Project 3: Applying Your Skills

Team Members
Haotian Shen
Qiaozhi Liu

Abstract

This project was a great application of the skills we learned from the lecture. By conducting data pre-processing and mining, we were able to gain useful insights into the heart disease dataset. We experimented with multiple classifiers, neural network models, and metrics to determine the initial model that performed the best and continued to improve upon it. By tuning the hyperparameters, we were able to find a balance between model complexity and predictive accuracy.

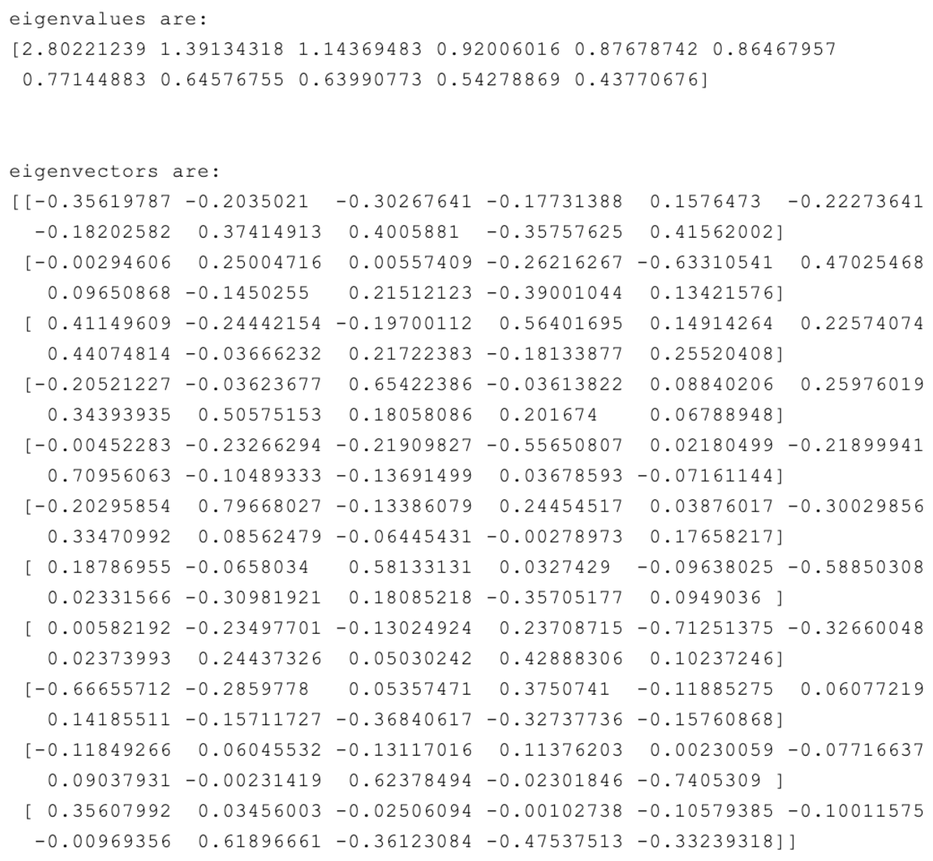
Reflection

Although aiming for a better predictive accuracy is important, we quickly learned that achieving a high test accuracy wasn't the only factor we needed to consider. We also needed to consider the true positive rate at a low false positive rate, which is crucial when dealing with medical data. By experimenting with multiple models, we were able to expand our knowledge and understanding of various machine learning methods. Throughout the project, we also discovered that having a good understanding and expertise of the dataset was equally as important as selecting the highest test accuracy model. It allowed us to make informed decisions about which model would provide us with the best results. Overall, this project provided us with a valuable opportunity to apply the concepts we learned in the lecture and to gain hands-on experience in data pre-processing, mining, and model tuning. It also highlighted the importance of selecting a model based on a variety of factors, including both test accuracy and true positive rate at a low false positive rate.

Task 1: Pre-processing, Data Mining, and Visualization

To begin our data analysis process, we will first visualize the dataset. However, as the provided data is high-dimensional, we must first employ PCA to reduce it to a two-dimensional space using the two most significant eigenvectors based on their respective eigenvalues.

After cleaning and normalizing our training data, X_train, we performed a principal component analysis (PCA) which yielded the following eigenvalues and eigenvectors:



```
eigenvalues are:
[2.80221239 1.39134318 1.14369483 0.92006016 0.87678742 0.86467957
 0.77144883 0.64576755 0.63990773 0.54278869 0.43770676]


eigenvectors are:
[[-0.35619787 -0.2035021  -0.30267641 -0.17731388  0.1576473  -0.22273641
  -0.18202582  0.37414913  0.4005881  -0.35757625  0.41562002]
 [-0.00294606  0.25004716  0.00557409 -0.26216267 -0.63310541  0.47025468
   0.09650868 -0.1450255   0.21512123 -0.39001044  0.13421576]
 [ 0.41149609 -0.24442154 -0.19700112  0.56401695  0.14914264  0.22574074
   0.44074814 -0.03666232  0.21722383 -0.18133877  0.25520408]
 [-0.20521227 -0.03623677  0.65422386 -0.03613822  0.08840206  0.25976019
   0.34393935  0.50575153  0.18058086  0.201674    0.06788948]
 [-0.00452283 -0.23266294 -0.21909827 -0.55650807  0.02180499 -0.21899941
   0.70956063 -0.10489333 -0.13691499  0.03678593 -0.07161144]
 [-0.20295854  0.79668027 -0.13386079  0.24454517  0.03876017 -0.30029856
   0.33470992  0.08562479 -0.06445431 -0.00278973  0.17658217]
 [ 0.18786955 -0.0658034   0.58133131  0.0327429  -0.09638025 -0.58850308
   0.02331566 -0.30981921  0.18085218 -0.35705177  0.0949036 ]
 [ 0.00582192 -0.23497701 -0.13024924  0.23708715 -0.71251375 -0.32660048
   0.02373993  0.24437326  0.05030242  0.42888306  0.10237246]
 [-0.66655712 -0.2859778   0.05357471  0.3750741  -0.11885275  0.06077219
   0.14185511 -0.15711727 -0.36840617 -0.32737736 -0.15760868]
 [-0.11849266  0.06045532 -0.13117016  0.11376203  0.00230059 -0.07716637
   0.09037931 -0.00231419  0.62378494 -0.02301846 -0.7405309 ]
 [ 0.35607992  0.03456003 -0.02506094 -0.00102738 -0.10579385 -0.10011575
  -0.00969356  0.61896661 -0.36123084 -0.47537513 -0.33239318]]
```

Figure 4: PCA analysis result of eigenvalues and eigenvectors using training data X_train.

Next, we plotted the two-dimensional data and the resulting plot is shown below. Upon observation, we can discern that there are no distinct clusters present within our projected 2-D data.

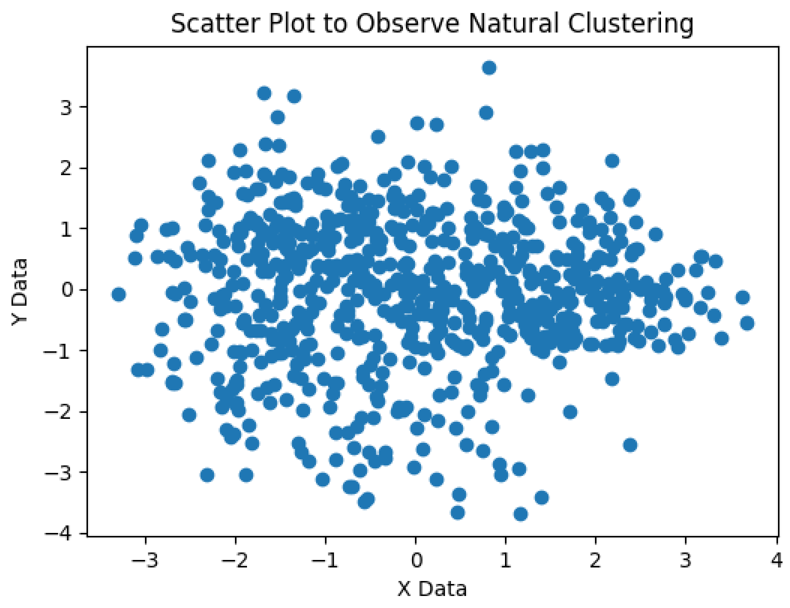


Figure 1: Natural clustering within the dataset using 2-D projected data.

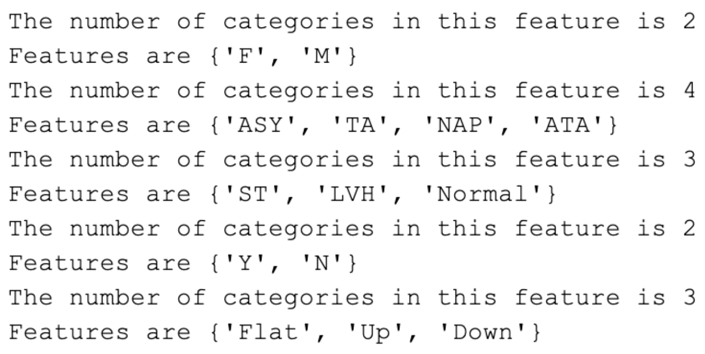
Given the numerous features in the provided data, with each having a different unit, data normalization or whitening is necessary as a preprocessing step. In order to accomplish this, the training and test sets should be combined, enabling normalization of the entire dataset on a uniform scale. Once combined, a preliminary analysis of the data is recommended to better understand the meaning and potential values of each column. As a starting point, a quick overview of the first 5 rows of the dataset is provided below.

```
    Age  Sex  ChestPainType  RestingBP  Cholesterol  FastingBS  RestingECG  \
0  66.0    1              4      146.0        278.0        0.0           2   
1  65.0    2              1      150.0        235.0        0.0           4   
2  63.0    2              1      150.0        223.0        0.0           4   
3  58.0    2              8      136.0        164.0        0.0           1   
4  54.0    2              8      192.0        283.0        0.0           2   

   MaxHR  ExerciseAngina  Oldpeak  ST_Slope  HeartDisease  
0  152.0               2      0.0         1           0.0  
1  120.0               1      1.5         1           1.0  
2  115.0               2      0.0         1           1.0  
3   99.0               1      2.0         1           1.0  
4  195.0               2      0.0         2           1.0  
```

Figure 2: The beginning five rows of the dataset for quick overview.

Prior to normalizing the data, we have discovered that some of the columns within the dataset contain categorical values. This necessitates the use of one-hot encoding in order to transform the categorical values into numerical values. For instance, if a feature contains 6 categories, it will be represented by the following binary numbers: 100000, 010000, 001000, 000100, 000010, 000001. The categorical value will ultimately be transformed into an integer representation of the binary vector. We will first analyze how many distinct categories exist within each categorical column. The image below displays the various classes contained within each categorical column.



```
The number of categories in this feature is 2
Features are {'F', 'M'}
The number of categories in this feature is 4
Features are {'ASY', 'TA', 'NAP', 'ATA'}
The number of categories in this feature is 3
Features are {'ST', 'LVH', 'Normal'}
The number of categories in this feature is 2
Features are {'Y', 'N'}
The number of categories in this feature is 3
Features are {'Flat', 'Up', 'Down'}
```

Figure 3: Categorical features in the raw data.

Once the data has been cleaned, our next step is to normalize it and then split it back into training and test sets. In order to identify which features are essential and should be included, we will employ both boxplot and RandomForestClassifier to process and gain a better understanding of the data.

The boxplot analysis reveals that the features of sex and fasting blood sugar levels exhibit no significant variations within their respective data. Therefore, we will exclude them from our input features and instead focus on Age, ChestPainType, RestingBP, Cholesterol, MaxHR, ExerciseAngina, Oldpeak, and ST_Slope, which correspond to index [0, 2, 3, 4, 7, 8, 9, 10]. These specific columns display high variance within their categories, indicating that they can contribute to differentiation between data points.

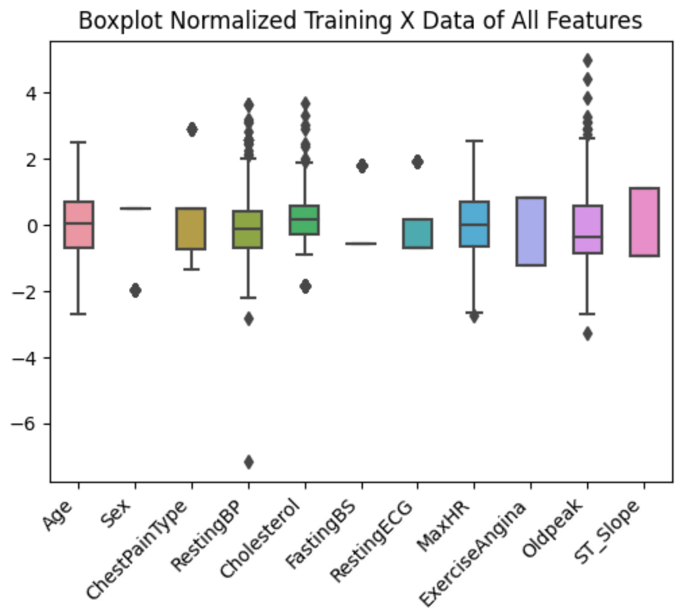
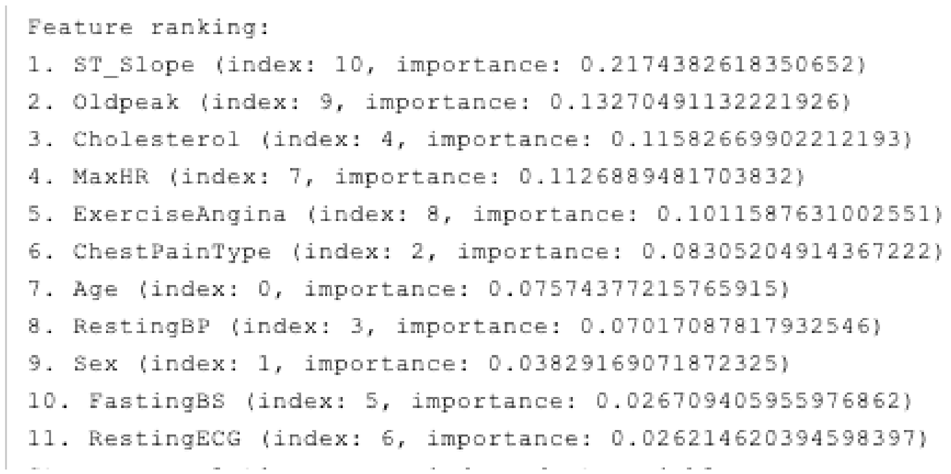


Figure 4: Boxplot to show the variations within each column of the training set data.

To further corroborate our boxplot analysis, we utilized RandomForestClassifier to fit on our dataset X and y, which will give us a ranking of important features. This was calculated based on the decrease in impurity when a particular feature is used for splitting. The result was below:



```
Feature ranking:
1. ST_Slope (index: 10, importance: 0.2174382618350652)
2. Oldpeak (index: 9, importance: 0.13270491132221926)
3. Cholesterol (index: 4, importance: 0.11582669902212193)
4. MaxHR (index: 7, importance: 0.1126889481703832)
5. ExerciseAngina (index: 8, importance: 0.1011587631002551)
6. ChestPainType (index: 2, importance: 0.08305204914367222)
7. Age (index: 0, importance: 0.07574377215765915)
8. RestingBP (index: 3, importance: 0.07017087817932546)
9. Sex (index: 1, importance: 0.03829169071872325)
10. FastingBS (index: 5, importance: 0.026709405955976862)
11. RestingECG (index: 6, importance: 0.026214620394598397)
```

Figure 5: Feature importance ranking derived using RandomForestClassifer from sklearn library

By discarding the 3 least important features, we still got the following features, Age, ChestPainType, RestingBP, Cholesterol, MaxHR, ExerciseAngina, Oldpeak, and ST_Slope, which emphasized our previous selection from the boxplot. Thus, for the input feature selection, we will use column index [0, 2, 3, 4, 7, 8, 9, 10]. This has successfully reduced our input features from 11 to only 8 for more efficient computation.

We also computed the correlations between each pair of independent variables to assess their relationships.

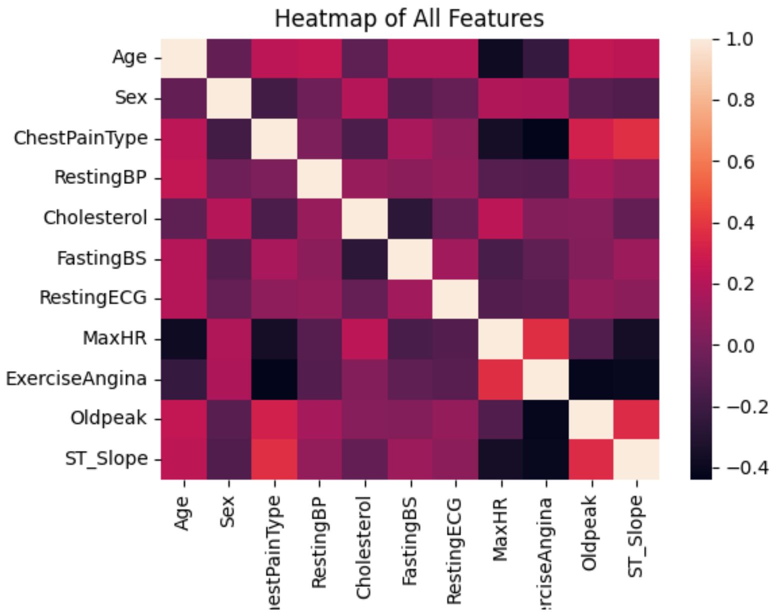
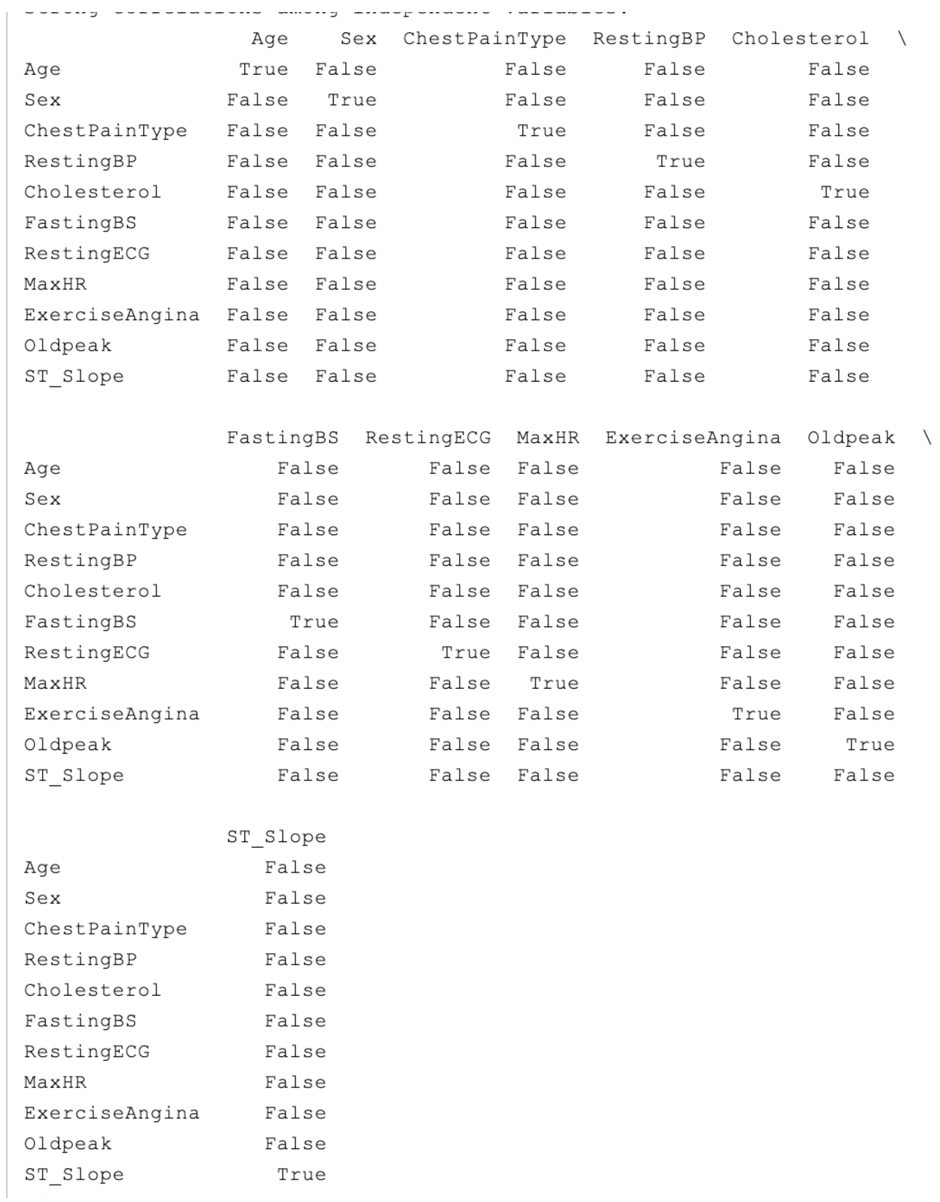


Figure 5: Heatmap of independent variables correlation

We utilized a true/false table to demonstrate the relationship between pairs of variables, utilizing a cutoff threshold of 0.5. The results revealed that none of the independent variables displayed any significant correlation with one another.



```
                  Age    Sex  ChestPainType  RestingBP  Cholesterol  \
Age              True  False          False      False        False
Sex             False   True          False      False        False
ChestPainType   False  False           True      False        False
RestingBP       False  False          False       True        False
Cholesterol     False  False          False      False         True
FastingBS       False  False          False      False        False
RestingECG      False  False          False      False        False
MaxHR           False  False          False      False        False
ExerciseAngina  False  False          False      False        False
Oldpeak         False  False          False      False        False
ST_Slope        False  False          False      False        False

                FastingBS  RestingECG  MaxHR  ExerciseAngina  Oldpeak  \
Age                 False       False  False           False    False
Sex                 False       False  False           False    False
ChestPainType       False       False  False           False    False
RestingBP           False       False  False           False    False
Cholesterol         False       False  False           False    False
FastingBS            True       False  False           False    False
RestingECG          False        True  False           False    False
MaxHR               False       False   True           False    False
ExerciseAngina      False       False  False            True    False
Oldpeak             False       False  False           False     True
ST_Slope            False       False  False           False    False

                ST_Slope
Age                False
Sex                False
ChestPainType      False
RestingBP          False
Cholesterol        False
FastingBS          False
RestingECG         False
MaxHR              False
ExerciseAngina     False
Oldpeak            False
ST_Slope            True
```

Figure 6: True/false table showing whether each pair has correlations.

By examining the first three eigenvalues and their corresponding eigenvectors, we can gain insight into:

Table 1: Potential correlations of different independent features within our dataset.

First feature	Second feature	Correlation direction
Age	Max HR	Negative
Age	Exercise Angina	Positive
Age	Oldpeak	Positive
Age	ST_Slope	Positive
Cholesterol level	Resting BP	Positive
Cholesterol level	Fasting BS	Negative
Cholesterol level	Oldpeak	Positive
Resting ECG	Resting BP	Negative

To investigate the significant signals within our features, we implemented a logistic regression using our normalized data to predict the test results. Initially, we achieved an accuracy of 0.855 using all of the independent variables.

Next, we exclusively utilized the 8 significant signals that we previously identified via boxplot analysis, namely: Age, ChestPainType, RestingBP, Cholesterol, MaxHR, ExerciseAngina, Oldpeak, and ST_Slope. By employing only these 8 independent features, we still achieved an impressive test set accuracy of 0.845 using logistic regression on our training set.

The minimal discrepancy between the two accuracy scores suggests that we can solely utilize these 8 features as our input features and still obtain sufficient variations and significant signals to ensure a successful model.

To further enhance our machine learning model, we can attempt to engineer features by adding polynomial features into our dataset. Since the Age, RestingBP, Cholesterol level, MaxHR, and ST_Slope may not exhibit a linear relationship with the prediction of heart disease, we can incorporate quadratic and cubic terms (polynomial features) into our dataset with the goal to improve prediction accuracy.

Task 2: Classification

Since the data is already cleaned and one-hot encoding is applied to all categorical data, the feature engineering is sufficient to yield a good prediction result. I have added more generated features by computing the squared and cubic term of cholesterol level, so the input feature size will be 10.

We have decided to use the following classifier model to train our training data and predict on the test set:

Available classifiers:
- Logistic Regression
- SGDCClassifier L2
- SGDCClassifier L1
- Ridge Classifier
- Perceptron
- BernoulliNB
- K-Nearest Neighbor
- Support Vector Machine
- Decision Tree Classifier
- Random Forest Classifier

Here are descriptions of each model used:

Logistic Regression: It predicts the probability of an outcome by describing the relationship between one dependent binary variable and one or more independent variables. It uses the logistic function to transform a linear combination of independent variables into a probability between 0 and 1.

Available hyper-parameters:
1. penalty: It controls the type of regularization used in the model. It can take either 'l1', 'l2', 'elasticnet', or 'none'. The default is 'l2'.
2. C: It is the inverse of regularization strength, which determines how much to penalize large coefficients. The smaller the value of C, the stronger the regularization. The default value is 1.0.
3. solver: It specifies the algorithm to use in the optimization problem. It can take either 'newton-cg', 'lbfgs', 'liblinear', 'sag', or 'saga'. The default is 'lbfgs'.
4. max_iter: It sets the maximum number of iterations for the solver to converge. The default is 100.
5. tol: It sets the tolerance for stopping criteria of the solver. The default is 1e-4.
6. fit_intercept: It specifies whether to include the intercept term in the model. The default is True.
7. class_weight: It is used to handle imbalanced classes by assigning weights to each class. It can take either 'balanced' or a dictionary of weights. The default is None.
8. multi_class: It specifies how to handle multi-class classification problems. It can take either 'ovr' for one-vs-rest or 'multinomial' for multinomial logistic regression. The default is 'ovr'.

9. n_jobs: It specifies the number of CPU cores to use for the computation. The default is 1, which means using a single core.

SGDCClassifier L1/L2: Stochastic gradient descent finds the minimum of an objective function (single, log, or modified_huber) and updates the parameters of the function. L1 means using L1 penalty and L2 means using L2 penalty. L1 can help with feature selection and reduce overfitting. L2 penalty is more suitable to perform slight changes in the weights and penalizes large weights more severely. It tends to drive towards a more balanced distribution of coefficients.

Available hyper-parameters:

1. loss: It specifies the loss function to use for the model. It can take either 'hinge' for linear SVM, 'log' for logistic regression, or 'modified_huber' for a smooth approximation of hinge loss. The default is 'hinge'.
2. penalty: It controls the type of regularization used in the model. It can take either 'l2', 'l1', 'elasticnet', or 'none'. The default is 'l2'.
3. alpha: It is the regularization parameter that determines the strength of the regularization. The default is 0.0001.
4. l1_ratio: It is the ratio between L1 and L2 regularization when using elasticnet penalty. The default is 0.15.
5. max_iter: It sets the maximum number of iterations for the solver to converge. The default is 1000.
6. tol: It sets the tolerance for stopping criteria of the solver. The default is 1e-3.
7. fit_intercept: It specifies whether to include the intercept term in the model. The default is True.
8. learning_rate: It specifies the learning rate for the optimizer. It can take either 'constant', 'optimal', 'invscaling', or 'adaptive'. The default is 'optimal'.
9. eta0: It is the initial learning rate for the optimizer. The default is 0.0.
10. power_t: It is the exponent for inverse scaling learning rate. The default is 0.5.
11. class_weight: It is used to handle imbalanced classes by assigning weights to each class. It can take either 'balanced' or a dictionary of weights. The default is None.
12. n_jobs: It specifies the number of CPU cores to use for the computation. The default is 1, which means using a single core.

Ridge Classifier: It is a linear classifier in machine learning that uses Ridge regression to regularize the coefficients of the linear model. It works by adding a regularization term to the least-squares objective function. The regularization term is a multiple of the sum of squares of the coefficients, multiplied by a hyperparameter alpha. The larger the value of alpha, the more the coefficients are penalized, which can help reduce overfitting.

Available hyper-parameters:

1. alpha: It is the regularization strength, which determines the amount of shrinkage applied to the coefficients. The default is 1.0.
2. fit_intercept: It specifies whether to include the intercept term in the model. The default is True.
3. normalize: It specifies whether to normalize the input features before fitting the model. The default is False.
4. solver: It specifies the algorithm to use for optimization. It can take either 'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', or 'saga'. The default is 'auto'.
5. max_iter: It sets the maximum number of iterations for the solver to converge. The default is None.
6. tol: It sets the tolerance for stopping criteria of the solver. The default is 1e-3.
7. class_weight: It is used to handle imbalanced classes by assigning weights to each class. It can take either 'balanced' or a dictionary of weights. The default is None.
8. random_state: It is used to seed the random number generator for reproducible results. The default is None.

Perceptron: It is a supervised neural network model used as a binary classifier and also performs as a linear classifier. The algorithm works by taking in a set of input features, which are each assigned a weight value. The input features are multiplied by their corresponding weights, and the resulting values are summed up to produce a single value. This value is then passed through a step function, which outputs a binary classification label (typically 1 or -1).

Available hyper-parameters:

1. penalty: It controls the type of regularization used in the model. It can take either 'l2', 'l1', or 'elasticnet'. The default is None.
2. alpha: It is the regularization parameter that determines the strength of the regularization. The default is 0.0001.
3. fit_intercept: It specifies whether to include the intercept term in the model. The default is True.
4. max_iter: It sets the maximum number of iterations for the solver to converge. The default is 1000.
5. tol: It sets the tolerance for stopping criteria of the solver. The default is 1e-3.
6. shuffle: It specifies whether to shuffle the training data at each iteration. The default is True.
7. verbose: It specifies the level of verbosity during training. The default is 0.
8. eta0: It is the initial learning rate for the optimizer. The default is 1.0.
9. n_jobs: It specifies the number of CPU cores to use for the computation. The default is 1, which means using a single core.
10. early_stopping: It specifies whether to stop training when the validation score stops improving. The default is False.
11. validation_fraction: It is the fraction of the training data to use for validation. The default is 0.1.
12. n_iter_no_change: It sets the maximum number of iterations without improvement before stopping. The default is 5.

BernoulliNB: It is used for binary data and it works by first estimating the probability of each feature being present in each class, based on the training data. Then, when given a new input, it calculates the probability of each class given the input features, using Bayes's theorem. The class with the highest probability is selected as the output.

Available hyper-parameters:

1. alpha: It is the smoothing parameter that determines the amount of smoothing applied to the probabilities. The default is 1.0.
2. binarize: It is the threshold value used to binarize the input features. The default is 0.0.
3. fit_prior: It specifies whether to learn class prior probabilities from the data or use a uniform prior. The default is True.
4. class_prior: It allows you to specify prior probabilities for each class. If None, uniform prior probabilities are used. The default is None.
5. alpha: It is the additive (Laplace/Lidstone) smoothing parameter. The default is 1.0.
6. binarize: It is the threshold used to binarize the input features. The default is 0.0.
7. fit_prior: It specifies whether to learn class prior probabilities from the data or use a uniform prior. The default is True.
8. class_prior: It allows you to specify prior probabilities for each class. If None, uniform prior probabilities are used. The default is None.

K-Nearest Neighbor: It is a non-parametric classification algorithm used in machine learning to predict the class of a new input based on the classes of its K nearest neighbors in the training data. In a binary classification problem, the algorithm works by first calculating the distances between the input sample and all the training samples in the feature space. The distance metric used can be Euclidean, Manhattan, or other distance measures depending on the data and the problem.

Available hyper-parameters:

1. n_neighbors: It is the number of nearest neighbors to use in the classification. The default is 5.
2. weights: It specifies the weight function used in prediction. It can take either 'uniform' for equal weights or 'distance' for weights proportional to the inverse of the distance. The default is 'uniform'.
3. algorithm: It specifies the algorithm used to compute the nearest neighbors. It can take either 'ball_tree', 'kd_tree', 'brute', or 'auto'. The default is 'auto'.
4. leaf_size: It sets the leaf size of the ball_tree or kd_tree algorithms. The default is 30.

5. p: It specifies the power parameter for the Minkowski metric used for distance computation. When p=1, this corresponds to the Manhattan distance, and when p=2, this corresponds to the Euclidean distance. The default is 2.
6. metric: It specifies the distance metric used for nearest neighbor search. It can take various distance metrics such as 'euclidean', 'manhattan', 'minkowski', 'chebyshev', and more. The default is 'minkowski'.
7. n_jobs: It specifies the number of CPU cores to use for the computation. The default is 1, which means using a single core.

Support Vector Machine: It works by finding the optimal hyperplane that separates the classes in the data, while maximizing the margin between the hyperplane and the data points. The margin is defined as the distance between the hyperplane and the closest data points from each class. In cases where the classes are not linearly separable, the SVM algorithm can use a technique called kernel trick, where it maps the input data into a higher-dimensional feature space where a linear decision boundary can be found.

Available hyper-parameters:

1. C: It is the regularization parameter that determines the trade-off between maximizing the margin and minimizing the classification error. The smaller the value of C, the stronger the regularization. The default is 1.0.
2. kernel: It specifies the kernel function used in the SVM algorithm. It can take various kernel functions such as 'linear', 'poly', 'rbf', 'sigmoid', and more. The default is 'rbf'.
3. degree: It is the degree of the polynomial kernel function. It is used when the kernel is 'poly'. The default is 3.
4. gamma: It is the kernel coefficient for 'rbf', 'poly', and 'sigmoid'. The default is 'scale', which is computed as 1 / (n_features * X.var()).
5. coef0: It is the independent term in the kernel function. It is used in 'poly' and 'sigmoid'. The default is 0.0.
6. shrinking: It specifies whether to use the shrinking heuristic. The default is True.
7. tol: It sets the tolerance for stopping criteria of the solver. The default is 1e-3.
8. cache_size: It sets the size of the kernel cache in MB. The default is 200.
9. class_weight: It is used to handle imbalanced classes by assigning weights to each class. It can take either 'balanced' or a dictionary of weights. The default is None.
10. decision_function_shape: It specifies the shape of the decision function. It can take either 'ovr' for one-vs-rest or 'ovo' for one-vs-one. The default is 'ovr'.
11. max_iter: It sets the maximum number of iterations for the solver to converge. The default is -1, which means no limit.
12. random_state: It is used to seed the random number generator for reproducible results. The default is None.

Decision Tree Classifier: It is a supervised algorithm to predict the class of a new input based on a decision tree built from the training data. In a binary classification problem, the algorithm works by creating a tree structure that recursively splits the data into two classes based on the values of the input features. During training, the algorithm starts with a root node that represents the entire training dataset. It then selects the best feature to split the data based on a criterion such as information gain or Gini impurity.

Available hyper-parameters:

1. criterion: It specifies the function used to measure the quality of a split. It can take either 'gini' for Gini impurity or 'entropy' for information gain. The default is 'gini'.
2. splitter: It specifies the strategy used to choose the split at each node. It can take either 'best' for the best split or 'random' for a random split. The default is 'best'.
3. max_depth: It is the maximum depth of the tree. The default is None, which means the nodes are expanded until all the leaves contain less than min_samples_split samples.
4. min_samples_split: It is the minimum number of samples required to split an internal node. The default is 2.
5. min_samples_leaf: It is the minimum number of samples required to be at a leaf node. The default is 1.

6. min_weight_fraction_leaf: It is the minimum weighted fraction of the sum total of the weights required to be at a leaf node. The default is 0.0.
7. max_features: It specifies the number of features to consider when looking for the best split. It can take either an integer, 'sqrt', 'log2', or None. The default is None.
8. random_state: It is used to seed the random number generator for reproducible results. The default is None.
9. max_leaf_nodes: It is the maximum number of leaf nodes in the tree. The default is None, which means unlimited.
10. min_impurity_decrease: It is the minimum impurity decrease required for a split to occur. The default is 0.0.
11. class_weight: It is used to handle imbalanced classes by assigning weights to each class. It can take either 'balanced' or a dictionary of weights. The default is None.

Random Forest Classifier: It is an ensemble algorithm that combines multiple decision tree classifiers to improve the accuracy and reduce the variance of the model. In a binary classification problem, the algorithm works by randomly selecting a subset of the features and a subset of the training data for each decision tree in the forest. This helps to reduce the correlation between the trees and improve the diversity of the ensemble.

Available hyper-parameters:
1. n_estimators: The number of trees in the forest. Default value is 100.
2. criterion: The function to measure the quality of a split. The supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Default value is "gini".
3. max_depth: The maximum depth of the tree. Default value is None, which means nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
4. min_samples_split: The minimum number of samples required to split an internal node. Default value is 2.
5. min_samples_leaf: The minimum number of samples required to be at a leaf node. Default value is 1.
6. max_features: The number of features to consider when looking for the best split. Supported values are "auto", "sqrt", "log2", or a float between 0 and 1. Default value is "auto", which means all features are considered.
7. max_leaf_nodes: The maximum number of leaf nodes in the tree. Default value is None, which means unlimited number of leaf nodes.
8. min_impurity_decrease: A node will be split if this split induces a decrease of the impurity greater than or equal to this value. Default value is 0.0.
9. bootstrap: Whether or not to use bootstrap samples when building trees. Default value is True.
10. random_state: The seed used by the random number generator.

Task 3: Evaluation

After we determine the type of classifier to use for test set prediction, we will generate all the metrics and confusion matrix for each model. The results are as follows;

Logistic Regression:
Training time = 0.0054628849029541016
Validation and Test time = 0.0009827613830566406
Training Accuracy = 0.8272980501392758

```
              precision    recall  f1-score   support

         0.0       0.80      0.80      0.80       315
         1.0       0.85      0.85      0.85       403

    accuracy                           0.83       718
   macro avg       0.82      0.82      0.82       718
weighted avg       0.83      0.83      0.83       718
```

```
Test Accuracy = 0.825
              precision    recall  f1-score   support

         0.0       0.82      0.81      0.81        95
         1.0       0.83      0.84      0.83       105

    accuracy                           0.82       200
   macro avg       0.82      0.82      0.82       200
weighted avg       0.82      0.82      0.82       200

Bias: 0.17270194986072418
Variance: 0.0022980501392758645
```

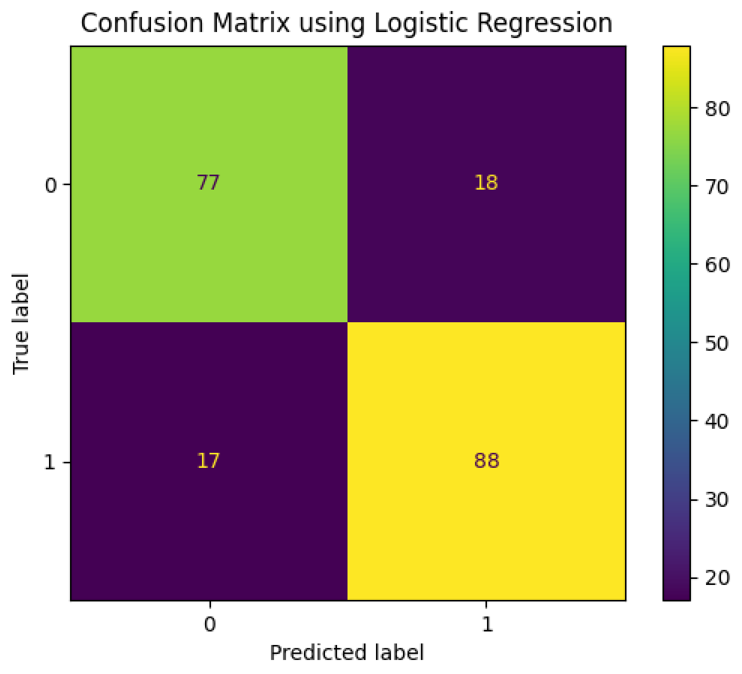


Figure 7: Confusion matrix and metrics using Logistic Regression.

```
SGDCClassifier L2:
Training time = 0.0019838809967041016
Validation and Test time = 0.0006787776947021484
Training Accuracy = 0.8119777158774373
              precision    recall  f1-score   support

         0.0       0.82      0.73      0.77       315
         1.0       0.80      0.88      0.84       403

    accuracy                           0.81       718
   macro avg       0.81      0.80      0.81       718
weighted avg       0.81      0.81      0.81       718

Test Accuracy = 0.815
```

```
              precision    recall  f1-score   support

         0.0       0.86      0.73      0.79        95
         1.0       0.78      0.90      0.84       105

    accuracy                           0.81       200
   macro avg       0.82      0.81      0.81       200
weighted avg       0.82      0.81      0.81       200

Bias: 0.18802228412256272
Variance: -0.0030222841225626684
```

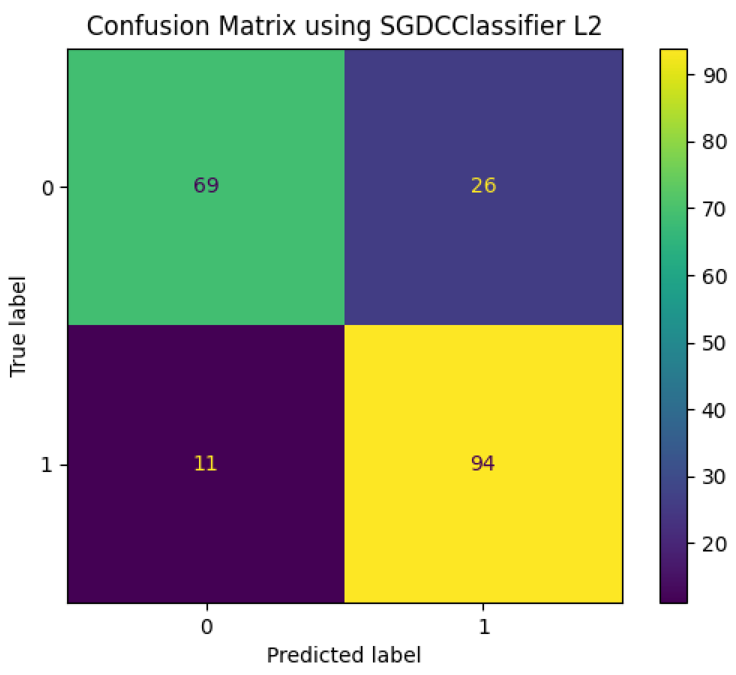


Figure 8: Confusion matrix and metrics using SGDCClassifier L2.

```
SGDCClassifier L1:
Training time = 0.0025501251220703125
Validation and Test time = 0.0006730556488037109
Training Accuracy = 0.8286908077994429
              precision    recall  f1-score   support

         0.0       0.84      0.75      0.79       315
         1.0       0.82      0.89      0.85       403

    accuracy                           0.83       718
   macro avg       0.83      0.82      0.82       718
weighted avg       0.83      0.83      0.83       718

Test Accuracy = 0.795
              precision    recall  f1-score   support
         0.0       0.81      0.75      0.78        95
         1.0       0.79      0.84      0.81       105

    accuracy                           0.80       200
   macro avg       0.80      0.79      0.79       200
weighted avg       0.80      0.80      0.79       200

Bias: 0.17130919220055707
Variance: 0.03369080779944289
```

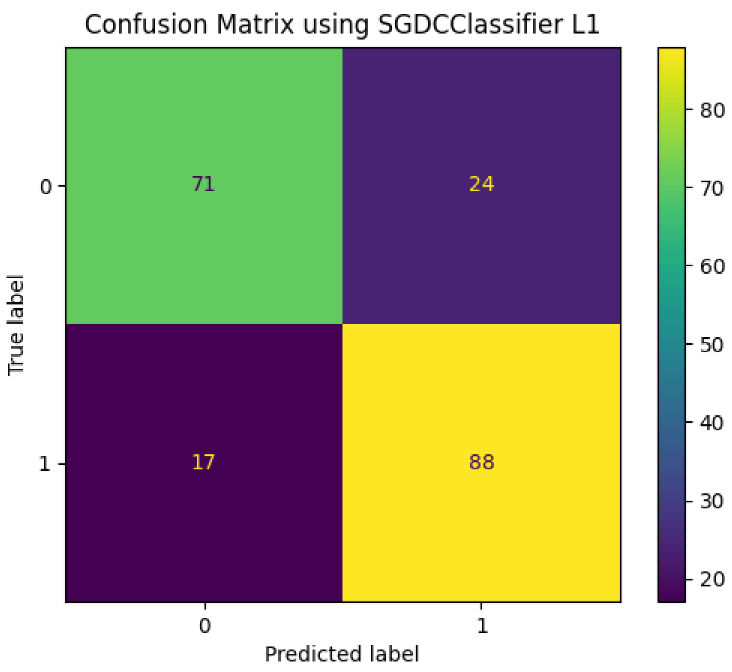


Figure 9: Confusion matrix and metrics using SGDCClassifier L1.

```
Ridge Classifier:
Training time = 0.004405021667480469
Validation and Test time = 0.0010349750518798828
Training Accuracy = 0.8189415041782729
              precision    recall  f1-score   support

         0.0       0.78      0.82      0.80       315
         1.0       0.85      0.82      0.84       403

    accuracy                           0.82       718
   macro avg       0.82      0.82      0.82       718
weighted avg       0.82      0.82      0.82       718

Test Accuracy = 0.84
              precision    recall  f1-score   support

         0.0       0.82      0.85      0.84        95
         1.0       0.86      0.83      0.84       105
    accuracy                           0.84       200
   macro avg       0.84      0.84      0.84       200
weighted avg       0.84      0.84      0.84       200

Bias: 0.18105849582172706
Variance: -0.02105849582172703
```

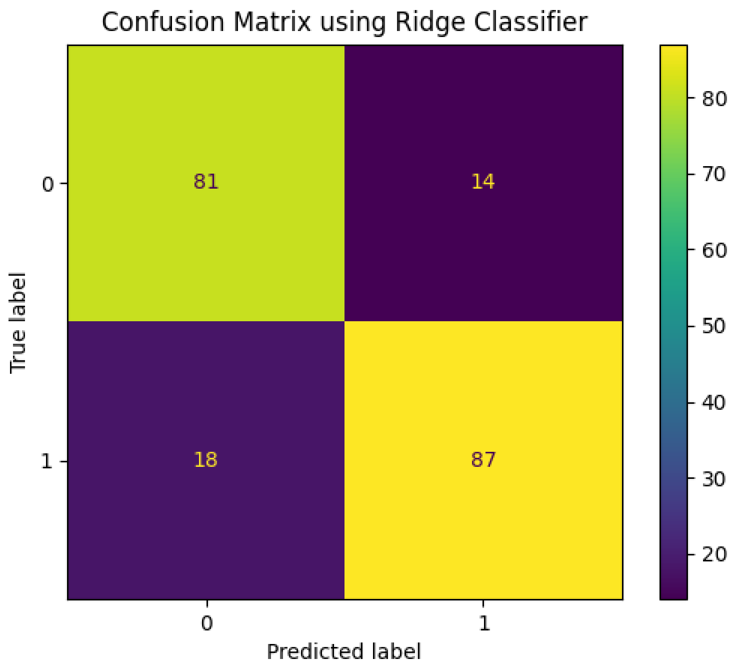


Figure 10: Confusion matrix and metrics using Ridge Classifier.

```
Perceptron:
Training time = 0.001180887222290039
Validation and Test time = 0.0006406307220458984
Training Accuracy = 0.8050139275766016
              precision    recall  f1-score   support

         0.0       0.78      0.77      0.78       315
         1.0       0.82      0.83      0.83       403

    accuracy                           0.81       718
   macro avg       0.80      0.80      0.80       718
weighted avg       0.80      0.81      0.80       718

Test Accuracy = 0.8
              precision    recall  f1-score   support

         0.0       0.81      0.76      0.78        95
         1.0       0.79      0.84      0.81       105

    accuracy                           0.80       200
   macro avg       0.80      0.80      0.80       200
weighted avg       0.80      0.80      0.80       200
```

Bias: 0.19498607242339838
Variance: 0.005013927576601573

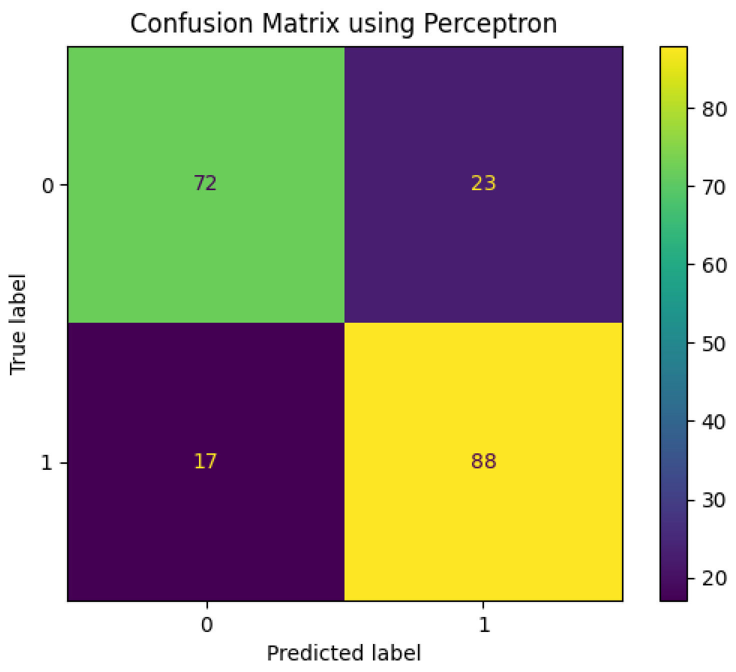


Figure 11: Confusion matrix and metrics using Perceptron.

```
BernoulliNB:
Training time = 0.0010740756988525391
Validation and Test time = 0.0007970333099365234
Training Accuracy = 0.8217270194986073
              precision    recall  f1-score   support

         0.0       0.79      0.80      0.80       315
         1.0       0.84      0.84      0.84       403

    accuracy                           0.82       718
   macro avg       0.82      0.82      0.82       718
weighted avg       0.82      0.82      0.82       718

Test Accuracy = 0.85
              precision    recall  f1-score   support

         0.0       0.84      0.84      0.84        95
         1.0       0.86      0.86      0.86       105

    accuracy                           0.85       200
   macro avg       0.85      0.85      0.85       200
weighted avg       0.85      0.85      0.85       200

Bias: 0.17827298050139273
Variance: -0.028272980501392708
```

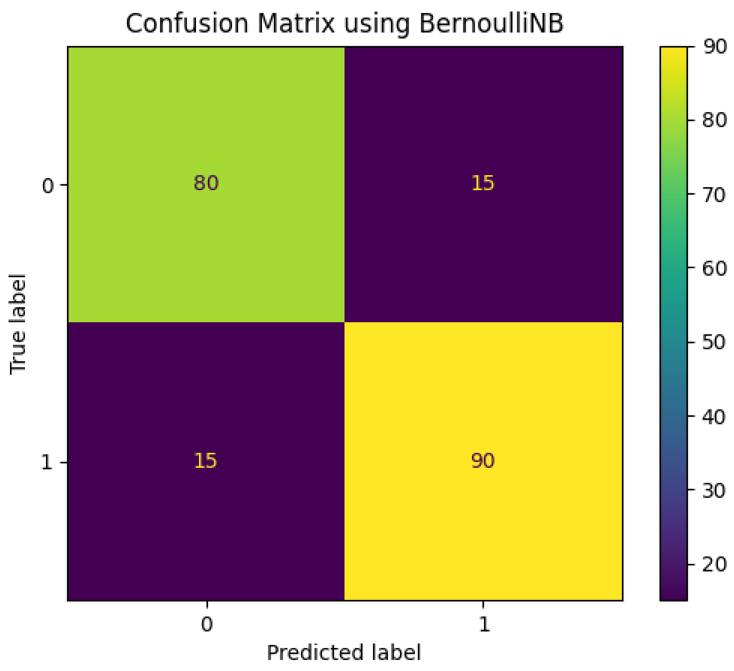


Figure 12: Confusion matrix and metrics using BernoulliNB.

```
K-Nearest Neighbor:
Training time = 0.0025551319122314453
Validation and Test time = 0.010808229446411133
Training Accuracy = 0.8732590529247911
              precision    recall  f1-score   support

         0.0       0.88      0.83      0.85       315
         1.0       0.87      0.91      0.89       403

    accuracy                           0.87       718
   macro avg       0.87      0.87      0.87       718
weighted avg       0.87      0.87      0.87       718

Test Accuracy = 0.85
              precision    recall  f1-score   support

         0.0       0.87      0.81      0.84        95
         1.0       0.84      0.89      0.86       105

    accuracy                           0.85       200
   macro avg       0.85      0.85      0.85       200
weighted avg       0.85      0.85      0.85       200

Bias: 0.1267409470752089
Variance: 0.023259052924791135
```

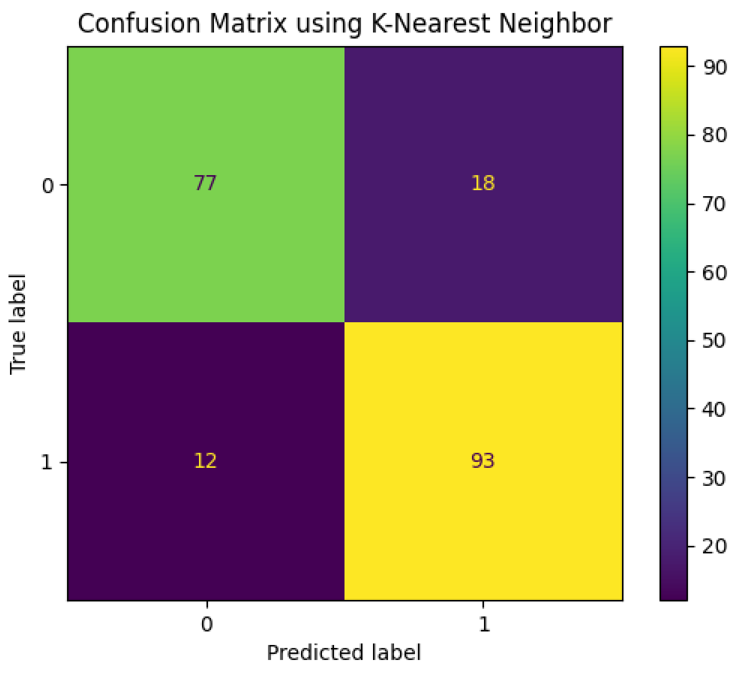


Figure 13: Confusion matrix and metrics using K-Nearest Neighbor.

```
Support Vector Machine:
Training time = 0.030160903930664062
Validation and Test time = 0.01674962043762207
Training Accuracy = 0.883008356545961
              precision    recall  f1-score   support

         0.0       0.90      0.83      0.86       315
         1.0       0.87      0.93      0.90       403

    accuracy                           0.88       718
   macro avg       0.89      0.88      0.88       718
weighted avg       0.88      0.88      0.88       718

Test Accuracy = 0.85
              precision    recall  f1-score   support

         0.0       0.87      0.80      0.84        95
         1.0       0.83      0.90      0.86       105

    accuracy                           0.85       200
   macro avg       0.85      0.85      0.85       200
weighted avg       0.85      0.85      0.85       200

Bias: 0.11699164345403901
Variance: 0.033008356545961015
```

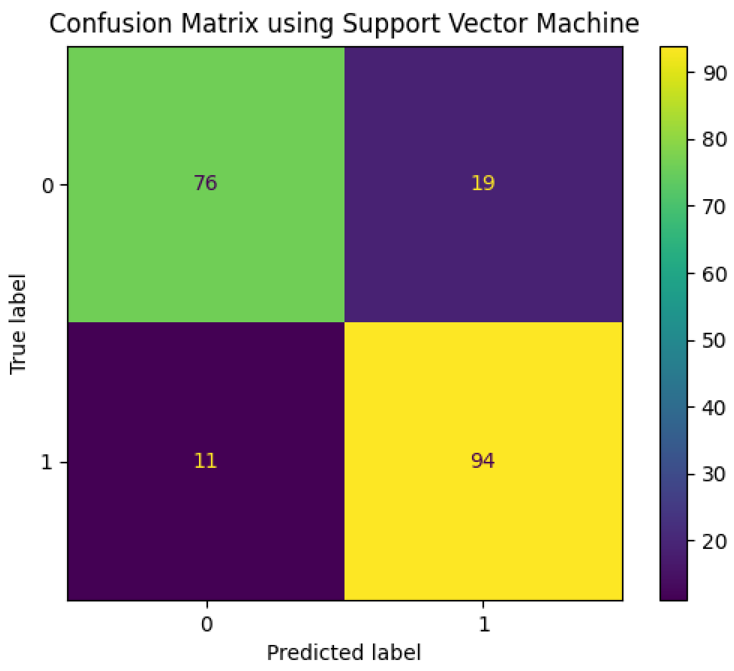


Figure 14: Confusion matrix and metrics using Support Vector Machine.

```
Decision Tree Classifier:
Training time = 0.002073049545288086
Validation and Test time = 0.0011370182037353516
Training Accuracy = 1.0
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       315
         1.0       1.00      1.00      1.00       403

    accuracy                           1.00       718
   macro avg       1.00      1.00      1.00       718
weighted avg       1.00      1.00      1.00       718

Test Accuracy = 0.79
              precision    recall  f1-score   support

         0.0       0.77      0.79      0.78        95
         1.0       0.81      0.79      0.80       105

    accuracy                           0.79       200
   macro avg       0.79      0.79      0.79       200
weighted avg       0.79      0.79      0.79       200

Bias: 0.0
Variance: 0.20999999999999996
```

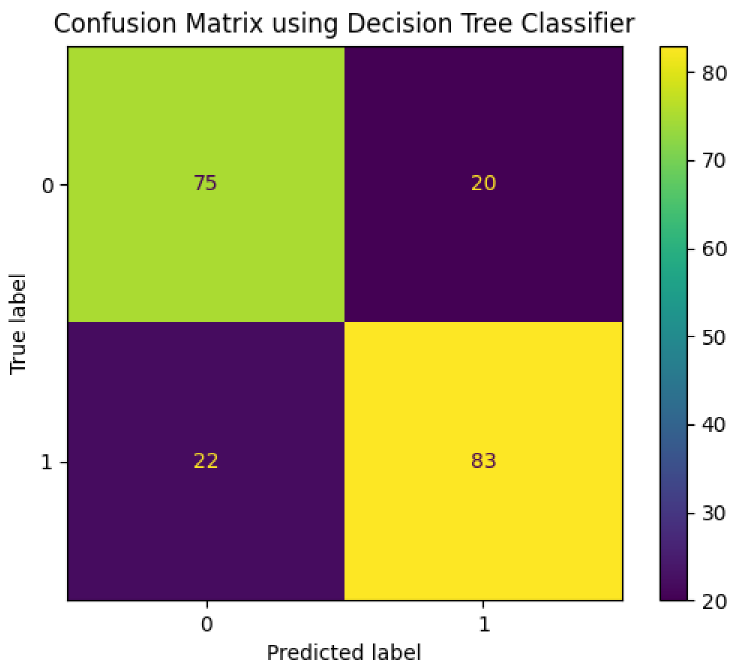


Figure 15: Confusion matrix and metrics using Decision Tree Classifier.

```
Random Forest Classifier:
Training time = 0.07437586784362793
Validation and Test time = 0.012320995330810547
Training Accuracy = 1.0
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       315
         1.0       1.00      1.00      1.00       403

    accuracy                           1.00       718
   macro avg       1.00      1.00      1.00       718
weighted avg       1.00      1.00      1.00       718

Test Accuracy = 0.85
              precision    recall  f1-score   support

         0.0       0.86      0.82      0.84        95
         1.0       0.84      0.88      0.86       105

    accuracy                           0.85       200
   macro avg       0.85      0.85      0.85       200
weighted avg       0.85      0.85      0.85       200

Bias: 0.0
Variance: 0.15000000000000002
```

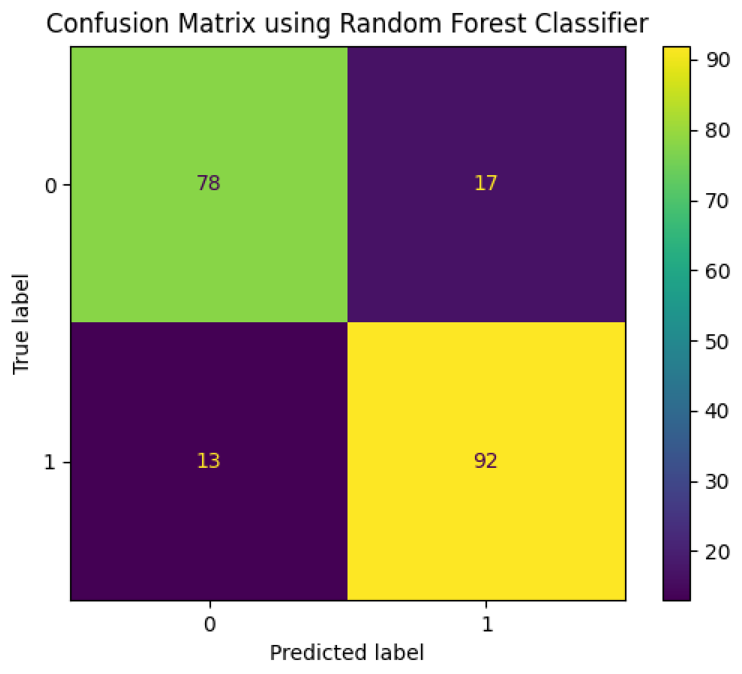


Figure 16: Confusion matrix and metrics using Random Forest Classifier.

Here is a quick summary of the performance metrics:

Table 2: Metrics of performance for each machine learning model

Model name	Test accuracy	Test positive F1-score	Bias	Variance
Logistic Regression	0.825	0.83	0.17270194986072418	0.0022980501392758645
SGDClassifier L2	0.815	0.84	0.18802228412256272	-0.0030222841225626684
SGDClassifier L1	0.795	0.81	0.17130919220055707	0.03369080779944289
Ridge Classifier	0.840	0.84	0.18105849582172706	-0.02105849582172703
Perceptron	0.800	0.81	0.19498607242339838	0.005013927576601573
BernoulliNB	0.850	0.86	0.17827298050139273	-0.028272980501392708
K-Nearest Neighbor	0.850	0.86	0.1267409470752089	0.023259052924791135
Support Vector Machine	0.850	0.86	0.11699164345403901	0.033008356545961015
Decision Tree Classifier	0.790	0.80	0	0.20999999999999996
Random Forest Classifier	0.85	0.86	0	0.15000000000000002

Both test accuracy and F1-score can be used to evaluate the performance of a model. The accuracy is the percentage of all correctly classified observations, while F1-score is the harmonic mean of precision and recall. A analysis of the pros and cons using F1-score and test accuracy are as below:

Table 3: Pros and cons of two performance metrics test accuracy vs F1-score

	Pros	Cons
Test accuracy	Easy to interpret	Does not consider how the data is distributed
F1-score	Consider how the data is distributed even when the data is highly imbalanced	Harder to interpret because it is a mixture of precision and recall

Both tables demonstrated that when the data was balanced, the test accuracy and F1-score exhibited nearly identical values. However, utilizing F1-score accounted for data distribution, providing a more definitive evaluation of model performance. As both scores displayed comparable values across all models, it was apparent that the dataset was balanced, allowing for the use of either metric to evaluate model performance.

Analysis of the F1-scores indicated that the Support Vector Machine, Random Forest Classifier, K-Nearest Neighbor, and Logistic Regression models were the top four performing models. To further explore these models, we generated ROC plots and scores.

The analysis of model performance on the heart disease dataset revealed a low bias and low variance nature of the data. The decision tree classifier exhibited the highest bias of 0.1949, indicating that the model did not underfit the training data and was a good representation of the dataset. It was able to capture the underlying patterns in the data without oversimplifying them.

Furthermore, the low variance values for all models suggested that they were not overly complex and were not overfitting the training data, a desirable characteristic of a well-performing model.

However, the test set accuracies for all models were below 90%, suggesting that there was still room for improvement. One way to achieve this was by adding more data to the dataset, which could help the model learn more generalized rules for prediction. By doing so, we could potentially yield more accurate results. In summary, the heart disease dataset exhibited low bias and low variance, and the models were not overfitting or oversimplifying the data. However, adding more data could potentially improve the accuracy of the models on the test set.

A Receiver Operating Characteristic (ROC) curve illustrates the performance of a binary classification model as the discrimination threshold is varied. The ROC curve is created by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) for different values of the threshold. Below are the ROC curves plotted for the four selected models:

Support Vector Machine:

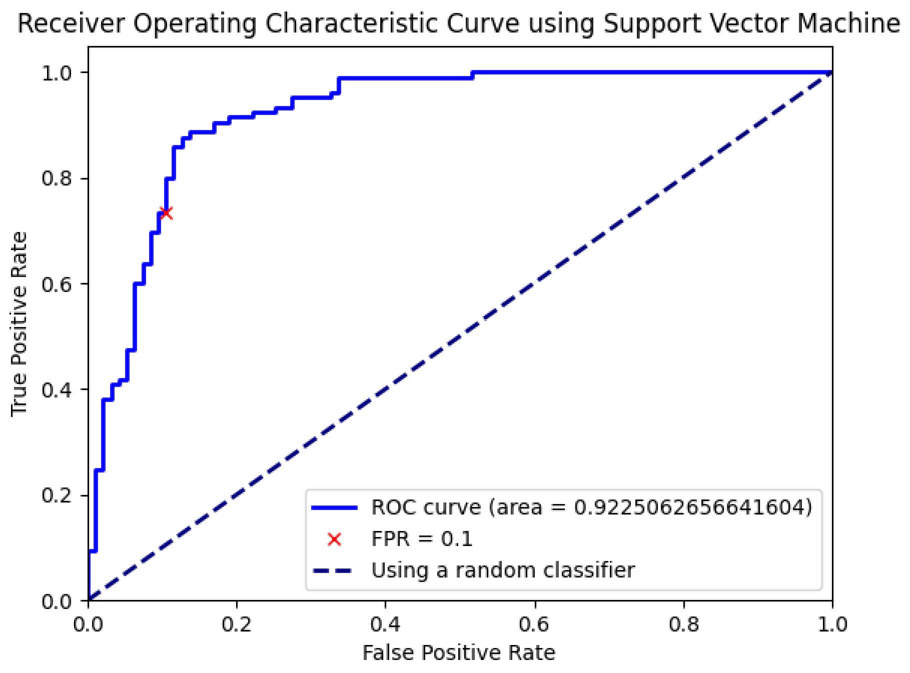


Figure 17: ROC curve using Support Vector Machine

Random Forest Classifier:

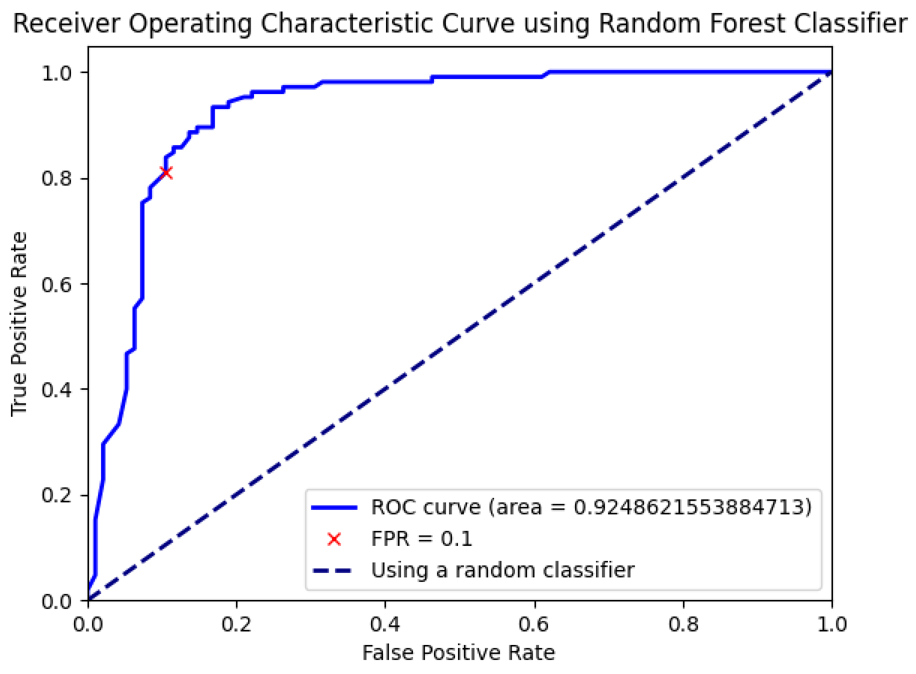


Figure 18: ROC curve using Random Forest Classifier

K-Nearest Neighbor:

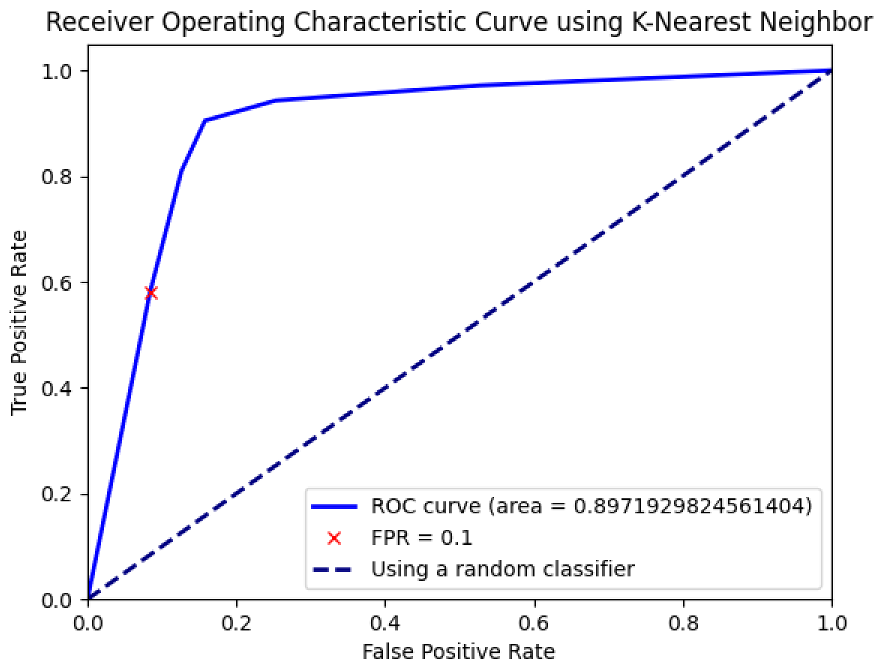


Figure 19: ROC curve using K-Nearest Neighbor

Logistic Regression:

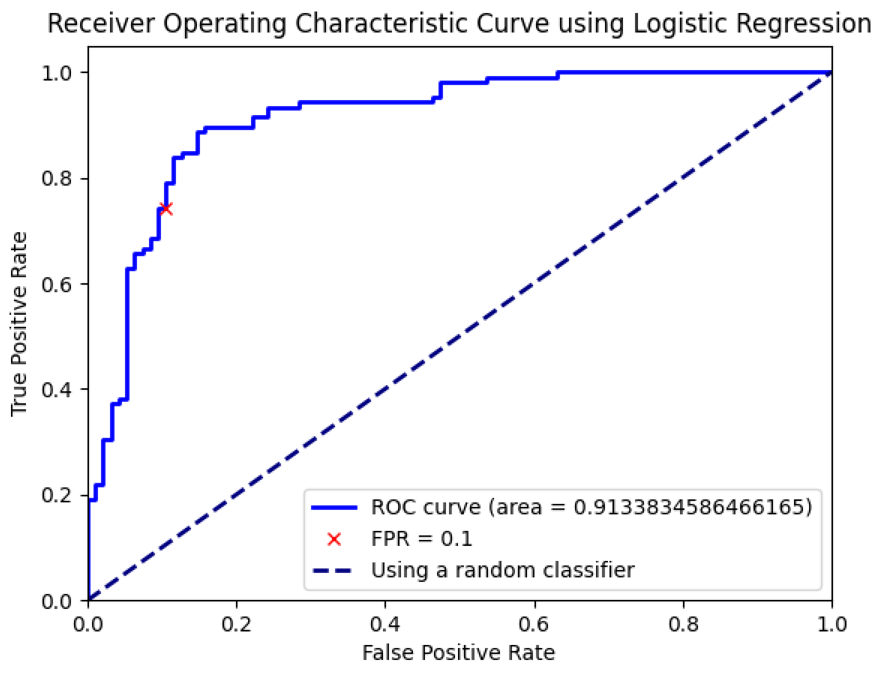


Figure 20: ROC curve using Logistic Regression

As we were dealing with the prediction of a medical condition, specifically heart disease, we exercised caution in selecting an appropriate operating point. To achieve our objective of minimizing false positives, which can result in the incorrect classification of healthy individuals as having heart disease, we needed to select a False Positive Rate (FPR) threshold that did not exceed 0.1. By selecting this FPR value, we were able to evaluate the True Positive Rate (TPR) of each model. The TPR values for each model are listed below.

Table 4: ROC-AUC of each model (part of extension)

	ROC-AUC
Logistic Regression	0.91338
K-Nearest Neighbor	0.89719
Support Vector Machine	0.92251
Random Forest Classifier	0.82486

Table 5: TPR of each model using FPR=0.1 as the operating point.

	TPR
Logistic Regression	0.7428571428571429
K-Nearest Neighbor	0.580952380952381
Support Vector Machine	0.7333333333333333
Random Forest Classifier	0.8095238095238095

Upon evaluating the True Positive Rate (TPR) of each model, we determined that both the Random Forest Classifier and K-Nearest Neighbor models exhibited similar accuracy in predicting heart disease, achieving high TPRs and low FPRs. As a result, we focused on tuning these two models further to achieve even more accurate and high-performing results.

Task 4: Iteration

As heart disease prediction requires a balance between high TPRs and low FPRs, our goal during the training iterations was to identify the best model that achieved the highest TPR at an FPR of 0.1. In some cases, a trade-off is required between achieving a higher TPR and balancing the two metrics. Therefore, it was important to ensure that our selected model was optimized for both high TPRs and low FPRs to achieve the best overall performance in predicting heart disease.

K-Nearest Neighbor:
Original model:
We will look at the K-Nearest Neighbor first. Originally, the KNeighborClassifier was using the default hyperparameter. Based on the scikit-learn library documentation, the default hyperparameter for KNeighborClassifier are:

- n_neighbors: 5
- weights: uniform
- algorithm: auto
- leaf_size: 30
- p: 2

The accuracy computed using KNeighborsClassifier is 87.5%
The true positive rate at fpr=0.1 is: 0.580952380952381

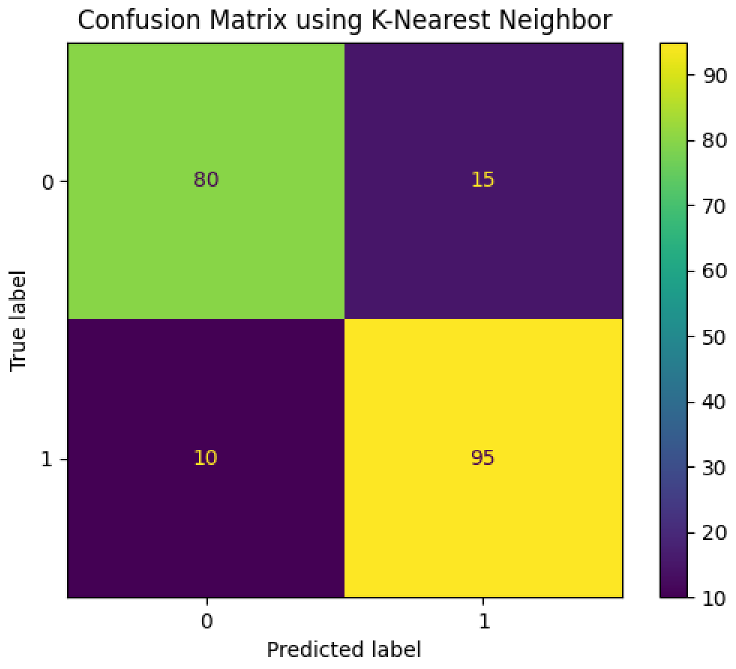


Figure 22: Original run confusion matrix using K-Nearest Neighbor classifier.

First iteration:
By increasing the number of neighbors utilized by KNeighborsClassifier, we aimed to enhance the TPR at a lower FPR rate. This was due to the fact that, as the n_neighbors value increased, the decision boundary became smoother and more regular. This allowed the classifier to capture more general patterns in the data, resulting in improved model performance. Additionally, utilizing a smaller leaf size resulted in a more balanced tree with a reduced depth. This change could affect the exact nearest neighbors that were identified during prediction.

- n_neighbors: 7
- weights: uniform
- algorithm: auto
- leaf_size: 15
- p: 2

The accuracy computed using KNeighborsClassifier is 87.0%

The true positive rate at fpr=0.1 is: 0.7333333333333333

Based on the significant increase in TPR from 0.58095 to 0.73333, we could tell that the model's performance at FPR was significantly enhanced. However, the test accuracy remained relatively consistent. These results indicate that the K-Nearest Neighbor model was successful in achieving our objective of improving the performance of heart disease prediction at a low FPR rate.

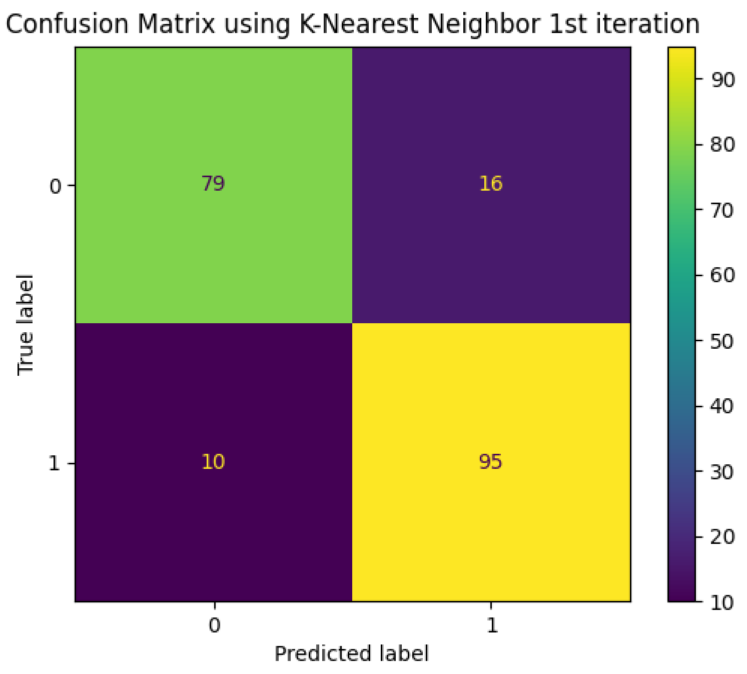


Figure 23: 1st iteration run confusion matrix using K-Nearest Neighbor classifier.

Second iteration:
For the second iteration, I maintained the n_neighbors and leaf_size hyperparameters while introducing a new metric called the Mahalanobis metric. This metric accounted for the covariance between different features, allowing for a more effective capture of the relationships between features and better adjustment for various scales and correlations. By implementing this new metric, we aimed to enhance the performance of the model in predicting heart disease.

- n_neighbors: 7
- weights: uniform
- algorithm: auto
- leaf_size: 15
- metric: mahalanobis
- metric_params: the covariance matrix
- p: 2

The accuracy computed using KNeighborsClassifier is 85.5%
The true positive rate at fpr=0.1 is: 0.7333333333333333

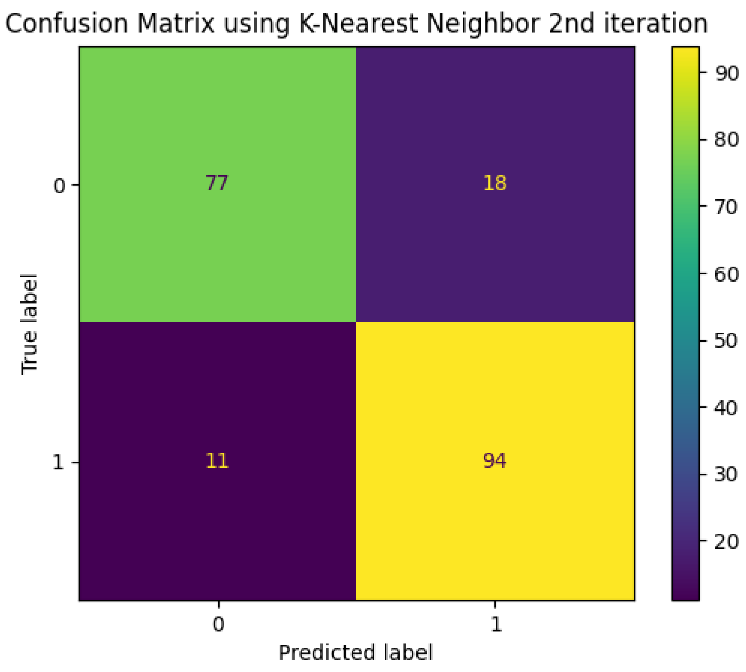


Figure 24: 2nd iteration run confusion matrix using K-Nearest Neighbor classifier.

Regrettably, this particular iteration did not lead to any improvements in the results, which could be due to the TPR and FPR thresholds we aimed to achieve in this particular scenario. To gain a better understanding of the actual performance, we may need to calculate the AUC-ROC metric. However, the overall test accuracy decreased by 2% from the first iteration, which suggests that when TPR is the control factor, the performance was not as strong as in the previous iteration.

Third iteration:
We will use an ensemble of KNeighborsClassifier and use voting techniques to come up with a more robust and generalized model.
combinations:
n_neighbors=5, leaf_size=15, weights='uniform',
n_neighbors=5, leaf_size=20, weights='uniform',
n_neighbors=7, leaf_size=15, weights='uniform'

As demonstrated above, we generated a list of models using various combinations of hyperparameters and passed them to the VotingClassifier. This classifier then utilized the average results of the individual models to generate the final predicted probabilities of the ensemble.

The accuracy computed using VotingClassifier is 87.5%
The true positive rate at fpr=0.1 is: 0.7333333333333333

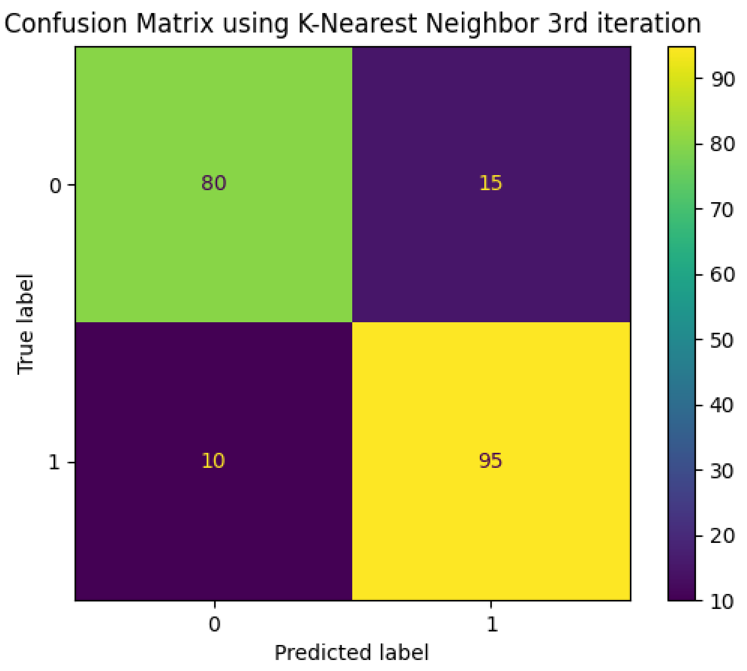


Figure 25: 3rd iteration run confusion matrix using K-Nearest Neighbor classifier.

After three iterations:
When considering the number of true positive predictions, both our original and third iteration models performed well by correctly identifying 80 positive cases. However, since we prioritize using the operating point at a low FPR rate that gives the highest TPR rate, the third iteration model performed the best. In comparison, the best K-Nearest Neighbor model that we can construct at an FPR of 0.1 is able to achieve a TPR of 0.733.

Random Forest Classifier:
Original model:
At the beginning of our analysis, the RandomForestModel utilized the default hyperparameters. According to the scikit-learn library, these default parameters are as follows:

- n_estimators: 100
- criterion: gini
- max_depth: None
- min_samples_split: 2
- min_samples_leaf: 1
- min_weight_fraction_leaf: 0.0
- max_features: sqrt
- max_leaf_nodes: None

The accuracy computed using RandomForestClassifier is 86.5%
The true positive rate at fpr=0.1 is: 0.8095238095238095

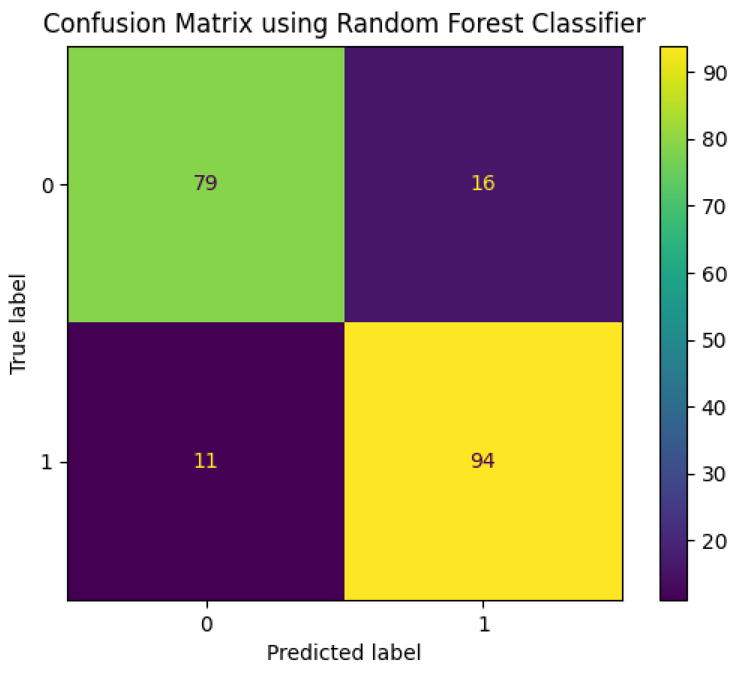


Figure 26: Original run confusion matrix using random forest classifier.

First iteration:
To begin the process of improving our model, we selected a max_depth of 5 as a starting point for our experimentation with fitting on the training set. Typically, setting a low max_depth value may lead to underfitting, where the model is too simplistic and unable to make accurate predictions on the test set. The hyperparameters that we will use are as follows:

- n_estimators: 100
- criterion: gini
- max_depth: 5
- min_samples_split: 2
- min_samples_leaf: 1
- min_weight_fraction_leaf: 0.0
- max_features: sqrt
- max_leaf_nodes: None

The accuracy computed using RandomForestClassifier is 87.0%
The true positive rate at fpr=0.1 is: 0.8

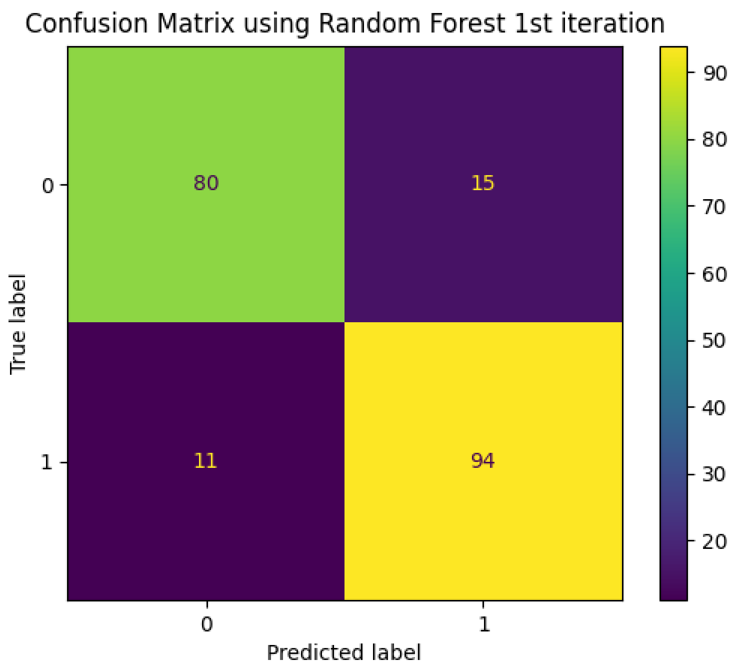


Figure 26: 1st iteration run confusion matrix using random forest classifier.

Based on the results, it appears that the first iteration using max_depth=5 did not improve the model's performance, potentially due to underfitting the data. Although the test accuracy did slightly increase by 0.5%, this improvement was not significant enough to provide us with enough confidence to make a definitive conclusion regarding the performance of this model.

Second iteration:
To improve the generalization performance of the model and prevent overfitting, we will adjust two hyperparameters. First, we will increase the max_depth value to 10, which will result in a deeper tree that can learn more complex patterns in the data. Second, we will increase the min_samples_split from the default 2 to 4, which controls the minimum number of samples required to split an internal node in the tree. Increasing this value will lead to a more shallow tree, which can improve the model's ability to generalize to new data.

- n_estimators: 100
- criterion: gini
- max_depth: 10
- min_samples_split: 4
- min_samples_leaf: 1
- min_weight_fraction_leaf: 0.0
- max_features: sqrt
- max_leaf_nodes: None

The accuracy computed using RandomForestClassifier is 88.5%
The true positive rate at fpr=0.1 is: 0.8380952380952381

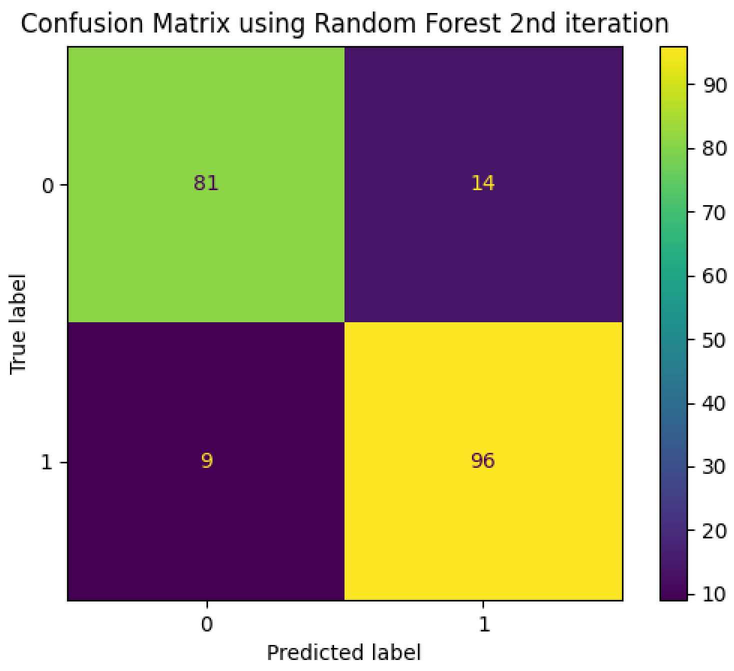


Figure 26: 2nd iteration run confusion matrix using random forest classifier.

Fortunately, we were able to improve the model performance by increasing the min_samples_split and using slightly deeper trees, which resulted in a substantial increase in TPR at an FPR of 0.1. This improvement was considerable, providing a 88.5% test accuracy and a TPR of 0.8381 at FPR=0.1, indicating that the model is highly reliable and provides our predictions with a high level of confidence.

Third iteration:
Next, we attempted to improve performance by reducing the number of decision trees in our random forest model and testing a different max_features function. Reducing the number of trees can potentially reduce the variance in the model, but it may also decrease model diversity, leading to a decline in overall performance. We also explored changing the max_features from 'sqrt' to None. Typically, setting this value to None implies considering all features for the best split, which could result in overfitting. However, since our dataset only includes eight features, we decided to use all the features to promote more robust learning on the dataset.

- n_estimators: 100
- criterion: gini
- max_depth: 10
- min_samples_split: 4
- min_samples_leaf: 1
- min_weight_fraction_leaf: 0.0
- max_features: None
- max_leaf_nodes: None

The accuracy computed using RandomForestClassifier is 86.0%
The true positive rate at fpr=0.1 is: 0.7904761904761904

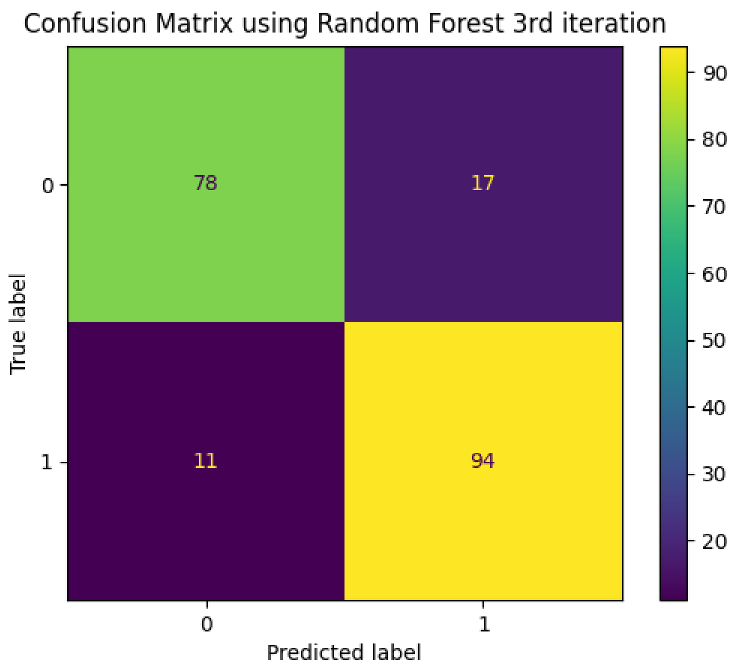


Figure 26: 3rd iteration run confusion matrix using random forest classifier.

Regrettably, this did not help and did not improve performance at low FPR.

After three iterations:
When considering the number of true positive predictions, our second iteration model outperforms the other models by correctly identifying 81 positive cases. Additionally, this model provides the highest true positive rate (TPR) at a false positive rate (FPR) of 0.1. In comparison, the best K-Nearest Neighbor model that we constructed can achieve a TPR of 0.8381 at an FPR of 0.1.

Extensions

Extension 1: Use more ML methods
We evaluated 10 different machine learning classifiers for predicting heart disease and identified the Support Vector Machine, Random Forest Classifier, K-Nearest Neighbor, and Logistic Regression models as the top 4 performers. After considering the tradeoff between computational cost and accuracy at our selected operating point, we concluded that the Random Forest Classifier is the best model for this task. This model offers a low computational cost while achieving outstanding accuracy at low FPR, which can be particularly beneficial for medical evaluations where accuracy is crucial.

Extension 2: Learn and apply some new-preprocessing or feature enhancement methods
In this project, I utilized a new preprocessing technique - one-hot encoding - which converts categorical data into binary vectors. This transformation generates highly discrete numerical data that doesn't obstruct our analysis. Additionally, we attempted to engineer polynomial features for MaxHR, Cholesterol, and other variables, but ultimately decided not to use them as they did not lead to a performance gain, despite conducting some proof-of-concept testing.

Extension 3: Performed three iterations on another classifier and see if the performance increases
We focused on enhancing the performance of our initial model, the K-Nearest Neighbor, and observed some progress in one of the iterations. To explore further opportunities for improvement, we conducted an additional experiment using the Random Forest Classifier. The detailed analysis and outcomes can be found in the Task 4 section.

Extension 4: Generate additional ROC curves and compare the area under the curve for multiple classifier results

We evaluated the performance of four models - Support Vector Machine, Random Forest Classifier, K-Nearest Neighbor, and Logistic Regression - using ROC analysis. The corresponding ROC-AUC values can be found in Table 4, which shows that the Support Vector Machine achieved the highest ROC-AUC score, while the Random Forest Classifier had the lowest. Therefore, the Support Vector Machine exhibited the greatest ability to differentiate between positive and negative categories, while the Random Forest Classifier performed well by achieving the best TPR at a low FPR.

However, we noted that the training dataset may not be optimal, as we discovered missing and incorrect data that could have negatively impacted performance. In the future, we plan to collect a new dataset with more features and fewer errors to retrain our models and improve their performance.

Extension 5: Using feedforward neural network model for prediction
We have built a feedforward neural network to predict heart disease. Some variations of the neural network was conducted to evaluate performance:

1. One input layer with 10 neurons and ReLU, no hidden layer, output layer with sigmoid activation function, trained for 100 epochs, batch size of 32
2. One input layer with 10 neurons and ReLU, 2 hidden layer with 16 and 8 neurons with ReLU, output layer with sigmoid activation function, trained for 100 epochs, batch size of 32
3. One input layer with 10 neurons and ReLU, use dropout rate of 0.2 at each layer, 2 hidden layer with 16 and 8 neurons with ReLU, output layer with sigmoid activation function, trained for 100 epochs, batch size of 32
4. One input layer with 32 neurons and Relu, 2 hidden layer with 16 and 8 neurons with ReLU, output layer with sigmoid activation function, trained for 50 epochs, batch size of 32
5. One input layer with 16 neurons and Relu, 2 hidden layer with 16 and 8 neurons with ReLU, output layer with sigmoid activation function, trained for 50 epochs, batch size of 32

After some proof of concept and iterations, we discovered that the best performing combination is the fifth variation.

Results using neural network:

```
Epoch 50/50
23/23 [==============================] - 0s 7ms/step - loss: 0.3086 - accuracy: 0.8719
2023-02-23 17:47:16.538594: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
7/7 [==============================] - 0s 12ms/step - loss: 0.3463 - accuracy: 0.8500
ANN Accuracy: 84.99999642372131%
2023-02-23 17:47:16.721591: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
7/7 [==============================] - 0s 4ms/step
Feedforward neural network tpr at fpr=0.1 is: 0.8
```

The feedforward neural network shows promising results, achieving a TPR of 0.8 at an FPR of 0.1, which is the same notable improvement compared to the Random Forest Classifier. However, it's important to note that overfitting can occur if we continue to increase the number of iterations on the training set, as this can cause a decrease in accuracy on the test set. Similarly, adding too many layers can also lead to reduced test set accuracy, making it crucial to find the optimal balance between model complexity and generalization performance.

Acknowledgement

I have looked at the documentation for the scikit-learn library, and I have also spoken with my professor to clarify my understanding of bias and variance.