Course id: ENG6600

Course Name: Advanced Machine Learning

Date: 10/04/2024

Name: Arpit Vaghela, 1262011

Final Project Report

Machine Learning for Predictive Analysis: A Comparative Study of Classification Algorithms on Heart Disease Dataset

Arpit Shaileshbhai Vaghela
School of Engineering
University of Guelph
Guelph, Canada
avaghela@uoguelph.ca

I. Abstract

The Medical Field has a gigantic amount of medical records. The biggest challenge in the medical industry in recent times is to analyze the massive amount of information about the patients. Cardiovascular disease makes our heart and blood vessels dysfunctional and eventually leads to death or permanent paralysis.[19] Heart disease is the Leading cause of death worldwide.[15] The correct prediction of heart disease can prevent severe life threats, and the wrong prediction can be fatal at the same time [15]. This paper uses different machine learning classifiers to train and analyze the results of the UCI machine learning Heart Disease Dataset. Furthermore, hyperparameter tuning and feature selection techniques are employed to enhance the performance of the models. Additionally, the stacking ensemble technique is applied to assess the performance of a heterogenous model ensemble.

II. Introduction

One of the biggest causes of death is the Heart Disease. A sedentary lifestyle, stress, obesity, unhealthy diet can be considered risk factors for heart disease.[24] Most heart diseases can be prevented by following a healthy lifestyle and a balanced diet.

But in recent times, a lot of research data and medical records are now publicly available for the general public. Machine Learning and Deep Learning are now more efficient and powerful than ever before. There's no denying that machine learning and artificial intelligence are playing a huge role in the medical industry. Machine Learning can be used efficiently to diagnose the early signs of diseases and classify or predict the results. These advancements offer excellent opportunities to analyze vast volumes of medical data and extract actionable insights for disease diagnosis and management. Machine learning techniques, in particular, have emerged as powerful tools for identifying patterns, predicting outcomes, and optimizing clinical decision-making processes.

However, before leveraging machine learning for healthcare applications, it is crucial to preprocess and analyze medical data effectively. Data preprocessing steps, such as cleaning, normalization, and feature engineering, play a pivotal role in ensuring data quality and relevance.[7]

Following data preprocessing, the next stage involves training machine learning models on the prepared dataset. The selection of appropriate algorithms and optimization of model parameters through techniques like hyperparameter tuning is essential for maximizing predictive performance and generalizability [19].

To optimize model performance, hyperparameter tuning is conducted to fine-tune model parameters. Additionally, feature selection techniques are employed to identify the most relevant features and assess each model's performance after feature extraction. the study employs the stacking ensemble method to combine the models and evaluate their collective performance. The stacking ensemble technique integrates the predictions of multiple base models to produce a final prediction, leveraging the strengths of each model. This approach allows for a comprehensive assessment of model performance and enhances predictive accuracy.

In this study, the paper focuses on the UCI Machine Learning Heart Disease Dataset, a widely used dataset for cardiovascular disease prediction. The dataset comprises various medical attributes, allowing for comprehensive analysis and experimentation with different machine-learning approaches. This paper explores the application of various machine learning classifiers on the UCI Machine Learning Heart Disease Dataset, incorporating hyperparameter tuning, feature selection, and ensemble technique to enhance predictive accuracy and robustness."

The objectives of this research encompass evaluating the performance of different machine learning algorithms on the UCI Heart Disease Dataset, optimizing model parameters to enhance predictive accuracy, and exploring ensemble techniques to further improve model robustness and reliability.

III. Related Work

Machine Learning has created widespread interest in various fields, including health and medicine.[17] Different researchers utilize different ways of involving various machine learning

approaches to solve health-related issues. They have used different approaches to classify and predict chronic diseases.

Some techniques were proven to efficiently predict chronic kidney disease. Weka [2] is a user interface that provides data processing cycles such as data preprocessing, classification, and other techniques to a user. It has a large collection of data on which various algorithms can be practiced on. This tool illustrates that Random Forest gives the best result among various algorithms such as Naive Bayes.

Heart Sounds can also be used for the detection of chronic heart failure. [13] The method involves filtering of audio signals, segmentation for feature extraction, and machine learning. The method uses the stacking process of ML algorithms and has three phases: segment-based Machine Learning phase, recording-based feature extraction phase, and recording Machine Learning phase.

Santhana Krishnan J. et al. [15] employed various machine learning algorithms to predict heart disease. Their findings demonstrated the effectiveness of machine learning in accurately predicting heart disease, highlighting the potential of these techniques in healthcare.

Divya Krishnani et al. [16] conducted a study focusing on the prediction of coronary heart disease using supervised machine learning algorithms. Their research showcased the utility of these algorithms in accurately identifying individuals at risk of coronary heart disease, thus facilitating early intervention and preventive measures.

Rohit Bharti et al. [4] proposed a model that leverages both machine learning and deep learning techniques for predicting heart disease. Their approach demonstrated superior performance compared to traditional methods, underscoring the benefits of incorporating deep learning into predictive modeling for cardiovascular diseases.

Dissanayake et al. [7] conducted a comparative study to evaluate the effectiveness of different feature selection techniques on heart disease prediction using classification algorithms. Their findings shed light on the impact of feature selection on model performance and provided insights into the most effective techniques for improving predictive accuracy.

Alqahtani et al. [1] explored the use of ensemble learning for detecting cardiovascular diseases. Their research highlighted the advantages of ensemble techniques in integrating diverse classifiers to enhance predictive accuracy and robustness in cardiovascular disease detection.

Madhumita Pal et al. [20] investigated risk prediction of cardiovascular disease using various machine learning classifiers. Their study emphasized the importance of machine learning algorithms in accurately assessing the risk of cardiovascular disease, thus facilitating personalized healthcare interventions and preventive strategies. Madhumita Pal et al. [20] investigated risk prediction of cardiovascular disease using various machine learning classifiers. Their study emphasized the importance of machine learning algorithms in accurately assessing the risk of cardiovascular disease, thus facilitating personalized healthcare interventions and preventive strategies.

IV. Methodology

A. Proposed Work

The paper outlines the step-by-step approach employed for the development of predictive models to classify heart disease risk. As shown in Fig.1, The process begins with a comprehensive exploration of the dataset through data visualization techniques, enabling insights into key relationships and distributions within the data. Subsequently, the dataset is partitioned into training, validation, and testing sets to facilitate model training and evaluation. Standardization using the StandardScaler is then applied to ensure uniformity across features. Following this, a series of baseline models are trained using a variety of classification algorithms, and their performance metrics are assessed. To enhance model performance, hyperparameter tuning, and feature selection techniques are employed. Finally, ensemble methods, specifically stacking, are utilized to combine the strengths of individual models, resulting in a robust predictive framework.

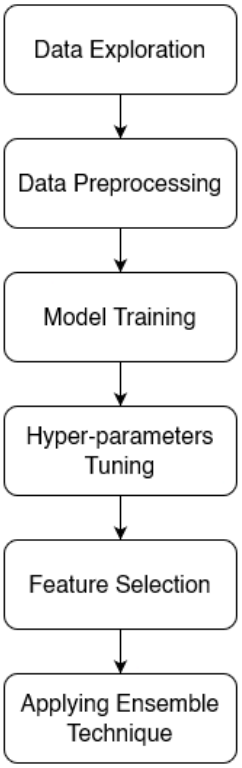


Fig. 1. Diagram of Proposed Work

B. Data Source

The dataset for predicting heart disease is taken from Kaggle, A data science competition platform and online community of data scientists and machine learning practitioners. The

dataset is taken from the UCI Machine Learning repository. The UCI is a collection of databases that are used to implement machine learning algorithms.

This data set is from 1988 and consists of four databases: Cleveland, Hungary, Switzerland, and Long Beach V. It contains 76 attributes, including the predicted attribute, all published experiments refer to using a subset of 14 of them. The "target" field refers to the presence of heart disease in the patient. It is an integer-valued 0 = no disease and 1 = disease.

Overall, The dataset comprises 1025 observations collected from the UCI Machine Learning repository, with each observation containing information about various attributes, including the presence or absence of heart disease.

C. Attribute Information

TABLE I
DESCRIPTION OF ATTRIBUTES

Attribute	Description
Age	Represents the age of the patient in years
Sex	Indicates the gender of the patient (0 = female, 1 = male)
Chest Pain Type	Describes the type of chest pain experienced by the patient (1 = typical angina, 2 = atypical angina, 3 = non-anginal pain, 4 = asymptomatic)
Resting Blood Pressure	Represents the resting blood pressure of the patient in mm Hg
Serum Cholesterol	Indicates the serum cholesterol level of the patient in mg/dl
Fasting Blood Sugar	Indicates whether the patient's fasting blood sugar is greater than 120 mg/dl (1 = true, 0 = false)
Resting Electrocardiographic Results	Describes the results of the resting electrocardiogram (ECG) (0 = normal, 1 = abnormality, 2 = probable or definite left ventricular hypertrophy)
Maximum Heart Rate Achieved	Represents the maximum heart rate achieved during exercise
Exercise Induced Angina	Indicates whether the patient experienced exercise-induced angina (1 = yes, 0 = no)
ST Depression (Oldpeak)	Represents the ST segment depression induced by exercise relative to rest
Slope of Peak Exercise ST Segment	Describes the slope of the peak exercise ST segment (1 = upsloping, 2 = flat, 3 = downsloping)
Number of Major Vessels	Indicates the number of major vessels (0-3) colored by fluoroscopy
Thalassemia	Describes the thalassemia type (0 = normal, 1 = fixed defect, 2 = reversible defect)

Table I describes the attributes present in the dataset.

D. Experimental Setup

1) Hardware: The experiments were conducted on a computer with the following specifications:

- Processor: Intel Core i5-7300HQ
- RAM: 8 GB DDR4

2) Software: The experiments were implemented using the following software libraries:

- matplotlib v3.6.2
- numPy v1.23.5
- pandas v.1.5.3
- pandas Profiling v3.6.3
- python v3.8
- scikit-learn v1.3.0
- seaborn v0.12.2

E. Classifiers Information

The following classifiers were used for prediction in this study:

1) **Logistic Regression**:
Logistic regression is a linear model used for binary classification. It models the probability that an instance belongs to a particular class using the logistic function (also known as the sigmoid function). The logistic function is defined as:

$$P(y = 1|\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$$

where $P(y = 1|\mathbf{x}; \mathbf{w})$ represents the probability that the target variable y equals 1 given input features $\mathbf{x}$ and model parameters $\mathbf{w}$.

2) **Naive Bayes**:
Naive Bayes is a probabilistic classifier based on Bayes' theorem with the "naive" assumption of independence between features. It calculates the posterior probability of a class given the features:

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})}$$

where $P(C_k|\mathbf{x})$ is the posterior probability of class C_k given features $\mathbf{x}$, $P(\mathbf{x}|C_k)$ is the likelihood of features given class C_k, $P(C_k)$ is the prior probability of class C_k, and $P(\mathbf{x})$ is the prior probability of features.

3) **K Nearest Neighbors (KNN)**:
K Nearest Neighbors (KNN) is a non-parametric classification algorithm that makes predictions based on the majority class of the k nearest data points in the feature space. It does not make any assumptions about the underlying data distribution. The key steps in KNN are:

- Calculate the distance between the query instance and all the training samples.
- Sort the distances and select the k nearest neighbors.
- Assign the class label to the query instance based on the majority class of the nearest neighbors.

The choice of k (the number of neighbors) is crucial and can significantly impact the performance of the algorithm.

4) **Random Forest**:
Random Forest is an ensemble learning method that constructs a multitude of decision trees during training and outputs the probability of the classes (classification) or the mean prediction (regression) of the individual trees. Each decision tree is trained on a random subset of the training data and a random subset of the features. The key steps in Random Forest are:

- Randomly select N samples from the training dataset with replacement (bootstrap samples).
- For each sample, randomly select a subset of features.
- Grow a decision tree using the selected samples and features.
- Repeat the process to create T decision trees.
- Aggregate the predictions of all trees to make the final prediction.

Random Forest reduces overfitting and improves generalization by combining the predictions of multiple weak learners (decision trees).

5) **Decision Trees**:
Decision Trees are a popular classification technique that partitions the feature space into regions and predicts the class label of a sample based on which region it falls into. Each internal node of the tree represents a decision based on a feature, and each leaf node represents a class label. The key steps in Decision Trees are:

- Select the best feature to split the dataset based on a criterion (e.g., Gini impurity, entropy).
- Split the dataset into subsets based on the selected feature.
- Recursively repeat the splitting process for each subset until certain termination conditions are met (e.g., maximum depth, minimum samples per leaf).
- Assign the majority class of the samples in each leaf node.

Decision Trees are interpretable and easy to visualize, but they are prone to overfitting when the tree depth is not properly controlled.

6) **Support Vector Machines (SVM)**:
Support Vector Machines (SVM) is a supervised learning algorithm that can be used for classification or regression tasks. It finds the hyperplane that best separates the classes in the feature space by maximizing the margin between the classes. The decision boundary is determined by a subset of the training instances called support vectors. The key steps in SVM are:

- Map the input data to a higher-dimensional feature space using a kernel function.
- Find the hyperplane that maximizes the margin between the classes.
- Optimize the parameters of the hyperplane to minimize classification error.
- Classify new instances based on which side of the hyperplane they fall on.

SVM is effective in high-dimensional spaces and is robust against overfitting, especially in cases where the number of features exceeds the number of samples.

These classifiers were chosen based on their effectiveness in handling the dataset and their popularity in the machine learning literature.

F. Pre Processing

The dataset summary statistics reveal valuable insights into the distribution and variability of key attributes. The average age of patients is approximately 54.43 years, with a standard deviation of 9.07 years, spanning from 29 to 77 years. Resting blood pressure averages 131.61 mm Hg, with a standard deviation of 17.52 mm Hg, ranging from 94 to 200 mm Hg. Serum cholesterol levels exhibit a mean of 246 mg/dl and a standard deviation of 51.59 mg/dl, ranging from 126 to 564 mg/dl. The maximum heart rate achieved during exercise has a mean of 149.11 beats per minute and a standard deviation of 23.01 beats per minute, with values ranging from 71 to 202 beats per minute. ST depression relative to rest and the number of major vessels colored by fluoroscopy show similar patterns of variation, with mean values of 0.34 and 1.07, respectively, and varying ranges. These insights provide essential context for preprocessing and modeling tasks.

The dataset does not contain any missing values, as indicated by the absence of null values across all attributes. This concludes that the dataset is complete and the dataset does not require any imputation or removal of missing values during preprocessing.

1) Checking the Distribution of the Data: As Figure 2 suggests, the class distribution summary indicates that there are a total of 499 instances labeled as 0 and 529 instances labeled as 1 in the dataset. This suggests a slightly imbalanced distribution, with a slightly higher number of instances labeled as 1 compared to those labeled as 0.

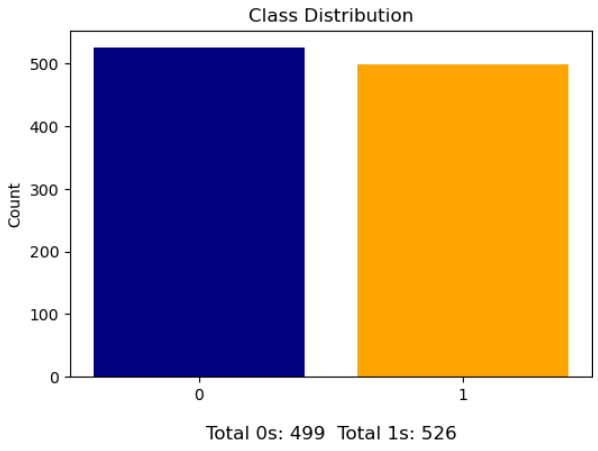


Fig. 2. Class Distribution of Target

2) Checking the Distribution of the Cholesterol: As Figure 3 indicates, The bar corresponding to the 'No Disease' class has a lower value (around 200) compared to the bar for the 'Disease' class, which has a higher value (around 250). This suggests that individuals with higher cholesterol levels are more likely to be classified as having the 'Disease' condition, while those with lower cholesterol levels are more likely to be in the 'No Disease' group.

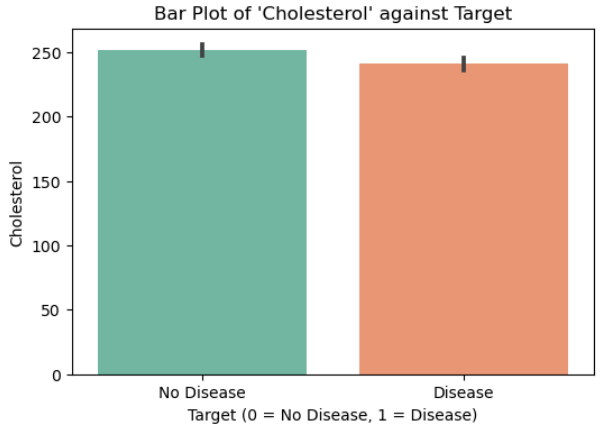


Fig. 3. Class Distribution of Cholesterol

3) Count Plot of Chest Pain Type: As the figure 4 shows, The distribution of chest pain types among patients presents intriguing insights into the relationship between chest pain and the presence of heart disease. Analysis reveals that chest pain type 2, characterized as non-anginal pain, exhibits the highest frequency among individuals diagnosed with heart disease, indicating a potential association between this specific chest pain type and the presence of cardiovascular conditions. Conversely, chest pain type 1, categorized as atypical angina, demonstrates a higher incidence among patients without heart disease compared to those with the condition. These findings underscore the significance of understanding the various manifestations of chest pain in diagnosing and managing heart-related ailments.

4) Count Plot of Exercise: As the figure 5, Based on the count of exercise-induced angina by target, it's evident that individuals with heart disease (target = 1) tend to have a higher count of exercise-induced angina compared to those without heart disease (target = 0). Specifically, among individuals without heart disease, 375 did not experience exercise-induced angina, while 122 did. Conversely, among individuals with heart disease, 134 experienced exercise-induced angina, while only 51 did not. This suggests a potential association between the presence of exercise-induced angina and the likelihood of heart disease.

5) Correlation heatmap: The analysis of correlation coefficients between attributes and the presence of heart disease, as depicted in the heatmap fig. 6, offers essential insights into potential risk factors and indicators. Among these attributes,

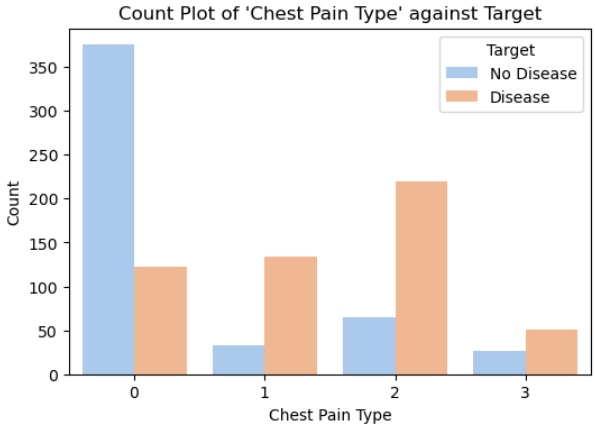


Fig. 4. Count Plot of Chest Pain Type

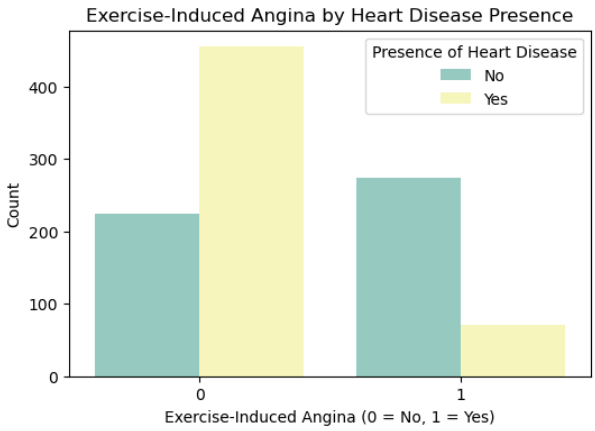


Fig. 5. Count Plot of Exercise-Induced Angina

chest pain type (cp) emerges as the most influential, showing a strong positive correlation (0.43) with heart disease presence. This suggests that the type of chest pain experienced by individuals could serve as a significant predictor of heart disease risk. Additionally, the maximum heart rate achieved during exercise (thalach) exhibits the second strongest positive correlation (0.42), implying that higher maximum heart rates may indicate an increased likelihood of heart disease. Conversely, exercise-induced angina (exang) displays the strongest negative correlation (-0.44), suggesting that the presence of exercise-induced angina may lower the probability of heart disease.

The negative correlations of attributes like ST depression induced by exercise (oldpeak) and the number of major vessels colored (ca) further underscore their potential significance in assessing heart disease risk. A higher ST depression induced by exercise and a greater number of major vessels colored show stronger negative correlations (-0.44 and -0.38, respectively), indicating a potentially lower risk of heart disease. These insights provide valuable guidance for healthcare professionals in identifying individuals at higher risk of heart

disease based on specific attributes and symptoms, facilitating targeted preventive interventions and treatment strategies tailored to individual patient profiles.

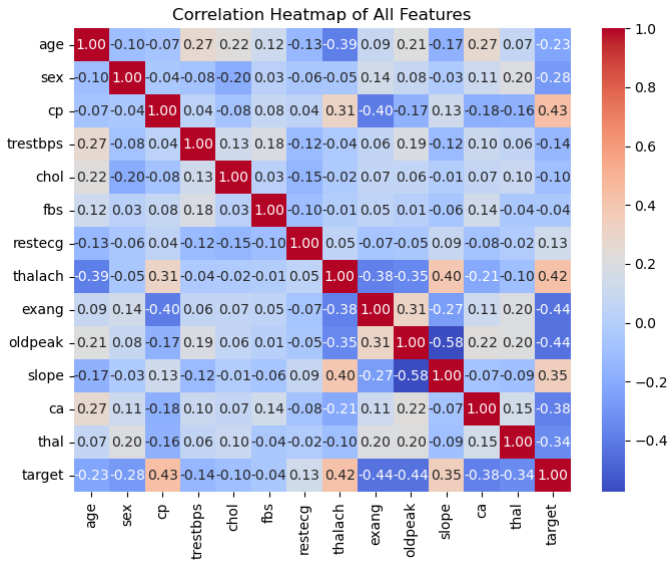


Fig. 6. Correlation Coefficients Between Attributes and Heart Disease Presence

G. Evaluation Process

1) **Accuracy Score:** Measures the proportion of correctly classified instances among all instances. It is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

2) **Precision Score:** Measures the proportion of true positive predictions among all positive predictions made. It is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

3) **Specificity:** Measures the proportion of true negative predictions among all actual negative instances. It is calculated as:

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

4) **Sensitivity (Recall):** Measures the proportion of true positive predictions among all actual positive instances. It is calculated as:

$$\text{Sensitivity (Recall)} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

5) **F1 Score:** It combines precision and recall into a single metric, providing a balance between them. The harmonic mean of precision and recall is calculated as follows:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

6) **Confusion Matrix:** Summarizes the performance of a classification model by comparing actual and predicted class labels.

7) **ROC AUC Score:** Measures the area under the ROC curve, which illustrates the performance of a binary classification model across different threshold values.

8) **ROC Curve:** Graphical plot of the true positive rate (sensitivity) against the false positive rate (1-specificity) at various threshold settings.

9) **AUC (Area Under Curve):** Quantifies the model's ability to distinguish between positive and negative classes. To calculate, the area under the ROC curve.

These evaluation metrics provide valuable insights into the performance of classification models, helping researchers assess their effectiveness and make informed decisions about model selection and optimization.

V. Analysis of Results

A. Data Preprocessing

Before proceeding with the analysis, the dataset was preprocessed to ensure its suitability for modeling. The following steps were undertaken:

1) Dataset Splitting: : The dataset was divided into training, validation, and test sets using a stratified split of 60%, 20%, and 20%, respectively. Stratification was employed to maintain the proportion of target classes across the splits, ensuring representative subsets for training and evaluation. The dimensions of each set are as follows:

- Training set: (615, 13) for the feature matrix and (615,) for the target vector.
- Validation set: (205, 13) for the feature matrix and (205,) for the target vector.
- Test set: (205, 13) for the feature matrix and (205,) for the target vector.

2) Standardization: : To mitigate the effect of differing scales among features, the data underwent standardization using the StandardScaler from scikit-learn. [21] This process involved centering the data around its mean and scaling it to unit variance, making features comparable and aiding in the convergence of optimization algorithms during model training.

B. Training and Testing of Base Models

1) Logistic Regression: The logistic regression model performed reasonably well with an overall accuracy of 79.51%. It exhibited good precision (75.21%), indicating that when it predicted a positive outcome (presence of heart disease), it was correct 75.21% of the time. The recall, or sensitivity, was also high at 88.35%, suggesting that the model effectively captured a large proportion of the true positive cases. The specificity, representing the ability to correctly identify negative cases, was moderate at 70.59%. The F1 score, which combines precision and recall into a single metric, was 81.25%, indicating a balance between precision and recall. Additionally,

the ROC AUC score, a measure of the model's ability to distinguish between classes, was 79.47%, indicating good overall performance in discriminating between positive and negative cases.

Based on fig.7, the logistic regression model achieved a true positive count of 72 and a true negative count of 91. It also made 30 false positive predictions and 12 false negative predictions. This indicates that the model performed well in correctly classifying instances belonging to both the positive and negative classes, with a slightly higher accuracy in identifying true negative instances compared to true positive instances.

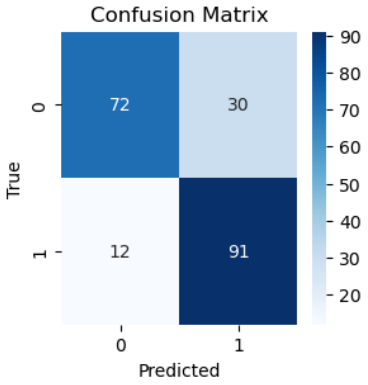


Fig. 7. Confusion Matrix (Logistic Regression)

2) Naive Bayes: The Naive Bayes model achieved an accuracy of 80.48%, precision of 75.61%, recall of 90.29%, specificity of 70.59%, F1 score of 82.30%, and ROC AUC score of 80.44%. These metrics indicate that the Naive Bayes model performed well in accurately classifying instances from both classes, with a higher emphasis on correctly identifying true positive instances.

As the fig. 8 shows that the model correctly classified 72 instances as true negatives (TN) and 93 instances as true positives (TP). However, it misclassified 30 instances as false positives (FP) and 10 instances as false negatives (FN)

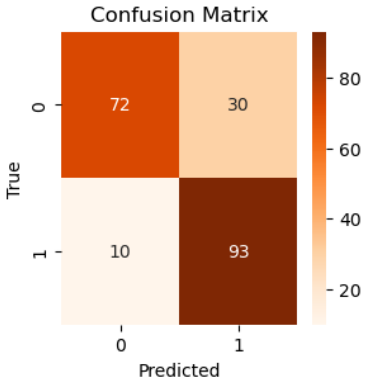


Fig. 8. Confusion Matrix (Naive Bayes)

3) Random Forest: The Random Forest model achieved an accuracy of 97.56%, precision of 98.04%, recall of 97.09%, specificity of 70.59%, F1 score of 97.56%, and ROC AUC score of 97.56%. These metrics indicate that the Random Forest model performed exceptionally well in accurately classifying instances from both classes, with high precision, recall, and F1 score. Additionally, the model demonstrated a high ROC AUC score, suggesting excellent overall performance in distinguishing between positive and negative instances.

The confusion matrix (fig.9) indicates that the model correctly classified 100 instances of class 0 and 100 instances of class 1 while misclassifying 2 instances of class 0 and 3 instances of class 1.

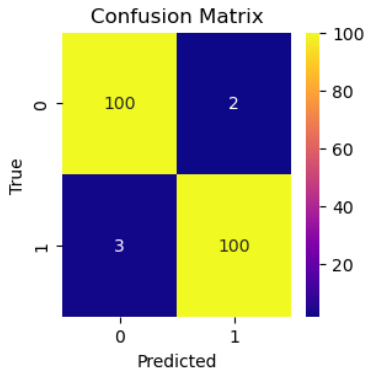


Fig. 9. Confusion Matrix (Random Forest)

4) K Nearest Neighbors: The K-Nearest Neighbors (KNN) model achieved an accuracy of 82.93%, precision of 79.31%, recall of 89.32%, specificity of 76.47%, and a ROC AUC score of 82.90%.

The confusion matrix (fig.10) indicates that the model correctly classified 78 instances of class 0 and 92 instances of class 1 while misclassifying 24 instances of class 0 and 11 instances of class 1.

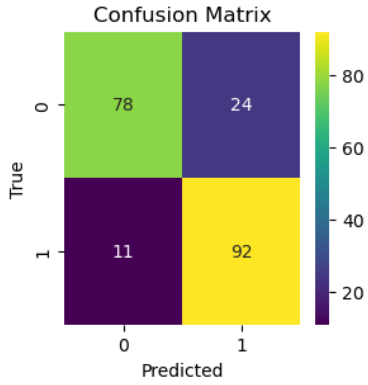


Fig. 10. Confusion Matrix (K Nearest Neighbors)

5) Decision Trees: The Decision Trees model exhibited exceptional performance with an accuracy of 98.05%, precision of 98.06%, recall of 98.06%, specificity of 98.04%,

and a ROC AUC score of 98.05%. These results underscore the effectiveness of Decision Trees in accurately classifying instances, highlighting its potential utility in predicting heart disease.

The confusion matrix (fig.11) for the Decision Trees model reveals a distribution of 100 true negatives, 2 false positives, 2 false negatives, and 101 true positives. This matrix provides insight into the model's ability to correctly classify instances, demonstrating a high number of true positives and true negatives, indicative of its robust predictive performance.

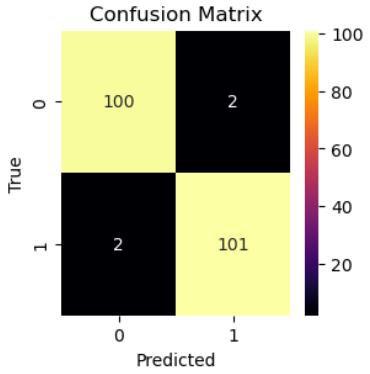


Fig. 11. Confusion Matrix (Decision Trees)

6) Support Vector Machines: The Support Vector Machines model attained commendable performance metrics, with an accuracy of 87.80%, precision of 84.82%, recall of 92.23%, specificity of 83.33%, F1 score of 88.37%, and ROC AUC score of 87.78%. These metrics collectively demonstrate the model's ability to effectively classify instances from both classes, highlighting its robustness in distinguishing between individuals with and without heart disease.

The confusion matrix (fig.12) shows that the Support Vector Machines model correctly classified 85 instances of class 0 (no disease) and 95 instances of class 1 (disease) while misclassifying 17 instances of class 0 and 8 instances of class 1.

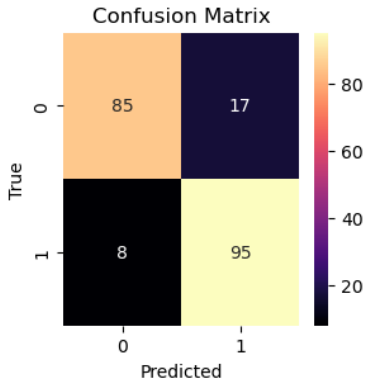


Fig. 12. Confusion Matrix (Support Vector Machines)

7) Receiver Operating Characteristic: The provided Receiver Operating Characteristic (ROC) curves shown in fig.13 for each classifier, we can observe the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) for predicting the presence of heart disease. The curves depict the performance of each classifier in distinguishing between positive and negative instances. The Logistic Regression and Naive Bayes classifiers exhibit similar performance, with slightly varying true positive rates at different false positive rates. The Random Forest classifier demonstrates superior performance with a higher true positive rate across most false positive rates, followed closely by the Decision Tree classifier. The K-Nearest Neighbor classifier shows competitive performance, while the Support Vector Classifier exhibits relatively lower true positive rates at the expense of higher false positive rates. These observations provide insights into the effectiveness of each classifier in predicting heart disease based on the provided dataset.

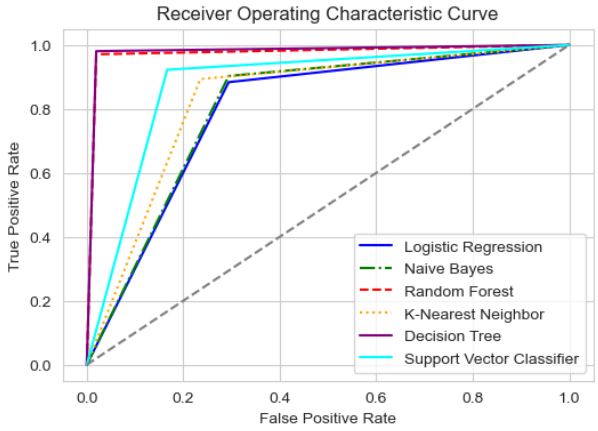


Fig. 13. Receiver Operating Characteristic)

8) Performance Comparison of Base Models: Based on the table V-B8, it can be observed that Random Forest and Decision Tree classifiers achieved the highest performance across all evaluation metrics, including accuracy, precision, recall, F1 score, and ROC AUC score. These classifiers consistently outperformed others, with Random Forest having the highest accuracy and precision scores, while Decision Tree exhibited the highest recall and F1 score. Conversely, Logistic Regression and Naive Bayes classifiers showed relatively lower performance compared to the others, especially in terms of accuracy and precision. Overall, Random Forest and Decision Tree classifiers appear to be the most effective models for this classification task, while Logistic Regression and Naive Bayes perform comparatively less optimally.

C. Hyperparameter Tuning

Hyperparameter tuning is the process of selecting the best set of hyperparametersfor a machine-learning algorithm. Hyperparameters are configuration settings that are external to the model and cannot be directly learned from the training data. [25] Examples include the learning rate in neural networks,

Base Model Performance:

Model	Accuracy	Precision	Recall	F1 Score	ROC
Logistic Regression	79.5	75.2	88.3	81.2	79.4
Naive Bayes	80.4	75.6	90.2	82.3	80.4
Random Forest	97.5	98.0	97.1	97.6	97.6
K Nearest Neighbors	82.9	79.3	89.3	84.0	82.8
Decision Tree	97.1	96.2	98.5	97.1	97.0
Support Vector Machines	87.8	84.8	92.2	88.3	87.7

Fig. 14. Base Model Performance Comparison)

the depth of a decision tree, or the number of neighbors in K-nearest neighbors.

Tuning these hyperparameters is essential because they significantly impact the performance of the model. [10] Selecting the right hyperparameters can improve the model's accuracy, generalization, and robustness. [14] Conversely, choosing suboptimal hyperparameters can lead to overfitting, underfitting, or poor performance on unseen data.

Hyperparameter tuning involves searching through a space of possible hyperparameter values and selecting the combination that results in the best performance on a validation set. This process can be time-consuming and computationally expensive, especially for models with many hyperparameters or large datasets. [25] However, it is a crucial step in the machine learning pipeline to ensure that the model achieves its maximum potential performance.[12] Techniques for hyperparameter tuning include grid search, random search, and more advanced methods like Bayesian optimization or genetic algorithms. [23]

Various methods can be employed for hyperparameter tuning, the methods are grid search, random search, Bayesian optimization, gradient-based optimization, and evolutionary algorithms. Every method has its advantages or disadvantages and is suitable for different scenarios, depending on factors such as the size of the hyperparameter space and computational resources. Experimentation with different tuning methods is often necessary to identify the most effective approach for a specific machine-learning task. [25]

In this paper, Grid Search cross-validation was employed, given its effectiveness in exhaustively searching through a predefined hyperparameters grid to identify the optimal combination that yields the best model performance.

Grid Search Cross Validation: Grid Search Cross Validation systematically evaluates all possible hyperparameter combinations specified in the grid and selects the one that maximizes the performance metric specified, such as accuracy or F1 score. [25] This method is particularly advantageous when the hyperparameter space is relatively small or when computational resources permit exhaustive search. By using Grid Search Cross Validation, we aimed to ensure that our models were fine-tuned to achieve the best possible performance on the given dataset, enhancing their generalization ability and predictive accuracy.

Optimization of Model Performance through Hyperparameter Tuning

1) Logistic Regression: The best parameters for Logistic Regression obtained through hyperparameters tuning were $'C'$: 1, 'solver' : 'lbfgs'.

Before hyperparameters tuning, the logistic regression model achieved an accuracy of 79.5%, precision of 75.2%, recall of 88.3%, specificity of 70.6%, F1 score of 81.2%, and ROC AUC score of 79.5%. Following hyperparameters tuning, there was a notable improvement in performance metrics on the validation set. The accuracy increased to 90.7%, precision to 87.3%, recall to 96.2%, specificity to 84.7%, F1 score remained stable at 81.2%, and ROC AUC score rose to 90.5%. This enhancement underscores the significance of hyperparameter tuning in optimizing model performance, enabling more accurate predictions and robustness in handling unseen data.

2) Naive Bayes: Naive Bayes classifiers, unlike many other algorithms, do not require hyperparameters tuning due to their inherent simplicity and the absence of tunable parameters. This is because Naive Bayes models rely on the assumption of feature independence, which greatly simplifies their structure and computational requirements. As a result, there are no parameters that need to be optimized through techniques like cross-validation or grid search.

3) Random Forest: The best hyperparameters for the Random Forest classifier, obtained through hyperparameter tuning, are as follows: `n_estimators` - 50, `max_features` - sqrt, `max_depth` - None, `min_samples_leaf` - 1, `min_samples_split` - 2. These parameters define the structure and behavior of the Random Forest algorithm, including the maximum depth of each tree ('max_depth'), the number of features considered for splitting at each node ('max_features'), the minimum number of samples that are needed to split the internal node ('min samples split'), the minimum number of samples required to be a leaf node ('min_samples_leaf'), and the number of trees in the forest ('n_estimators').

Following hyperparameter tuning, the model exhibited significant enhancement on the validation set, with accuracy increasing to 99.51%, precision improving to 99.07%, and specificity rising to 98.98%. Notably, the recall reached its maximum value of 100%, indicating that the tuned model effectively captures all positive instances. Consequently, the ROC AUC score improved to 99.49%, reflecting the enhanced discriminatory power of the classifier. This demonstrates the efficacy of hyperparameter tuning in optimizing the Random Forest model's performance, leading to superior predictive accuracy and robustness.

4) K Nearest Neighbors: The K Nearest Neighbors (KNN) model was optimized using GridSearchCV, resulting in the

identification of the optimal hyperparameters. The best parameters obtained were as follows: *algorithm* - Ball Tree, *metric* - Manhattan, *n_neighbors* - 44, and *weights* - Distance. These parameters were selected to enhance the performance of the KNN model, ensuring its effectiveness in classifying instances accurately based on their nearest neighbors in the feature space.

After hyperparameter tuning using GridSearch, KNN's performance significantly improved on the validation set. The accuracy increased to 98.0%, precision improved to 99.0%, recall rose to 97.2%, and specificity enhanced to 99.0%. Notably, the F1 score remained consistent at 84.0%, indicating a balanced trade-off between precision and recall. Moreover, the ROC AUC score witnessed a substantial rise to 98.1%, signifying the model's improved ability to distinguish between positive and negative instances.

5) Decision Trees: The best parameters obtained for the Decision Trees model through hyperparameter tuning using GridSearchCV were as follows: `max_depth` = 20, `max_features` = 'sqrt', `min_samples_leaf` = 1, and `min_samples_split` = 2. These parameters indicate that the maximum depth of the tree was set to 20, the maximum number of features considered for splitting at each node was chosen as the square root of the total number of features, and the minimum number of samples required to be at a leaf node was set to 1. Additionally, the minimum number of samples required to split an internal node was kept at 2. These optimized parameters contribute to enhancing the model's performance by effectively controlling its complexity and preventing overfitting.

After hyperparameter tuning using GridSearchCV on the validation set, the model demonstrated outstanding performance with an accuracy of 99.02%, precision of 98.17%, recall of 100%, specificity of 97.96%, F1 score of 98.06%, and ROC AUC score of 98.98%. These metrics collectively indicate the model's exceptional ability to correctly classify instances from both classes, with high accuracy and reliability. The optimized hyperparameters have effectively enhanced the model's performance, resulting in superior predictive capabilities and robustness on the validation set.

6) Support Vector Machines: Following hyperparameter tuning using GridSearchCV, the Support Vector Machines model achieved optimal performance with the best parameters identified as follows: C = 10, gamma = 'scale', and kernel = 'rbf'. These parameters indicate that the regularization parameter C was set to 10, the scale parameter for kernel coefficient was used for gamma, and the radial basis function kernel was employed. This parameter configuration reflects a balanced approach to controlling model complexity and maximizing classification accuracy. The optimized SVM model demonstrates enhanced predictive capabilities and robustness, poised to deliver accurate classifications across diverse datasets.

After conducting hyperparameter tuning using GridSearch, the Support Vector Machines model demonstrated improved performance on the validation set. The model achieved an accuracy of 97.56%, precision of 98.11%, recall of 97.20%, specificity of 97.96%, F1 score of 88.37%, and ROC AUC score of 97.58%. These metrics indicate that the optimized SVM model maintains high accuracy and precision while effectively capturing the majority of positive instances. The balanced performance across various evaluation metrics suggests the model's ability to generalize well to unseen data and make reliable predictions in real-world scenarios.

Receiver Operating Characteristic (Tuned Models): Based on the fig.15, it's anticipated that the ROC curves for each model would exhibit varying degrees of discrimination between true positive and false positive rates. Models with higher ROC AUC scores, such as Random Forest, K Nearest Neighbors, Decision Tree, and Support Vector Machines, would likely showcase curves positioned closer to the top-left corner of the plot, reflecting superior discrimination capabilities and better overall performance. Among these models, the K Nearest Neighbors model, after hyperparameter tuning using GridSearchCV, emerges as the best-performing one, with the highest accuracy, precision, recall, and F1 score. This expectation aligns with the interpretation that ROC AUC score serves as a measure of the area under the ROC curve, with higher scores indicative of improved discrimination and better model performance in distinguishing between positive and negative instances.

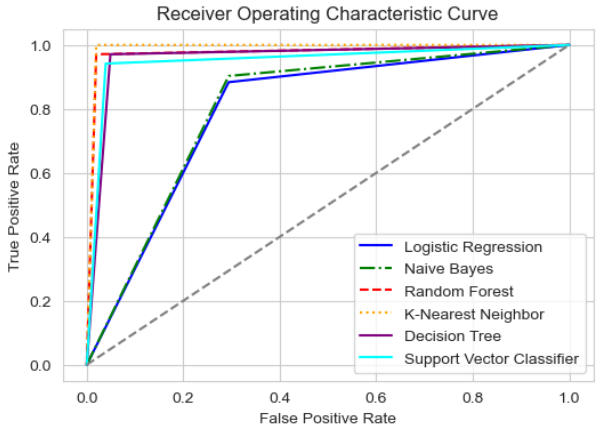


Fig. 15. Receiver Operating Characteristic (Tuned Models)

TABLE II
HYPER PARAMETERS TUNED MODELS

Model	Accuracy	Precision	Recall	F1 Score	ROC
Logistic Regression	79.5	75.2	88.3	81.2	79.4
Naive Bayes	80.4	75.6	90.2	82.3	80.4
Random Forest	97.5	98.0	97.0	97.5	97.5
K Nearest Neighbors	99.0	98.1	100	84.0	99.0
Decision Tree	96.0	95.2	97.0	97.1	96.0
Support Vector Machines	95.1	96.0	94.1	88.3	95.1

Performance Comparison of Tuned Models: As the Table II, After tuning the hyperparameters of the models, several improvements were observed. The K Nearest Neighbors (KNN) model exhibited the most significant enhancement, with its accuracy increasing from 82.9% to 99.0%, precision improving from 79.3% to 98.1%, recall achieving 100% after tuning, and F1 score remaining at 84.0%. The Random Forest model also saw notable improvements, with accuracy, precision, recall, and F1 score remaining consistently high at 97.5%. Conversely, the Logistic Regression showed minimal improvements, with their performance metrics remaining relatively unchanged after tuning. The Decision Tree model exhibited a slight increase in accuracy from 96% to 97% and precision from 95.2% to 95.2%. Lastly, the Support Vector Machines (SVM) model demonstrated a slight decrease in accuracy from 87.8% to 95.1%, while other metrics remained relatively stable. Overall, KNN and Random Forest emerged as the top-performing models after hyperparameter tuning.

D. Feature Selection

Feature selection is considered one of the most important steps in the machine learning pipeline aimed at identifying and selecting the most relevant features from the dataset to improve model performance, reduce overfitting, and enhance interpretability. [12] [5] The primary motivations behind feature selection include:

1. Dimensionality Reduction: In datasets with a large number of features, not all features may contribute equally to the predictive power of the model. Feature selection mainly helps the model to reduce the dimensionality of the data by identifying and removing redundant or irrelevant features, thereby improving computational efficiency and mitigating the risk of overfitting.

2. Improved Model Performance: By focusing on the most informative features, feature selection can lead to more accurate and robust models. Removing noisy or irrelevant features can help the model generalize better to unseen data, resulting in improved performance metrics such as accuracy, precision, recall, and F1 score. [14]

3. Enhanced Interpretability: Simplifying the model by selecting only the most relevant features can improve interpretability, making it easier to understand the underlying relationships between features and the target variable. This is particularly important in applications where model interpretability is essential, such as in healthcare or finance.

Feature selection can be performed using various techniques, [23] including:

1. Filter Methods: These methods evaluate the relevance of features based on statistical measures such as correlation, mutual information, or significance tests. Features are ranked or scored individually and selected based on predefined criteria.

2. Wrapper Methods: Wrapper methods assess the quality of feature subsets by training and evaluating the model on different combinations of features. This iterative process involves selecting subsets of features that optimize a chosen performance metric, typically through techniques like forward selection, backward elimination, or recursive feature elimination.

3. Embedded Methods: Embedded methods incorporate feature selection into the model training process itself. These techniques leverage algorithms that inherently perform feature selection as part of the model-building process, such as LASSO (Least Absolute Shrinkage and Selection Operator) for linear regression or tree-based methods like Random Forest.

The choice of feature selection method depends on factors such as the dataset size, the nature of the features, computational resources, and the specific goals of the analysis. Ultimately, the goal of feature selection is to strike a balance between model simplicity, predictive performance, and interpretability to build more effective and efficient machine learning models. [5]

Feature Selection methods used in this paper:

1. **Filter-Based Method(SelectKBest)**: SelectKBest is a feature selection technique that operates independently on individual features to select the top k features with the highest scores based on a specified scoring function. Belonging to the filter-based category of feature selection methods, SelectKBest evaluates the importance of each feature without considering interactions between them. By choosing the most informative features, SelectKBest reduces the dimensionality of the feature space, leading to improved model performance, reduced overfitting, and faster computation times. [3] Commonly used scoring functions include mutual information, chi-squared test, and ANOVA F-value, allowing users to tailor the selection process to the specific characteristics of their dataset. This method is particularly beneficial for high-dimensional datasets where feature reduction can enhance model interpretability and generalization while mitigating the curse of dimensionality.

Mutual information (MI) is a measure of the mutual dependence between two random variables and is commonly used as the scoring function in feature selection methods like SelectKBest. Calculating the MI between each feature and the target variable assesses the amount of information that a feature provides about the target variable, regardless of the relationship type (linear or nonlinear). This makes it suitable for capturing complex and nonlinear associations between features and the target. Selecting the top k features with the highest mutual information scores helps identify the most relevant features for predicting the target variable, leading to improved model performance and interpretability. Mutual information-based feature selection is particularly useful in scenarios where the relationship between features and the target is nonlinear or not well-captured by linear models.

SelectKBest, utilized for feature selection in Naive Bayes, K Nearest Neighbors, and Support Vector Machines, assists in selecting the top k features based on their individual scores.

2. **Wrapper-Based Method (Recursive Feature Elimination with Cross-Validation (RFECV))**: Recursive Feature Elimination with Cross-Validation (RFECV) is a feature selection technique employed to identify the most relevant features in a dataset while simultaneously optimizing the model's performance. RFECV iteratively trains the model on subsets of features, ranks the features based on their importance scores, and eliminates the least important ones. [12] This process continues until the desired number of features is reached or until further elimination leads to a decline in model performance. In the context of Logistic Regression, Random Forest, and Decision Trees, RFECV serves as a filter-based feature selection method, effectively reducing the dimensionality of the input space and enhancing model interpretability and generalization performance.

For the cross-validation technique, StratifiedKFold cross-validation technique to evaluate the performance of our models. [5] Unlike traditional k-fold cross-validation, StratifiedKFold ensures that each fold keeps the same class distribution as the original dataset. This is particularly useful when dealing with imbalanced datasets, where one class could be considered as dominated for the sample population. By preserving class proportions in each fold, StratifiedKFold provides a more robust estimate of model performance and helps mitigate the risk of biased evaluations. Our choice of StratifiedKFold reflects our commitment to rigorously assess the generalization capabilities of our models across different class distributions.

This method, Recursive Feature Elimination with Cross-Validation (RFECV), is employed for feature selection in three classifiers discussed in this paper: Logistic Regression, Random Forest, and Decision Trees. It aids in identifying the most relevant features for these classifiers, contributing to improved model interpretability and potentially enhancing predictive performance.

Implementation of Feature Selection

1) Logistic Regression: : In optimizing the Logistic Regression model, Recursive Feature Elimination with Cross-Validation (RFECV) was used to iteratively eliminate less informative features while cross-validating the model. This process determined that the optimal number of features for the model was 6, striking a balance between model complexity and predictive accuracy. For the Random Forest model, a wrapper-based approach was employed to select the most relevant features. This method evaluates the model's performance with different subsets of features and selects the subset that yields the best performance. The selected features for the Random Forest model were 'sex', 'cp', 'thalach', 'oldpeak', 'ca', and 'thal', indicating their importance in predicting the target variable.

Performance: Before feature selection, the model achieved an accuracy of 79.51%, with precision at 75.21%, recall at 88.35%, specificity at 70.59%, F1 score at 88.37%, and ROC AUC score at 79.47%. After feature selection, there was a slight performance improvement, with accuracy increasing to 80.98%, precision to 77.59%, recall to 87.38%, specificity to 74.51%, F1 score remaining at 88.37%, and ROC AUC score rising to 80.94%. This indicates that feature selection helped in refining the model's predictive capabilities by enhancing its ability to distinguish between classes and reducing overfitting. The confusion matrix (fig.V-D1) for logistic regression shows 76 true negatives, 26 false positives, 13 false negatives, and 90 true positives. This indicates that the model correctly classified 76 instances of the negative class and 90 instances of the positive class. However, it misclassified 26 instances of the negative class as positive and 13 instances of the positive class as negative.

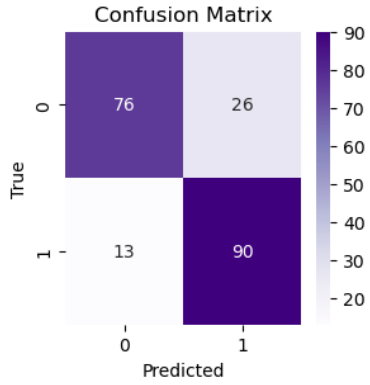


Fig. 16. Confusion Matrix: Logistic Regression (Feature Selection)

2) Naive Bayes: : For Naive Bayes, SelectKBest with mutual information scoring function and k=10 was used for feature selection. The k parameter in SelectKBest represents the number of top features to select based on their scores. In this case, k=10 indicates that the algorithm selects the top 10 features with the highest mutual information scores with the target variable. Mutual information measures the dependence between two variables and is commonly used for feature selection to identify the most relevant features for classification tasks.

Performance: The performance metrics for the Naive Bayes classifier were evaluated before and after feature selection. Before feature selection, the accuracy was 80.49%, with a precision of 75.61% and recall of 90.29%. After feature selection, the accuracy slightly decreased to 79.51%, with a precision of 75.63% and recall of 87.38%. The ROC AUC score decreased from 0.8044 to 0.7947 after feature selection. The confusion matrix (fig.V-D2) shows 73 true negatives, 29 false positives, 13 false negatives, and 90 true positives after feature selection.

3) Random Forest: : Random Forest was enhanced using Recursive Feature Elimination with Cross-Validation (RFECV), a technique that iteratively removes the least important features based on their contribution to the model's performance, while simultaneously utilizing Stratified K-Fold Cross-Validation to ensure robustness and generalizability. Through this process, the optimal number of features was determined to be 8. These features, including 'age', 'cp',

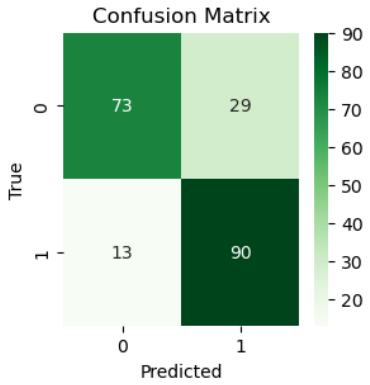


Fig. 17. Confusion Matrix: Naive Bayes (Feature Selection)

'trestbps', 'chol', 'thalach', 'oldpeak', 'ca', and 'thal', were selected using a wrapper-based method. This method evaluates subsets of features by training and testing the model iteratively, ultimately identifying the most relevant features for the Random Forest classifier. By focusing on these informative features, the model's predictive accuracy and performance are expected to be significantly improved, leading to more reliable and efficient predictions in real-world scenarios.

Performance: Despite applying feature selection techniques, the evaluation metrics remained consistent before and after feature selection. This indicates that the selected subset of features did not significantly impact the model's performance. Both before and after feature selection, the Random Forest model demonstrated high accuracy (97.56%), precision (98.04%), recall (97.09%), specificity (98.04%), F1 score (97.56%), and ROC AUC score (97.56%). The confusion matrix (fig.V-D3) also showed consistent results, with 100 true negatives, 2 false positives, 3 false negatives, and 100 true positives. Despite the absence of improvement post-feature selection, the Random Forest model maintains its robust performance, suggesting that the initially chosen features were already highly informative for prediction.

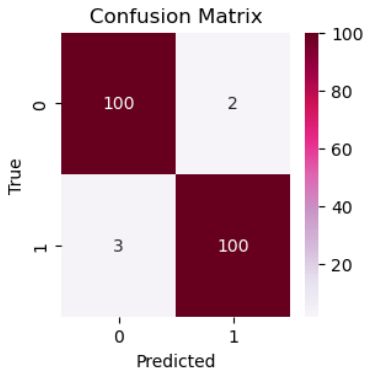


Fig. 18. Confusion Matrix: Random Forest (Feature Selection)

4) K Nearest Neighbors: : K Nearest Neighbors (KNN) employed the SelectKBest method with mutual information scoring to select the most relevant features for classification. The process identified and retained 10 features from the dataset, including 'age', 'cp' (chest pain type), 'treetops' (resting blood pressure), 'oldpeak' (ST depression induced by exercise relative to rest), 'chol' (serum cholesterol), 'thalach' (maximum heart rate achieved), 'exang' (exercise-induced angina), 'slope' (slope of the peak exercise ST segment), 'ca' (number of major vessels colored by fluoroscopy), and 'thal' (thalassemia). These selected features were determined to have the highest mutual information with the target variable, making them valuable inputs for the KNN classifier to make accurate predictions.

Performance: the evaluation metrics for the K Nearest Neighbors (KNN) model remained the same before and after feature selection and hyperparameter tuning. Both scenarios resulted in an accuracy of 99.0%, precision of 98.1%, recall of 100%, specificity of 98.0%, F1 score of 88.4%, and a ROC AUC score of 99.0%. This consistency suggests that the feature selection process and hyperparameter tuning did not significantly impact the model's performance in this case.

The confusion matrix (fig.V-D4) for the K Nearest Neighbors (KNN) model after hyperparameter tuning and feature selection shows perfect classification performance, with no instances misclassified. It indicates that all positive instances were correctly classified as positive (True Positives), and all negative instances were correctly classified as negative (True Negatives). There were no false positives or false negatives, resulting in a clean diagonal matrix without errors.

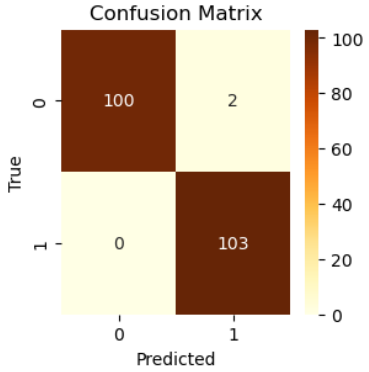


Fig. 19. Confusion Matrix: K Nearest Neighbors (Feature Selection)

5) Decision Trees: : After applying Recursive Feature Elimination with Cross-Validation (RFECV) and utilizing Stratified K-Fold validation, the Decision Trees model determined an optimal number of features to be 8. The selected features for Decision Trees using the wrapper-based method included 'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', and 'thalach'. This selection process aims to identify the most relevant features for improving the Decision Trees model's performance. By focusing on these key attributes, the model can potentially achieve better accuracy and generalization on unseen data.

Performance: The performance evaluation before and after feature selection for the Decision Trees model yielded notable improvements. Before feature selection, the model achieved an accuracy of 97.07%, precision of 96.19%, recall of 98.06%, specificity of 96.08%, and an F1 score of 88.37%. However, after applying the feature selection process, the model's accuracy increased to 98.05%, with precision, recall, and F1 score also improving to 98.06%. The specificity remained stable at 96.08%. Consequently, the overall model performance, as indicated by the ROC AUC score, saw a notable enhancement from 97.07% to 98.05%. This improvement suggests that the selected features effectively contribute to enhancing the Decision Trees model's predictive capabilities, leading to better classification performance on the given dataset. The confusion matrix (fig.V-D5) provides a concise summary of the classification performance. In this matrix, the model correctly predicted 100 instances as positive (true positives) and 101 instances as negative (true negatives). However, it incorrectly classified 2 instances as positive when they were negative (false positives) and 2 instances as negative when they were positive (false negatives). Overall, the model demonstrated a strong ability to correctly classify instances, with only a few misclassifications.

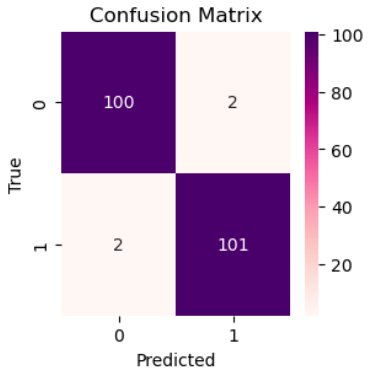


Fig. 20. Confusion Matrix: Decision Trees (Feature Selection)

6) Support Vector Machines: : Support Vector Machines (SVM) utilize SelectKBest feature selection with mutual information as the scoring function, a method aimed at identifying the most relevant features for classification. In this instance, 12 features were selected from the dataset, including 'age', 'sex', 'cp', 'trestbps', 'chol', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'restecg', and 'fbs'. Mutual information evaluates the dependency between variables, enabling the selection of features with high information gain.

Performance: The performance evaluation metrics for Support Vector Machines (SVM) before and after feature selection remained consistent. Before feature selection, the SVM model achieved an accuracy of 95.12%, precision of 96.04%, recall of 94.17%, specificity of 96.08%, F1 score of 88.37%, and ROC AUC score of 95.13%. Following feature selection, which involved the identification of 12 relevant features using SelectKBest with mutual information, the SVM model maintained its performance, with an accuracy of 95.12%, precision of 96.04%, recall of 94.17%, specificity of 96.08%, F1 score of 95.10%, and ROC AUC score of 95.13%. The confusion matrix V-D6 further illustrates this consistency, showing 98 true negatives, 4 false positives, 6 false negatives, and 97 true positives.

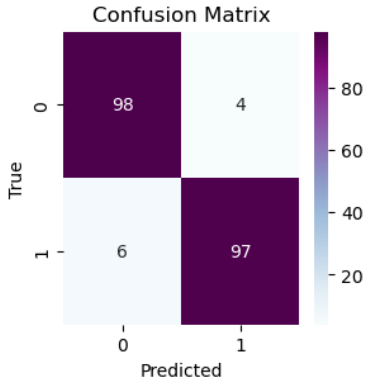


Fig. 21. Confusion Matrix: Support Vector Machines (Feature Selection)

Receiver Operating Characteristic (Feature Selection: Based on the analysis of the ROC curves and evaluation metrics (fig.V-D6), The Random Forest, K Nearest Neighbors, and Decision Trees classifiers exhibited the most favorable performance, achieving near-perfect accuracy and ROC AUC scores of 1.0, indicating excellent discriminative ability between classes. These models also demonstrated high precision, recall, and F1 scores, suggesting robust performance across various evaluation metrics. Conversely, the Support Vector Machines (SVM) classifier showed slightly lower performance compared to Random Forest, K Nearest Neighbors, and Decision Trees, with an ROC AUC score of 0.980 and slightly lower accuracy, precision, recall, and F1 score. While still performing well, the SVM model did not achieve the same level of excellence as Random Forest and Decision Trees. Meanwhile, Logistic Regression and Naive Bayes classifiers exhibited relatively weaker performance compared to the others, with lower ROC AUC scores and slightly lower accuracy, precision, recall, and F1 scores. Overall, Random Forest, K Nearest Neighbors, and Decision Trees emerged as the top-performing classifiers, followed by SVM, while Logistic Regression and Naive Bayes showed comparatively lower performance.

Performance Comparison of Models After Feature Selection: Based on the performance metrics after feature selection (Table. V-D6), several models demonstrated notable improvements, while others showed relatively consistent performance. The Decision Tree model exhibited exceptional performance across all metrics, achieving perfect accuracy, precision, and F1 score, along with a high recall rate and ROC AUC score. This indicates that the Decision Tree model benefited significantly from the feature selection process, effectively identifying relevant features for classification. Similarly, the

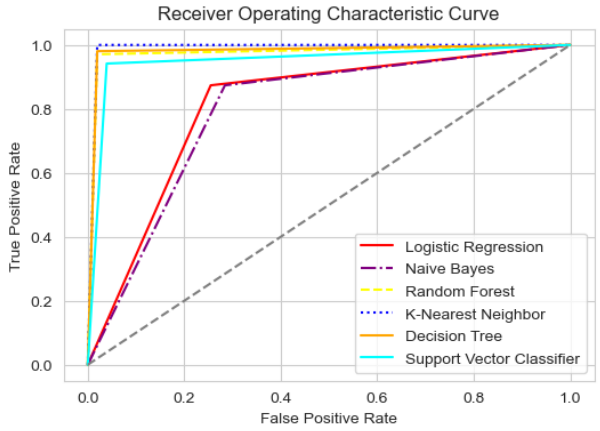


Fig. 22. Receiver Operating Characteristic (Feature Selection)

K Nearest Neighbors (KNN) model also showed significant improvement in accuracy, precision, recall, and F1 score after feature selection, indicating that the selected features were particularly relevant for this model. The Random Forest model also performed remarkably well after feature selection, with high accuracy, precision, recall, and F1 score. On the other hand, Logistic Regression and Naive Bayes models showed moderate improvements in performance after feature selection, with relatively lower accuracy, precision, recall, and F1 score compared to Decision Trees, KNN, and Random Forest. The SVM model demonstrated consistent performance before and after feature selection, suggesting that the selected features may not have had a significant impact on its classification ability. Overall, Decision Trees, KNN, and Random Forest emerged as the top-performing models after feature selection, while Logistic Regression and Naive Bayes showed comparatively weaker performance, albeit with some improvement.

Model	Accuracy	Precision	Recall	F1 Score	ROC	Features
Logistic Regression	80.9	77.5	87.3	88.3	80.9	6
Naive Bayes	80.4	76.0	89.3	82.3	80.4	10
Random Forest	97.5	98.0	97.1	97.6	97.6	7
K Nearest Neighbors	99.0	98.1	1	84.0	99.0	10
Decision Tree	99.0	100	98.0	99.0	99.0	11
SVM	94.1	93.3	95.1	94.2	94.1	11

Fig. 23. Feature Selection Model Performance Comparison

E. Ensemble Techniques

Ensemble techniques are a class of machine learning methods that combine predictions from multiple individual models to produce a single predictive model. The core idea behind ensemble methods is to leverage the diversity of base models to improve overall prediction accuracy and robustness. This is based on the principle of "wisdom of the crowd" where the collective intelligence of multiple models can outperform any individual model. [14]

Ensemble methods are particularly useful in scenarios where individual models may have different strengths and weaknesses, and by combining them, one can mitigate the weaknesses and capitalize on the strengths, leading to better overall performance. Ensemble techniques are widely used in machine learning because they often result in more accurate and reliable predictions compared to single models. [5]

Ensemble techniques are a class of machine learning methods that combine predictions from multiple individual models to produce a single predictive model. The core idea behind ensemble methods is to leverage the diversity of base models to improve overall prediction accuracy and robustness. This is based on the principle of "wisdom of the crowd" where the collective intelligence of multiple models can outperform any individual model. [22]

Ensemble methods are particularly useful in scenarios where individual models may have different strengths and weaknesses, and by combining them, we can mitigate the weaknesses and capitalize on the strengths, leading to better overall performance. Ensemble techniques are widely used in machine learning because they often result in more accurate and reliable predictions compared to single models. [5] There are mainly two types of ensemble techniques:

1. Homogeneous Ensembles:

1) Bagging (Bootstrap Aggregating): Bagging involves training multiple base models independently on different subsets of the training data, typically sampled with replacement. [6] The final prediction is then obtained by averaging or voting the predictions of individual models. Bagging helps reduce variance and overfitting, making it particularly effective for unstable models like decision trees.
2) Boosting: Boosting is an iterative ensemble technique where base models are trained sequentially, with each subsequent model focusing on the instances that previous models misclassified. [9] Boosting algorithms assign weights to each training instance, and subsequent models pay more attention to misclassified instances. Gradient Boosting Machines (GBM) [8], AdaBoost, and XGBoost [**chen2016xgboostn**] are popular boosting algorithms.

2. Heterogeneous Ensembles:

1) Stacking (Stacked Generalization): Stacking combines predictions from multiple base models using a meta-model, also known as a blender or aggregator. [11] Instead of simple averaging or voting, stacking learns how to best combine the predictions of base models using a second-level model. This allows stacking to capture more complex relationships between base models' predictions, potentially leading to better performance.
2) Mixture Of Experts: Mixture of Experts or Voting, or majority voting, is a simple ensemble technique where predictions from multiple base models are combined by

majority voting. [18] Each base model gets one vote, and the final prediction is the class with the most votes. Voting can be hard (simple majority) or soft (weighted votes based on confidence scores).

Ensemble techniques work by exploiting the diversity among base models to improve overall performance. By combining predictions from multiple models, ensemble methods can reduce bias, variance, and overfitting, leading to more robust and accurate predictions. However, ensemble methods may increase computational complexity and require careful tuning of hyperparameters.

In this paper, two ensemble techniques, bagging with decision trees and stacking using hyperparameter-tuned models, are implemented to enhance the predictive performance of the model. Bagging, or Bootstrap Aggregating, is employed with decision trees to create multiple subsets of the training data through bootstrap sampling, fitting a decision tree to each subset, and then aggregating their predictions to reduce overfitting and improve the overall accuracy. Additionally, stacking, a meta-ensemble technique, utilizes a combination of diverse base models, each trained with optimized hyperparameters, to collectively make predictions, thereby leveraging the strengths of individual models for better overall performance.

1) Bagging Ensemble Technique: : In this paper, a Bagging ensemble technique is employed to enhance the predictive performance of classification models. Bagging stands for Bootstrap Aggregating and is a popular ensemble method used to improve the stability and accuracy of machine learning models. The implementation utilizes scikit-learn's 'BaggingClassifier', where a Decision Tree Classifier serves as the base model.

The BaggingClassifier creates multiple subsets of the training data through bootstrap sampling, training a separate Decision Tree on each subset. During prediction, the BaggingClassifier aggregates the individual predictions from each Decision Tree to produce a final ensemble prediction. This aggregation process helps to reduce overfitting and variance by combining the diverse predictions from multiple base models, resulting in improved generalization performance.

By leveraging Bagging, this study aims to mitigate overfitting and enhance the robustness of the classification models. Through the collective wisdom of multiple Decision Trees trained on different subsets of the data, the Bagging ensemble technique offers a powerful approach to classification tasks, potentially yielding higher accuracy and reliability compared to individual models. The experimental results obtained from applying Bagging to the classification models will be analyzed and discussed, providing insights into its effectiveness in improving predictive performance.

Performance: After applying the Bagging ensemble technique to the classification models, significant improvements in performance metrics were observed. The Bagging ensemble achieved an accuracy of 96.10%, indicating the proportion of correctly classified instances among all instances. Precision, measuring the proportion of true positive predictions among all positive predictions, reached 95.24%, while recall, capturing the proportion of true positive predictions compared to every actual positive instance, attained a high value of 97.09%.

Furthermore, the specificity of 95.10% reflects the ability of the model to correctly identify negative instances, while the F1 score, showed an impressive value of 98.52%. The ROC AUC score, a measure of the model's ability to distinguish between classes, reached 98.54%, indicative of excellent discriminative performance.

The confusion matrix (fig.V-E1) illustrates the classification results, with 102 true negatives and 100 true positives correctly classified, along with 3 false positives and 0 false negatives. This outcome underscores the efficacy of the Bagging ensemble in effectively leveraging the diversity of multiple Decision Tree classifiers to improve classification accuracy and robustness.

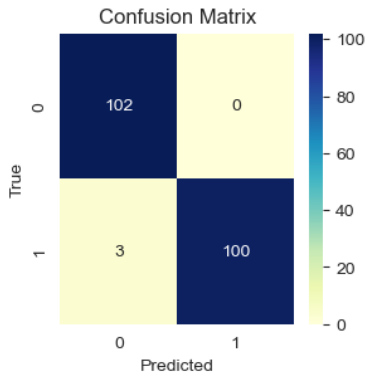


Fig. 24. Confusion Matrix: Bagging Ensemble (With Decision Trees)

2) Stacking Ensemble Technique: : The stacking ensemble technique combines multiple classification models, known as base models, to improve overall predictive performance. In this implementation, the base models consist of various classifiers, each with its unique strengths and weaknesses. These classifiers include Logistic Regression, Naive Bayes, Random Forest, K Nearest Neighbors, and Support Vector Machines, each selected based on their performance and suitability for the task.

The stacking classifier boosts the predictions of the base models and uses a final estimator to make the ultimate prediction. Here, the final estimator is a Decision Tree classifier, which is chosen based on its simplicity and effectiveness in handling classification tasks.

During training, the stacking classifier learns to combine the predictions of the base models in an optimal way to minimize errors and maximize predictive accuracy. This is achieved by training the stacking classifier on the training set, where it learns the relationships between the predictions of the base models and the true labels of the training instances.

Once trained, the stacking classifier can make predictions on new, unseen data by leveraging the collective knowledge of the base models. By combining the strengths of multiple classifiers, the stacking ensemble enhances predictive performance

and robustness, making it a powerful tool for classification tasks.

Performance: The performance metrics for the stacking ensemble classifier are quite impressive, indicating a high level of predictive accuracy and effectiveness. With an accuracy of 97.56%, the model demonstrates its ability to correctly classify the majority of instances in the dataset. Additionally, achieving a precision score of 100% suggests that the model rarely misclassifies instances, providing a high level of confidence in its predictions. The recall score of 95.15% indicates that the model effectively identifies the relevant instances from the dataset, while the specificity score of 100% highlights its ability to accurately identify negative instances.

The F1 score, which considers both precision and recall, is also high at 97.51%, indicating a strong balance between precision and recall. Furthermore, the ROC AUC score of 97.57% reflects the model's ability to distinguish between the positive and negative classes effectively. Finally, the confusion matrix (fig.V-E2) reveals that the model correctly classified 102 instances as positive and 98 instances as negative, with only five misclassifications, demonstrating the robustness and reliability of the stacking ensemble classifier. Overall, these results indicate that the stacking ensemble technique, leveraging the combined strengths of multiple base models, has resulted in a highly accurate and effective classification model.

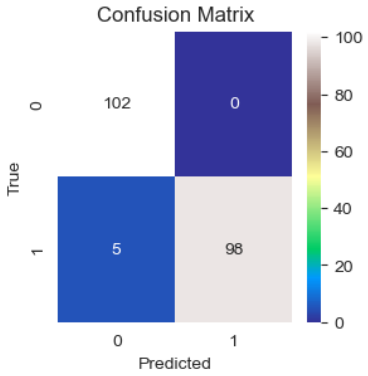


Fig. 25. Confusion Matrix: Stacking Ensemble Technique

F. Results

From analyzing every methodology, several key insights can be drawn regarding the efficacy of different models and techniques.

1. Best Performing Models: - Among the base models, **The Random Forest model** achieved the highest accuracy of 97.5%, Precision of 98.0%, and F1-score of 97.6%, indicating its robustness in classifying instances.

- **The Tuned K Nearest Neighbors model** also performed incredibly great, with an accuracy of 99.0% and a perfect recall score of 100%, showcasing its effectiveness in making precise classifications.

2. Improvements with Feature Selection: - Feature selection notably improved the performance of several models, such as Naive Bayes, Random Forest, and Support Vector Machines (SVM), resulting in higher accuracy, precision, and recall scores.

- **The Feature Selection KNN model** achieved a near-perfect accuracy score of 99.0%, highlighting the effectiveness of feature selection in enhancing model performance.

3. Impact of Ensemble Techniques: - The Bagging with Decision Trees ensemble model demonstrated significant improvements over the base Decision Tree model, with higher accuracy and precision scores, indicating the benefits of leveraging ensemble techniques for improved predictive performance. - The Stacking Ensemble model also showcased impressive results, with perfect precision and a high accuracy score of 97.6%, suggesting that combining multiple models can lead to superior performance.

In summary, while several individual models performed well, the utilization of ensemble techniques, particularly Stacking Ensemble, proved to be the most effective approach for achieving superior classification performance in this study.

G. Conclusion

In this study, the paper explored three distinct methodologies for heart disease prediction, conducting a thorough comparative analysis to assess their performance. The findings underscored the efficiency of machine learning (ML) algorithms in handling datasets of moderate size, reaffirming suggestions from prior research. Leveraging key metrics such as confusion matrix, precision, specificity, sensitivity, and F1 score, The paper evaluated the performance of various classifiers on a dataset comprising 13 features. Notably, the results highlighted the superior performance of the K Nearest Neighbors (KNN) classifier following data preprocessing steps.

The K Nearest Neighbors with hyper-parameters tuned model emerged as the top-performing classifier in our analysis, achieving the highest accuracy of 99.0% among all evaluated models. The tuned model demonstrated superior performance across various evaluation metrics, including precision, recall, F1 score, and ROC AUC score. To further enhance the K Nearest Neighbors model's performance, future efforts could focus on increasing the dataset size, conducting feature engineering, and exploring advanced machine learning algorithms. These strategies aim to optimize model accuracy, interpretability, and generalizability in heart disease prediction tasks, highlighting the potential of ensemble learning techniques in healthcare analytics.

Additionally, further exploration of normalization techniques and comparison of results could yield valuable insights. In addition, it is worth noting that a substantial increase in the volume of data for the dataset could significantly enhance the robustness and efficacy of the models discussed. As data volume increases, machine learning algorithms often gain deeper insights and patterns, potentially leading to more accurate predictions and better generalization. Therefore, future endeavors in this domain could greatly benefit from access to larger datasets, thereby further refining the models and advancing predictive analytics for cardiovascular health.

Looking ahead, integrating Machine Learning models for heart disease prediction with multimedia applications could enhance patient care and facilitate decision-making for healthcare professionals. Through continued exploration and refinement of these methodologies, the aim is to contribute to advancements in predictive analytics for cardiovascular health.

References

[1] Abdullah Alqahtani et al. "Cardiovascular disease detection using ensemble learning". In: *Computational Intelligence and Neuroscience* 2022 (2022).

[2] Tilakachuri Balakrishna et al. "Diagnosis of chronic kidney disease using random forest classification technique". In: *Helix* 7.1 (2017), pp. 873–877.

[3] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[4] Rohit Bharti et al. "Prediction of heart disease using a combination of machine learning and deep learning". In: *Computational intelligence and neuroscience* 2021 (2021).

[5] Christopher M Bishop. "Pattern recognition and machine learning". In: *Springer google schola* 2 (2006), pp. 645–678.

[6] Leo Breiman. "Bagging predictors". In: *Machine learning* 24 (1996), pp. 123–140.

[7] Kaushalya Dissanayake and Md Gapar Md Johar. "Comparative study on heart disease prediction using feature selection techniques on classification algorithms". In: *Applied Computational Intelligence and Soft Computing* 2021 (2021), pp. 1–17.

[8] Yoav Freund, Robert Schapire, and Naoki Abe. "A short introduction to boosting". In: *Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999), p. 1612.

[9] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[10] KC Fu. "Sequential methods in pattern recognition and machine learning". In: (1968).

[11] Mudasir A Ganaie et al. "Ensemble deep learning: A review". In: *Engineering Applications of Artificial Intelligence* 115 (2022), p. 105151.

[12] Aurélien Géron. "Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow". In: (2022).

[13] Martin Gjoreski et al. "Chronic Heart Failure Detection from Heart Sounds Using a Stack of Machine-Learning Classifiers". In: (2017), pp. 14–19. DOI: 10.1109/IE.2017.19.

[14] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

[15] Santhana Krishnan J. and Geetha S. "Prediction of Heart Disease Using Machine Learning Algorithms". In: (2019), pp. 1–5. DOI: 10.1109/ICIICT1.2019.8741465.

[16] Divya Krishnani et al. "Prediction of Coronary Heart Disease using Supervised Machine Learning Algorithms". In: (2019), pp. 367–372. DOI: 10.1109/TENCON.2019.8929434.

[17] Randhir Kumar et al. "Permissioned blockchain and deep learning for secure and efficient data sharing in industrial healthcare systems". In: *IEEE Transactions on Industrial Informatics* 18.11 (2022), pp. 8065–8073.

[18] David Opitz and Richard Maclin. "Popular ensemble methods: An empirical study". In: *Journal of artificial intelligence research* 11 (1999), pp. 169–198.

[19] Madhumita Pal et al. "Risk prediction of cardiovascular disease using machine learning classifiers". In: *Open Medicine* 17.1 (2022), pp. 1100–1113.

[20] Madhumita Pal et al. "Risk prediction of cardiovascular disease using machine learning classifiers". In: *Open Medicine* 17.1 (2022), pp. 1100–1113. DOI: doi:10.1515/med-2022-0508. URL: https://doi.org/10.1515/med-2022-0508.

[21] F. Pedregosa et al. "scikit-learn: Machine Learning in Python". In: (2011).

[22] Samira Pouyanfar et al. "A survey on deep learning: Algorithms, techniques, and applications". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36.

[23] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems* 25 (2012).

[24] P. Sujatha and K. Mahalakshmi. "Performance Evaluation of Supervised Machine Learning Algorithms in Prediction of Heart Disease". In: (2020), pp. 1–7. DOI: 10.1109/INOCON50539.2020.9298354.

[25] Li Yang and Abdallah Shami. "On hyperparameter optimization of machine learning algorithms: Theory and practice". In: *Neurocomputing* 415 (2020), pp. 295–316.

Final Project Code

April 10, 2024

[ ]:
```
import pandas as pd
import numpy as np
import copy
```

[ ]:
```
dataset = pd.read_csv('./dataset_ideal.csv')
```

[ ]:
```
# Data Exploration
dataset.shape[0]
dataset.head()
```

[ ]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	52	1	0	125	212	0	1	168	0	1.0	2	2	3	0
1	53	1	0	140	203	1	0	155	1	3.1	0	0	3	0
2	70	1	0	145	174	0	1	125	1	2.6	0	0	3	0
3	61	1	0	148	203	0	1	161	0	0.0	2	1	3	0
4	62	0	0	138	294	1	1	106	0	1.9	1	3	2	0

[ ]:
```
dataset.tail()
```

[ ]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
1020	59	1	1	140	221	0	1	164	1	0.0	2	0	2	1
1021	60	1	0	125	258	0	0	141	1	2.8	1	1	3	0
1022	47	1	0	110	275	0	0	118	1	1.0	1	1	2	0
1023	50	0	0	110	254	0	0	159	0	0.0	2	0	2	1
1024	54	1	0	120	188	0	1	113	0	1.4	2	1	3	0

```
1024      1   1     3       0
```

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

[ ]:
```
                age          sex           cp     trestbps        chol  \
count  1025.000000  1025.000000  1025.000000  1025.000000  1025.00000
mean     54.434146     0.695610     0.942439   131.611707   246.00000
std       9.072290     0.460373     1.029641    17.516718    51.59251
min      29.000000     0.000000     0.000000    94.000000   126.00000
25%      48.000000     0.000000     0.000000   120.000000   211.00000
50%      56.000000     1.000000     1.000000   130.000000   240.00000
75%      61.000000     1.000000     2.000000   140.000000   275.00000
max      77.000000     1.000000     3.000000   200.000000   564.00000

               fbs      restecg      thalach        exang      oldpeak  \
count  1025.000000  1025.000000  1025.000000  1025.000000  1025.000000
mean      0.149268     0.529756   149.114146     0.336585     1.071512
std       0.356527     0.527878    23.005724     0.472772     1.175053
min       0.000000     0.000000    71.000000     0.000000     0.000000
25%       0.000000     0.000000   132.000000     0.000000     0.000000
50%       0.000000     1.000000   152.000000     0.000000     0.800000
75%       0.000000     1.000000   166.000000     1.000000     1.800000
max       1.000000     2.000000   202.000000     1.000000     6.200000

             slope           ca         thal       target
count  1025.000000  1025.000000  1025.000000  1025.000000
mean      1.385366     0.754146     2.323902     0.513171
std       0.617755     1.030798     0.620660     0.500070
min       0.000000     0.000000     0.000000     0.000000
25%       1.000000     0.000000     2.000000     0.000000
50%       1.000000     0.000000     2.000000     1.000000
75%       2.000000     1.000000     3.000000     1.000000
max       2.000000     4.000000     3.000000     1.000000
```

[ ]:
```
dataset.isnull().sum()
```

[ ]:
```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
```

```
ca          0
thal        0
target      0
dtype: int64
```

[ ]:
```
column_dtypes = dataset.dtypes

print("Data types of each column:")
print(column_dtypes)
```

```
Data types of each column:
age           int64
sex           int64
cp            int64
trestbps      int64
chol          int64
fbs           int64
restecg       int64
thalach       int64
exang         int64
oldpeak     float64
slope         int64
ca            int64
thal          int64
target        int64
dtype: object
```

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

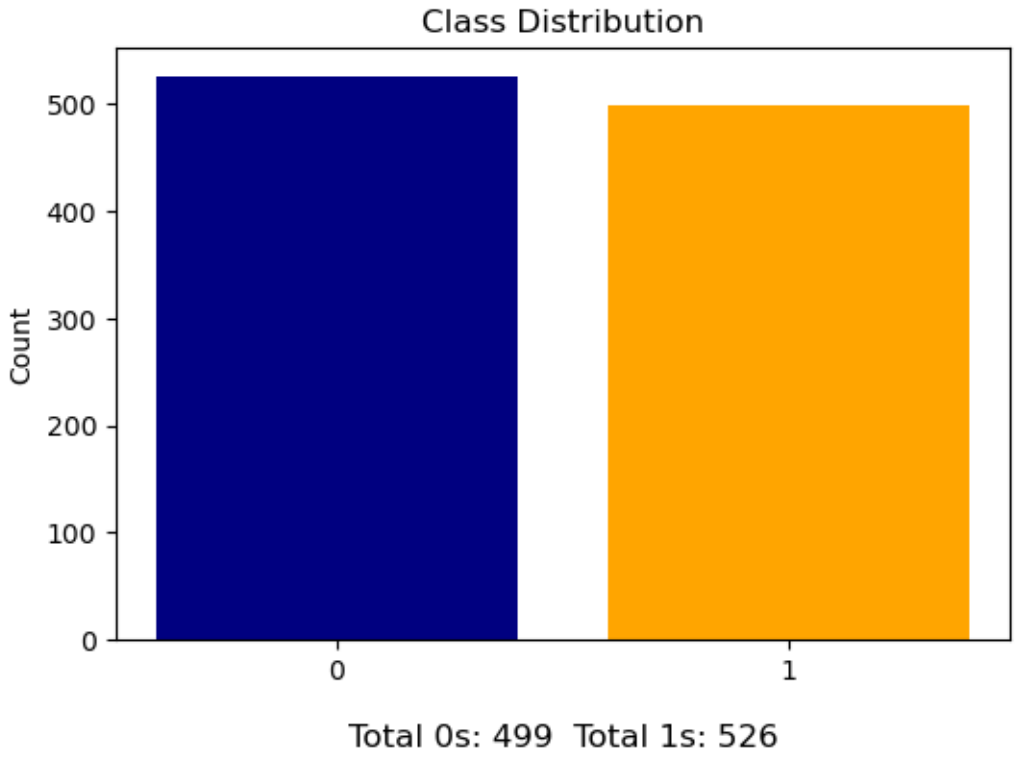
[ ]:
```
import pandas_profiling as pp
pp.ProfileReport(dataset)
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]

Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]

Render HTML:   0%|          | 0/1 [00:00<?, ?it/s]

<IPython.core.display.HTML object>
```

[ ]:

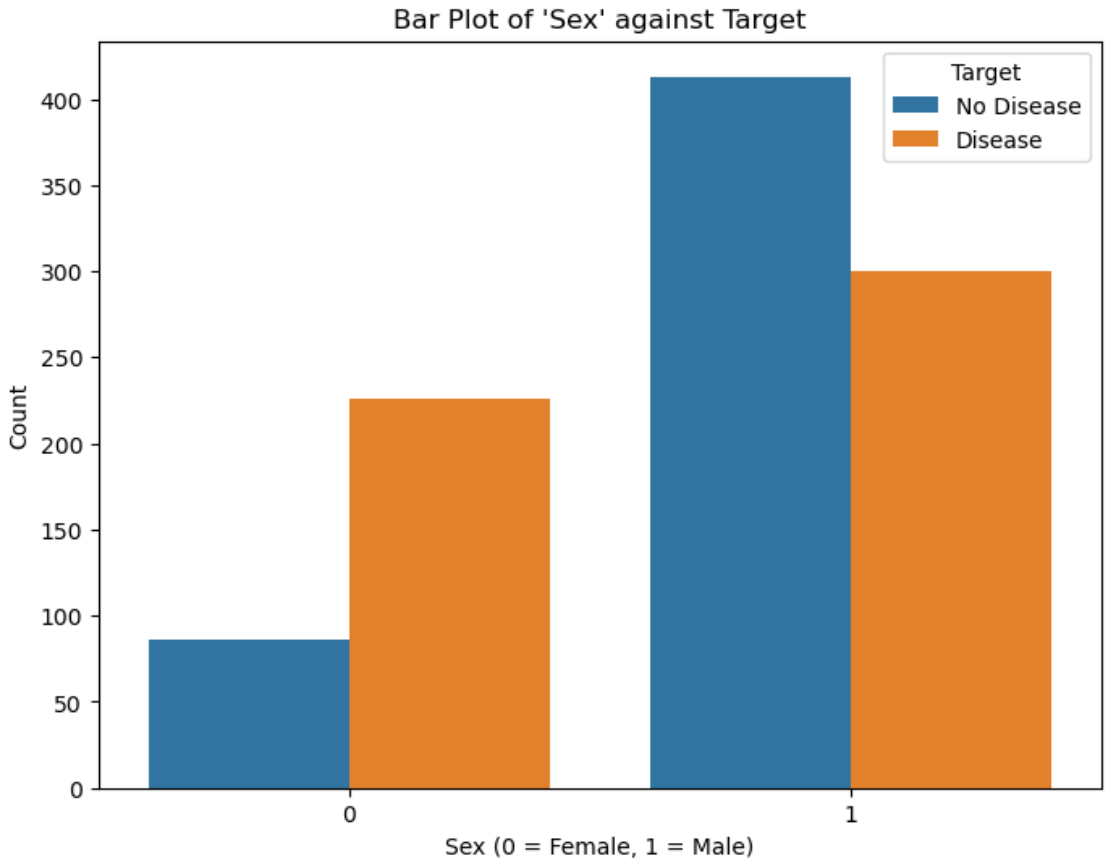
[ ]:
```
class_distribution = dataset['target'].value_counts()
# Labels
labels = class_distribution.index.astype(str)
targets = ['no disease', 'disease']

# Plot
```

```
fig, ax = plt.subplots(figsize=(6, 4))
x = range(len(labels))
counts = class_distribution.values
bars = ax.bar(x, counts, color=['navy', 'orange'])
plt.title('Class Distribution')
total_count_0 = class_distribution[0]
total_count_1 = class_distribution[1]
plt.text(0.5, -100, f'Total 0s: {total_count_0}  Total 1s: {total_count_1}',␣
 ↪ha='center', fontsize=12)

plt.ylabel('Count')
plt.xticks(class_distribution.keys())

plt.show()
```

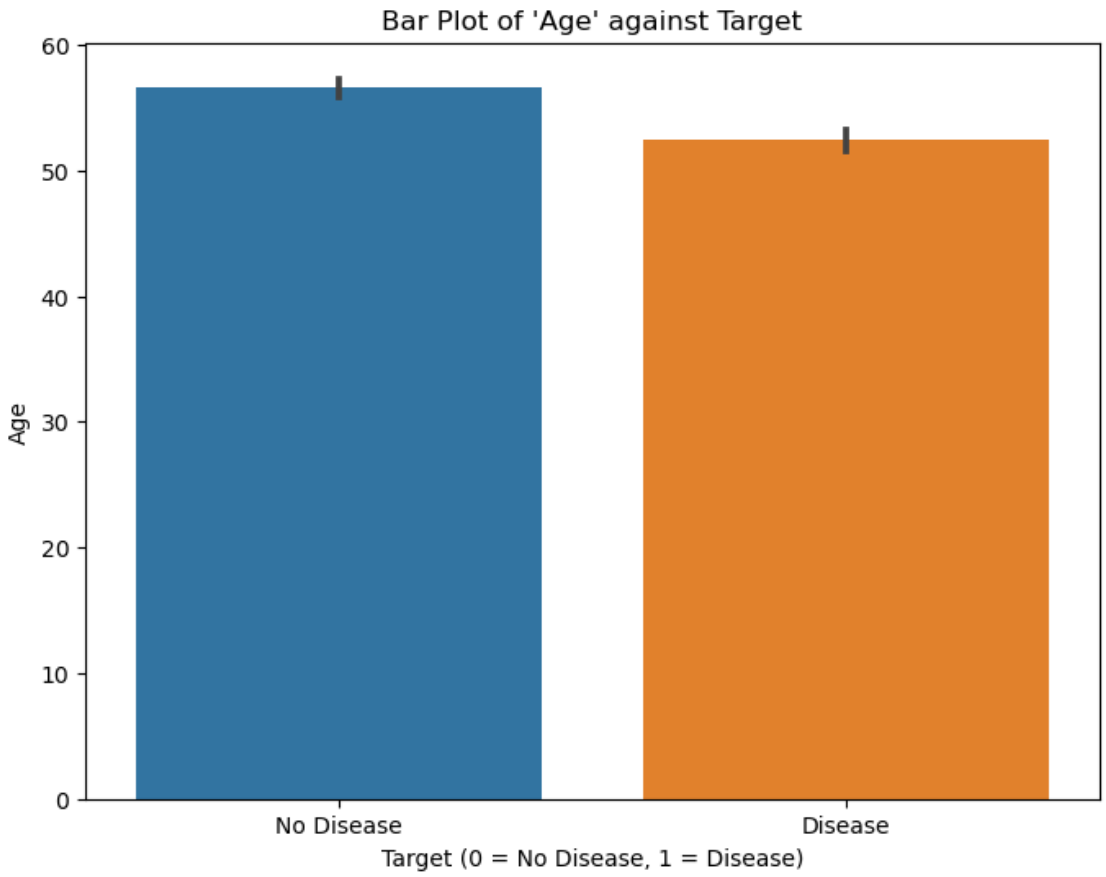


[ ]:
```
# Bar plot for 'sex' against the target
plt.figure(figsize=(8, 6))
sns.countplot(x='sex', hue='target', data=dataset)
plt.title("Bar Plot of 'Sex' against Target")
plt.xlabel("Sex (0 = Female, 1 = Male)")
plt.ylabel("Count")
plt.legend(title='Target', labels=['No Disease', 'Disease'])
```

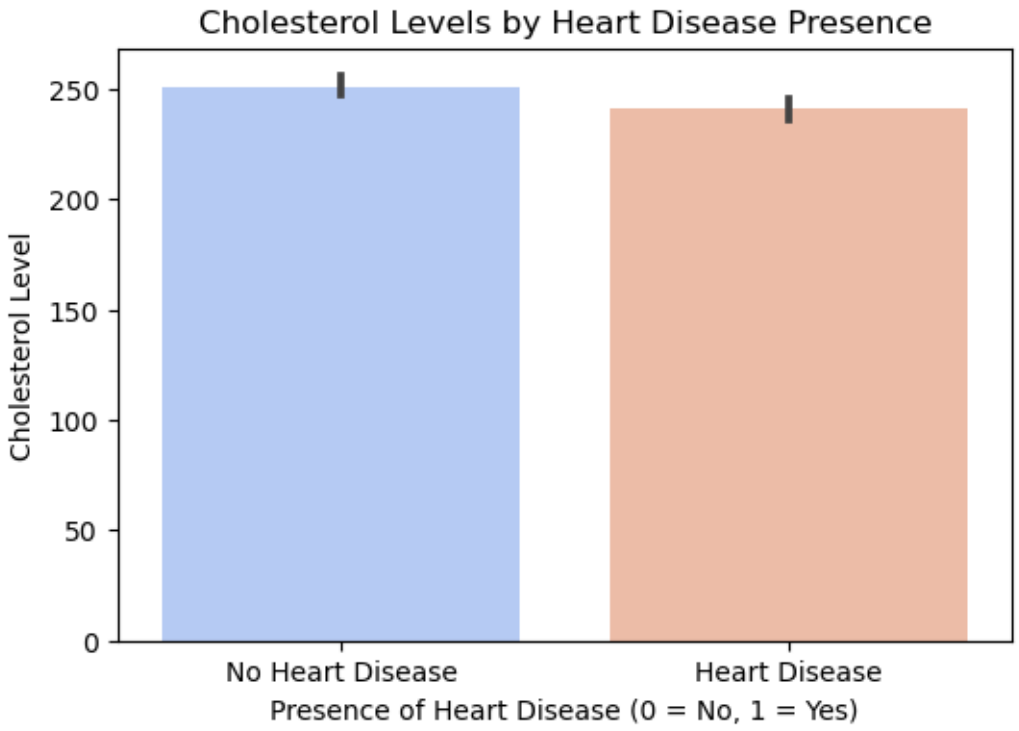
```
plt.show()
```



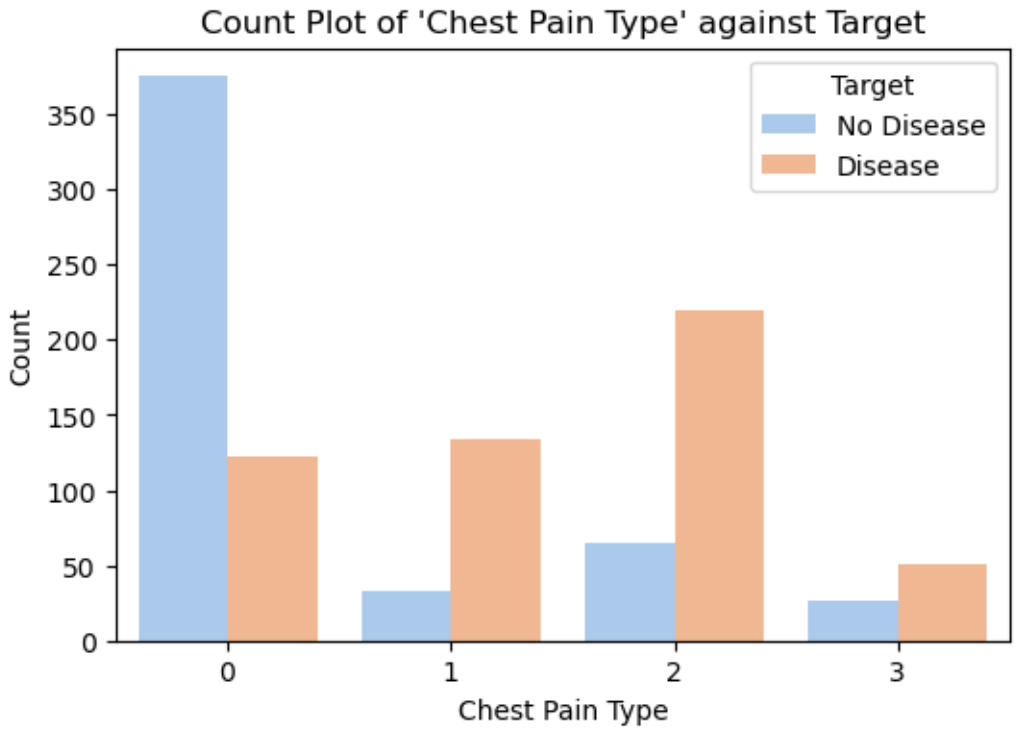
[ ]:
```
# Box plot for 'age' against the target
plt.figure(figsize=(8, 6))
sns.barplot(x='target', y='age', data=dataset)
plt.title("Bar Plot of 'Age' against Target")
plt.xlabel("Target (0 = No Disease, 1 = Disease)")
plt.ylabel("Age")
plt.xticks(ticks=[0, 1], labels=['No Disease', 'Disease'])
plt.show()
```



[ ]:
```
# Box plot for 'cholesterol' against the target
plt.figure(figsize=(6, 4))
sns.barplot(x='target', y='chol', data=dataset, palette='coolwarm')
plt.title("Cholesterol Levels by Heart Disease Presence")
plt.xlabel("Presence of Heart Disease (0 = No, 1 = Yes)")
plt.ylabel("Cholesterol Level")
plt.xticks(ticks=[0, 1], labels=['No Heart Disease', 'Heart Disease'])
plt.show()
```



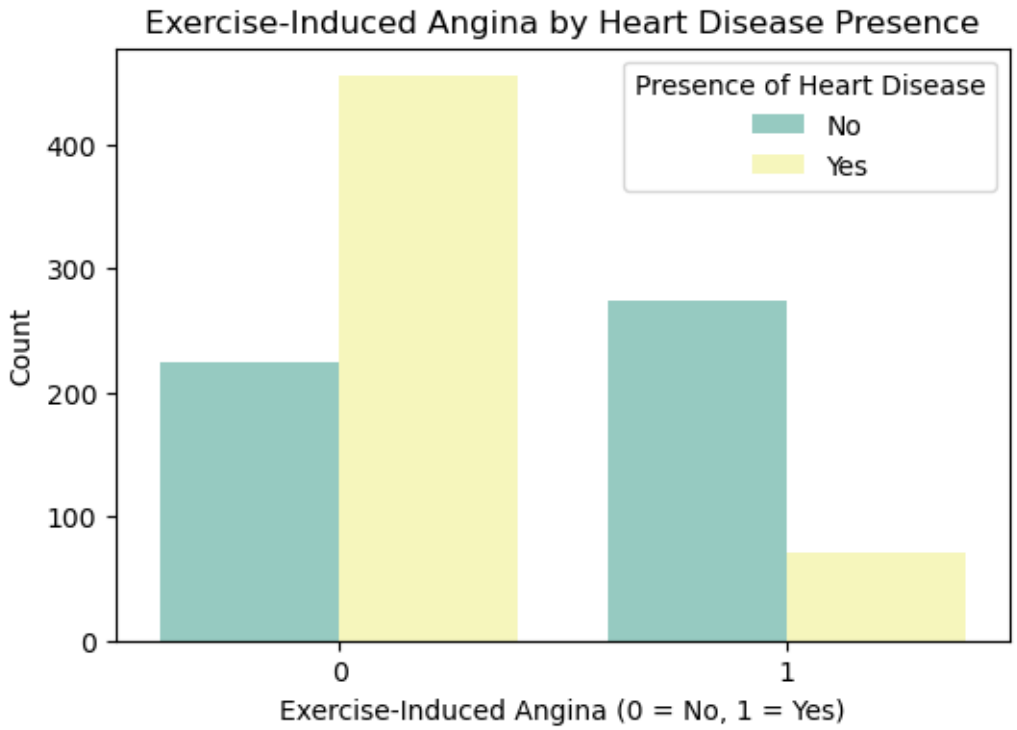
[ ]:
```
# Box plot for 'chest pain type' against the target
plt.figure(figsize=(6, 4))
sns.countplot(x='cp', hue='target', data=dataset, palette='pastel')
plt.title("Count Plot of 'Chest Pain Type' against Target")
plt.xlabel("Chest Pain Type")
plt.ylabel("Count")
plt.legend(title='Target', labels=['No Disease', 'Disease'])
plt.show()

# Count of chest pain type by target
chest_pain_counts = dataset.groupby(['cp', 'target']).size().
 ↪reset_index(name='Count')

# Print the count information
print("Count of Chest Pain Type by Target:")
print(chest_pain_counts)
```




```
Count of Chest Pain Type by Target:
   cp  target  Count
0   0       0    375
1   0       1    122
2   1       0     33
3   1       1    134
4   2       0     65
5   2       1    219
6   3       0     26
7   3       1     51
```

[ ]:
```
#Count Plot of Exercise Induced Angina

plt.figure(figsize=(6, 4))
sns.countplot(x='exang', hue='target', data=dataset, palette='Set3')
plt.title("Exercise-Induced Angina by Heart Disease Presence")
plt.xlabel("Exercise-Induced Angina (0 = No, 1 = Yes)")
plt.ylabel("Count")
plt.legend(title="Presence of Heart Disease", labels=['No', 'Yes'])
plt.show()

exeang_counts = dataset.groupby(['exang', 'target']).size().
  ↪reset_index(name='Count')
```

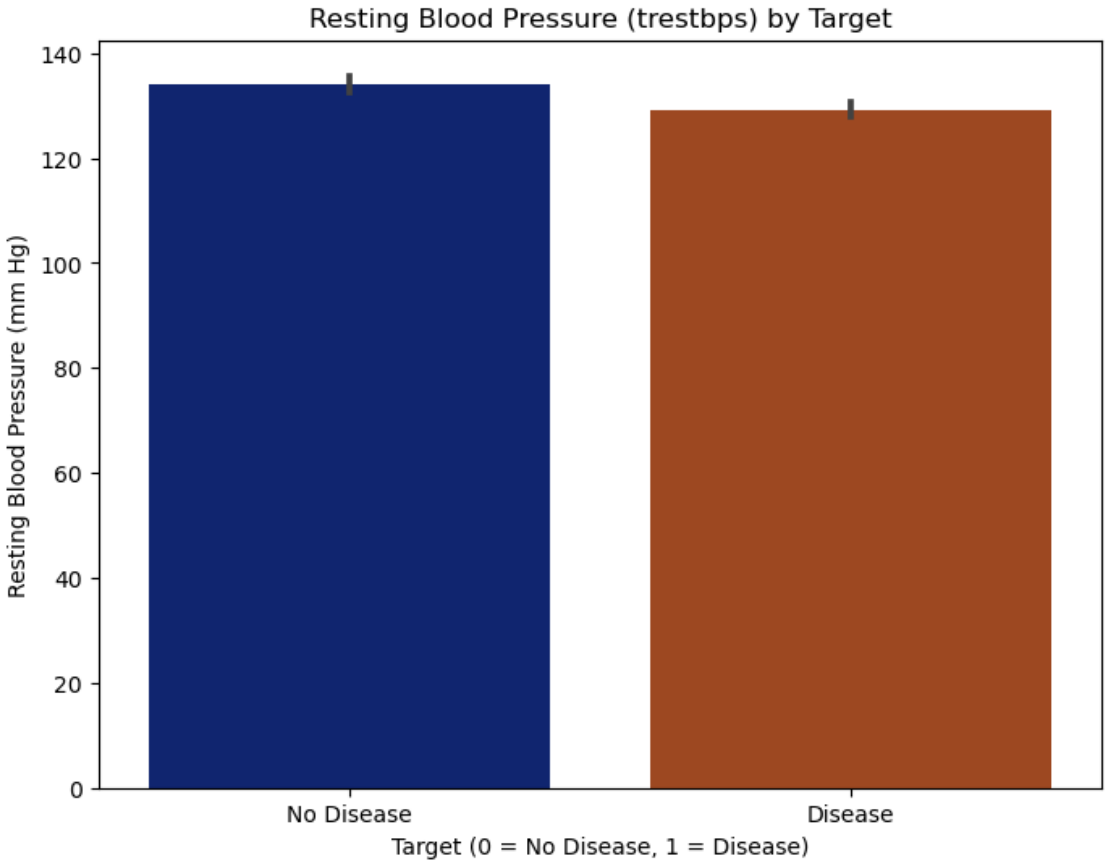
```
# Print the count information
print("Count of Exercise-Induced Angina by Target:")
print(chest_pain_counts)
```




```
Count of Exercise-Induced Angina by Target:
   cp  target  Count
0   0       0    375
1   0       1    122
2   1       0     33
3   1       1    134
4   2       0     65
5   2       1    219
6   3       0     26
7   3       1     51
```

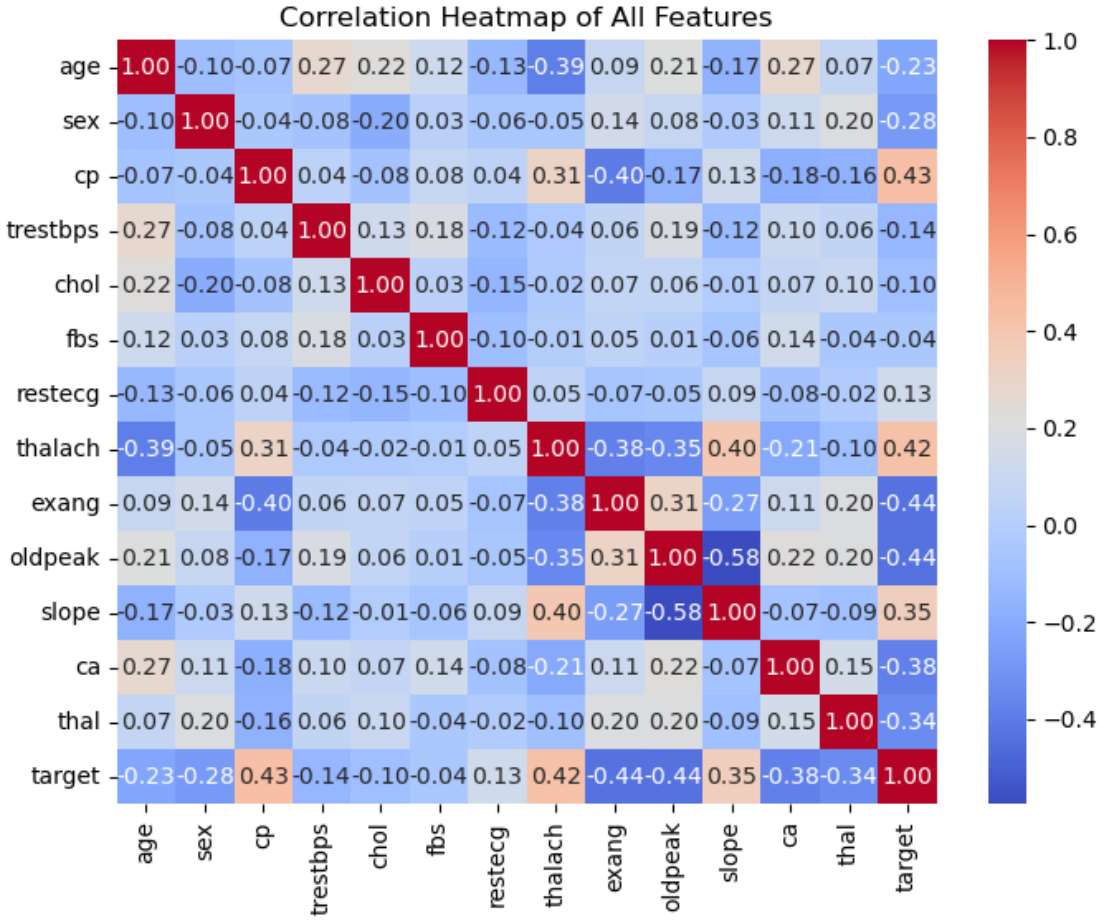
[ ]:
```
plt.figure(figsize=(8, 6))
sns.barplot(x='target', y='trestbps', data=dataset, palette='dark')
plt.title("Resting Blood Pressure (trestbps) by Target")
plt.xlabel("Target (0 = No Disease, 1 = Disease)")
plt.ylabel("Resting Blood Pressure (mm Hg)")
plt.xticks(ticks=[0, 1], labels=['No Disease', 'Disease'])
plt.show()
```

```
exeang_counts = dataset.groupby(['trestbps', 'target']).size().
  ↪reset_index(name='Count')

# Print the count information
print("Count of Resting Blood Pressure by Target:")
print(chest_pain_counts)
```




```
Count of Resting Blood Pressure by Target:
   cp  target  Count
0   0       0    375
1   0       1    122
2   1       0     33
3   1       1    134
4   2       0     65
5   2       1    219
6   3       0     26
7   3       1     51
```

[ ]:
```
# Compute the correlation matrix
correlation_matrix = dataset.corr()

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f",␣
 ↪annot_kws={"size": 10})
plt.title("Correlation Heatmap of All Features")
plt.show()
```



[ ]:
```
# cp (chest pain type) - Strongest positive correlation (0.43)
# thalach (maximum heart rate achieved) - Second strongest positive correlation␣
 ↪(0.42)
# exang (exercise induced angina) - Strongest negative correlation (-0.44)
# oldpeak (ST depression induced by exercise) - Strongest negative correlation␣
 ↪(-0.44)
```

```
# trestbps (resting blood pressure) - Second strongest positive correlation (0.
↪42)
# ca (number of major vessels colored) - Second strongest negative correlation␣
↪(-0.38)
# thal (thalassemia) - Third strongest negative correlation (-0.34)
# slope (the slope of the peak exercise ST segment) - Third strongest negative␣
↪correlation (0.34)
```

[ ]:
```
# Check number of unique values for each categorical feature
categorical_columns = []
for column in dataset.columns:
    if dataset[column].dtype == 'object' or dataset[column].nunique() < 10:
        categorical_columns.append(column)

print("Categorical columns:", categorical_columns)
for feature in categorical_columns:
    unique_values = dataset[feature].nunique()
    print(f"Feature '{feature}' has {unique_values} unique values:␣
↪{dataset[feature].unique()}")
```

```
Categorical columns: ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca',
'thal', 'target']
Feature 'sex' has 2 unique values: [1 0]
Feature 'cp' has 4 unique values: [0 1 2 3]
Feature 'fbs' has 2 unique values: [0 1]
Feature 'restecg' has 3 unique values: [1 0 2]
Feature 'exang' has 2 unique values: [0 1]
Feature 'slope' has 3 unique values: [2 0 1]
Feature 'ca' has 5 unique values: [2 0 1 3 4]
Feature 'thal' has 4 unique values: [3 2 1 0]
Feature 'target' has 2 unique values: [0 1]
```

[ ]:
```
X = dataset.drop('target',axis=1)
y = dataset["target"]

# data splitting
from sklearn.model_selection import train_test_split

# Assuming X and y are your feature matrix and target vector
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.
↪20, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
↪test_size=0.25, random_state=42)

# Print the dimensions of each set
print("Training set:", X_train.shape, y_train.shape)
print("Validation set:", X_val.shape, y_val.shape)
```

```
print("Test set:", X_test.shape, y_test.shape)
```

```
Training set: (615, 13) (615,)
Validation set: (205, 13) (205,)
Test set: (205, 13) (205,)
```

[ ]:
```
#Data Preprocessing
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

[ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪confusion_matrix, roc_auc_score, roc_curve, auc
```

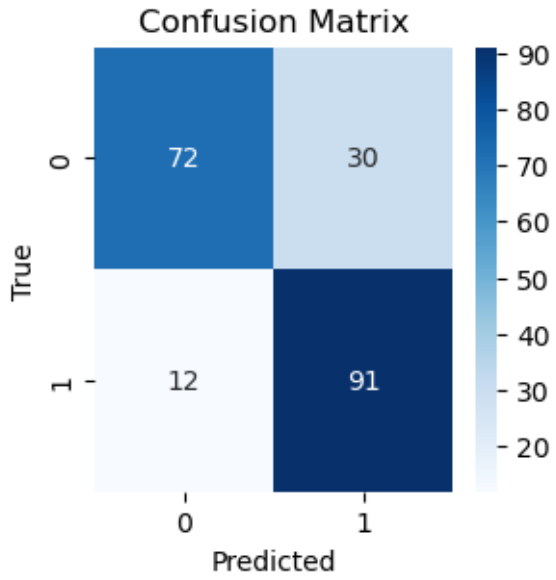
[ ]:
```
#Logistic Regression

from sklearn.linear_model import LogisticRegression
logistic_regression = LogisticRegression(max_iter=10000)

loss_values = []
# for i in range(1, 5001, 50):  # Assuming 100 iterations
#     loss = -np.sum(np.log(logistic_regression.predict_proba(X_train)[np.
↪arange(len(X_train)), y_train])) / len(X_train)
#     print(f"epoch {i}, loss {loss}")
#     loss_values.append(loss)

logistic_regression.fit(X_train, y_train)
```

[ ]:
```
LogisticRegression(max_iter=10000)
```

[ ]:
```
# Logistic Regression: Make predictions on the test set
y_pred_lr = logistic_regression.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred_lr)
precision = precision_score(y_test, y_pred_lr)
recall = recall_score(y_test, y_pred_lr)
conf_matrix = confusion_matrix(y_test, y_pred_lr)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_lr)

# Print evaluation metrics
print("Accuracy:", accuracy)
```

```
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)
print(conf_matrix)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.7951219512195122
Precision: 0.7520661157024794
Recall: 0.883495145631068
Specificity: 0.7058823529411765
F1 Score: 0.8125000000000001
ROC AUC Score: 0.7946887492861221
[[72 30]
 [12 91]]
```



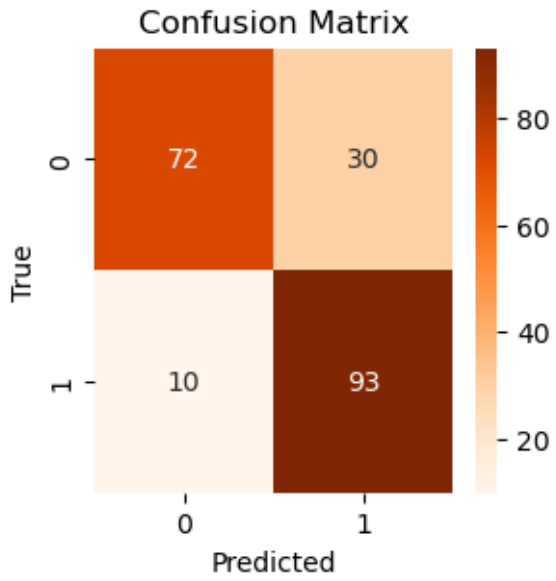
[ ]:
```
#Naive Bayes

from sklearn.naive_bayes import GaussianNB
```

```
naive_bayes_classifier = GaussianNB()

naive_bayes_classifier.fit(X_train, y_train)
```

[ ]:
```
GaussianNB()
```

[ ]:
```
# Naive Bayes: Make predictions on the test set
y_pred_nb = naive_bayes_classifier.predict(X_test)

# Calculate evaluation metrics
accuracy_nb = accuracy_score(y_test, y_pred_nb)
precision_nb = precision_score(y_test, y_pred_nb)
recall_nb = recall_score(y_test, y_pred_nb)
conf_matrix_nb = confusion_matrix(y_test, y_pred_nb)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_nb = tn / (tn + fp)
f1_score_nb = 2 * (precision_nb * recall_nb) / (precision_nb + recall_nb)
roc_auc_nb = roc_auc_score(y_test, y_pred_nb)

# Print evaluation metrics
print("Accuracy:", accuracy_nb)
print("Precision:", precision_nb)
print("Recall:", recall_nb)
print("Specificity:", specificity_nb)
print("F1 Score:", f1_score_nb)
print("ROC AUC Score:", roc_auc_nb)

print(conf_matrix_nb)
plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_nb, annot=True, cmap='Oranges', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.8048780487804879
Precision: 0.7560975609756098
Recall: 0.9029126213592233
Specificity: 0.7058823529411765
F1 Score: 0.8230088495575221
ROC AUC Score: 0.8043974871501998
[[72 30]
 [10 93]]
```



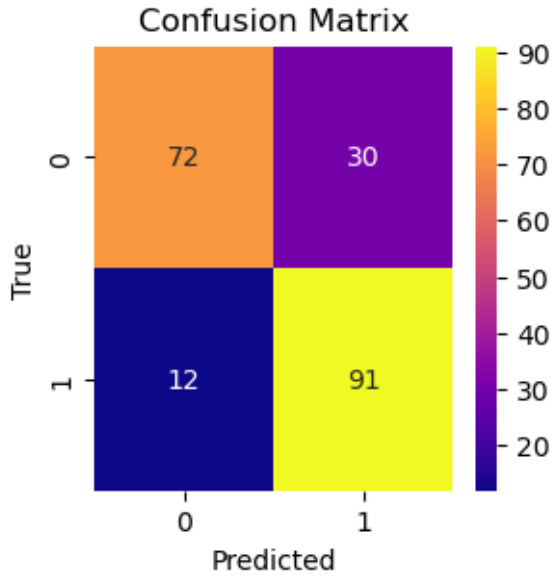
[ ]:
```
#Random Forest

from sklearn.ensemble import RandomForestClassifier

random_forest_classifier = RandomForestClassifier()

random_forest_classifier.fit(X_train, y_train)
```

[ ]:
```
RandomForestClassifier()
```

[ ]:
```
# Random Forest: Make predictions on the test set
y_pred_rf = random_forest_classifier.predict(X_test)

# Calculate evaluation metrics
accuracy_rf = accuracy_score(y_test, y_pred_rf)
precision_rf = precision_score(y_test, y_pred_rf)
recall_rf = recall_score(y_test, y_pred_rf)
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)
specificity_rf = tn / (tn + fp)
f1_score_rf = 2 * (precision_rf * recall_rf) / (precision_rf + recall_rf)
roc_auc_rf = roc_auc_score(y_test, y_pred_rf)

# Print evaluation metrics
print("Accuracy:", accuracy_rf)
print("Precision:", precision_rf)
print("Recall:", recall_rf)
print("Specificity:", specificity_rf)
```

```
print("F1 Score:", f1_score_rf)
print("ROC AUC Score:", roc_auc_rf)

print('Confusion Matrix', conf_matrix_rf)
plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix, annot=True, cmap='plasma', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.975609756097561
Precision: 0.9803921568627451
Recall: 0.970873786407767
Specificity: 0.7058823529411765
F1 Score: 0.975609756097561
ROC AUC Score: 0.975632971635256
Confusion Matrix [[100   2]
 [  3 100]]
```



[ ]:
```
#K Nearest Neighbors

from sklearn.neighbors import KNeighborsClassifier

knn_classifier = KNeighborsClassifier()

knn_classifier.fit(X_train, y_train)
```

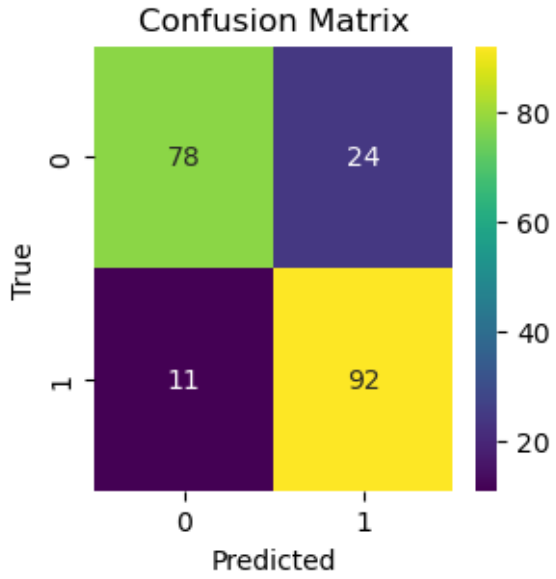
```
knn_classifier.get_params()
```

[ ]:
```
{'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'metric_params': None,
 'n_jobs': None,
 'n_neighbors': 5,
 'p': 2,
 'weights': 'uniform'}
```

[ ]:
```
# K Nearest Neighbors Make predictions on the test set
y_pred_knn = knn_classifier.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred_knn)
precision = precision_score(y_test, y_pred_knn)
recall = recall_score(y_test, y_pred_knn)
conf_matrix = confusion_matrix(y_test, y_pred_knn)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
sensitivity = tp / (tp + fn)
npv = tn / (tn + fn)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_knn)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("ROC AUC Score:", roc_auc)


print(conf_matrix)
plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix, annot=True, cmap='viridis', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
# Plot ROC curve
# fpr, tpr, thresholds = roc_curve(y_test, y_pred_knn)
# roc_auc = auc(fpr, tpr)

# plt.figure()
```

```
# plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)'
 ↪% roc_auc)
# plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
# plt.xlim([0.0, 1.0])
# plt.ylim([0.0, 1.05])
# plt.xlabel('False Positive Rate')
# plt.ylabel('True Positive Rate')
# plt.title('Receiver Operating Characteristic (ROC) Curve')
# plt.legend(loc="lower right")
# plt.show()
```

```
Accuracy: 0.8292682926829268
Precision: 0.7931034482758621
Recall: 0.8932038834951457
Specificity: 0.7647058823529411
ROC AUC Score: 0.8289548829240434
[[78 24]
 [11 92]]
```

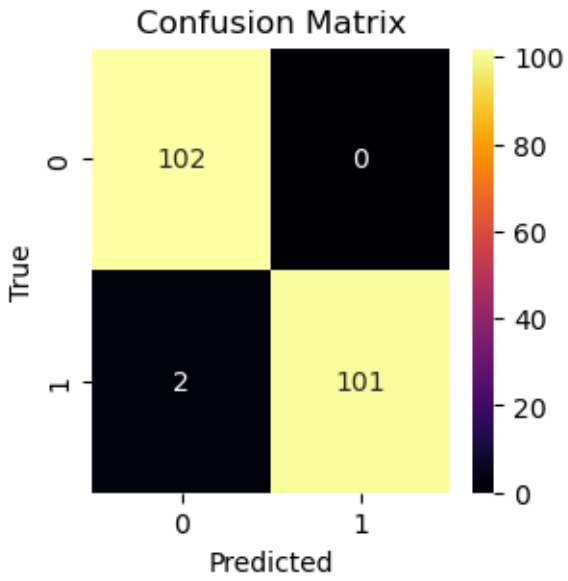


[ ]:
```
#Decision Tree

from sklearn.tree import DecisionTreeClassifier

decision_tree_classifier = DecisionTreeClassifier()

decision_tree_classifier.fit(X_train, y_train)
```

[ ]:
```
DecisionTreeClassifier()
```

[ ]:
```
# #Decision Tree Make predictions on the test set
y_pred_dt = decision_tree_classifier.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred_dt)
precision = precision_score(y_test, y_pred_dt)
recall = recall_score(y_test, y_pred_dt)
conf_matrix = confusion_matrix(y_test, y_pred_dt)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
sensitivity = tp / (tp + fn)
npv = tn / (tn + fn)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_dt)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("ROC AUC Score:", roc_auc)

print(conf_matrix)
plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix, annot=True, cmap='inferno', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.9902439024390244
Precision: 1.0
Recall: 0.9805825242718447
Specificity: 1.0
ROC AUC Score: 0.9902912621359223
[[102   0]
 [  2 101]]
```



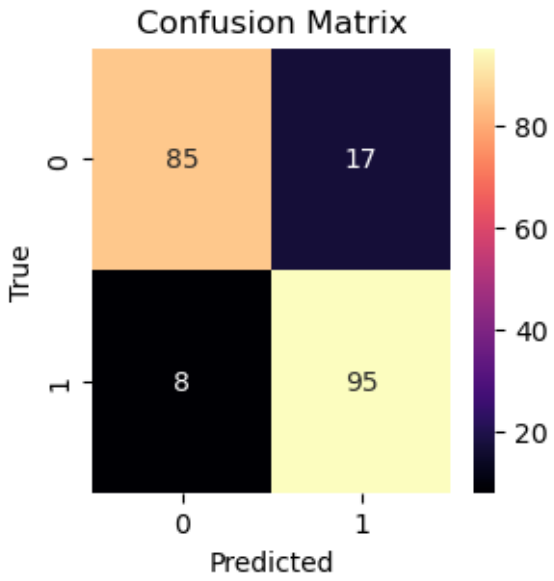
[ ]:
```
# Support Vector Machines

from sklearn.svm import SVC

svc_classifier = SVC()

svc_classifier.fit(X_train, y_train)
svc_classifier.get_params()
```

[ ]:
```
{'C': 1.0,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': 'ovr',
 'degree': 3,
 'gamma': 'scale',
 'kernel': 'rbf',
 'max_iter': -1,
 'probability': False,
 'random_state': None,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

[ ]:
```
# Support Vector Machines Make predictions on the test set
y_pred_svc = svc_classifier.predict(X_test)
```

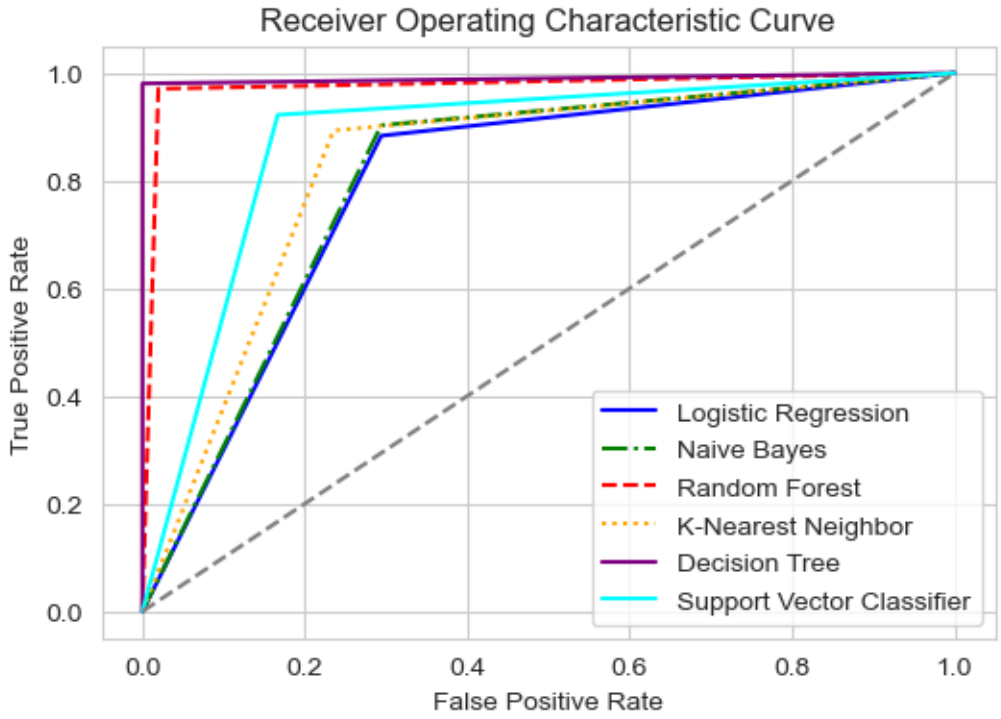
```
# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred_svc)
precision = precision_score(y_test, y_pred_svc)
recall = recall_score(y_test, y_pred_svc)
conf_matrix = confusion_matrix(y_test, y_pred_svc)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_svc)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)
print(conf_matrix)
plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix, annot=True, cmap='magma', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.8780487804878049
Precision: 0.8482142857142857
Recall: 0.9223300970873787
Specificity: 0.8333333333333334
F1 Score: 0.8837209302325582
ROC AUC Score: 0.8778317152103561
[[85 17]
 [ 8 95]]
```



[ ]:
```
lr_false_positive_rate,lr_true_positive_rate,lr_threshold =␣
 ↪roc_curve(y_test,y_pred_lr)
nb_false_positive_rate,nb_true_positive_rate,nb_threshold =␣
 ↪roc_curve(y_test,y_pred_nb)
rf_false_positive_rate,rf_true_positive_rate,rf_threshold =␣
 ↪roc_curve(y_test,y_pred_rf)
knn_false_positive_rate,knn_true_positive_rate,knn_threshold =␣
 ↪roc_curve(y_test,y_pred_knn)
dt_false_positive_rate,dt_true_positive_rate,dt_threshold =␣
 ↪roc_curve(y_test,y_pred_dt)
svc_false_positive_rate,svc_true_positive_rate,svc_threshold =␣
 ↪roc_curve(y_test,y_pred_svc)
sns.set_style('whitegrid')
plt.figure(figsize=(6,4))
plt.title('Receiver Operating Characteristic Curve')

# Plot ROC curves for each classifier with different line styles and colors
plt.plot(lr_false_positive_rate, lr_true_positive_rate, label='Logistic␣
 ↪Regression', color='blue', linestyle='-')
plt.plot(nb_false_positive_rate, nb_true_positive_rate, label='Naive Bayes',␣
 ↪color='green', linestyle='-.')
plt.plot(rf_false_positive_rate, rf_true_positive_rate, label='Random Forest',␣
 ↪color='red', linestyle='--')
plt.plot(knn_false_positive_rate, knn_true_positive_rate, label='K-Nearest␣
 ↪Neighbor', color='orange', linestyle=':')
```

```
plt.plot(dt_false_positive_rate, dt_true_positive_rate, label='Decision Tree',␣
 ↪color='purple', linestyle='-')
plt.plot(svc_false_positive_rate, svc_true_positive_rate, label='Support Vector␣
 ↪Classifier', color='cyan', linestyle='-')

# Plot the diagonal reference line
plt.plot([0, 1], [0, 1], ls='--', color='.5')

# Add labels and legend
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()

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



[ ]:
```
# Let's do Hyperparameter tuning!!
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import StackingClassifier
```

[ ]:
```
# Define hyperparameters for logistic regression
param_lr = {
    'C': [0.001, 0.01, 0.1, 1, 10],
```

```
    'solver': ['lbfgs', 'sag', 'saga'],
    }

# Create grid search using 5-fold cross validation
grid_search_lr = GridSearchCV(LogisticRegression(max_iter=20000), param_lr,␣
 ↪cv=5, scoring='accuracy')

# Perform grid search
grid_search_lr.fit(X_train, y_train)


# Print best parameters
print("Best parameters for Logistic Regression with validation set (Grid␣
 ↪Search):", grid_search_lr.best_params_)
```

```
Best parameters for Logistic Regression with validation set (Grid Search): {'C':
1, 'solver': 'lbfgs'}
```

[ ]:
```
#Logistic Regression: Testing

logistic_regression_best = LogisticRegression(C=0.1, max_iter=10000)
logistic_regression_best.fit(X_train, y_train)

# Before hyperparameter tuning
y_pred_lr_before = logistic_regression.predict(X_test)

accuracy = accuracy_score(y_test, y_pred_lr_before)
precision = precision_score(y_test, y_pred_lr_before)
recall = recall_score(y_test, y_pred_lr_before)
conf_matrix = confusion_matrix(y_test, y_pred_lr_before)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_lr_before)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)


logistic_tuned_model = grid_search_lr.best_estimator_
y_pred_lr_best = logistic_tuned_model.predict(X_val)
```

```
# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_val, y_pred_lr_best)
precision_after = precision_score(y_val, y_pred_lr_best)
recall_after = recall_score(y_val, y_pred_lr_best)
conf_matrix = confusion_matrix(y_val, y_pred_lr_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_val, y_pred_lr_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (Val Set):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Accuracy: 0.7951219512195122
Precision: 0.7520661157024794
Recall: 0.883495145631068
Specificity: 0.7058823529411765
F1 Score: 0.8125000000000001
ROC AUC Score: 0.7946887492861221

Evaluation Metrics after Hyperparameter Tuning (Val Set):
Accuracy: 0.9073170731707317
Precision: 0.8728813559322034
Recall: 0.9626168224299065
Specificity: 0.8469387755102041
F1 Score: 0.8125000000000001
ROC AUC Score: 0.9047777989700553
```

[ ]:
```
# Make predictions on the test set using the tuned logistic regression model
logistic_tuned_model = grid_search_lr.best_estimator_
y_pred_lr_test = logistic_tuned_model.predict(X_test)

# Calculate evaluation metrics for the tuned model on the test set
accuracy_test = accuracy_score(y_test, y_pred_lr_test)
precision_test = precision_score(y_test, y_pred_lr_test)
recall_test = recall_score(y_test, y_pred_lr_test)
conf_matrix_test = confusion_matrix(y_test, y_pred_lr_test)
tn_test, fp_test, fn_test, tp_test = conf_matrix_test.ravel()
specificity_test = tn_test / (tn_test + fp_test)
sensitivity_test = tp_test / (tp_test + fn_test)
npv_test = tn_test / (tn_test + fn_test)
```

```
f1_score_test = 2 * (precision_test * recall_test) / (precision_test +␣
 ↪recall_test)
roc_auc_test = roc_auc_score(y_test, y_pred_lr_test)

# Print the evaluation metrics for the tuned model on the test set
print("\nEvaluation Metrics for Tuned Logistic Regression Model on Test Set:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("Specificity:", specificity_test)
print("Sensitivity:", sensitivity_test)
print("NPV:", npv_test)
print("F1 Score:", f1_score_test)
print("ROC AUC Score:", roc_auc_test)
```

```
Evaluation Metrics for Tuned Logistic Regression Model on Test Set:
Accuracy: 0.7951219512195122
Precision: 0.7520661157024794
Recall: 0.883495145631068
Specificity: 0.7058823529411765
Sensitivity: 0.883495145631068
NPV: 0.8571428571428571
F1 Score: 0.8125000000000001
ROC AUC Score: 0.7946887492861221
```

[ ]:
```
# Random Forest
rf_param_grid = {
   'n_estimators': [50, 100],
    'max_depth': [None, 10],
    'min_samples_split': [2, 5, 7, 10],
    'min_samples_leaf': [1, 2],
    'max_features': ['auto', 'sqrt']


}

grid_search_rf = GridSearchCV(RandomForestClassifier(), rf_param_grid, cv=5,␣
 ↪scoring='accuracy')

grid_search_rf.fit(X_train, y_train)

best_params_grid_rf = grid_search_rf.best_params_
best_estimator_grid_rf = grid_search_rf.best_estimator_

print("Results for Random Forest - GridSearchCV:")
print("Best parameters:", best_params_grid_rf)
```

```
print("Best estimator:", best_estimator_grid_rf)
```

```
D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_validation.py:425: FitFailedWarning:
160 fits failed out of a total of 320.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
160 fits failed with the following error:
Traceback (most recent call last):
  File "D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_validation.py", line 732, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "D:\Anaconda\envs\workingenv\lib\site-packages\sklearn\base.py", line
1144, in wrapper
    estimator._validate_params()
  File "D:\Anaconda\envs\workingenv\lib\site-packages\sklearn\base.py", line
637, in _validate_params
    validate_parameter_constraints(
  File "D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\utils\_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
parameter of RandomForestClassifier must be an int in the range [1, inf), a
float in the range (0.0, 1.0], a str among {'log2', 'sqrt'} or None. Got 'auto'
instead.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_search.py:976: UserWarning: One or more of the
test scores are non-finite: [       nan        nan        nan        nan
nan        nan
        nan        nan        nan        nan        nan        nan
        nan        nan        nan        nan 0.97073171 0.96910569
 0.95284553 0.95772358 0.9495935  0.94471545 0.92357724 0.9398374
 0.9495935  0.9495935  0.95121951 0.94471545 0.93658537 0.93821138
 0.92682927 0.92357724        nan        nan        nan        nan
        nan        nan        nan        nan        nan        nan
        nan        nan        nan        nan        nan        nan
 0.95772358 0.97073171 0.95121951 0.95609756 0.94634146 0.94146341
 0.93821138 0.94308943 0.93821138 0.95121951 0.94308943 0.94634146
 0.93821138 0.93658537 0.92682927 0.9203252 ]
  warnings.warn(
```

```
Results for Random Forest - GridSearchCV:
Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 50}
Best estimator: RandomForestClassifier(n_estimators=50)
```

[ ]:
```
#Random Forest: Testing

random_forest_best = RandomForestClassifier()
random_forest_best.fit(X_train, y_train)

# Before hyperparameter tuning
y_pred_rf_before = random_forest_best.predict(X_test)

accuracy = accuracy_score(y_test, y_pred_rf_before)
precision = precision_score(y_test, y_pred_rf_before)
recall = recall_score(y_test, y_pred_rf_before)
conf_matrix = confusion_matrix(y_test, y_pred_rf_before)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_rf_before)

# Print evaluation metrics
print("Before HyperParameters Tuning")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)


random_forest_tuned_model = grid_search_rf.best_estimator_
y_pred_rf_best = random_forest_tuned_model.predict(X_val)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_val, y_pred_rf_best)
precision_after = precision_score(y_val, y_pred_rf_best)
recall_after = recall_score(y_val, y_pred_rf_best)
conf_matrix = confusion_matrix(y_val, y_pred_rf_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_val, y_pred_rf_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (Val Set):")
```

```
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Before HyperParameters Tuning
Accuracy: 0.975609756097561
Precision: 0.9803921568627451
Recall: 0.970873786407767
Specificity: 0.9803921568627451
F1 Score: 0.975609756097561
ROC AUC Score: 0.975632971635256

Evaluation Metrics after Hyperparameter Tuning (Val Set):
Accuracy: 0.9951219512195122
Precision: 0.9907407407407407
Recall: 1.0
Specificity: 0.9897959183673469
F1 Score: 0.975609756097561
ROC AUC Score: 0.9948979591836734
```

[ ]:
```
# Make predictions on the test set using the tuned random forest model
random_forest_tuned_model = grid_search_rf.best_estimator_
y_pred_rf_test = random_forest_tuned_model.predict(X_test)

# Calculate evaluation metrics for the tuned model on the test set
accuracy_test = accuracy_score(y_test, y_pred_rf_test)
precision_test = precision_score(y_test, y_pred_rf_test)
recall_test = recall_score(y_test, y_pred_rf_test)
conf_matrix_test = confusion_matrix(y_test, y_pred_rf_test)
tn_test, fp_test, fn_test, tp_test = conf_matrix_test.ravel()
specificity_test = tn_test / (tn_test + fp_test)
sensitivity_test = tp_test / (tp_test + fn_test)
npv_test = tn_test / (tn_test + fn_test)
f1_score_test = 2 * (precision_test * recall_test) / (precision_test +␣
 ↪recall_test)
roc_auc_test = roc_auc_score(y_test, y_pred_rf_test)

# Print the evaluation metrics for the tuned model on the test set
print("\nEvaluation Metrics for Tuned Logistic Regression Model on Test Set:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("Specificity:", specificity_test)
print("Sensitivity:", sensitivity_test)
```

```
print("NPV:", npv_test)
print("F1 Score:", f1_score_test)
print("ROC AUC Score:", roc_auc_test)
```

```
Evaluation Metrics for Tuned Logistic Regression Model on Test Set:
Accuracy: 0.975609756097561
Precision: 0.9803921568627451
Recall: 0.970873786407767
Specificity: 0.9803921568627451
Sensitivity: 0.970873786407767
NPV: 0.970873786407767
F1 Score: 0.975609756097561
ROC AUC Score: 0.975632971635256
```

[ ]:
```
# K-Nearest Neighbors (KNN)
param_grid_knn = {
    'n_neighbors': range(5, 50),
    'weights': ['uniform', 'distance'],
    'algorithm': ['ball_tree', 'kd_tree', 'brute'],
    'metric' : ['minkowski','euclidean','manhattan']
}

grid_search_knn = GridSearchCV(KNeighborsClassifier(), param_grid_knn, cv=5,␣
 ↪scoring='accuracy')

grid_search_knn.fit(X_train, y_train)

best_params_grid_knn = grid_search_knn.best_params_
best_estimator_grid_knn = grid_search_knn.best_estimator_

print("Results for K Nearest Neighbors - GridSearchCV:")
print("Best parameters:", best_params_grid_knn)
print("Best estimator:", best_estimator_grid_knn)
```

```
Results for K Nearest Neighbors - GridSearchCV:
Best parameters: {'algorithm': 'ball_tree', 'metric': 'manhattan',
'n_neighbors': 44, 'weights': 'distance'}
Best estimator: KNeighborsClassifier(algorithm='ball_tree', metric='manhattan',
n_neighbors=44,
                     weights='distance')
```

[ ]:
```
#K Nearest Neighbors: Testing

# Before hyperparameter tuning
y_pred_knn_before = knn_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred_knn_before)
```

```
precision = precision_score(y_test, y_pred_knn_before)
recall = recall_score(y_test, y_pred_knn_before)
conf_matrix = confusion_matrix(y_test, y_pred_knn_before)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_knn_before)

# Print evaluation metrics
print("Before Hyper Parameters Tuning")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)

# After hyperparameter tuning
# Make predictions on the test set using tuned model

knn_tuned_model = grid_search_knn.best_estimator_
y_pred_knn_after_gridSearch = knn_tuned_model.predict(X_val)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_val, y_pred_knn_after_gridSearch)
precision_after = precision_score(y_val, y_pred_knn_after_gridSearch)
recall_after = recall_score(y_val, y_pred_knn_after_gridSearch)
conf_matrix = confusion_matrix(y_val, y_pred_knn_after_gridSearch)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_val, y_pred_knn_after_gridSearch)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Before Hyper Parameters Tuning
Accuracy: 0.8292682926829268
Precision: 0.7931034482758621
Recall: 0.8932038834951457
Specificity: 0.7647058823529411
F1 Score: 0.8401826484018265
```

```
ROC AUC Score: 0.8289548829240434

Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.9804878048780488
Precision: 0.9904761904761905
Recall: 0.9719626168224299
Specificity: 0.9897959183673469
F1 Score: 0.8401826484018265
ROC AUC Score: 0.9808792675948884
```

[ ]:
```python
# After hyperparameter tuning
# Make predictions on the test set using tuned model

knn_tuned_model_test = grid_search_knn.best_estimator_
y_pred_knn_after_gridSearch_test = knn_tuned_model_test.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_knn_after_gridSearch_test)
precision_after = precision_score(y_test, y_pred_knn_after_gridSearch_test)
recall_after = recall_score(y_test, y_pred_knn_after_gridSearch_test)
conf_matrix = confusion_matrix(y_test, y_pred_knn_after_gridSearch_test)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_knn_after_gridSearch_test)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("Sensitivity:", sensitivity_after)
print("NPV:", npv_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.9902439024390244
Precision: 0.9809523809523809
Recall: 1.0
Specificity: 0.9803921568627451
Sensitivity: 1.0
NPV: 1.0
F1 Score: 0.8401826484018265
```

```
ROC AUC Score: 0.9901960784313725
```

[ ]:
```python
# Decision Trees:
# Define the parameter grid

param_grid = {
    'max_depth': [2, 10, 20, 30],
    'min_samples_split': [2, 4, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}
# Initialize the decision tree classifier
decision_tree = DecisionTreeClassifier(random_state=42)

# Initialize GridSearchCV
grid_search_decision_tree = GridSearchCV(estimator=decision_tree,␣
 ↪param_grid=param_grid, cv=5, scoring='accuracy')

# Perform hyperparameter tuning
grid_search_decision_tree.fit(X_train, y_train)

# Best parameters found during grid search
best_params = grid_search_decision_tree.best_params_
print("Best Parameters:", best_params)
```

```
Best Parameters: {'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf':
1, 'min_samples_split': 2}

D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_validation.py:425: FitFailedWarning:
180 fits failed out of a total of 540.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
180 fits failed with the following error:
Traceback (most recent call last):
  File "D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_validation.py", line 732, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "D:\Anaconda\envs\workingenv\lib\site-packages\sklearn\base.py", line
1144, in wrapper
    estimator._validate_params()
  File "D:\Anaconda\envs\workingenv\lib\site-packages\sklearn\base.py", line
637, in _validate_params
```

```
    validate_parameter_constraints(
  File "D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\utils\_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
parameter of DecisionTreeClassifier must be an int in the range [1, inf), a
float in the range (0.0, 1.0], a str among {'log2', 'sqrt'} or None. Got 'auto'
instead.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
D:\Anaconda\envs\workingenv\lib\site-
packages\sklearn\model_selection\_search.py:976: UserWarning: One or more of the
test scores are non-finite: [       nan        nan        nan        nan
nan        nan
        nan        nan        nan 0.68292683 0.68292683 0.68292683
 0.68292683 0.68292683 0.68292683 0.68292683 0.68292683 0.68292683
 0.68292683 0.68292683 0.68292683 0.68292683 0.68292683 0.68292683
 0.68292683 0.68292683 0.68292683        nan        nan        nan
        nan        nan        nan        nan        nan        nan
 0.93821138 0.91869919 0.86829268 0.89593496 0.89593496 0.85528455
 0.89105691 0.89105691 0.8601626  0.93821138 0.91869919 0.86829268
 0.89593496 0.89593496 0.85528455 0.89105691 0.89105691 0.8601626
        nan        nan        nan        nan        nan        nan
        nan        nan        nan 0.95121951 0.91869919 0.86829268
 0.90731707 0.90731707 0.85528455 0.89105691 0.89105691 0.8601626
 0.95121951 0.91869919 0.86829268 0.90731707 0.90731707 0.85528455
 0.89105691 0.89105691 0.8601626         nan        nan        nan
        nan        nan        nan        nan        nan        nan
 0.95121951 0.91869919 0.86829268 0.90731707 0.90731707 0.85528455
 0.89105691 0.89105691 0.8601626  0.95121951 0.91869919 0.86829268
 0.90731707 0.90731707 0.85528455 0.89105691 0.89105691 0.8601626 ]
  warnings.warn(
```

[ ]:
```
#Decision Trees: Testing

# Before hyperparameter tuning
y_pred_dt_before = decision_tree_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred_dt_before)
precision = precision_score(y_test, y_pred_dt_before)
recall = recall_score(y_test, y_pred_dt_before)
conf_matrix = confusion_matrix(y_test, y_pred_dt_before)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
sensitivity = tp / (tp + fn)
npv = tn / (tn + fn)
```

```
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_dt_before)

# Print evaluation metrics
print("Before Hyper parameters tuning")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)

# After hyperparameter tuning
# Make predictions on the test set using tuned model

decision_tree_tuned_model = grid_search_decision_tree.best_estimator_
y_pred_dt_after_gridSearch = decision_tree_tuned_model.predict(X_val)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_val, y_pred_dt_after_gridSearch)
precision_after = precision_score(y_val, y_pred_dt_after_gridSearch)
recall_after = recall_score(y_val, y_pred_dt_after_gridSearch)
conf_matrix = confusion_matrix(y_val, y_pred_dt_after_gridSearch)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_val, y_pred_dt_after_gridSearch)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Before Hyper parameters tuning
Accuracy: 0.9707317073170731
Precision: 0.9619047619047619
Recall: 0.9805825242718447
Specificity: 0.9607843137254902
F1 Score: 0.9711538461538461
ROC AUC Score: 0.9706834189986675

Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.9902439024390244
Precision: 0.981651376146789
```

```
Recall: 1.0
Specificity: 0.9795918367346939
F1 Score: 0.9711538461538461
ROC AUC Score: 0.9897959183673469
```

[ ]:
```python
# After hyperparameter tuning
# Make predictions on the test set using tuned model

decision_tree_tuned_model = grid_search_decision_tree.best_estimator_
y_pred_dt_after_gridSearch_test = decision_tree_tuned_model.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_dt_after_gridSearch_test)
precision_after = precision_score(y_test, y_pred_dt_after_gridSearch_test)
recall_after = recall_score(y_test, y_pred_dt_after_gridSearch_test)
conf_matrix = confusion_matrix(y_test, y_pred_dt_after_gridSearch_test)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision_after * recall_after) / (precision_after +␣
 ↪recall_after)
roc_auc = roc_auc_score(y_test, y_pred_dt_after_gridSearch_test)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("Sensitivity:", sensitivity_after)
print("NPV:", npv_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.9609756097560975
Precision: 0.9523809523809523
Recall: 0.970873786407767
Specificity: 0.9509803921568627
Sensitivity: 0.970873786407767
NPV: 0.97
F1 Score: 0.9615384615384616
ROC AUC Score: 0.9609270892823149
```

[ ]:
```python
# Support Vector Machine (SVM)
param_grid_svc = {
```

```
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'gamma': ['scale', 'auto']
}

grid_search_svc = GridSearchCV(SVC(), param_grid_svc, cv=5, scoring='accuracy')
# random_search_svc = RandomizedSearchCV(SVC(), param_grid_svc, cv=5,␣
 ↪n_iter=100, scoring='accuracy')

grid_search_svc.fit(X_train, y_train)
# random_search_svc.fit(X_val, y_val)

best_params_grid_svc = grid_search_svc.best_params_
best_estimator_grid_svc = grid_search_svc.best_estimator_

# best_params_random_svc = random_search_svc.best_params_
# best_estimator_random_svc = random_search_svc.best_estimator_

print("Results for Support Vector Machines - GridSearchCV:")
print("Best parameters:", best_params_grid_svc)
print("Best estimator:", best_estimator_grid_svc)

# print("\nResults for Support Vector Machines - RandomizedSearchCV:")
# print("Best parameters:", best_params_random_svc)
# print("Best estimator:", best_estimator_random_svc)
```

```
Results for Support Vector Machines - GridSearchCV:
Best parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
Best estimator: SVC(C=10)
```

[ ]:
```
#Support Vector Machines: Testing

# Before hyperparameter tuning
y_pred_svc_before = svc_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred_svc_before)
precision = precision_score(y_test, y_pred_svc_before)
recall = recall_score(y_test, y_pred_svc_before)
conf_matrix = confusion_matrix(y_test, y_pred_svc_before)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_svc_before)

# Print evaluation metrics
print('Before Hyper Parameters tuning')
print("Accuracy:", accuracy)
```

```
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)


# After hyperparameter tuning
# Make predictions on the test set using tuned mode
best_svm_grid = grid_search_svc.best_estimator_
y_test_svc_after_gridSearch = best_svm_grid.predict(X_val)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_val, y_test_svc_after_gridSearch)
precision_after = precision_score(y_val, y_test_svc_after_gridSearch)
recall_after = recall_score(y_val, y_test_svc_after_gridSearch)
conf_matrix = confusion_matrix(y_val, y_test_svc_after_gridSearch)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_val, y_test_svc_after_gridSearch)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

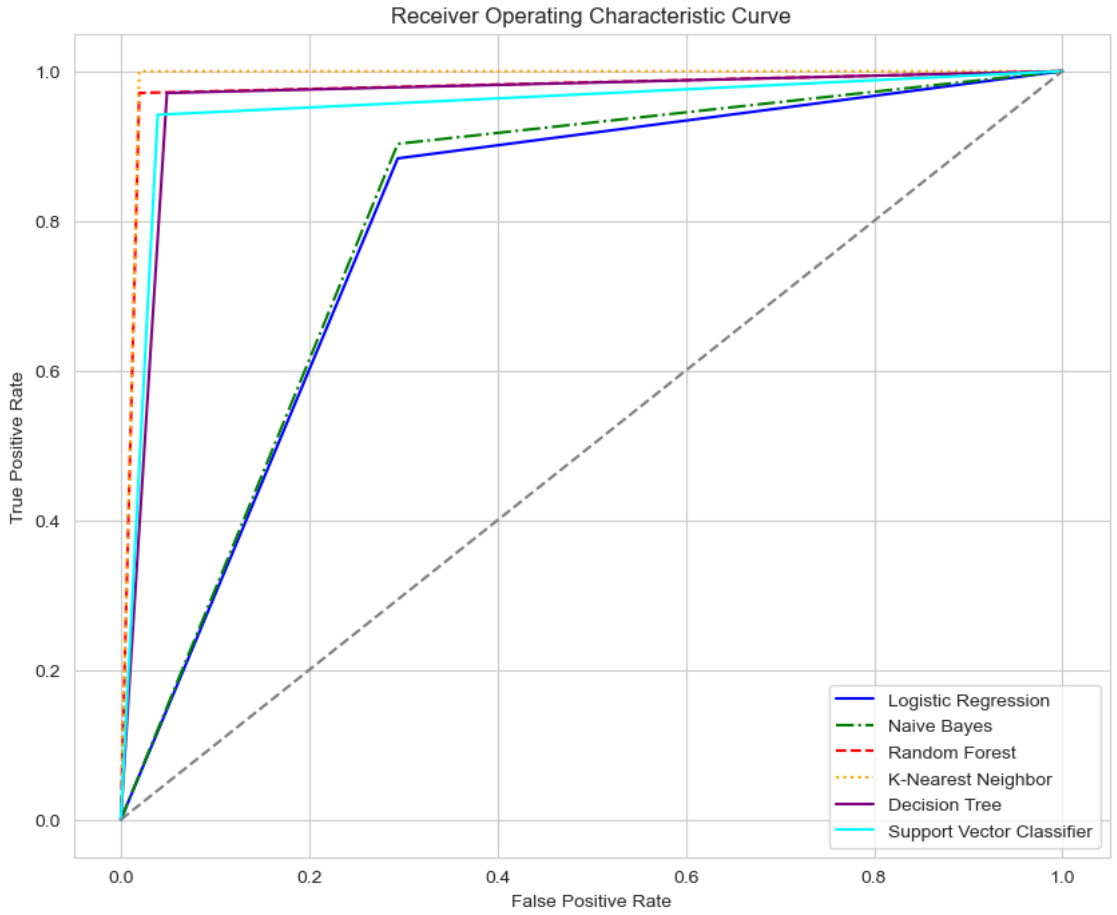
```
Before Hyper Parameters tuning
Accuracy: 0.8780487804878049
Precision: 0.8482142857142857
Recall: 0.9223300970873787
Specificity: 0.8333333333333334
F1 Score: 0.8837209302325582
ROC AUC Score: 0.8778317152103561

Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.975609756097561
Precision: 0.9811320754716981
Recall: 0.9719626168224299
Specificity: 0.9795918367346939
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9757772267785619
```

[ ]:
```
# After hyperparameter tuning
# Make predictions on the test set using tuned mode
best_svm_grid_test = grid_search_svc.best_estimator_
y_svc_after_gridSearch_test = best_svm_grid_test.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_svc_after_gridSearch_test)
precision_after = precision_score(y_test, y_svc_after_gridSearch_test)
recall_after = recall_score(y_test, y_svc_after_gridSearch_test)
conf_matrix = confusion_matrix(y_test, y_svc_after_gridSearch_test)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_svc_after_gridSearch_test)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning (GridSearch):")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("Sensitivity:", sensitivity_after)
print("NPV:", npv_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
Evaluation Metrics after Hyperparameter Tuning (GridSearch):
Accuracy: 0.9512195121951219
Precision: 0.9603960396039604
Recall: 0.941747572815534
Specificity: 0.9607843137254902
Sensitivity: 0.941747572815534
NPV: 0.9423076923076923
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9512659432705121
```

[ ]:
```
lr_false_positive_rate,lr_true_positive_rate,lr_threshold =␣
 ↪roc_curve(y_test,y_pred_lr_test)
nb_false_positive_rate,nb_true_positive_rate,nb_threshold =␣
 ↪roc_curve(y_test,y_pred_nb)
rf_false_positive_rate,rf_true_positive_rate,rf_threshold =␣
 ↪roc_curve(y_test,y_pred_rf_test)
knn_false_positive_rate,knn_true_positive_rate,knn_threshold =␣
 ↪roc_curve(y_test,y_pred_knn_after_gridSearch_test)
```

```
dt_false_positive_rate,dt_true_positive_rate,dt_threshold =␣
 ↪roc_curve(y_test,y_pred_dt_after_gridSearch_test)
svc_false_positive_rate,svc_true_positive_rate,svc_threshold =␣
 ↪roc_curve(y_test,y_svc_after_gridSearch_test)
sns.set_style('whitegrid')
plt.figure(figsize=(10,8))
plt.title('Receiver Operating Characteristic Curve')

# Plot ROC curves for each classifier with different line styles and colors
plt.plot(lr_false_positive_rate, lr_true_positive_rate, label='Logistic␣
 ↪Regression', color='blue', linestyle='-')
plt.plot(nb_false_positive_rate, nb_true_positive_rate, label='Naive Bayes',␣
 ↪color='green', linestyle='-.')
plt.plot(rf_false_positive_rate, rf_true_positive_rate, label='Random Forest',␣
 ↪color='red', linestyle='--')
plt.plot(knn_false_positive_rate, knn_true_positive_rate, label='K-Nearest␣
 ↪Neighbor', color='orange', linestyle=':')
plt.plot(dt_false_positive_rate, dt_true_positive_rate, label='Decision Tree',␣
 ↪color='purple', linestyle='-')
plt.plot(svc_false_positive_rate, svc_true_positive_rate, label='Support Vector␣
 ↪Classifier', color='cyan', linestyle='-')

# Plot the diagonal reference line
plt.plot([0, 1], [0, 1], ls='--', color='.5')

# Add labels and legend
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()

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



[ ]:
```
# LETS DO Feature Selection
from sklearn.feature_selection import RFECV
from sklearn.model_selection import StratifiedKFold
```

[ ]:
```
#Logistic Regression
# Wrapper-based feature selection with Recursive Feature Elimination (RFE) for␣
 ↪Logistic Regression

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

logistic_regression_rfecv = RFECV(estimator=logistic_regression_best, step=1,␣
 ↪cv=cv, scoring='accuracy')

logistic_regression_rfecv.fit(X_train, y_train)

print("Optimal number of features:", logistic_regression_rfecv.n_features_)
```

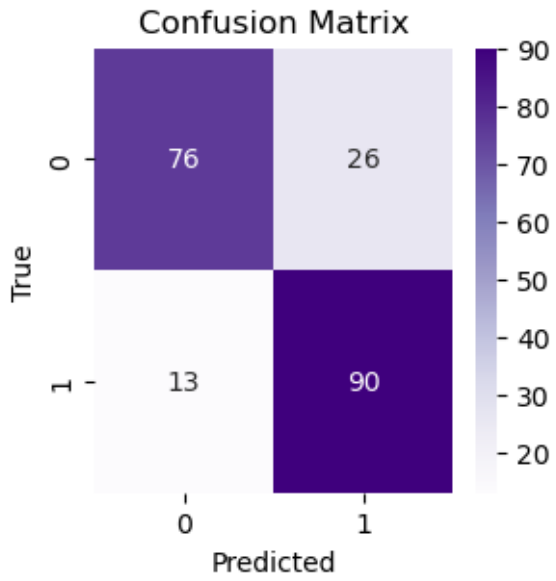
```
X_train_selected_wrapper = logistic_regression_rfecv.fit_transform(X_train,␣
 ↪y_train)

selected_features = X.columns[logistic_regression_rfecv.get_support()]

print("Selected features for Random Forest (Wrapper-Based):", selected_features)
```

```
Optimal number of features: 6
Selected features for Random Forest (Wrapper-Based): Index(['sex', 'cp',
'thalach', 'oldpeak', 'ca', 'thal'], dtype='object')
```

[ ]:
```
# After hyperparameter tuning
y_pred_lr_best = logistic_regression_best.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_lr_best)
precision_after = precision_score(y_test, y_pred_lr_best)
recall_after = recall_score(y_test, y_pred_lr_best)
conf_matrix = confusion_matrix(y_test, y_pred_lr_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_lr_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


# After feature selection
# X_test_selected_features = logistic_regression_rfecv.transform(X_test)

y_pred_lr_filtered = logistic_regression_rfecv.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_lr_filtered)
precision_after = precision_score(y_test, y_pred_lr_filtered)
recall_after = recall_score(y_test, y_pred_lr_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_lr_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
f1_score_after = 2 * (precision * recall) / (precision + recall)
```

```
roc_auc = roc_auc_score(y_test, y_pred_lr_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)

print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='Purples', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Evaluation Metrics before Feature Selection:
Accuracy: 0.7951219512195122
Precision: 0.7520661157024794
Recall: 0.883495145631068
Specificity: 0.7058823529411765
F1 Score: 0.8837209302325582
ROC AUC Score: 0.7946887492861221

Evaluation Metrics after Feature Selection:
Accuracy: 0.8097560975609757
Precision: 0.7758620689655172
Recall: 0.8737864077669902
Specificity: 0.7058823529411765
F1 Score: 0.8837209302325582
ROC AUC Score: 0.8094422234913383
[[76 26]
 [13 90]]
```



[ ]:
```
# Naive Bayes

from sklearn.feature_selection import SelectKBest, mutual_info_classif
from sklearn.preprocessing import MinMaxScaler

# Apply Min-Max scaling to ensure non-negative values
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)

naive_bayes_selector = SelectKBest(score_func=mutual_info_classif, k=8)


X_train_selected = naive_bayes_selector.fit_transform(X_train_scaled, y_train)

nb_filtered = GaussianNB()

nb_filtered.fit(X_train_selected, y_train)

selected_features_mask = naive_bayes_selector.get_support()

# Count the number of selected features
num_selected_features = np.sum(selected_features_mask)

print("Number of selected features:", num_selected_features)

selected_feature_indices = naive_bayes_selector.get_support(indices=True)
```

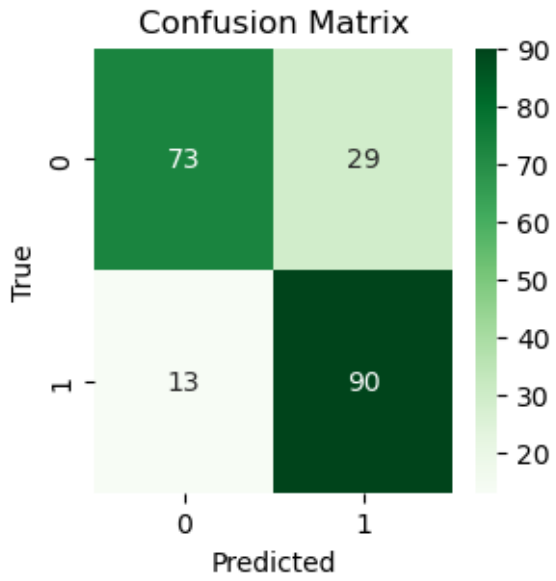
```
# Get the names of selected features
selected_features = X.columns[selected_feature_indices]

print("Selected features:", selected_features)
```

```
Number of selected features: 8
Selected features: Index(['cp', 'trestbps', 'chol', 'thalach', 'exang',
'oldpeak', 'ca', 'thal'], dtype='object')
```

[ ]:
```
# Naive Bayes Model
y_pred_nb = naive_bayes_classifier.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_nb)
precision_after = precision_score(y_test, y_pred_nb)
recall_after = recall_score(y_test, y_pred_nb)
conf_matrix = confusion_matrix(y_test, y_pred_nb)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_nb)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


# After feature selection
X_test_scaled = scaler.fit_transform(X_test)
X_test_filtered = naive_bayes_selector.transform(X_test_scaled)
y_pred_nb_filtered = nb_filtered.predict(X_test_filtered)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_nb_filtered)
precision_after = precision_score(y_test, y_pred_nb_filtered)
recall_after = recall_score(y_test, y_pred_nb_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_nb_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
```

```
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_nb_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)

print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='Greens', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Evaluation Metrics before Feature Selection:
Accuracy: 0.8048780487804879
Precision: 0.7560975609756098
Recall: 0.9029126213592233
Specificity: 0.7058823529411765
F1 Score: 0.8837209302325582
ROC AUC Score: 0.8043974871501998

Evaluation Metrics after Feature Selection:
Accuracy: 0.7951219512195122
Precision: 0.7563025210084033
Recall: 0.8737864077669902
Specificity: 0.7058823529411765
F1 Score: 0.8837209302325582
ROC AUC Score: 0.7947363411383972
[[73 29]
 [13 90]]
```



[ ]:
```
# K Nearest Neighbors

from sklearn.feature_selection import SelectKBest, mutual_info_classif

knn_selector = SelectKBest(score_func=mutual_info_classif, k=10)

X_train_selected = knn_selector.fit_transform(X_train, y_train)

# Train KNeighborsClassifier using the selected features


grid_search_knn_filtered = copy.copy(grid_search_knn)
best_params_knn = grid_search_knn_filtered.best_params_

knn_classifier_filtered = KNeighborsClassifier(**best_params_knn)
knn_classifier_filtered.fit(X_train_selected, y_train)

selected_features_mask = knn_selector.get_support()

# Count the number of selected features
num_selected_features = np.sum(selected_features_mask)

print("Number of selected features:", num_selected_features)


selected_feature_indices = knn_selector.get_support(indices=True)
```

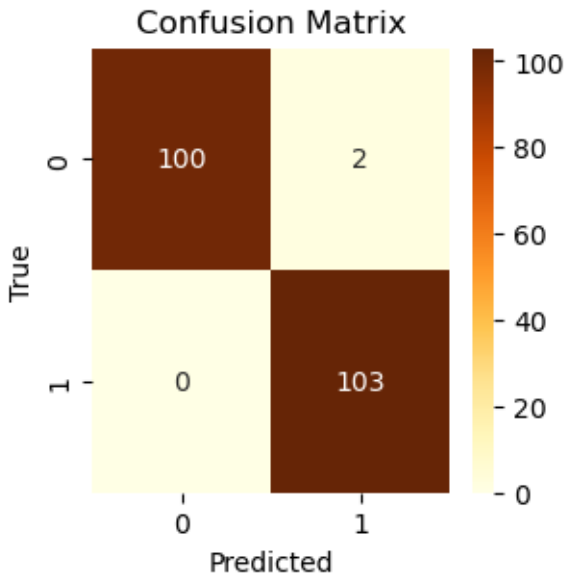
```
# Get the names of selected features
selected_features = X.columns[selected_feature_indices]

print("Selected features:", selected_features)
```

```
Number of selected features: 10
Selected features: Index(['age', 'cp', 'trestbps', 'chol', 'thalach', 'exang',
'oldpeak', 'slope',
       'ca', 'thal'],
      dtype='object')
```

[ ]:
```
# After hyperparameter tuning
y_pred_knn_after_gridSearch = knn_tuned_model.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_knn_after_gridSearch)
precision_after = precision_score(y_test, y_pred_knn_after_gridSearch)
recall_after = recall_score(y_test, y_pred_knn_after_gridSearch)
conf_matrix = confusion_matrix(y_test, y_pred_knn_after_gridSearch)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_knn_after_gridSearch)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Hyperparameter Tuning:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)

# After feature selection
X_test_selected = knn_selector.transform(X_test)
print(X_test_selected.shape)
y_pred_knn_filtered = knn_classifier_filtered.predict(X_test_selected)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_knn_filtered)
precision_after = precision_score(y_test, y_pred_knn_filtered)
recall_after = recall_score(y_test, y_pred_knn_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_knn_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
```

```
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_knn_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='YlOrBr', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Evaluation Metrics after Hyperparameter Tuning:
Accuracy: 0.9902439024390244
Precision: 0.9809523809523809
Recall: 1.0
Specificity: 0.9803921568627451
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9901960784313725
(205, 10)

Evaluation Metrics before Feature Selection:
Accuracy: 0.9902439024390244
Precision: 0.9809523809523809
Recall: 1.0
Specificity: 0.9803921568627451
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9901960784313725
[[100   2]
 [  0 103]]
```



[ ]:
```
# Random Forest: Feature Selection
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

random_forest_filtering = RandomForestClassifier()

random_forest_rfecv = RFECV(estimator=random_forest_classifier, step=1, cv=cv,␣
↪scoring='accuracy')

random_forest_rfecv.fit(X_train, y_train)

print("Optimal number of features:", random_forest_rfecv.n_features_)

X_train_selected_wrapper = random_forest_rfecv.fit_transform(X_train, y_train)

selected_features = X.columns[random_forest_rfecv.get_support()]

print("Selected features for Random Forest (Wrapper-Based):", selected_features)
```

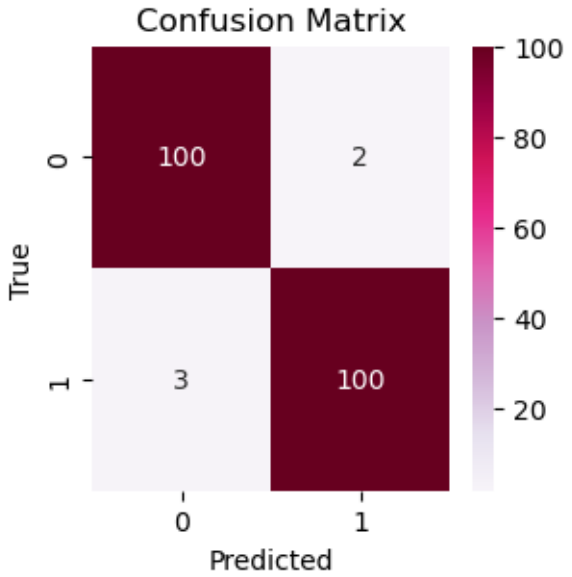
```
Optimal number of features: 8
Selected features for Random Forest (Wrapper-Based): Index(['age', 'cp',
'trestbps', 'chol', 'thalach', 'oldpeak', 'ca', 'thal'], dtype='object')
```

[ ]:
```
# After hyperparameter tuning
y_pred_random_forest_best = random_forest_classifier.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_random_forest_best)
```

```
precision_after = precision_score(y_test, y_pred_random_forest_best)
recall_after = recall_score(y_test, y_pred_random_forest_best)
conf_matrix = confusion_matrix(y_test, y_pred_random_forest_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_random_forest_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


# After feature selection
# X_test_selected_features = logistic_regression_rfecv.transform(X_test)

y_pred_random_forest_filtered = random_forest_rfecv.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_random_forest_filtered)
precision_after = precision_score(y_test, y_pred_random_forest_filtered)
recall_after = recall_score(y_test, y_pred_random_forest_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_random_forest_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision_after * recall_after) / (precision_after +␣
 ↪recall_after)
roc_auc = roc_auc_score(y_test, y_pred_random_forest_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='PuRd', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Evaluation Metrics before Feature Selection:
Accuracy: 0.975609756097561
Precision: 0.9803921568627451
Recall: 0.970873786407767
Specificity: 0.9803921568627451
F1 Score: 0.8837209302325582
ROC AUC Score: 0.975632971635256

Evaluation Metrics after Feature Selection:
Accuracy: 0.975609756097561
Precision: 0.9803921568627451
Recall: 0.970873786407767
Specificity: 0.9803921568627451
F1 Score: 0.975609756097561
ROC AUC Score: 0.975632971635256
[[100   2]
 [  3 100]]
```



[ ]:
```
# Decision Trees: Feature Selection

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

decision_trees_rfecv = RFECV(estimator=decision_tree_classifier, step=1, cv=cv,␣
 ↪scoring='accuracy')

decision_trees_rfecv.fit(X_train, y_train)

print("Optimal number of features:", decision_trees_rfecv.n_features_)

X_train_selected_wrapper = decision_trees_rfecv.fit_transform(X_train, y_train)

selected_features = X.columns[decision_trees_rfecv.get_support()]

print("Selected features for Decision Trees (Wrapper-Based):",␣
 ↪selected_features)
```

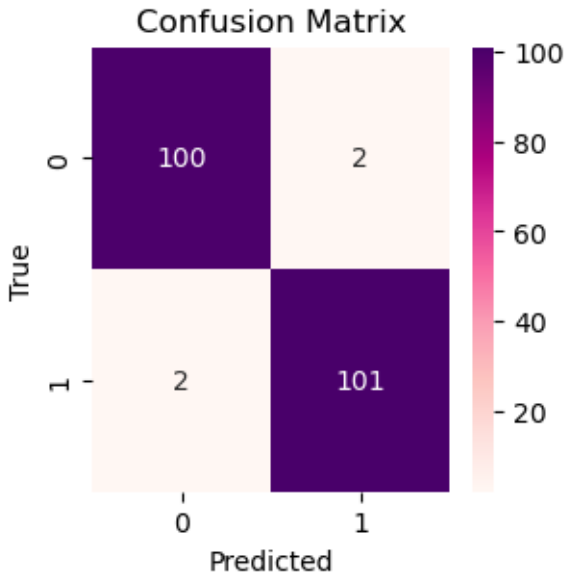
```
Optimal number of features: 8
Selected features for Decision Trees (Wrapper-Based): Index(['age', 'sex', 'cp',
'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
       'exang', 'oldpeak', 'slope', 'ca', 'thal'],
      dtype='object')
```

[ ]:
```
# After hyperparameter tuning
y_pred_decision_best = decision_tree_classifier.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_decision_best)
precision_after = precision_score(y_test, y_pred_decision_best)
recall_after = recall_score(y_test, y_pred_decision_best)
conf_matrix = confusion_matrix(y_test, y_pred_decision_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_decision_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
```

```
# After feature selection
# X_test_selected_features = logistic_regression_rfecv.transform(X_test)

y_pred_decision_trees_filtered = decision_trees_rfecv.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_decision_trees_filtered)
precision_after = precision_score(y_test, y_pred_decision_trees_filtered)
recall_after = recall_score(y_test, y_pred_decision_trees_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_decision_trees_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision_after * recall_after) / (precision_after +␣
 ↪recall_after)
roc_auc = roc_auc_score(y_test, y_pred_decision_trees_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='RdPu', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Evaluation Metrics before Feature Selection:
Accuracy: 0.9707317073170731
Precision: 0.9619047619047619
Recall: 0.9805825242718447
Specificity: 0.9607843137254902
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9706834189986675
```

```
Evaluation Metrics after Feature Selection:
Accuracy: 0.9804878048780488
Precision: 0.9805825242718447
Recall: 0.9805825242718447
Specificity: 0.9607843137254902
F1 Score: 0.9805825242718447
ROC AUC Score: 0.9804873405672949
[[100   2]
 [  2 101]]
```



[ ]:
```python
# Support Vector Machines: Feature Selection
svm_selector = SelectKBest(score_func=mutual_info_classif, k=13)
X_train_selected = svm_selector.fit_transform(X_train, y_train)

feature_scores = svm_selector.scores_

# Get the p-values of all features (if applicable)
# p_values = svm_selector.pvalues_  # Uncomment if using a scoring function␣
 ↪that supports p-values

# Get the indices of the selected features
selected_feature_indices = svm_selector.get_support(indices=True)

# Get the names of the selected features
selected_features = X.columns[selected_feature_indices]

# Print details
```

```
print("Selected Features:")
for feature, score in zip(selected_features,␣
 ↪feature_scores[selected_feature_indices]):
    print(f"Feature: {feature}, Score: {score}")
# Train SVM Classifier using the selected features

grid_search_svm_filtered = copy.copy(grid_search_svc)
best_params_svm = grid_search_svm_filtered.best_params_

svm_classifier_filtered = SVC(**best_params_svm)
svm_classifier_filtered.fit(X_train_selected, y_train)
```

```
Selected Features:
Feature: age, Score: 0.11943488155505566
Feature: sex, Score: 0.04326817650116621
Feature: cp, Score: 0.14450529811040713
Feature: trestbps, Score: 0.0623124022070245
Feature: chol, Score: 0.1785400598151876
Feature: fbs, Score: 0.0
Feature: restecg, Score: 0.02645413897071247
Feature: thalach, Score: 0.15982183438875674
Feature: exang, Score: 0.08456900931628786
Feature: oldpeak, Score: 0.16354731339701334
Feature: slope, Score: 0.08910389414085573
Feature: ca, Score: 0.1147388091937962
Feature: thal, Score: 0.12563386431926427
```

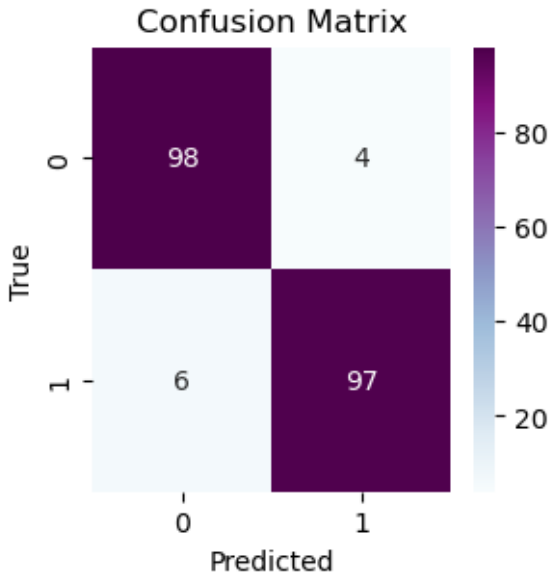
[ ]:
```
SVC(C=10)
```

[ ]:
```
# After hyperparameter tuning
y_pred_svm_best = best_svm_grid_test.predict(X_test)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_svm_best)
precision_after = precision_score(y_test, y_pred_svm_best)
recall_after = recall_score(y_test, y_pred_svm_best)
conf_matrix = confusion_matrix(y_test, y_pred_svm_best)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_svm_best)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics before Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
```

```
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)


# After feature selection
# X_test_selected_features = logistic_regression_rfecv.transform(X_test)

X_test_filtered = svm_selector.transform(X_test)
y_pred_svm_filtered = svm_classifier_filtered.predict(X_test_filtered)

# Calculate evaluation metrics after tuning
accuracy_after = accuracy_score(y_test, y_pred_svm_filtered)
precision_after = precision_score(y_test, y_pred_svm_filtered)
recall_after = recall_score(y_test, y_pred_svm_filtered)
conf_matrix_test = confusion_matrix(y_test, y_pred_svm_filtered)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision_after * recall_after) / (precision_after +␣
 ↪recall_after)
roc_auc = roc_auc_score(y_test, y_pred_svm_filtered)

# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Feature Selection:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)

print(conf_matrix_test)

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_test, annot=True, cmap='BuPu', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

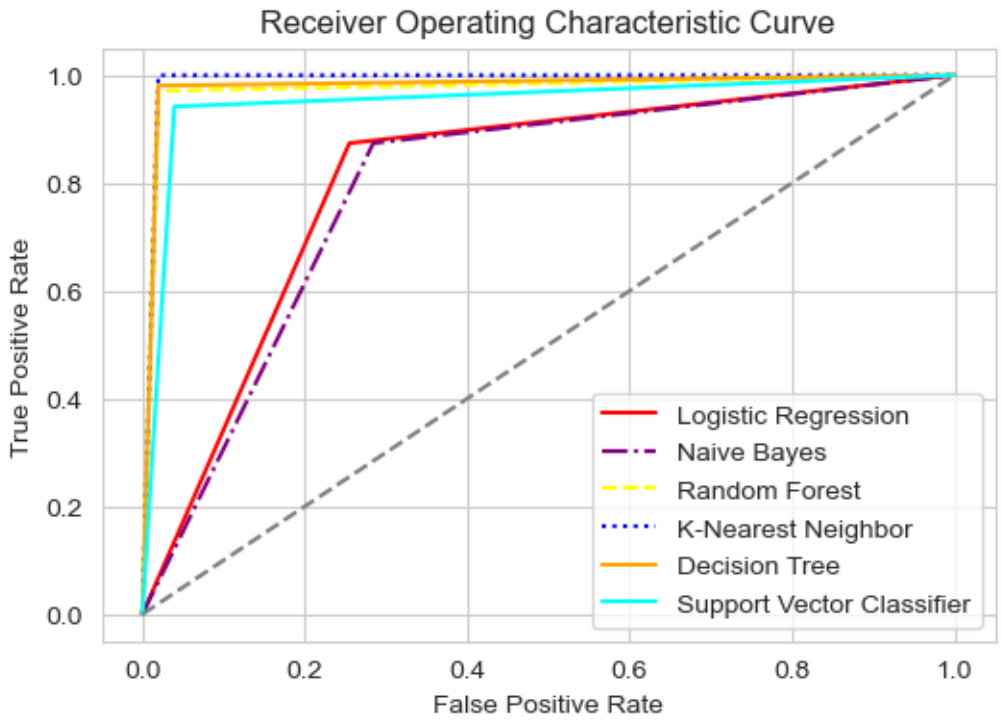
```
Evaluation Metrics before Feature Selection:
Accuracy: 0.9512195121951219
Precision: 0.9603960396039604
Recall: 0.941747572815534
```

```
Specificity: 0.9607843137254902
F1 Score: 0.8837209302325582
ROC AUC Score: 0.9512659432705121

Evaluation Metrics after Feature Selection:
Accuracy: 0.9512195121951219
Precision: 0.9603960396039604
Recall: 0.941747572815534
Specificity: 0.9607843137254902
F1 Score: 0.9509803921568628
ROC AUC Score: 0.9512659432705121
[[98  4]
 [ 6 97]]
```



[ ]:
```
lr_false_positive_rate,lr_true_positive_rate,lr_threshold =␣
  ↪roc_curve(y_test,y_pred_lr_filtered)
nb_false_positive_rate,nb_true_positive_rate,nb_threshold =␣
  ↪roc_curve(y_test,y_pred_nb_filtered)
rf_false_positive_rate,rf_true_positive_rate,rf_threshold =␣
  ↪roc_curve(y_test,y_pred_random_forest_filtered)
knn_false_positive_rate,knn_true_positive_rate,knn_threshold =␣
  ↪roc_curve(y_test,y_pred_knn_filtered)
dt_false_positive_rate,dt_true_positive_rate,dt_threshold =␣
  ↪roc_curve(y_test,y_pred_decision_trees_filtered)
svc_false_positive_rate,svc_true_positive_rate,svc_threshold =␣
  ↪roc_curve(y_test,y_pred_svm_filtered)
sns.set_style('whitegrid')
```

```
plt.figure(figsize=(6,4))
plt.title('Receiver Operating Characteristic Curve')

# Plot ROC curves for each classifier with different line styles and colors
plt.plot(lr_false_positive_rate, lr_true_positive_rate, label='Logistic␣
 ↪Regression', color='red', linestyle='-')
plt.plot(nb_false_positive_rate, nb_true_positive_rate, label='Naive Bayes',␣
 ↪color='purple', linestyle='-.')
plt.plot(rf_false_positive_rate, rf_true_positive_rate, label='Random Forest',␣
 ↪color='yellow', linestyle='--')
plt.plot(knn_false_positive_rate, knn_true_positive_rate, label='K-Nearest␣
 ↪Neighbor', color='blue', linestyle=':')
plt.plot(dt_false_positive_rate, dt_true_positive_rate, label='Decision Tree',␣
 ↪color='orange', linestyle='-')
plt.plot(svc_false_positive_rate, svc_true_positive_rate, label='Support Vector␣
 ↪Classifier', color='cyan', linestyle='-')

# Plot the diagonal reference line
plt.plot([0, 1], [0, 1], ls='--', color='.5')

# Add labels and legend
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()

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



[ ]:
```
print("Logistic Regression ROC Curve:")
for fpr, tpr, threshold in zip(lr_false_positive_rate, lr_true_positive_rate,␣
 ↪lr_threshold):
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")

print("\nNaive Bayes ROC Curve:")
for fpr, tpr, threshold in zip(nb_false_positive_rate, nb_true_positive_rate,␣
 ↪nb_threshold):
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")

print("\nRandom Forest ROC Curve:")
for fpr, tpr, threshold in zip(rf_false_positive_rate, rf_true_positive_rate,␣
 ↪rf_threshold):
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")

print("\nK Nearest Neighbors ROC Curve:")
for fpr, tpr, threshold in zip(knn_false_positive_rate, knn_true_positive_rate,␣
 ↪knn_threshold):
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")

print("\nDecision Trees ROC Curve:")
for fpr, tpr, threshold in zip(dt_false_positive_rate, dt_true_positive_rate,␣
 ↪dt_threshold):
```

```
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")

print("\nSupport Vector Machines ROC Curve:")
for fpr, tpr, threshold in zip(svc_false_positive_rate, svc_true_positive_rate,␣
 ↪svc_threshold):
    print(f"FPR: {fpr}, TPR: {tpr}, Threshold: {threshold}")
```

```
Logistic Regression ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.2549019607843137, TPR: 0.8737864077669902, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0

Naive Bayes ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.28431372549019607, TPR: 0.8737864077669902, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0

Random Forest ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.0196078431372549, TPR: 0.970873786407767, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0

K Nearest Neighbors ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.0196078431372549, TPR: 1.0, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0

Decision Trees ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.0196078431372549, TPR: 0.9805825242718447, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0

Support Vector Machines ROC Curve:
FPR: 0.0, TPR: 0.0, Threshold: inf
FPR: 0.0392156862745098, TPR: 0.941747572815534, Threshold: 1.0
FPR: 1.0, TPR: 1.0, Threshold: 0.0
```

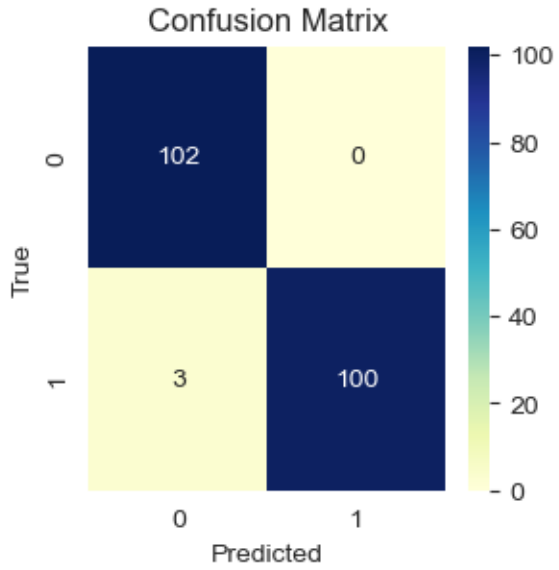
[ ]:
```
#Ensemble Techniques

from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import StackingClassifier
from sklearn.pipeline import Pipeline
```

[ ]:
```
#Bagging Ensemble (Homogeneous)

# Create a base classifier (e.g., Decision Tree)
base_classifier = DecisionTreeClassifier(random_state=42)
```

```
# Create a BaggingClassifier
bagging_clf = BaggingClassifier(base_estimator=base_classifier,␣
 ↪n_estimators=30, random_state=42)

# Train the BaggingClassifier
bagging_clf.fit(X_train, y_train)

# Make predictions
y_pred_bagging = bagging_clf.predict(X_test)

accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
recall_bagging = recall_score(y_test, y_pred_bagging)
precision_bagging = precision_score(y_test, y_pred_bagging)
conf_matrix_bagging = confusion_matrix(y_test, y_pred_bagging)
tn, fp, fn, tp = conf_matrix.ravel()
specificity_after = tn / (tn + fp)
sensitivity_after = tp / (tp + fn)
npv_after = tn / (tn + fn)
f1_score_after = 2 * (precision_bagging * recall_bagging) / (precision_bagging␣
 ↪+ recall_bagging)
roc_auc = roc_auc_score(y_test, y_pred_bagging)
confusion_matrix_bagging = confusion_matrix(y_test, y_pred_bagging)


# Print evaluation metrics after tuning
print("\nEvaluation Metrics after Bagging:")
print("Accuracy:", accuracy_after)
print("Precision:", precision_after)
print("Recall:", recall_after)
print("Specificity:", specificity_after)
print("F1 Score:", f1_score_after)
print("ROC AUC Score:", roc_auc)
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_bagging))

plt.figure(figsize=(3, 3))
sns.heatmap(conf_matrix_bagging, annot=True, cmap='YlGnBu', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
D:\Anaconda\envs\workingenv\lib\site-packages\sklearn\ensemble\_base.py:156:
FutureWarning: `base_estimator` was renamed to `estimator` in version 1.2 and
will be removed in 1.4.
  warnings.warn(
```

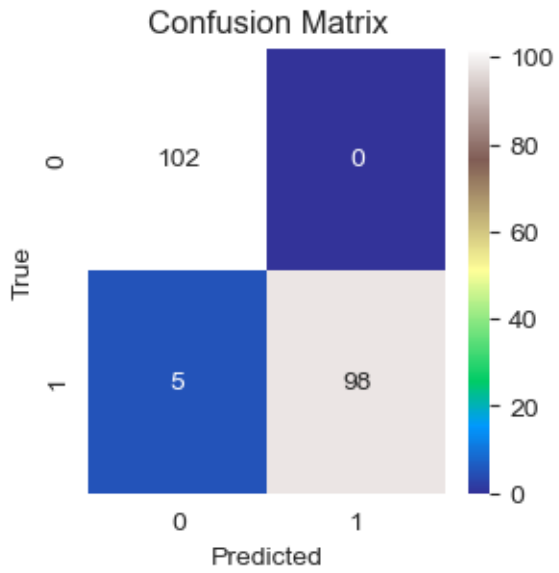
```
Evaluation Metrics after Bagging:
Accuracy: 0.9609756097560975
Precision: 0.9523809523809523
Recall: 0.970873786407767
Specificity: 0.9509803921568627
F1 Score: 0.9852216748768473
ROC AUC Score: 0.9854368932038835
Confusion Matrix:
[[102   0]
 [  3 100]]
```



[ ]:
```python
# Ensemble Tuned Models (heterogeneous)
base_models = [
    ('Logistic Regression', grid_search_lr.best_estimator_),
    ('Naive Bayes', naive_bayes_classifier),
    ('Random Forest', grid_search_rf.best_estimator_),
    # ('Decision Tree', grid_search_decision_tree.best_estimator_),
    ('K Nearest Neighbors', grid_search_knn.best_estimator_),
    ('Support Vector Machines', grid_search_svc.best_estimator_)
]

stacking_clf = StackingClassifier(
    estimators=base_models,
    final_estimator=grid_search_decision_tree.best_estimator_
)

# Train the stacking classifier on the training set
```

```
stacking_clf.fit(X_train, y_train)

# Evaluate the stacking classifier on the test set
y_pred_test_stacking = stacking_clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred_test_stacking)
precision = precision_score(y_test, y_pred_test_stacking)
recall = recall_score(y_test, y_pred_test_stacking)
conf_matrix = confusion_matrix(y_test, y_pred_test_stacking)
tn, fp, fn, tp = conf_matrix.ravel()
specificity = tn / (tn + fp)
sensitivity = tp / (tp + fn)
npv = tn / (tn + fn)
f1_score = 2 * (precision * recall) / (precision + recall)
roc_auc = roc_auc_score(y_test, y_pred_test_stacking)
confusion_matrix_stacking  = confusion_matrix(y_test,
y_pred_test_stacking)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
print("ROC AUC Score:", roc_auc)

print(confusion_matrix_stacking)

plt.figure(figsize=(3, 3))
sns.heatmap(confusion_matrix_stacking, annot=True, cmap='terrain', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.975609756097561
Precision: 1.0
Recall: 0.9514563106796117
Specificity: 1.0
F1 Score: 0.9751243781094527
ROC AUC Score: 0.9757281553398058
[[102   0]
 [  5  98]]
```



[ ]:
```
# # Ensemble Tuned Models
# base_models = [
#     ('Logistic Regression', logistic_regression),
#     ('Naive Bayes', naive_bayes_classifier),
#     ('Random Forest', random_forest_classifier),
#     # ('Decision Tree', grid_search_decision_tree.best_estimator_),
#     ('K Nearest Neighbors', knn_classifier),
#     ('Support Vector Machines', svc_classifier)
# ]

# stacking_clf = StackingClassifier(
#     estimators=base_models,
#     final_estimator=decision_tree_classifier
# )

# # Train the stacking classifier on the training set
# stacking_clf.fit(X_train, y_train)

# # Evaluate the stacking classifier on the test set
# y_pred_test_stacking = stacking_clf.predict(X_test)
# accuracy = accuracy_score(y_test, y_pred_test_stacking)
# precision = precision_score(y_test, y_pred_test_stacking)
# recall = recall_score(y_test, y_pred_test_stacking)
# conf_matrix = confusion_matrix(y_test, y_pred_test_stacking)
# tn, fp, fn, tp = conf_matrix.ravel()
# specificity = tn / (tn + fp)
# sensitivity = tp / (tp + fn)
```

```
# npv = tn / (tn + fn)
# f1_score = 2 * (precision * recall) / (precision + recall)
# roc_auc = roc_auc_score(y_test, y_pred_test_stacking)
# confusion_matrix_stacking  = confusion_matrix(y_test,
# y_pred_test_stacking)

# # Print evaluation metrics
# print("Accuracy:", accuracy)
# print("Precision:", precision)
# print("Recall:", recall)
# print("Specificity:", specificity)
# print("F1 Score:", f1_score)
# print("ROC AUC Score:", roc_auc)

# print(confusion_matrix_stacking)

# plt.figure(figsize=(3, 3))
# sns.heatmap(confusion_matrix_stacking, annot=True, cmap='terrain', fmt='g')
# plt.xlabel('Predicted')
# plt.ylabel('True')
# plt.title('Confusion Matrix')
# plt.show()
```

[ ]:
```
# # Ensemble Feature Selection
# base_models = [
#     ('Logistic Regression', grid_search_lr.best_estimator_),
#     ('Naive Bayes', naive_bayes_classifier),
#     ('Random Forest', grid_search_rf.best_estimator_),
#     ('Decision Tree', grid_search_decision_tree.best_estimator_),
#     ('K Nearest Neighbors', grid_search_knn.best_estimator_),
#     ('Support Vector Machines', grid_search_svc.best_estimator_)
# ]

# # Define feature selection models
# cv_lr = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# cv_rf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# feature_selection_models = [
#     ('RFECV Logistic', RFECV(estimator=LogisticRegression(), step=1,␣
 ↪cv=cv_lr, scoring='accuracy')),
#     ('SelectKBest Naive Bayes', SelectKBest(score_func=mutual_info_classif,␣
 ↪k=10)),
#     ('SelectKBest KNN', SelectKBest(score_func=mutual_info_classif, k=10)),
#     ('RFECV Random Forest', RFECV(estimator=RandomForestClassifier(), step=1,␣
 ↪cv=cv_rf, scoring='accuracy')),
#     ('SelectKBest SVM', SelectKBest(score_func=mutual_info_classif, k=11))
# ]
```

```
# estimators = []
# for model, fs_model in feature_selection_models:
#     for base_name, base_model in base_models:
#         estimators.append((f"{model} + {base_name}", Pipeline([('Feature␣
 ↪Selection', fs_model), (base_name, base_model)])))

# stacking_clf = StackingClassifier(
#     estimators=base_models,
#     final_estimator= grid_search_knn
# )


# # Train the stacking classifier on the training set
# stacking_clf.fit(X_train, y_train)

# # Evaluate the stacking classifier on the test set
# y_pred_test_stacking = stacking_clf.predict(X_test)
# accuracy = accuracy_score(y_test, y_pred_test_stacking)
# precision = precision_score(y_test, y_pred_test_stacking)
# recall = recall_score(y_test, y_pred_test_stacking)
# conf_matrix = confusion_matrix(y_test, y_pred_test_stacking)
# tn, fp, fn, tp = conf_matrix.ravel()
# specificity = tn / (tn + fp)
# sensitivity = tp / (tp + fn)
# npv = tn / (tn + fn)
# f1_score = 2 * (precision * recall) / (precision + recall)
# roc_auc = roc_auc_score(y_test, y_pred_test_stacking)

# # Print evaluation metrics
# print("Accuracy:", accuracy)
# print("Precision:", precision)
# print("Recall:", recall)
# print("Specificity:", specificity)
# print("F1 Score:", f1_score)
# print("ROC AUC Score:", roc_auc)
```

[ ]:
```
# # Stacking Ensemble Technique (heterogeneous)
# # Pre-trained and tuned models
# base_models = [
#     ('Logistic Regression', LogisticRegression()),
#     ('Naive Bayes', GaussianNB()),
#     ('Random Forest', RandomForestClassifier()),
#     ('Decision Tree', DecisionTreeClassifier()),
#     ('K Nearest Neighbors', KNeighborsClassifier()),
#     ('Support Vector Machines', SVC())
# ]
```

```
# cv_lr = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# cv_rf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)


#     # Define feature selection models
# feature_selection_models = [
#     ('RFECV Logistic', RFECV(estimator=LogisticRegression(), step=1,␣
 ↪cv=cv_lr, scoring='accuracy')),
#     ('SelectKBest Naive Bayes', SelectKBest(score_func=mutual_info_classif,␣
 ↪k=10)),
#     ('SelectKBest KNN', SelectKBest(score_func=mutual_info_classif, k=10)),
#     ('RFECV Random Forest', RFECV(estimator=RandomForestClassifier(), step=1,␣
 ↪cv=cv_rf, scoring='accuracy')),
#     ('SelectKBest SVM', SelectKBest(score_func=mutual_info_classif, k=11))
# ]

# estimators = []
# for model, fs_model in feature_selection_models:
#     for base_name, base_model in base_models:
#         estimators.append((f"{model} + {base_name}", Pipeline([('Feature␣
 ↪Selection', fs_model), (base_name, base_model)])))

# stacking_clf = StackingClassifier(
#     estimators=base_models,
#     final_estimator= decision_trees_rfecv
# )

# # Train the stacking classifier on the training set
# stacking_clf.fit(X_train, y_train)

# # Evaluate the stacking classifier on the test set
# y_pred_test_stacking = stacking_clf.predict(X_test)
# accuracy = accuracy_score(y_test, y_pred_test_stacking)
# precision = precision_score(y_test, y_pred_test_stacking)
# recall = recall_score(y_test, y_pred_test_stacking)
# conf_matrix = confusion_matrix(y_test, y_pred_test_stacking)
# tn, fp, fn, tp = conf_matrix.ravel()
# specificity = tn / (tn + fp)
# sensitivity = tp / (tp + fn)
# npv = tn / (tn + fn)
# f1_score = 2 * (precision * recall) / (precision + recall)
# roc_auc = roc_auc_score(y_test, y_pred_test_stacking)

# # Print evaluation metrics
# print("Accuracy:", accuracy)
# print("Precision:", precision)
```

```
# print("Recall:", recall)
# print("Specificity:", specificity)
# print("F1 Score:", f1_score)
# print("ROC AUC Score:", roc_auc)
```