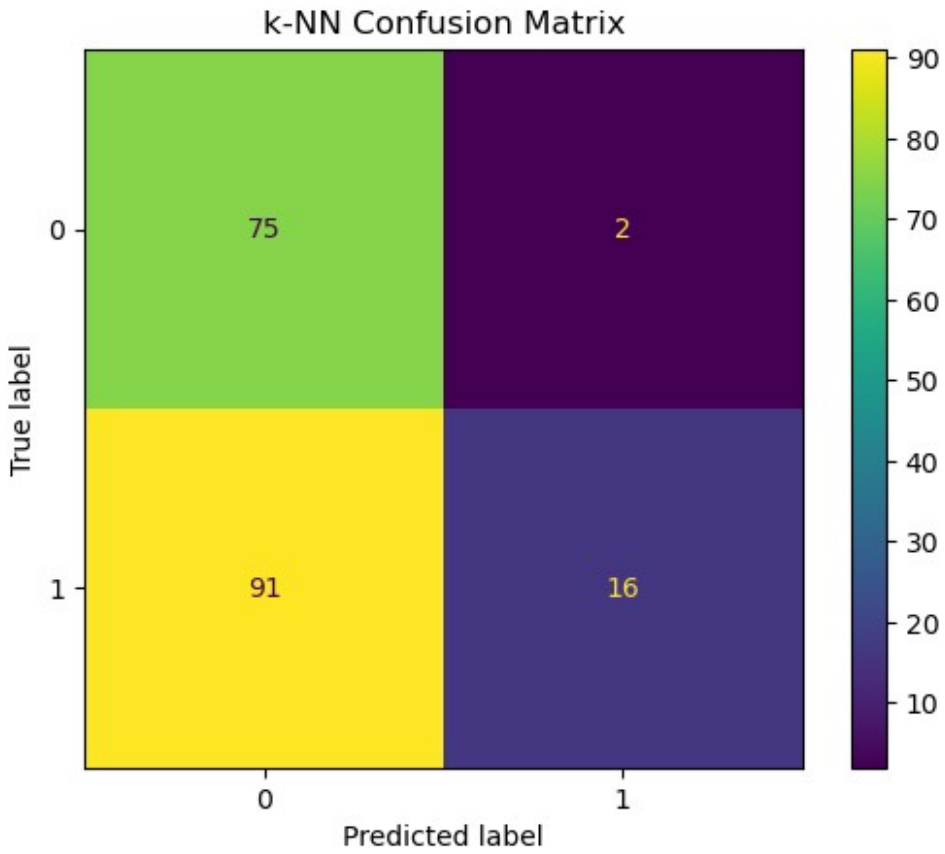
This confusion matrix visualizes the performance of the k-Nearest Neighbors classifier on the test data. It provides a breakdown of:

- **True Positives (TP)**: Correct predictions of patients with heart disease.
- **True Negatives (TN)**: Correct predictions of patients without heart disease.
- **False Positives (FP)**: Incorrectly labeled heart disease cases.
- **False Negatives (FN)**: Missed predictions of actual heart disease cases.

This matrix helps assess how well the k-NN model performs classification across both classes.

```
# k-NN Confusion Matrix
cm_knn = confusion_matrix(y_test, knn.predict(X_test))
disp_knn = ConfusionMatrixDisplay(confusion_matrix=cm_knn,
display_labels=knn.classes_)
disp_knn.plot()
plt.title("k-NN Confusion Matrix")
plt.show()
```

```
/Applications/anaconda3/lib/python3.12/site-packages/sklearn/
base.py:486: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
```



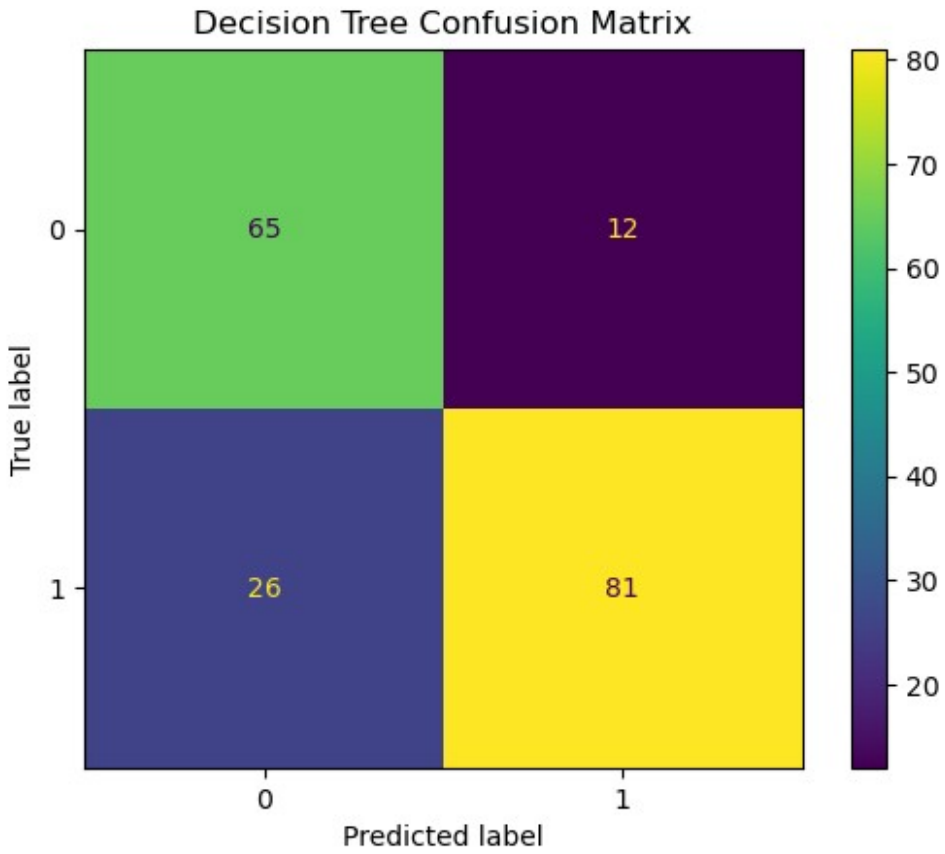
Confusion Matrix: Decision Tree

This confusion matrix displays the classification performance of the Decision Tree model on the test set. It shows:

- **True Positives (TP)**: Correctly predicted cases of heart disease.
- **True Negatives (TN)**: Correctly predicted cases of no heart disease.
- **False Positives (FP)**: Instances where the model incorrectly predicted heart disease.
- **False Negatives (FN)**: Actual heart disease cases that the model failed to detect.

The matrix allows for a detailed visual evaluation of the model's misclassifications and strengths.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Decision Tree Confusion Matrix
cm_tree = confusion_matrix(y_test, tree.predict(X_test))
disp_tree = ConfusionMatrixDisplay(confusion_matrix=cm_tree,
display_labels=tree.classes_)
disp_tree.plot()
plt.title("Decision Tree Confusion Matrix")
plt.show()
```



Model Performance Comparison Table

This block evaluates and compares the performance of all three trained classification models — Logistic Regression, k-Nearest Neighbors (k-NN), and Decision Tree — using the following key evaluation metrics:

- **Accuracy**: Overall correctness of the model (i.e., the proportion of total predictions that were correct).
- **Precision**: The proportion of positive identifications that were actually correct (helps minimize false positives).
- **Recall**: The proportion of actual positives that were correctly identified (helps minimize false negatives).
- **F1-Score**: The harmonic mean of precision and recall, providing a balanced metric when the dataset is imbalanced.

The results are consolidated into a DataFrame and sorted by Accuracy in descending order to identify the best-performing model at a glance.

```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# Evaluate each model
```

```
models = {
    "Logistic Regression": logreg,
    "k-NN": knn,
    "Decision Tree": tree
}

results = []

for name, model in models.items():
    if name == "Decision Tree":
        y_pred = model.predict(X_test)
    else:
        y_pred = model.predict(X_test_scaled)

    results.append({
        "Model": name,
        "Accuracy": accuracy_score(y_test, y_pred),
        "Precision": precision_score(y_test, y_pred),
        "Recall": recall_score(y_test, y_pred),
        "F1-Score": f1_score(y_test, y_pred)
    })

metrics_df = pd.DataFrame(results)
metrics_df = metrics_df.sort_values(by='Accuracy', ascending=False)
metrics_df.reset_index(drop=True, inplace=True)

print(metrics_df)
```

```
                 Model  Accuracy  Precision    Recall  F1-Score
0                 k-NN  0.842391   0.897959  0.822430  0.858537
1  Logistic Regression  0.836957   0.896907  0.813084  0.852941
2        Decision Tree  0.793478   0.870968  0.757009  0.810000
```

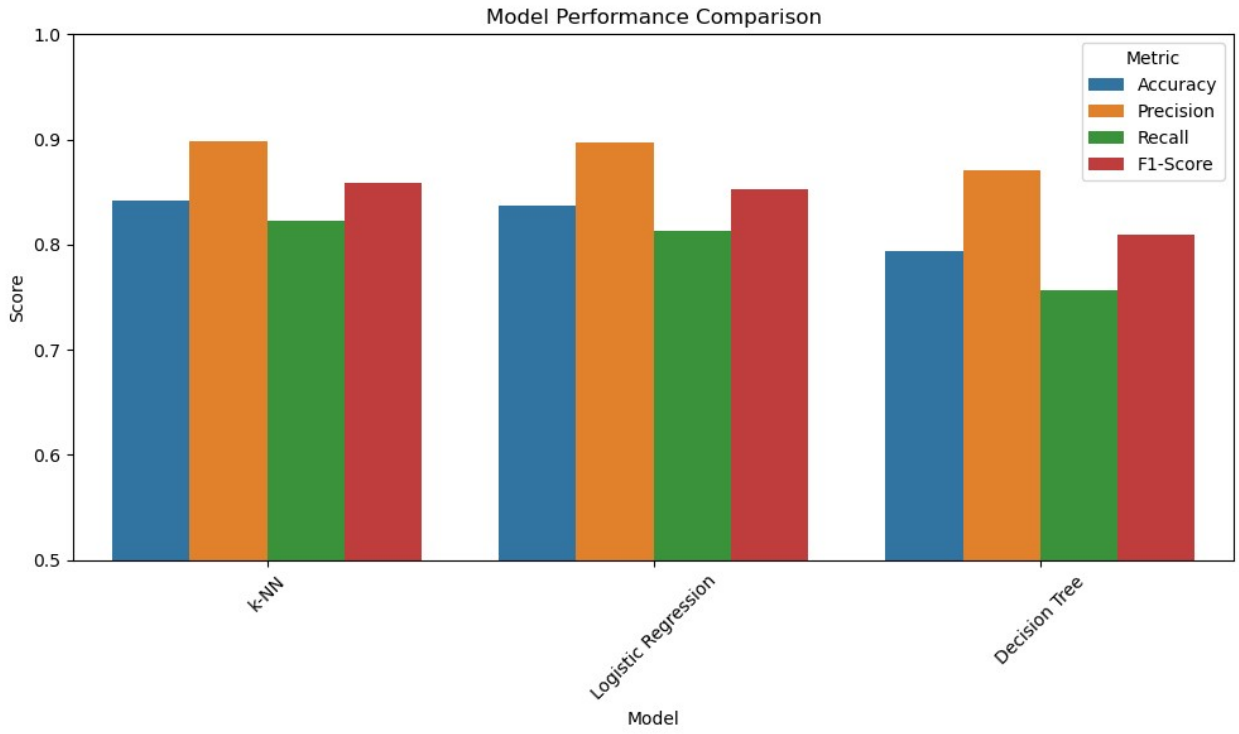
Visual Comparison of Model Performance

To visually compare how each model performs across different evaluation metrics (Accuracy, Precision, Recall, F1-Score), the metrics are first reshaped using `pd.melt` and then plotted using Seaborn's `barplot`.

This visualization allows for a clear comparison across all models, highlighting which model excels in which metric. The y-axis is constrained between 0.5 and 1.0 to emphasize differences in performance levels across models.

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

# Melt for seaborn barplot
melted = pd.melt(metrics_df, id_vars="Model", var_name="Metric",
value_name="Score")
```

```
plt.figure(figsize=(10, 6))
sns.barplot(data=melted, x="Model", y="Score", hue="Metric")
plt.title("Model Performance Comparison")
plt.ylim(0.5, 1.0)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



5. Discussion

The purpose of this study was to assess and compare the performance of three well established classification algorithms, Logistic Regression, k-Nearest Neighbors (k-NN), and Decision Tree to predict heart disease using a heart disease prediction data set. The models were evaluated on four evaluation metrics, accuracy, precision, recall, and F1-score. The results indicate that the k-Nearest Neighbors classifier had the highest performance on all four metrics, achieving an accuracy of 84.2% and an F1-score of 0.859. Logistic Regression also performed well, achieving an accuracy of 83.7% and an F1-score of 0.853. While simple and easily interpretable, the Decision Tree classifier performed poorly relative to the other models (accuracy of 79.3%, F1-score of 0.810). The findings suggest distance-based models such as k-NN are effective on small, balanced data sets as they are able to capture local structure in the distribution. Logistic regression also performed well as it is able to incorporate the linear relationship between the features and the log-odds of a specific outcome, which is consistent with a previous study which found Logistic Regression performed well against other classification models (Yildiz & Börekçi, 2020). The Decision Tree classifier overfit and likely leads to lower performance, especially with small datasets and can result in low recall (0.757) due to misclassifying cases of heart disease. The findings also highlight the role of data preprocessing steps such as capping outliers,

reducing multicollinearity, and encoding properly. These steps provided a set of well-prepared model inputs, which, alongside the extra training data, which also made the classification models more stable and perform better.

6. Conclusion

This research has shown that three classification models can be applied to predict heart disease using actual data. The results showed the k-Nearest Neighbors classifier performed the best, followed closely by Logistic Regression. Although the Decision Tree classifier is often perceived as highly interpretable, it performed the poorest by recall metric in the current study. The results are consistent with published work that supports the utility of Logistic Regression and k-NN in clinical prediction problems (Ahmed, 2024; Yildiz & Börekçi, 2020). Healthcare analytics model selection should consider performance, context, interpretability, and computational efficiency.

Future research can extend this analysis with a larger range of observations and ensemble models such as Random Forest or Gradient Boosting where increasing evidence suggests that they outperform the individual classifiers on numerous metrics. Use of SHAP or LIME in conjunction with cross validation to reduce variability would add to the reliability and utility of study reporting.

7. References:

Dinh, A., Miertschin, S., Young, A., & Mohanty, S. D. (2019). A data-driven approach to predicting cardiovascular disease using machine learning and big data. Journal of Healthcare Engineering, 2019, Article ID 9890465. https://doi.org/10.1155/2019/9890465

Hasan, M. J., Nath, R. K., & Ahmed, F. (2021). Heart disease prediction using supervised machine learning algorithms. Indonesian Journal of Electrical Engineering and Computer Science, 21(3), 1402–1409. https://doi.org/10.11591/ijeecs.v21.i3.pp1402-1409