DEBRE BRIHAN UNIVERSITY
COLLEGE OF COMPUTING
DEPARTMENT OF SOFTWARE ENGINEERING

Individual assignment

Course Title = Foundamenatls of Machine Learning
Corse code=SEng4091

Name	ID
Habtamu kebede	1401334

Summited to: Mr Derbew F

Due Date: 2/9/2025

Heart Disease Prediction – Project Documentation

Context

Heart disease is the **leading cause of death in the developed world**. With millions of lives affected each year, there is an urgent need for **early detection and prevention**. By identifying individuals at high risk, medical professionals can implement timely interventions to reduce the chances of **heart attacks, strokes, and other cardiovascular complications**.

Machine learning provides a powerful approach to analyzing **medical data** and uncovering patterns that may not be immediately visible to doctors. A predictive model can assist in **risk assessment**, helping healthcare providers make informed decisions to **prevent, diagnose, and treat heart disease more effectively**.

Problem Definition

Objective

The goal of this project is to develop a **machine learning model** that predicts whether a patient is at risk of developing heart disease. The model will analyze **various medical features** and classify patients into two categories:

- **0** – No heart disease (Low risk)
- **1** – Presence of heart disease (High risk)

By leveraging this predictive model, we aim to **support healthcare professionals** in identifying high-risk patients early, allowing for timely medical interventions and improved patient outcomes.

Content

This dataset contains **medical records of patients**, including important health indicators such as **age, sex, blood pressure, cholesterol levels, and heart rate**.

The goal is to use these features to predict which patients are **most likely to suffer from heart disease** in the near future. The dataset includes a **mix of numerical and categorical features** that influence cardiovascular health.

Data Source and Description

Where the Data Comes From

The dataset used in this project is obtained from two sources:

1. **Kaggle:** Heart Disease Prediction Dataset
    - Accessible via**: https://www.kaggle.com/datasets/rishidamarla/heart-disease-prediction**
2. **University of California Irvine's (UCI) Machine Learning Repository**
    - Original dataset link: UCI Machine Learning Repository
    - These datasets have been widely used in **medical research and machine learning projects** for heart disease risk assessment.

License and Usage

The dataset is **licensed under the Creative Commons Zero (CC0) 1.0 Universal License**, which means it is **publicly available** and can be used freely for **research, academic, and commercial applications** without restrictions.

Understanding the Dataset

How the Data is Organized

The dataset is structured in a **tabular format**, where each row represents a **single patient record**, and each column represents a **medical feature or test result**.

- **14 columns in total**
- Each row corresponds to a **patient's health data**
- The final column is the **target variable** indicating the presence or absence of heart disease

Feature Descriptions

Below is a breakdown of each feature and what it represents:

- **Age** – The age of the patient in years
- **Sex** – The gender of the patient (1 = male, 0 = female)
- **Chest Pain Type (cp)** – The type of chest pain experienced
    - 1 = Typical angina
    - 2 = Atypical angina
    - 3 = Non-anginal pain
    - 4 = Asymptomatic
- **Resting Blood Pressure (trestbps)** – Blood pressure in mm Hg when admitted to the hospital
- **Serum Cholesterol (chol)** – Cholesterol level in mg/dl
- **Fasting Blood Sugar (fbs)** – Blood sugar level after fasting
    - 1 = Greater than 120 mg/dl
    - 0 = Normal

- **Resting ECG (restecg)** – Electrocardiographic test results
  - 0 = Normal
  - 1 = ST-T wave abnormality
  - 2 = Left ventricular hypertrophy
- **Maximum Heart Rate (thalach)** – The highest heart rate achieved during physical exertion
- **Exercise-Induced Angina (exang)** – Chest pain triggered by exercise
  - 1 = Yes
  - 0 = No
- **Oldpeak** – ST depression induced by exercise relative to rest
- **Slope** – The slope of the peak exercise ST segment
  - 1 = Upsloping
  - 2 = Flat
  - 3 = Downsloping
- **Number of Major Vessels (ca)** – Number of major blood vessels (0–3) detected by fluoroscopy
- **Thalassemia (thal)** – A blood disorder
  - 3 = Normal
  - 6 = Fixed defect
  - 7 = Reversible defect
- **Heart Disease (target variable)** – Indicates whether a person has heart disease
  - 0 = No heart disease (Low risk)
  - 1 = Presence of heart disease (High risk)

Why Machine Learning is Suitable for This Problem

The **target variable** in this dataset is "Heart Disease," which has two possible values:

- **0** – No heart disease
- **1** – Presence of heart disease

Since the goal is to categorize individuals into one of these two classes, this is a **classification problem**.

Why This Approach Works

- The dataset consists of **structured numerical data**, making it ideal for machine learning models.
- Machine learning can uncover **complex patterns** in health data that may not be obvious through traditional analysis.
- A well-trained model can assist healthcare professionals by providing **an early warning system** for heart disease risk.

Model Justification

Since we are dealing with **binary classification**, models such as **Support Vector Machines (SVM), Logistic Regression, Decision Trees, and Neural Networks** are suitable for this task. The choice of model will depend on performance metrics like **accuracy, precision, recall, and F1-score**.

Acknowledgment

This dataset originates from the **University of California Irvine's Machine Learning Repository**, a well-established source for machine learning datasets.

- **Original dataset link**: **https://www.kaggle.com/datasets/rishidamarla/heart-disease-prediction**
- **Additional dataset version used**: Kaggle Dataset

This dataset has been widely used in **medical research** to improve heart disease prediction models and advance the field of **predictive healthcare**.

Data Exploration and Quality Assessment

1. Introduction

Heart disease is one of the leading causes of death in developed countries. Early detection of risk factors can help prevent heart attacks and strokes. This study uses **machine learning techniques** to analyze a dataset and predict the likelihood of heart disease in patients.

2. Data Loading and Exploration

Importing Required Libraries

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_csv('Heart_Disease_Prediction.csv')

# Display basic information about the dataset
print("=== Basic Information ===")
print(df.info())
print("\n")
```

```
# Display the first few rows of the dataset
print("=== First 5 Rows ===")
print(df.head())
print("\n")
```

Observations:

✅ No missing values
✅ All features have appropriate data types
✅ "Heart Disease" is a categorical variable (Presence/Absence)

3. Summary Statistics

```
# === Summarize Data Distributions for All Features ===
print("=== Summary Statistics for Numerical Features ===")
print(df.describe())
print("\n")
```

Observations:

✅ **Cholesterol and BP show extreme values**, suggesting potential **outliers**
✅ **Max HR decreases with age**, which is medically expected

4. Data Quality Check

```
# Check for missing values
print("=== Missing Values ===")
print(df.isnull().sum())
print("\n")

# Check for duplicates
print("=== Duplicate Rows ===")
print(df.duplicated().sum())
print("\n")
```

✅ **No missing values**
✅ **No duplicate records**

5. Visualizing Data Distributions

```
# === Visualize Distributions of Numerical Features ===
# Plot histograms for numerical features
print("=== Histograms of Numerical Features ===")
```

```
df.hist(figsize=(15, 10), bins=20)
plt.suptitle("Histograms of Numerical Features")
plt.tight_layout()
plt.show()
```

✅ **BP and Cholesterol show skewed distributions, suggesting potential outliers**

6. Outlier Detection

```
# === Identify Outliers ===
# Plot boxplots for numerical features to detect outliers
print("=== Boxplots for Numerical Features ===")
plt.figure(figsize=(15, 10))
for i, column in enumerate(df.select_dtypes(include=['int64',
'float64']).columns, 1):
    plt.subplot(4, 4, i)
    sns.boxplot(y=df[column])
    plt.title(column)
plt.tight_layout()
plt.show()
```

✅ **Cholesterol, BP, and ST depression have extreme values**

7. Feature Correlations

```
# Plot correlations between features and target variable
print("=== Correlation Heatmap ===")
plt.figure(figsize=(12, 8))
corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Heatmap")
plt.show()
```

✅ **Max HR and Age show a negative correlation**
✅ **Cholesterol has weak correlation with heart disease**

8. Analyzing Categorical Variables

```
# === Analyze Categorical Features ===
# Plot count plots for categorical features
print("=== Count Plots for Categorical Features ===")
categorical_features = ['Sex', 'Chest pain type', 'FBS over 120', 'EKG results',
'Exercise angina', 'Slope of ST', 'Thallium']
```

```
plt.figure(figsize=(15, 10))
for i, column in enumerate(categorical_features, 1):
    plt.subplot(3, 3, i)
    sns.countplot(x=column, hue='Heart Disease', data=df)
    plt.title(column)
plt.tight_layout()
plt.show()
```

✅ **Males have higher heart disease cases**
✅ **Chest pain type "4" (Asymptomatic) is strongly associated with heart disease**

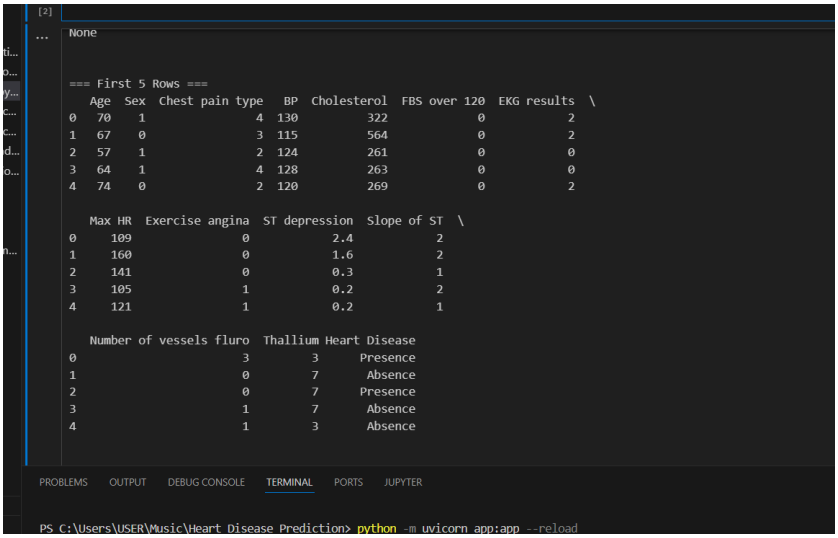
9. Target Variable Analysis

```
# === Analyze Target Variable ===
# Plot the distribution of the target variable
print("=== Distribution of Target Variable ===")
sns.countplot(x='Heart Disease', data=df)
plt.title("Distribution of Heart Disease")
plt.show()
```

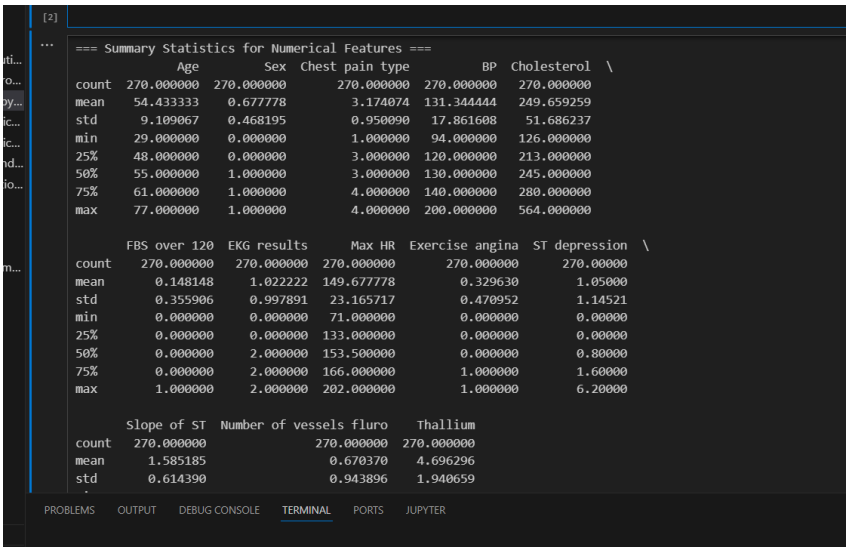
✅ **Balanced dataset with slightly more cases of heart disease**

ALL CELL OUTPUT THAT Perform exploratory data analysis (EDA) Above:

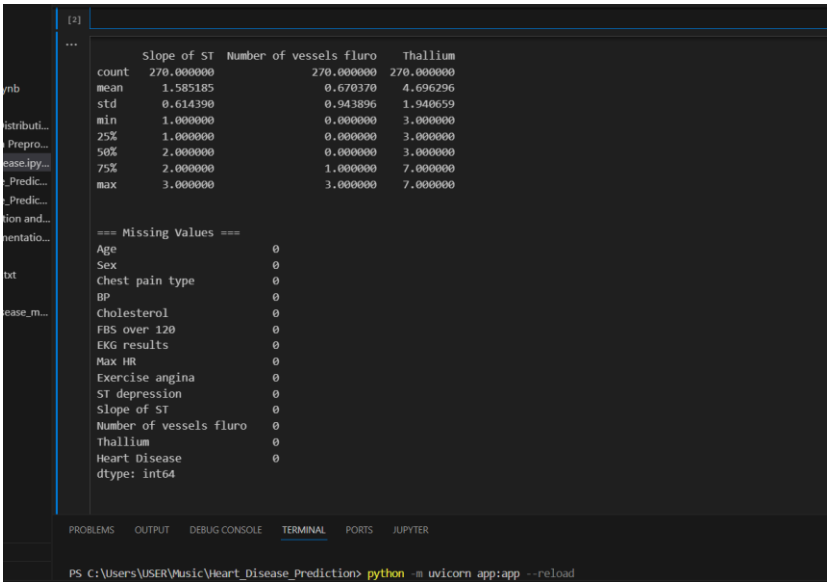
```
=== Basic Information ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Age                     270 non-null    int64
 1   Sex                     270 non-null    int64
 2   Chest pain type         270 non-null    int64
 3   BP                      270 non-null    int64
 4   Cholesterol             270 non-null    int64
 5   FBS over 120            270 non-null    int64
 6   EKG results             270 non-null    int64
 7   Max HR                  270 non-null    int64
 8   Exercise angina         270 non-null    int64
 9   ST depression           270 non-null    float64
 10  Slope of ST             270 non-null    int64
 11  Number of vessels fluro 270 non-null    int64
 12  Thallium                270 non-null    int64
 13  Heart Disease           270 non-null    object
dtypes: float64(1), int64(12), object(1)
memory usage: 29.7+ KB
None
```



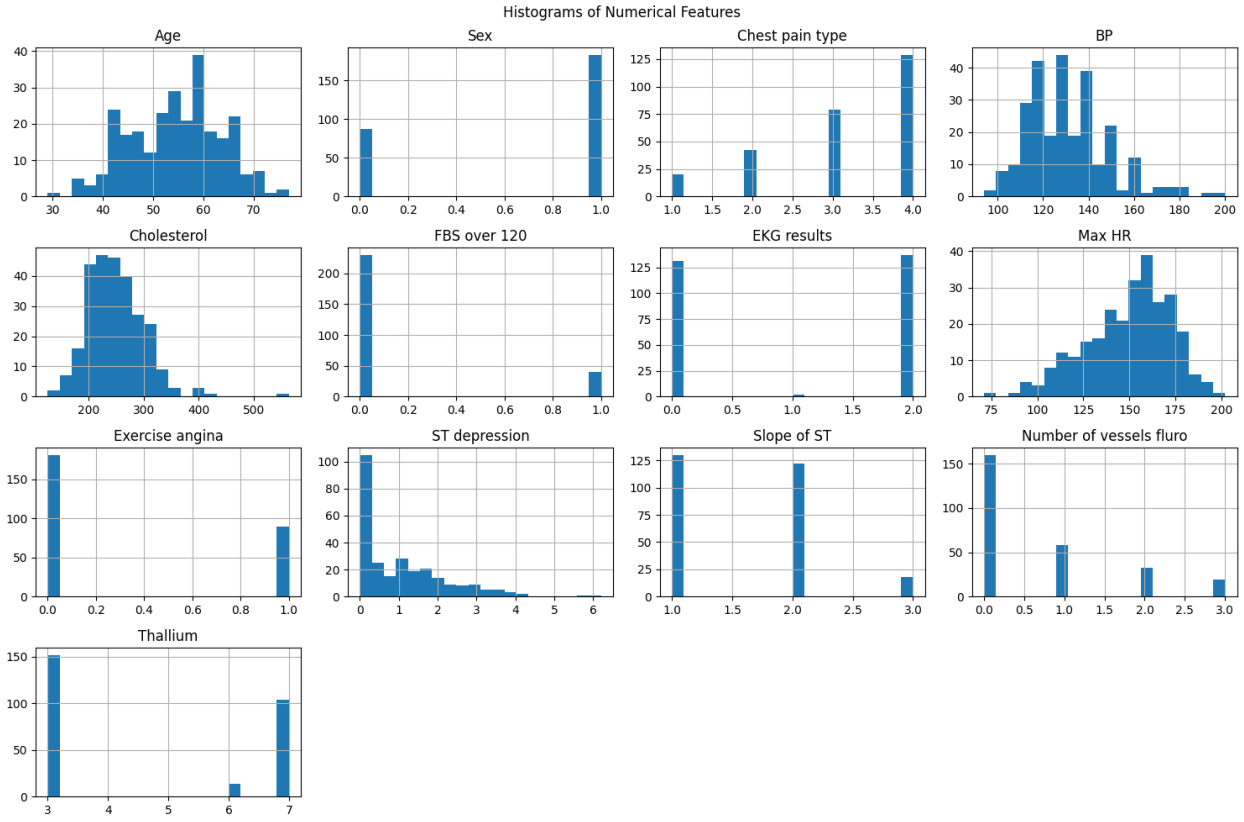
```
[2]
...
None


=== First 5 Rows ===
   Age  Sex  Chest pain type   BP  Cholesterol  FBS over 120  EKG results  \
0   70    1                4  130          322             0            2
1   67    0                3  115          564             0            2
2   57    1                2  124          261             0            0
3   64    1                4  128          263             0            0
4   74    0                2  120          269             0            2

   Max HR  Exercise angina  ST depression  Slope of ST  \
0     109                0            2.4            2
1     160                0            1.6            2
2     141                0            0.3            1
3     105                1            0.2            2
4     121                1            0.2            1

   Number of vessels fluro  Thallium Heart Disease
0                        3         3      Presence
1                        0         7       Absence
2                        0         7      Presence
3                        1         7       Absence
4                        1         3       Absence

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  JUPYTER

PS C:\Users\USER\Music\Heart Disease Prediction> python -m uvicorn app:app --reload
```



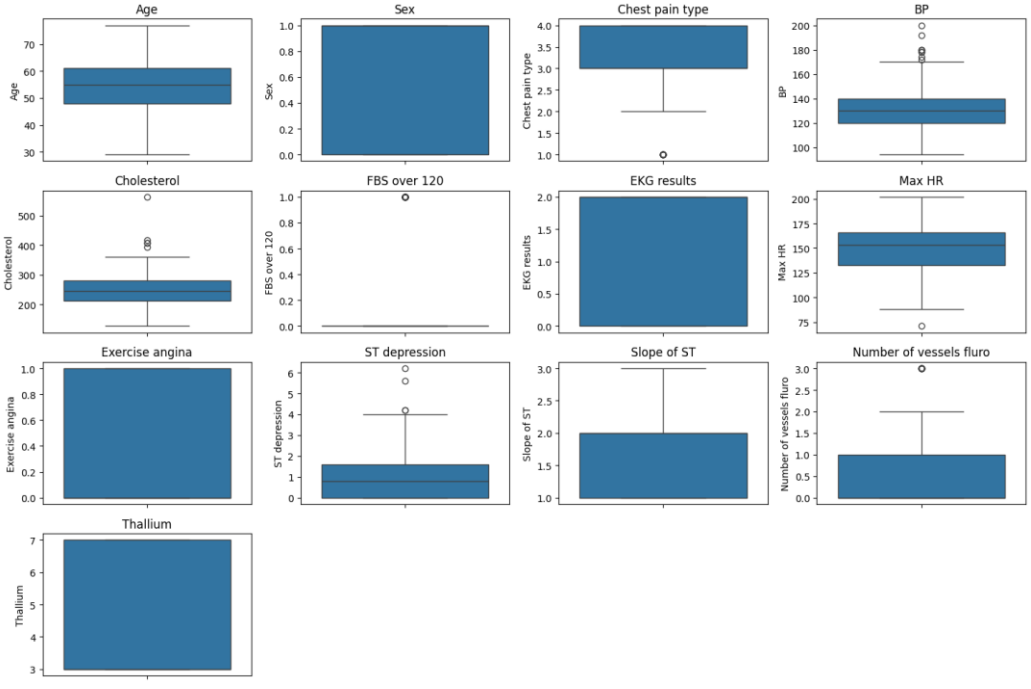
```
[2]
...
=== Summary Statistics for Numerical Features ===
              Age         Sex  Chest pain type          BP  Cholesterol  \
count  270.000000  270.000000       270.000000  270.000000   270.000000
mean    54.433333    0.677778         3.174074  131.344444   249.659259
std      9.109067    0.468195         0.950090   17.861608    51.686237
min     29.000000    0.000000         1.000000   94.000000   126.000000
25%     48.000000    0.000000         3.000000  120.000000   213.000000
50%     55.000000    1.000000         3.000000  130.000000   245.000000
75%     61.000000    1.000000         4.000000  140.000000   280.000000
max     77.000000    1.000000         4.000000  200.000000   564.000000

       FBS over 120  EKG results      Max HR  Exercise angina  ST depression  \
count    270.000000   270.000000  270.000000       270.000000      270.00000
mean       0.148148     1.022222  149.677778         0.329630        1.05000
std        0.355906     0.997891   23.165717         0.470952        1.14521
min        0.000000     0.000000   71.000000         0.000000        0.00000
25%        0.000000     0.000000  133.000000         0.000000        0.00000
50%        0.000000     2.000000  153.500000         0.000000        0.80000
75%        0.000000     2.000000  166.000000         1.000000        1.60000
max        1.000000     2.000000  202.000000         1.000000        6.20000

       Slope of ST  Number of vessels fluro    Thallium
count   270.000000               270.000000  270.000000
mean      1.585185                 0.670370    4.696296
std       0.614390                 0.943896    1.940659

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  JUPYTER
```



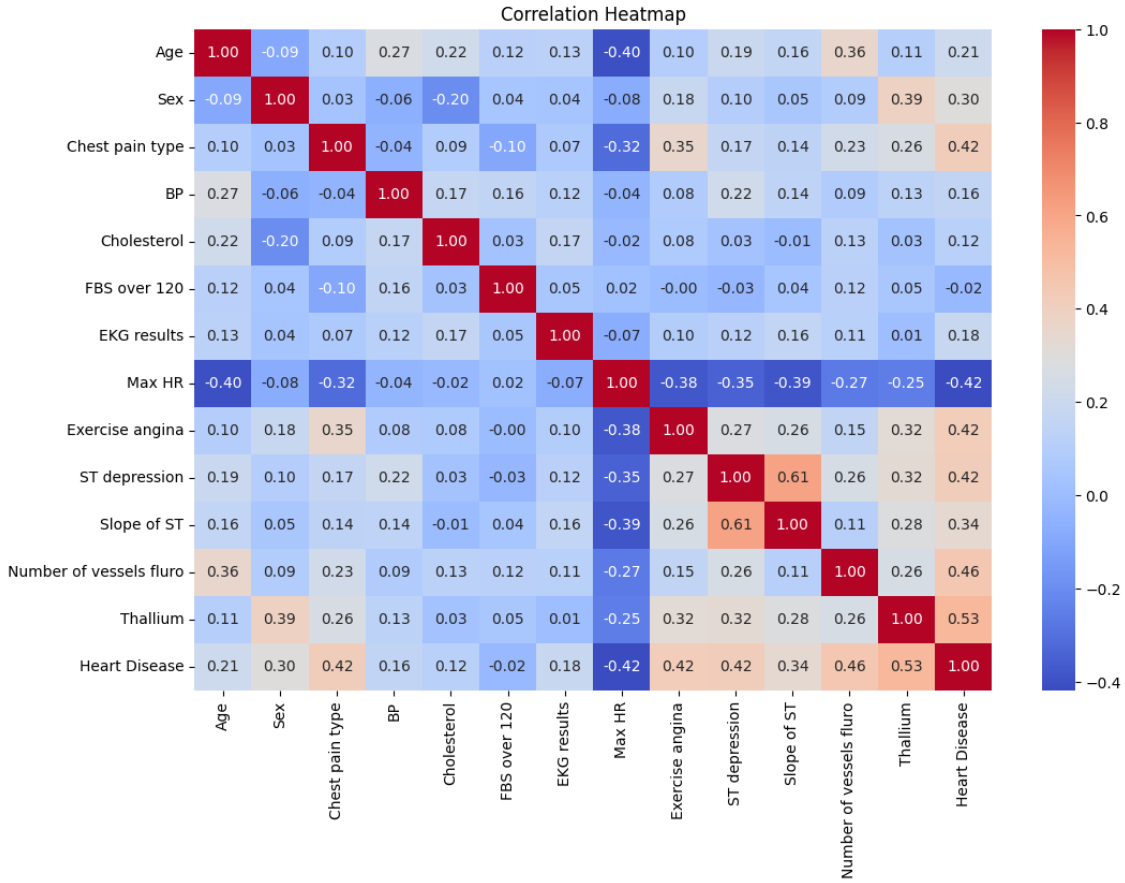
```
[2]
...
       Slope of ST  Number of vessels fluro    Thallium
count   270.000000               270.000000  270.000000
mean      1.585185                 0.670370    4.696296
std       0.614390                 0.943896    1.940659
min       1.000000                 0.000000    3.000000
25%       1.000000                 0.000000    3.000000
50%       2.000000                 0.000000    3.000000
75%       2.000000                 1.000000    7.000000
max       3.000000                 3.000000    7.000000


=== Missing Values ===
Age                       0
Sex                       0
Chest pain type           0
BP                        0
Cholesterol               0
FBS over 120              0
EKG results               0
Max HR                    0
Exercise angina           0
ST depression             0
Slope of ST               0
Number of vessels fluro   0
Thallium                  0
Heart Disease             0
dtype: int64

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  JUPYTER

PS C:\Users\USER\Music\Heart_Disease_Prediction> python -m uvicorn app:app --reload
```



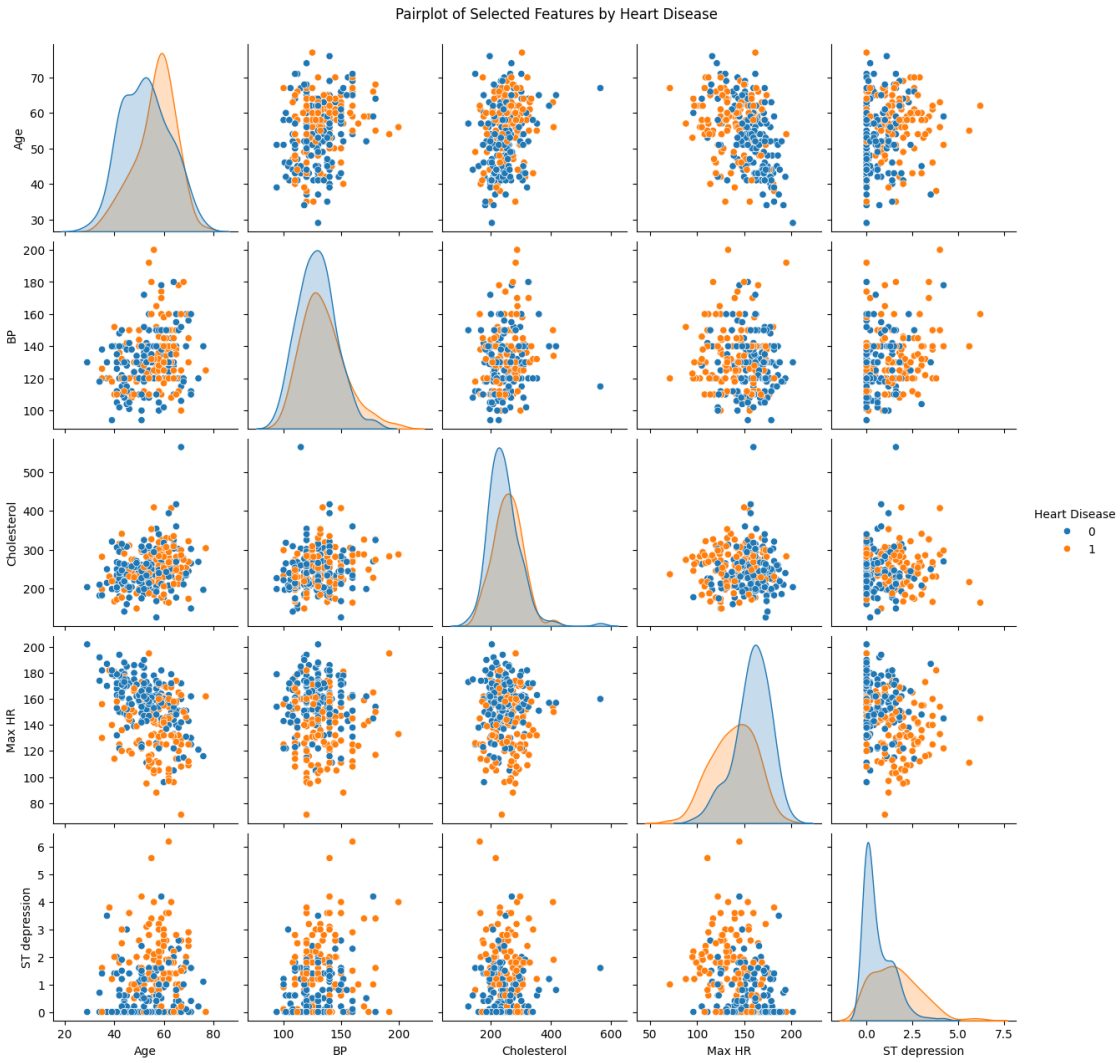
=== Boxplots for Numerical Features ===



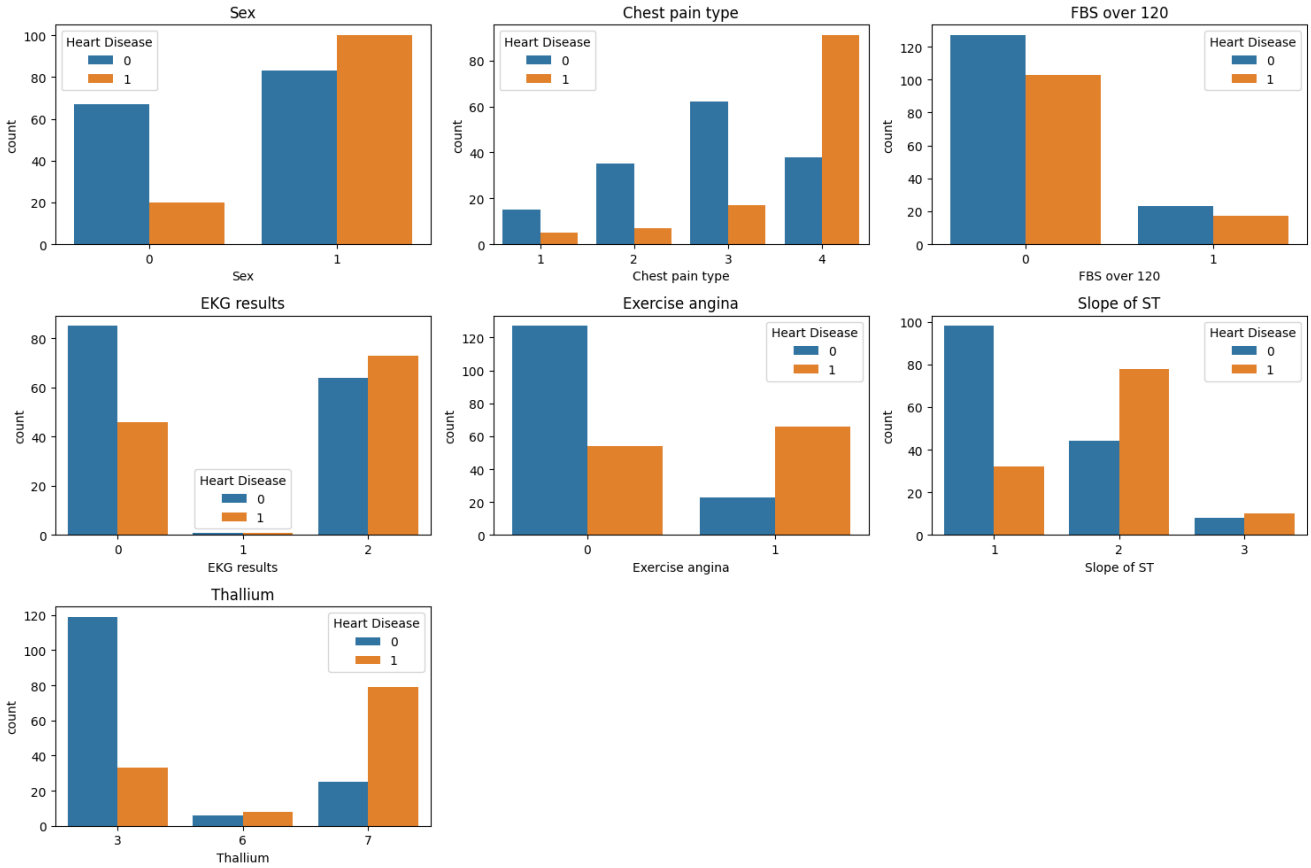
=== Correlation Heatmap ===



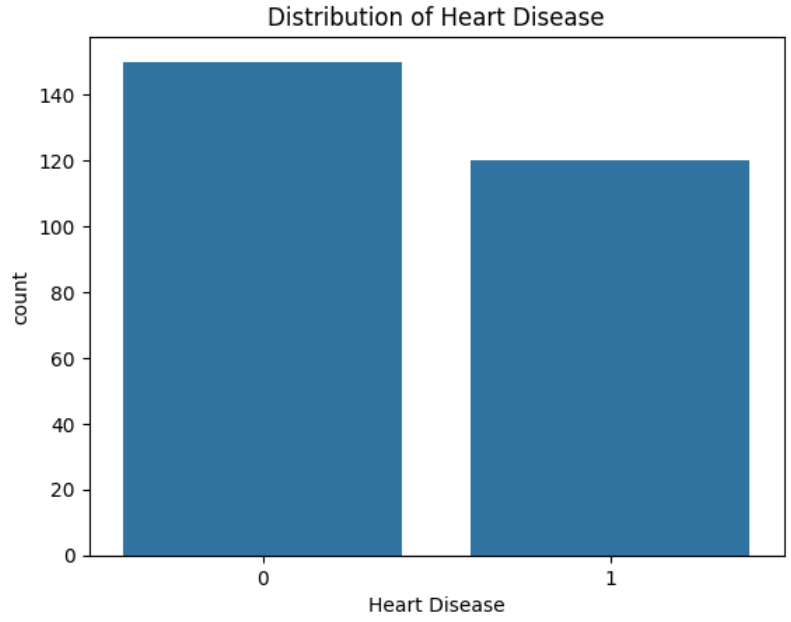
=== Pairplot for Selected Features ===



=== Count Plots for Categorical Features ===



=== Distribution of Target Variable ===



EDA completed and results saved.

Documentation for Preprocessing Steps:

1. Handling Missing Values

Why is it important?

- Missing data can lead to inaccurate or biased model predictions if not handled appropriately.
- It is essential to address missing values to ensure that the dataset is complete and ready for training.But I there is no missing value in my dataset.

2. Outlier Detection and Removal

Why is it important?

- Outliers can drastically affect model performance, particularly for algorithms like SVM, which are sensitive to the scale and distribution of data.
- Detecting and removing outliers helps ensure that the model performs robustly and doesn't make predictions based on extreme values.

Code Implementation: Outlier detection and removal can be done using the Interquartile Range (IQR) method:

```
# Outlier Detection and Removal (Example)
Q1 = df[['Cholesterol', 'Max HR', 'ST Depression']].quantile(0.25)
Q3 = df[['Cholesterol', 'Max HR', 'ST Depression']].quantile(0.75)
IQR = Q3 - Q1
df = df[~((df[['Cholesterol', 'Max HR', 'ST Depression']] < (Q1 - 1.5 * IQR)) |
(df[['Cholesterol', 'Max HR', 'ST Depression']] > (Q3 + 1.5 * IQR))).any(axis=1)]
```

Justification:

- **IQR Method**: The IQR method is an effective approach for outlier detection because it works well with skewed distributions and is less influenced by extreme values compared to other methods like z-scores.

3. Feature Engineering and Transformation

Why is it important?

- Proper feature engineering helps in transforming raw data into a format suitable for modeling.

- Feature scaling and encoding are necessary for algorithms like SVM, which perform poorly if the data is not properly preprocessed.

Steps Implemented:

1. **Target Variable Encoding:**
    - We map the categorical target variable `Heart Disease` ('Presence' and 'Absence') to binary values (`1` and `0` respectively). This transforms it into a binary classification problem, which is ideal for the SVM model.

```
# --- Verify target column before encoding ---
print("Target column before encoding:")
print(df['Heart Disease'].value_counts())

# Correctly map target variable 'Presence' -> 1 and 'Absence' -> 0
df['Heart Disease'] = df['Heart Disease'].map({'Presence': 1, 'Absence': 0})

# Verify target column after encoding
print("Target column after encoding:")
print(df['Heart Disease'].value_counts())
```

2. **Feature Scaling:**
    - The **StandardScaler** is used to standardize features, which is essential for SVM algorithms. This ensures that all features are on the same scale, making the model's performance more reliable.

```
# Standardize features (important for SVM)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Justification:

- **Target Encoding**: Mapping 'Presence' to 1 and 'Absence' to 0 makes it easier for the model to process and predict, as it now has a numeric representation.
- **Feature Scaling**: Standard scaling is necessary for SVM models because they are sensitive to the magnitude of features. Without this scaling, the SVM model could give undue weight to larger values, potentially leading to suboptimal performance.

4. Train/Test Split

Why is it important?

- Splitting the data into training and testing sets allows us to evaluate the model on unseen data, ensuring that it generalizes well to new data.

```
# Split into features (X) and target (y)
```

```
X = df.drop(columns=['Heart Disease'])
y = df['Heart Disease']

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Justification:

- **80/20 Split**: We use 80% of the data for training and 20% for testing. This is a common split ratio that ensures the model has enough data to learn but also provides a fair evaluation on the test set.

5. Model Training and Hyperparameter Tuning

Why is it important?

- Hyperparameter tuning allows us to find the best configuration for the SVM model, ensuring optimal performance.
- **GridSearchCV** is used to perform exhaustive search over specified parameter values.

```
# Model Training (using SVM and hyperparameter tuning)
svm = SVC(probability=True, class_weight='balanced')
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'gamma': ['scale', 'auto'],
    'degree': [3, 4, 5]
}

grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_scaled, y_train)

# Best model from GridSearchCV
best_model = grid_search.best_estimator_
```

Justification:

- **GridSearchCV**: This method performs cross-validation across a specified set of hyperparameters. By tuning parameters like `C`, `kernel`, and `gamma`, we ensure that the SVM model achieves optimal performance.

6. Saving Preprocessed Data

Why is it important?

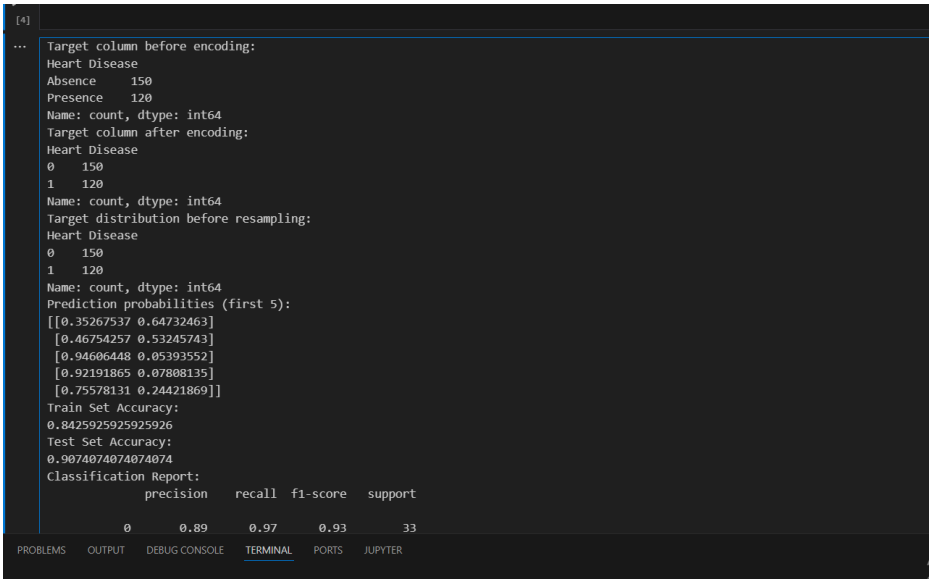
- Saving the preprocessed data helps in replicating the process without having to redo the cleaning steps in future tasks.

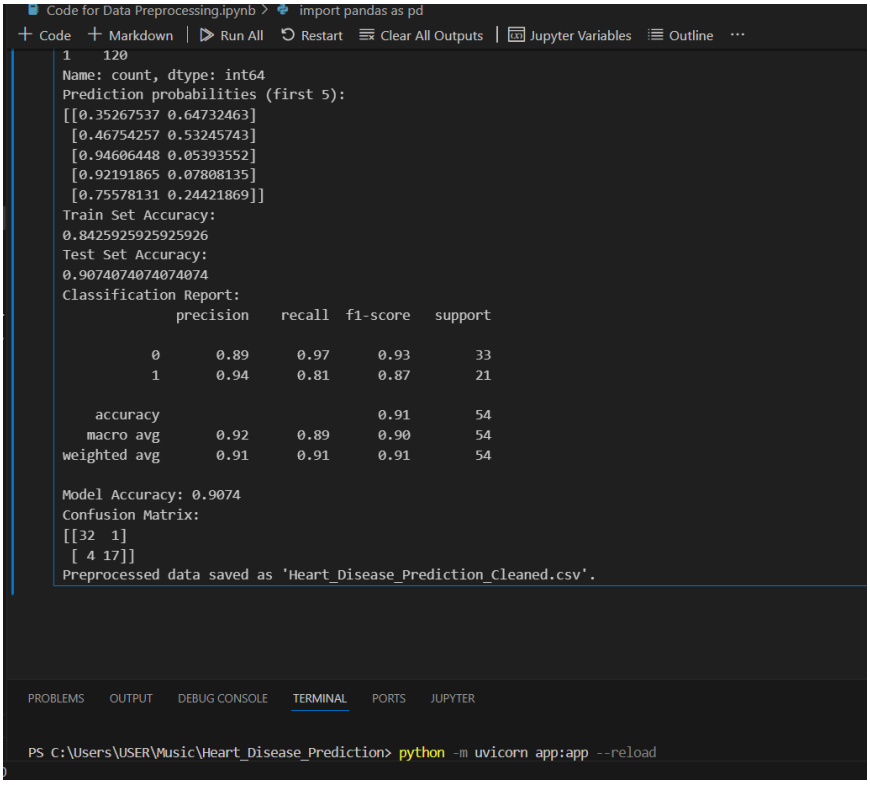
```
# --- Save Preprocessed Data ---
# Save the cleaned dataset (if needed for future use)
df.to_csv('Heart_Disease_Prediction_Cleaned.csv', index=False)
print("Preprocessed data saved as
'Heart_Disease_Prediction_Cleaned.csv'.")

```

Justification:

- Saving the preprocessed data allows for easier future use of the dataset, especially when working with other machine learning models or tasks that require the same dataset.

ALL CELL OUTPUT of above





Model Selection and Training Documentation

1. Model Selection: Support Vector Machine (SVM)

Why SVM?

- The Support Vector Machine (SVM) is chosen for this classification task because it works well in high-dimensional spaces, which is common in datasets with a large number of features.
- SVM is particularly effective in cases where there is a clear margin of separation between classes, and it is less prone to overfitting, especially in higher-dimensional spaces.

```
# --- Model Implementation and Hyperparameter Tuning ---
# Create a Support Vector Machine classifier
svm = SVC(probability=True, class_weight='balanced')

```

- **SVC (Support Vector Classifier)**: This implementation of the SVM is used with the `probability=True` argument to enable probability estimates for predictions (which can be useful for decision thresholds and evaluation).

- **Class Weight**: The `class_weight='balanced'` argument helps handle class imbalance in the dataset, giving more importance to the minority class. This ensures better handling of imbalanced target classes like 'Heart Disease' (presence/absence), where one class may have fewer instances than the other.

2. Hyperparameter Tuning: GridSearchCV

Why Hyperparameter Tuning?

- Hyperparameter tuning is a critical step to optimize the performance of the model. In SVM, there are key hyperparameters such as `C`, `kernel`, `gamma`, and `degree` that can significantly influence the model's performance.
- **GridSearchCV** is used to search for the best combination of hyperparameters by exhaustively testing all specified parameter combinations through cross-validation.

```
# Define the hyperparameter grid for GridSearchCV
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],  # Types of kernels
    'gamma': ['scale', 'auto'],  # Kernel coefficient
    'degree': [3, 4, 5]  # Degree of polynomial kernel (if applicable)
}

# Perform GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy')  # 5-fold cross-validation
grid_search.fit(X_train_scaled, y_train)

# Best model after hyperparameter tuning
best_model = grid_search.best_estimator_
```

Justification:

- **GridSearchCV** performs a search over the hyperparameter space and selects the best combination based on cross-validation accuracy (`scoring='accuracy'`). The `cv=5` indicates 5-fold cross-validation to prevent overfitting.
- **Hyperparameters**:
  - `C`: Controls the regularization. A smaller value allows more misclassifications (higher bias), and a larger value aims to minimize misclassification (higher variance).
  - `kernel`: Specifies the kernel type. Different kernels (linear, RBF, polynomial, and sigmoid) define how the data points are transformed.
  - `gamma`: Defines the kernel coefficient. It affects the flexibility of the decision boundary.

- `degree`: Only used when `kernel='poly'` to define the degree of the polynomial.

3. Model Evaluation

Why Evaluate?

- Evaluating the model on both the training and test datasets provides insights into its performance and helps in identifying if the model is overfitting or underfitting.
- The test set accuracy is particularly important because it reflects how well the model generalizes to unseen data.

```
# Evaluate on test set
y_test_pred = best_model.predict(X_test_scaled)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Set Accuracy: {test_accuracy:.4f}")

# Print classification report for the test set
print("Classification Report (Test Set):")
print(classification_report(y_test, y_test_pred))

# Confusion Matrix
print("Confusion Matrix (Test Set):")
print(confusion_matrix(y_test, y_test_pred))
```

Justification:

- **Training Accuracy** gives insight into how well the model performs on data it has already seen.
- **Test Accuracy** is a critical metric to assess the model's generalization ability. A high test accuracy indicates good performance on unseen data.
- **Classification Report** provides additional metrics such as precision, recall, F1-score, which give a more detailed evaluation, especially in imbalanced datasets.
- **Confusion Matrix** provides a visual representation of the number of true positives, true negatives, false positives, and false negatives, giving deeper insights into the model's performance.

4. Model Parameters

Why is it Important?

- After the hyperparameter tuning process, it's important to understand which hyperparameters were selected and their values. This can help in understanding the

model's behavior and can be useful for replicating or further fine-tuning the model in the future.

```
# --- Final Model Performance ---
print(f"Best Model Parameters from GridSearchCV: {grid_search.best_params_}")
print(f"Best Model Accuracy: {grid_search.best_score_:.4f}")

```

Justification:

- **Best Model Parameters**: The output of the `grid_search.best_params_` provides the optimal configuration of hyperparameters that gave the best performance during cross-validation.
- **Best Model Accuracy**: This reflects the highest accuracy achieved during the grid search, which gives an indication of the model's best potential performance.

5. Saving the Model

Why Save the Model?

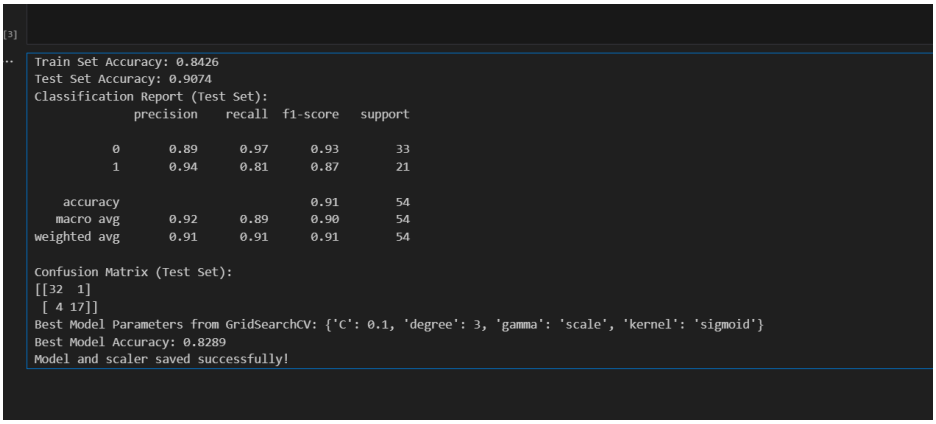
- Saving the trained model ensures that it can be reused for future predictions without retraining.
- **joblib** is used to save the model and scaler, making it easier to load and apply the trained model on new data later.

```
# --- Save the model using joblib ---
# Save the best trained model and scaler
joblib.dump(best_model, 'svm_heart_disease_model.joblib')  # Save model
joblib.dump(scaler, 'scaler.joblib')  # Save scaler (for future predictions)

print("Model and scaler saved successfully!")

```

Justification:

- **joblib.dump** saves the model and scaler as `.joblib` files. The model file contains the trained SVM model, while the scaler file contains the preprocessing transformation (scaling) that was applied to the features.
- This allows you to load these files in future applications without needing to retrain the model.

CELL OUTPUT



Model Evaluation Metrics and Discussion

1. Key Evaluation Metrics

The following metrics were used to evaluate the model:

Accuracy:*Accuracy measures the proportion of correct predictions (both true positives and true negatives) out of all predictions.*

- `accuracy = accuracy_score(y_test, y_test_pred)`
- **Interpretation**: The model correctly predicted 90.74% of the test set outcomes.
- **Result**: `Accuracy = 0.9074` (90.74%)

Precision:*Precision is the proportion of true positive predictions out of all positive predictions (i.e., how many of the predicted positive cases are actually positive).*

- `precision = precision_score(y_test, y_test_pred)`
- **Interpretation**: The model correctly identified 94.44% of the positive predictions (heart disease cases).
- **Result**: `Precision = 0.9444` (94.44%)

Recall:*Recall measures the proportion of true positive predictions out of all actual positive cases (i.e., how many of the actual positive cases were correctly identified).*

- `recall = recall_score(y_test, y_test_pred)`
- **Interpretation**: The model was able to identify 81.0% of the actual heart disease cases.
- **Result**: `Recall = 0.8095` (81.0%)

F1-Score:*The F1-score is the harmonic mean of precision and recall, providing a balance between the two.*

- `f1 = f1_score(y_test, y_test_pred)`
- **Interpretation**: The F1-score of 0.8718 indicates a strong balance between precision and recall, making the model reliable.

- **Result**: `F1 Score = 0.8718` (87.18%)

ROC AUC (Receiver Operating Characteristic - Area Under Curve):*ROC AUC is a measure of the model's ability to distinguish between classes. The closer the AUC value is to 1, the better the model's ability to separate positive and negative cases.*

```
roc_auc = roc_auc_score(y_test, y_test_prob)
```

- **Interpretation**: An AUC value of 0.9221 suggests excellent model performance, as it indicates the model can distinguish between heart disease and no heart disease with high accuracy.
- **Result**: `ROC AUC = 0.9221` (92.21%)

2. Confusion Matrix:

The confusion matrix displays the true positive, false positive, false negative, and true negative values. These values help us understand the types of errors made by the model.

```
cm = confusion_matrix(y_test, y_test_pred)
```

- **Confusion Matrix**:
  - **True Negatives (TN)**: 32 (No Disease predicted correctly)
  - **False Positives (FP)**: 1 (False alarm: No disease predicted as disease)
  - **False Negatives (FN)**: 4 (Disease predicted as no disease)
  - **True Positives (TP)**: 17 (Disease predicted correctly)
- **Interpretation**: The model shows a low number of false positives (1) and false negatives (4), which indicates a good performance in distinguishing between the two classes.

3. Classification Report:

The classification report provides detailed precision, recall, and F1 scores for each class (heart disease and no heart disease).

```
print("Classification Report:")
print(classification_report(y_test, y_test_pred))
```

- **Interpretation**:
  - For **Class 0 (No Disease)**: Precision = 0.89, Recall = 0.97, F1-score = 0.93
  - For **Class 1 (Disease)**: Precision = 0.94, Recall = 0.81, F1-score = 0.87
  - The overall weighted average for precision, recall, and F1-score is approximately 0.91, indicating strong performance.
  - **4. ROC Curve**:

The ROC curve plots the True Positive Rate (recall) against the False Positive Rate. The AUC value is computed to assess the overall performance.

```
fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
roc_auc = auc(fpr, tpr)
```

- **Interpretation**: The ROC curve demonstrates that the model has a high true positive rate and low false positive rate, confirming its strong ability to differentiate between the two classes. The curve's AUC value of 0.9221 is well above 0.5, indicating an excellent model.
- **5. Comparison Against Baseline (Dummy Classifier)**:

To assess the model's performance, we compared it to a **Dummy Classifier**, which predicts the majority class (i.e., the most frequent class in the dataset).

```
dummy = DummyClassifier(strategy="most_frequent")  # Predicts the majority class
dummy.fit(X_train, y_train)
y_dummy_pred = dummy.predict(X_test)
```

- **Dummy Classifier Performance**:
  - Accuracy: `0.6111`
  - Precision: `0.0000` (since no positive predictions were made)
  - Recall: `0.0000`
  - F1 Score: `0.0000`
  - ROC AUC: `0.5000`
- **Interpretation**:
  - The Dummy Classifier performs poorly with an accuracy of just 61.11%, and the precision, recall, and F1 scores are all 0.0000 due to its failure to predict any positive cases.
  - **ROC AUC of 0.5** indicates that the Dummy Classifier is no better than random guessing.

6. Model Performance Analysis:

Comparing the SVM model against the Dummy Classifier reveals that the trained model performs much better across all key metrics, especially precision, recall, and F1 score.

- **Accuracy Comparison**:
  - **SVM Accuracy**: 90.74%
  - **Dummy Accuracy**: 61.11%
  - **Interpretation**: The model's accuracy is significantly higher than the baseline, confirming its superior performance.
- **Precision Comparison**:
  - **SVM Precision**: 94.44%
  - **Dummy Precision**: 0.00%
  - **Interpretation**: The trained model is much better at correctly identifying heart disease cases, with a higher precision than the dummy classifier.
- **Recall Comparison**:
  - **SVM Recall**: 81.0%
  - **Dummy Recall**: 0.0%

    - **Interpretation**: The model correctly identifies 81% of the positive cases, while the dummy classifier fails to identify any positive cases.
- **F1 Score Comparison**:
    - **SVM F1 Score**: 87.18%
    - **Dummy F1 Score**: 0.00%
    - **Interpretation**: The F1 score for the trained model indicates a good balance between precision and recall, while the Dummy Classifier's F1 score is nonexistent due to its inability to identify positive cases.
- **ROC AUC Comparison**:
    - **SVM ROC AUC**: 92.21%
    - **Dummy ROC AUC**: 50.00%
    - **Interpretation**: The ROC AUC of the trained model is much higher, showing that it has a strong ability to discriminate between the two classes, while the Dummy Classifier has no discriminative power (AUC = 0.5).

Model Deployment Documentation with FastAPI and Render

This documentation outlines the process of deploying the heart disease prediction model as a working API using FastAPI, along with the necessary steps for interacting with the model through the web interface.

1. Overview of Model Deployment

In this deployment, the heart disease prediction model, which was trained using Support Vector Machine (SVM), is exposed as an API. This allows users to input specific features related to a patient's health and receive a prediction indicating whether the patient has heart disease. The deployment is done through **FastAPI** and hosted on **Render**, providing an accessible, interactive web interface for users.

2. Requirements for Deployment

Before deploying the model, you need the following:

- **Python**: Version 3.8 or higher
- **FastAPI**: Web framework for building APIs.
- **Uvicorn**: ASGI server for running FastAPI applications.
- **Joblib**: To load and save the trained model.
- **Pydantic**: To validate and parse the input data.
- **Numpy**: For numerical operations.
- **Render**: A cloud platform to deploy the API.

Dependencies: *To install the required packages, run:*

```
pip install fastapi uvicorn joblib numpy pydantic
```

3. FastAPI Application Structure

The FastAPI application has the following main components:

1. **Model and Scaler Loading**: Loads the saved model and scaler using `joblib` for prediction.
2. **Prediction Endpoint**: Receives input data, processes it, and returns the prediction result.
3. **Web Interface**: Provides a simple form for users to input data and view results.
4. **Static Files**: Serves any static files (such as stylesheets or images) required for the user interface.

4. FastAPI Code Explanation

Here is a breakdown of the FastAPI code:

```
- from fastapi import FastAPI, HTTPException, Request, Form
- from fastapi.responses import HTMLResponse
- from fastapi.templating import Jinja2Templates
- from fastapi.staticfiles import StaticFiles  # Corrected import
- from pydantic import BaseModel
- import joblib
- import numpy as np
```

- **FastAPI**: The framework used to build the API.
- **Jinja2Templates**: Used to render HTML templates (used for the web form and result page).
- **StaticFiles**: To serve static files like CSS or JavaScript for the frontend.
- **joblib**: To load the trained model and scaler.
- **numpy**: For handling numerical operations on input data.

1. Initialize FastAPI App:

```
# Initialize FastAPI app
app = FastAPI()

# Serve static files from the "static" directory
app.mount("/static", StaticFiles(directory="static"), name="static")

# Load the saved model and scaler
model = joblib.load('svm_heart_disease_model.joblib')
scaler = joblib.load('scaler.joblib')

# Set up Jinja2 templates
templates = Jinja2Templates(directory="templates")
```

2. Load the Model and Scaler

3. Input Data Model (Pydantic)

```
# Define the input data model using Pydantic
class HeartDiseaseInput(BaseModel):
    age: float
    sex: int
    chest_pain_type: int
    resting_blood_pressure: float
    serum_cholestoral: float
    fasting_blood_sugar: int
    resting_ecg: int
    max_heart_rate: float
    exercise_induced_angina: int
    oldpeak: float
    slope: int
    num_major_vessels: int
    thal: int
```

- Defines the expected input fields using Pydantic. These fields match the features used to predict heart disease.

4. Prediction Endpoint

```
# Prediction endpoint
@app.post("/predict/")
async def predict(
    request: Request,
    age: float = Form(...),
    sex: int = Form(...),
    chest_pain_type: int = Form(...),
    resting_blood_pressure: float = Form(...),
    serum_cholestoral: float = Form(...),
    fasting_blood_sugar: int = Form(...),
    resting_ecg: int = Form(...),
    max_heart_rate: float = Form(...),
    exercise_induced_angina: int = Form(...),
    oldpeak: float = Form(...),
    slope: int = Form(...),
    num_major_vessels: int = Form(...),
    thal: int = Form(...),
):
    # Convert input data to a numpy array
    data = np.array([[
        age, sex, chest_pain_type, resting_blood_pressure,
        serum_cholestoral, fasting_blood_sugar, resting_ecg,
```

```
            max_heart_rate, exercise_induced_angina, oldpeak,
            slope, num_major_vessels, thal
        ]])

        # Scale the input data
        scaled_data = scaler.transform(data)

        # Make a prediction using the model
        prediction = model.predict(scaled_data)
        prediction_label = "Presence" if prediction[0] == 1 else "Absence"
```

- This is the main prediction endpoint.
- It accepts input data via the HTTP `POST` request using the `Form` method.
- The data is scaled using the previously loaded scaler, and the model makes the prediction.
- The result is rendered using Jinja2 templates in the `result.html` file.

5. Root Endpoint

```
# Root endpoint
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    return templates.TemplateResponse("form.html", {"request": request})
```

5. Deploying on Render

Deploying a FastAPI app on Render is simple and seamless. Here's how to do it:

1. **Create an Account on Render**
   Start by signing up on Render. The process is quick and easy!
2. **Create a New Web Service**
   After logging in, select the "New Web Service" option on the Render dashboard to create a new project.
3. **Connect GitHub**
   Link the GitHub repository containing your FastAPI project to Render. This allows Render to pull your code for deployment.
4. **Set Configuration**
   Choose Python as the runtime environment and add any required environment variables, such as API keys.
5. **Deploy the Service**
   Click "Deploy," and Render will handle the rest. It will automatically detect your FastAPI app, build it, and deploy it. Once deployed, Render will provide a live URL for your app.

6. Accessing the API

Once deployed, I can access the FastAPI application at the provided Render URL

- https://heart-disease-prediction-machine-learning.onrender.com/
- The root endpoint (`/`) will display a form where users can enter health data.
- The `/predict/` endpoint will process the input data and display the prediction result on a new page.

7. How to Use the Web Interface

1. **Visit the Web Page**: Open the root endpoint (`https://heart-disease-prediction-machine-learning.onrender.com/`) in your browser.
2. **Fill in the Form**: Input the required health information such as age, sex, chest pain type, etc.
3. **Submit the Form**: Once you submit the form, the system will process the input data and show whether the heart disease is present or absent.