HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

MACHINE LEARNING CAPSTONE PROJECT

HEART DISEASE PREDICTION

Instructor:	Assoc. Prof. Than Quang Khoat
Students:	Doan Minh Viet 20210933
	Do Hoang Tuan 20214939
	Hoang Tu Quyen 20214929
	Nguyen Ba Duong 20214886
	Nguyen Viet Trung 20214934

Hanoi, June 2023

Table of Contents

Abstract

This report investigates the performance of different machine learning (ML) algorithms for heart disease prediction tasks. We analyze and compare the results obtained from five popular algorithms: Logistic Regression, Support Vector Machines (SVM), Random Forest, K Nearest Neighbors (KNN), and Artificial Neural Network (ANN).

To conduct our study, we use a dataset consisting of 729 labeled records and 11 features related to heart health. We begin by performing data preprocessing techniques, including data transformation and handling missing values. The dataset is then split into a training set and a test set in a 75% : 25% ratio.

Next, we train the ML models (Logistic Regression, SVM, KNN, Random Forest, ANN) on the training set and evaluate their performance on the test set. To find the optimal hyperparameters for each algorithm, we employ Grid Search, a method which systematically explores a range of parameter combinations.

After training and prediction, we utilize various evaluation metrics, including accuracy, precision, recall, F1 score, and confusion matrix, to assess the performance of the heart disease prediction models.

I. Introduction

1. Context

Cardiovascular diseases (CVDs) are the leading cause of death globally, responsible for 32% of all deaths (in 2019). Four out of 5 CVD deaths are due to heart attacks and strokes, and one-third of these deaths occur prematurely in people under 70 years of age. Heart failure is a common event caused by CVDs.

People with cardiovascular disease or who are at high cardiovascular risk (due to the presence of one or more risk factors such as hypertension, diabetes, hyperlipidaemia or already established disease) need early detection and management wherein a machine learning model can be of great help.

2. Attribute Information

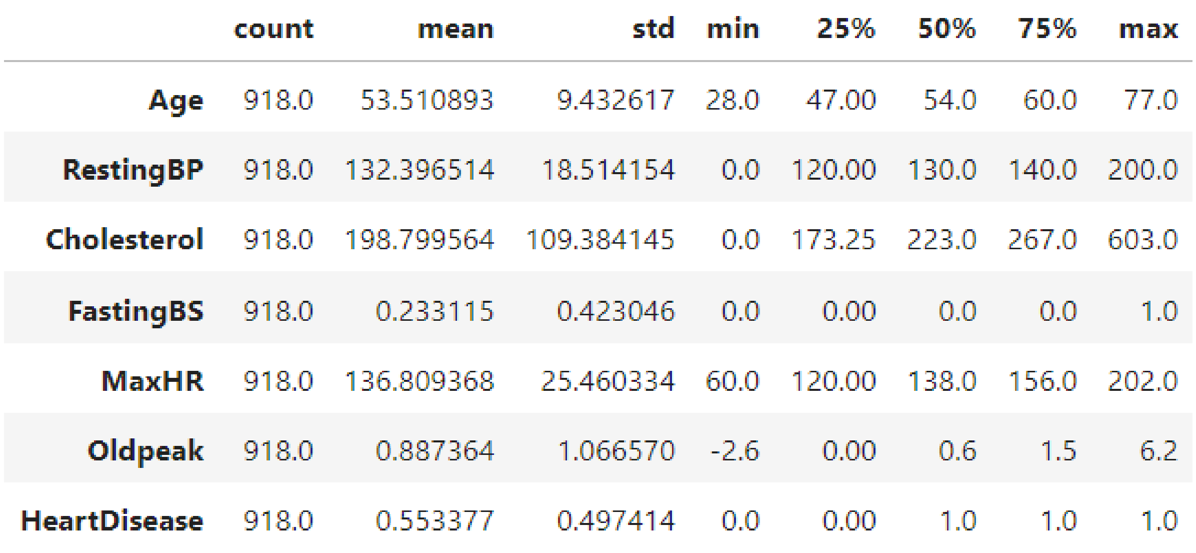
Age	Age of the patient [years]
Sex	Sex of the patient [M: Male, F: Female]
ChestPainType	Chest pain type [TA: Typical Angina, ATA: Atypical Angina, NAP: Non-Anginal Pain, ASY: Asymptomatic]
RestingBP	Resting blood pressure [mm Hg]
Cholesterol	Serum cholesterol [mm/dl]
FastingBS	Fasting blood sugar [1: if FastingBS > 120 mg/dl, 0: otherwise]

RestingECG	Resting electrocardiogram results [Normal: Normal, ST: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV), LVH: showing probable or definite left ventricular hypertrophy by Estes' criteria]
MaxHR	Maximum heart rate achieved [Numeric value between 60 and 202]
ExerciseAngina	Exercise-induced angina [Y: Yes, N: No]
Oldpeak	ST [Numeric value measured in depression]
ST_Slope	The slope of the peak exercise ST segment [Up: upsloping, Flat: flat, Down: downsloping]
HeartDisease	Output class [1: Heart disease, 0: Normal]

II. Exploratory Data Analysis (EDA)

Exploratory data analysis (EDA) is one of the most important steps before doing anything in every tabular dataset problem. As we used Heart Failure Prediction Dataset, EDA helps us gain insight into the dataset by visualizing charts and detecting any errors, outliers as well as understanding different data patterns of attributes. In this section, we tried to depict what we have done and how we preprocessed the dataset.

1. Data Insight and Visualization:



	count	mean	std	min	25%	50%	75%	max
Age	918.0	53.510893	9.432617	28.0	47.00	54.0	60.0	77.0
RestingBP	918.0	132.396514	18.514154	0.0	120.00	130.0	140.0	200.0
Cholesterol	918.0	198.799564	109.384145	0.0	173.25	223.0	267.0	603.0
FastingBS	918.0	0.233115	0.423046	0.0	0.00	0.0	0.0	1.0
MaxHR	918.0	136.809368	25.460334	60.0	120.00	138.0	156.0	202.0
Oldpeak	918.0	0.887364	1.066570	-2.6	0.00	0.6	1.5	6.2
HeartDisease	918.0	0.553377	0.497414	0.0	0.00	1.0	1.0	1.0

Figure 1. Numerical Value Information

Upon examining the Numerical Value Information (Figure 1.), we observed that both the Cholesterol and RestingBP variables have a minimum value of 0, which is highly unusual. We suspected that these values had been mislabeled and should actually be null or missing values. Therefore, we modify these values to represent null values before proceeding with visualizing the dataset.

```
# There is one record with RestingBP = 0, which is impossible.
# We should change it to nan and then impute it later.
heart['RestingBP'] = heart['RestingBP'].replace(0, np.nan)
```

```
heart[heart['Cholesterol'] == 0].shape
# There are a lot of records with Cholesterol = 0, which is impossible too.
heart['Cholesterol'] = heart['Cholesterol'].replace(0, np.nan)
```

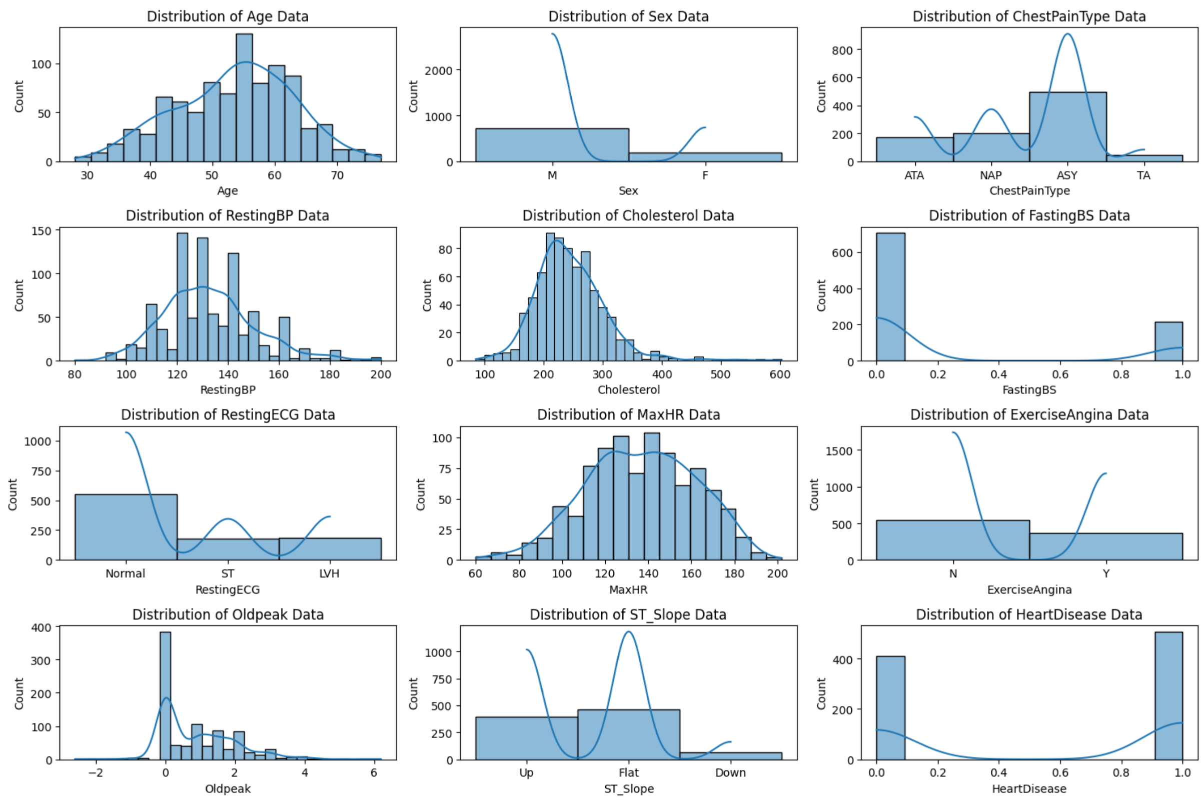


Figure 2. Data visualization for every attributes

2. Data Preparation

Data preparation is a pre-processing step that involves cleansing, transforming, and consolidating data. In this part, we will illustrate how we preprocessed our dataset based on the previous section.

2.1 Data Cleansing

During this step, we conducted a thorough examination to identify any missing or inappropriate values that deviated significantly from the expected descriptions for each attribute. Our analysis revealed that the only attributes containing missing values were Cholesterol and RestingBP, which were also the initial sources of the inappropriate values.

We proceeded to address the missing values in Cholesterol and RestingBP. To accomplish this, we used the K Nearest Neighbor (kNN) imputer from the scikit-learn library to impute these values after completing the scaling and encoding step.

```
Age 0               RestingECG 0
Sex 0               MaxHR 0
ChestPainType 0     ExerciseAngina 0
RestingBP 1         Oldpeak 0
Cholesterol 172     ST_Slope 0
FastingBS 0         HeartDisease 0
```

2.2 Feature Selection

In this step, we would like to remove redundant attributes and some attributes which have a high correlation value with other attributes which are based on a heatmap (Figure 3).

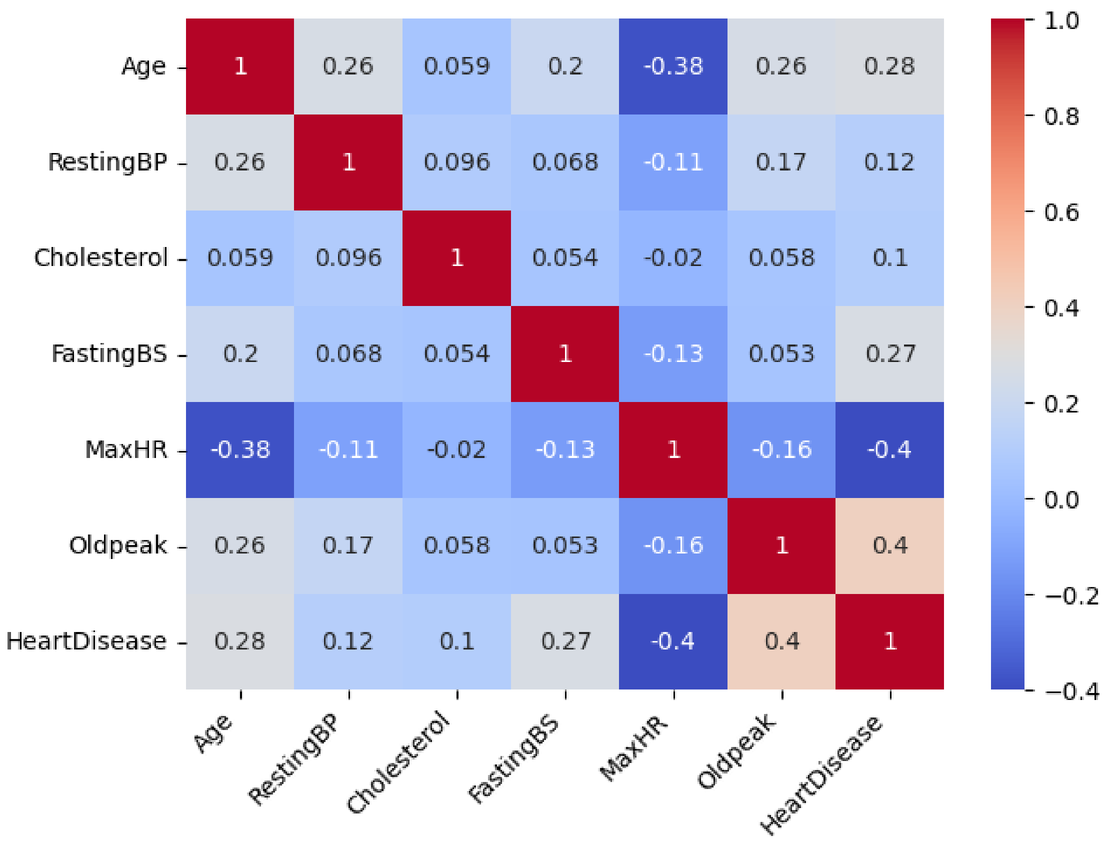


Figure 3. Correlation heatmap

As we can see, there are no variables that have high correlation with the others, so in this step, we did not reduce any variables.

2.3 Variable Transformations

Machine learning models do not understand the units of the values of the features. It treats the input just as a simple number but does not understand the true meaning of that value. Thus, it becomes necessary to scale the data. We use scaling for numerical features and encoding for categorical features.

2.3.1 Scaling

We have 2 options for data scaling: Normalization and Standardization. As most of the algorithms assume the data to be normally (Gaussian) distributed, Normalization is done for features whose data does not display normal distribution and standardization is carried out for features that are normally distributed where their values are huge or very small as compared to other features.

Normalization: Oldpeak feature is normalized as it displays a right skewed data distribution. (Figure 2)

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- Standardization : Age, RestingBP, Cholesterol and MaxHR features are scaled down because these features are normally distributed. (Figure 2)

$$z = \frac{X - \mu}{\sigma}$$

$$\mu = Mean$$

$$\sigma = Standard\ Deviation$$

2.3.2 Encoding

- Non tree - based models:

For categorical columns, we use label encoding for categorical features with 2 unique values, and one-hot encoding for categorical features with more than 2 unique values.

- Tree - based models:

We use label encoding for all categorical columns.
To deal with missing values, we dropped all categorical columns with more than 2 values in data then used kNN imputer to find their suitable value, after that insert those columns into the dataset again.

2.4 Handling missing data by KNN Imputer:

$$d_{xy} = \sqrt{weight \cdot square\ distance\ from\ present\ coordinates}$$

where

$$weight = \frac{Total\ number\ of\ coordinates}{Number\ of\ present\ coordinates}$$

- Identify the missing values in the "RestingBP" or "Cholesterol" attribute.
- For each missing value, find the k nearest neighbors (data points with non-missing values) based on a distance metric such as Euclidean distance or Manhattan distance. The k nearest neighbors are typically chosen based on a predefined value or through cross-validation.
- Calculate the average or weighted average of the non-missing values from the k nearest neighbors and assign this value as the imputed value for the missing entry.

3. Data Splitting

We randomly split the dataset into 2 parts: train set and test set with the ratio 75%:25%.

III. Modelling

1. Logistic Regression

Logistic Regression is an example of a classification algorithm which is used to find a relationship between features and the probability of a particular outcome used in binary classification problems.

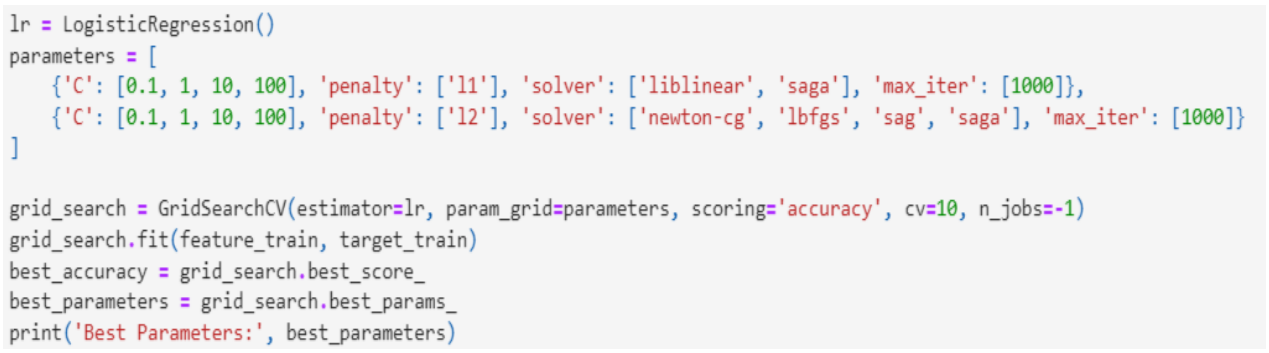
Denote y_i is the real result of patient i^{th} (= 1 if the patient has heart disease and = 0 otherwise), $\widehat{y_i}$ is the probability that the patient i^{th} has heart disease and $\widehat{y_i}$ can be calculated by sigmoid function:

$$\widehat{y_i} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n)}}$$

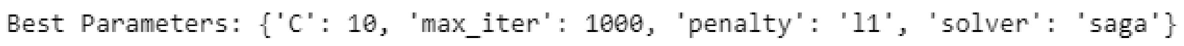
where $X_1, X_2, ... X_n$ are explanatory variables of all features after transforming data and $\beta_0, \beta_1, ... \beta_n$ are parameters. We need to minimize the lost function:

$$Loss = -\sum_{i=1}^{N} ((y_i * log(\widehat{y_i}) + (1 - y_i) * log(1 - \widehat{y_i}))$$

by finding all parameters β_0, β_1, ... β_n where N is the number of samples of the train dataset.
To do this, we used Grid Search to find best hyperparameters including ‘c’, ‘penalty’,”solver’ and ‘max-iter’ and we get the best hyperparameters:

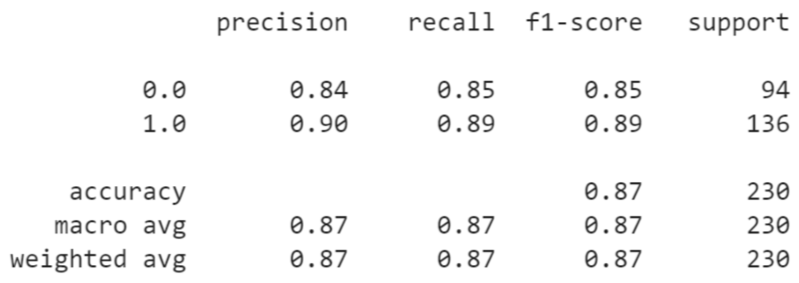


```
lr = LogisticRegression()
parameters = [
    {'C': [0.1, 1, 10, 100], 'penalty': ['l1'], 'solver': ['liblinear', 'saga'], 'max_iter': [1000]},
    {'C': [0.1, 1, 10, 100], 'penalty': ['l2'], 'solver': ['newton-cg', 'lbfgs', 'sag', 'saga'], 'max_iter': [1000]}
]

grid_search = GridSearchCV(estimator=lr, param_grid=parameters, scoring='accuracy', cv=10, n_jobs=-1)
grid_search.fit(feature_train, target_train)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
print('Best Parameters:', best_parameters)
```



```
Best Parameters: {'C': 10, 'max_iter': 1000, 'penalty': 'l1', 'solver': 'saga'}
```

We train the model in train set with these best hyperparameters and used classification report to evaluate the model in test set:



```
              precision    recall  f1-score   support

         0.0       0.84      0.85      0.85        94
         1.0       0.90      0.89      0.89       136

    accuracy                           0.87       230
   macro avg       0.87      0.87      0.87       230
weighted avg       0.87      0.87      0.87       230
```

2. Support Vector Machine (SVM)

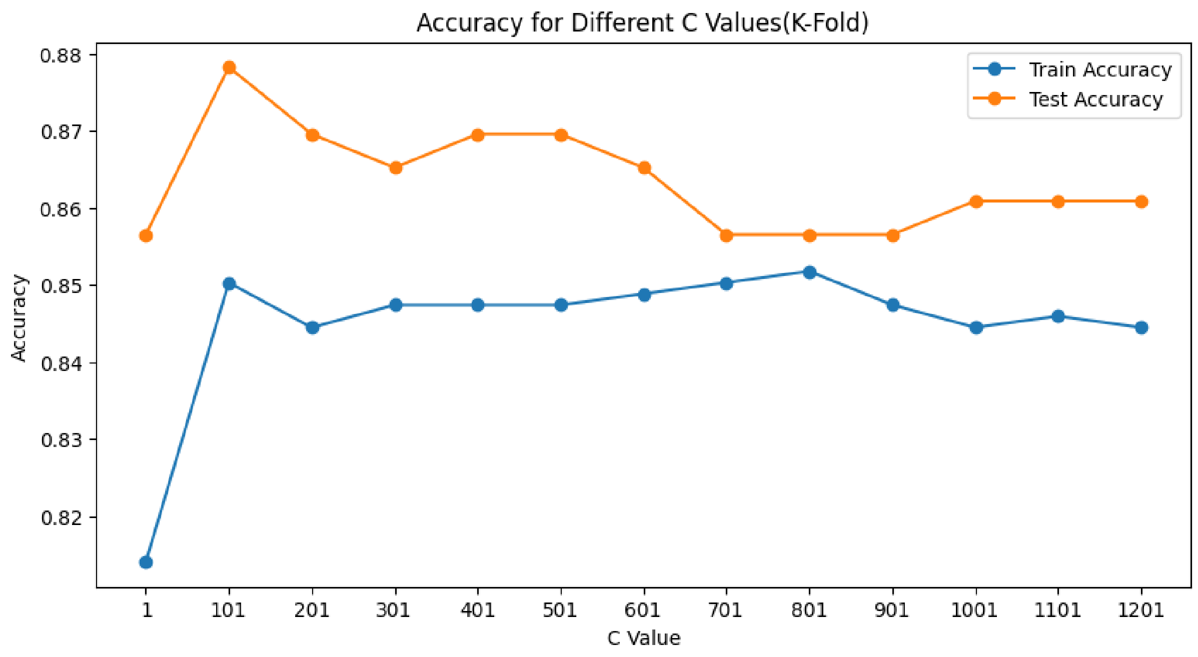
Support Vector Machines (SVM) is a classification and regression algorithm that finds an optimal hyperplane to separate classes by maximizing the margin between them. It can solve linear and non-linear problems and work well for many practical problems.

For nonlinear problems, we transform the input into another space, which often has higher dimensions, so that the projection of data is linearly separable (Kernel functions) then use Linear SVM in the new space. When using kernel functions, we will find best hyperparameters like the choice of kernel (polynomial, rbf, sigmod), C (trade-off between accuracy and margin), and gamma (reach of training examples in RBF kernel).

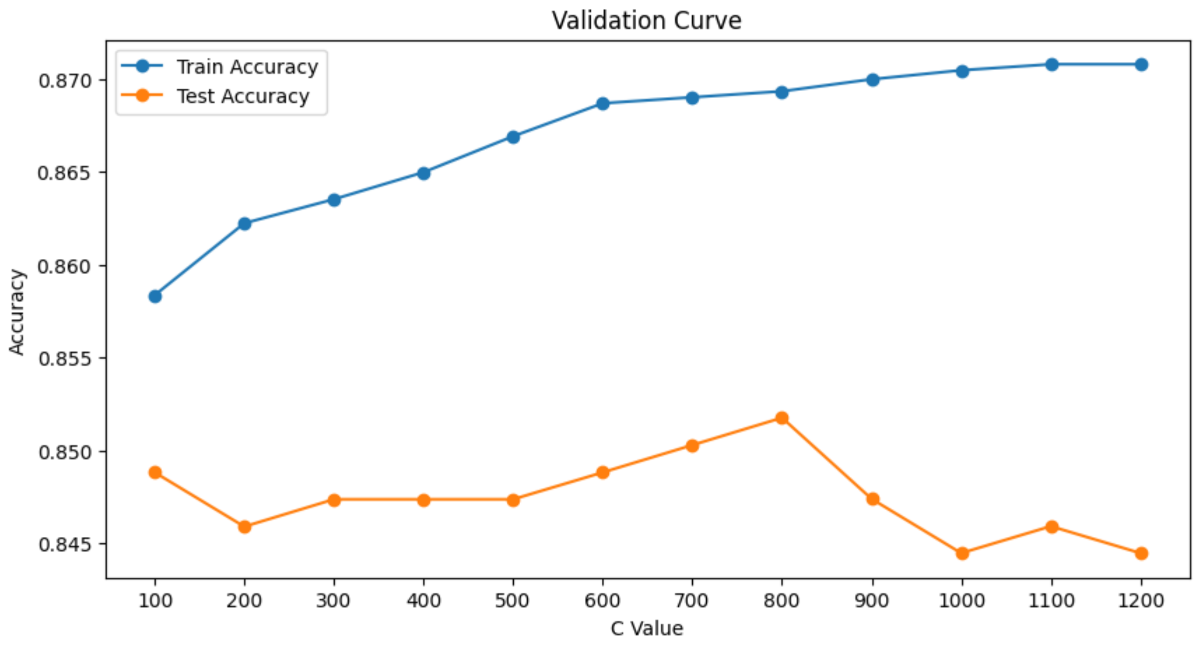
For finding the best parameters, we use grid search to find best C value, kernel type and also best gamma for RBF kernel:

```
#grid search for best parameters
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
svm = SVC()
parameters = [{'C': [0.1, 1, 10, 100, 1000], 'kernel': ['linear']},
                {'C': [0.1, 1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma': [0.1, 0.01, 0.001, 0.0001]}]
grid_search = GridSearchCV(estimator=svm, param_grid=parameters, scoring='accuracy', cv=10, n_jobs=-1)
grid_search.fit(feature_train, target_train)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
print('Best Parameters:', best_parameters)
```

After using grid search, we find that the best parameters for our training set are C: 100, kernel: rbf, gamma: 0.001. We will plot the k - fold for different C values with grid search kernel and gamma:



As we can see, C = 100 has the highest test and train accuracy. We also use validation curve to show the sensitivity between changes in our model's accuracy with changes C value:



As the value of C increases, both the accuracy of the training set is higher, but the accuracy of the test set decreases, the model tends to overfit. So, we can conclude that C = 100 would be the ideal value of C.

After find the best parameters, we also use classification report to evaluate the model:

```
              precision    recall  f1-score   support

         0.0       0.85      0.83      0.84        94
         1.0       0.88      0.90      0.89       136

    accuracy                           0.87       230
   macro avg       0.87      0.86      0.86       230
weighted avg       0.87      0.87      0.87       230
```

3. Ensemble of k-NN models

An ensemble of k-nearest neighbors (k-NN) models is a technique in machine learning where multiple KNN models are combined to make predictions. Ensemble methods aim to improve the overall performance and generalization ability of individual models by leveraging the diversity and collective wisdom of multiple models.

- k-NN is a simple and intuitive algorithm used for both classification and regression tasks. Given a new data point, it finds the k nearest neighbors in the training dataset based on a distance metric (in this model we use Euclidean distance) and assigns a class label or predicts a value based on the labels/values of those neighbors.

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- Ensemble learning combines multiple models to obtain better predictive performance than what individual models can achieve. The idea behind ensemble methods is that by combining the predictions of multiple models, the errors made by one model can be compensated by the strengths of others, leading to improved overall accuracy and robustness.

An overview how all base classifiers are trained:

- Each classifier is trained on a different subset of the training data. In this case, the base classifiers are instances of the KNeighborsClassifier from the scikit-learn library.
- For each base classifier, a new KNeighborsClassifier object is created with the current k value (we implement a “for” loop for k in k_values = range(1,11)).
- A random subset of the training data is selected by randomly choosing indices using np.random.choice().
- The current classifier is then fitted to the subset of training features and targets using the fit() method.
- The trained classifier is appended to the “classifiers” list.

Weighted Voting:

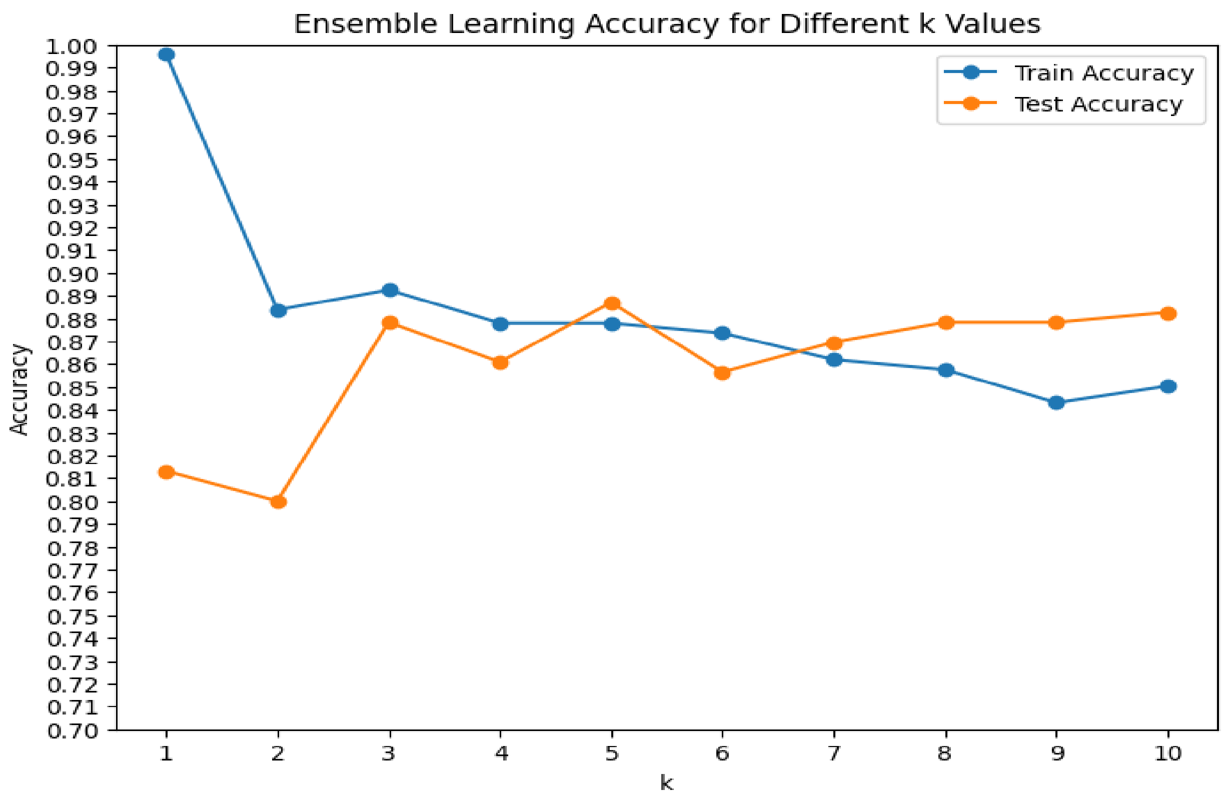
- Still within the for loop of k_values, after training the base classifiers, the code proceeds to perform predictions and calculate weights for each classifier.
- Three empty lists: “weights”, “train_predictions”, and “test_predictions”, are created to store the weights and predictions.
- For each classifier in “classifiers”, predictions are made on both the training and test sets using the predict() method.
- The accuracy of each classifier is computed using accuracy_score().
- The weight of each classifier is calculated by dividing its training accuracy by the sum of training accuracies for all classifiers to normalize the weight.
- The weight is then appended to the “weights” list, and the predictions are appended to the “train_predictions” and “test_predictions” lists.

Combining Predictions:

- For each classifier, the predictions are multiplied by their corresponding weights and added to the ensemble predictions.
- The ensemble predictions are rounded to obtain class labels.

Evaluating Ensemble Performance:

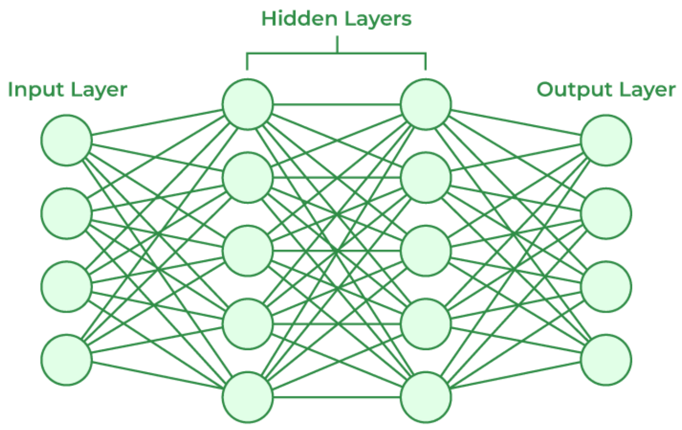
- The accuracy of the ensemble model is evaluated using accuracy_score() for both train set and test set



```
Best k value: 5
Accuracy for best k value: 0.8869565217391304
RAM Consumption: 3.17578125 MB
Time Taken: 2.1531295776367188 seconds
Classification Report for Best k value (5):
              precision    recall  f1-score   support

         0.0       0.88      0.84      0.86        94
         1.0       0.89      0.92      0.91       136

    accuracy                           0.89       230
   macro avg       0.89      0.88      0.88       230
weighted avg       0.89      0.89      0.89       230
```

4. Artificial Neural Network (MLP)

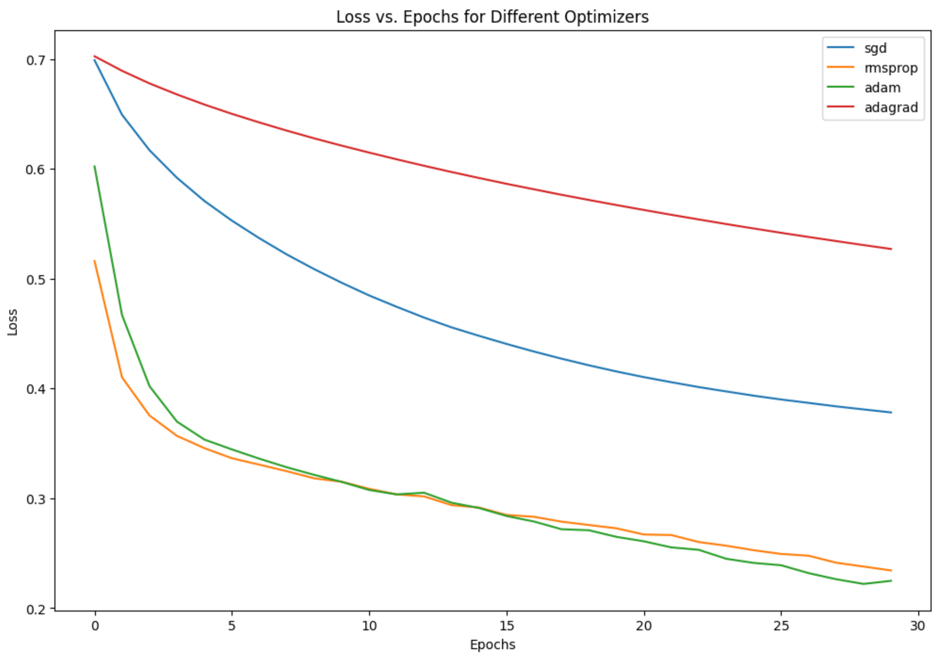
ANN is a computational model inspired by the structure and function of the human brain's neural networks.



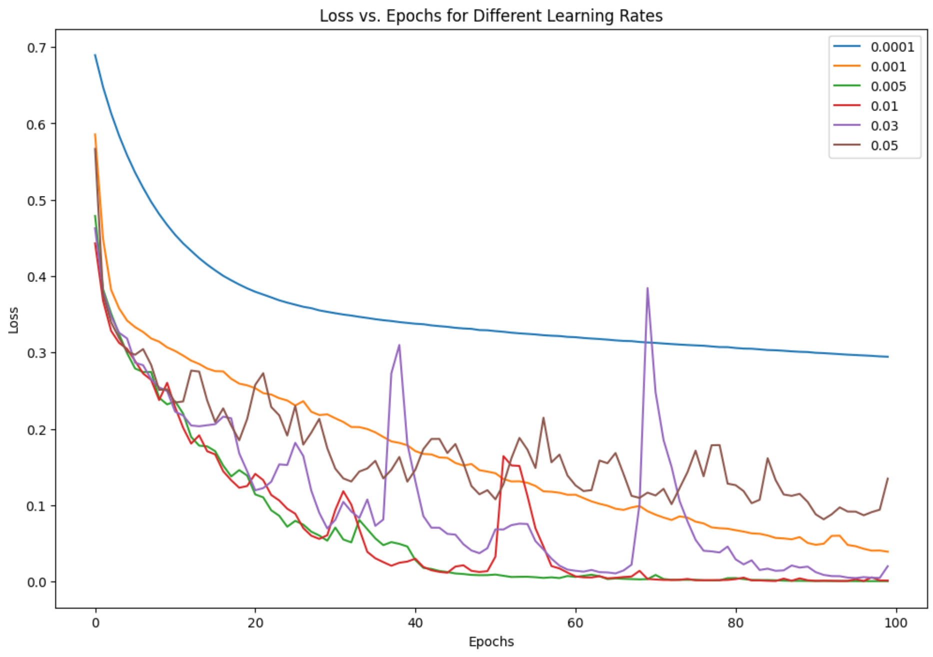
example of a ANN

This algorithm is implemented using the keras library defined in Tensorflow. The model contains some modified parameters:

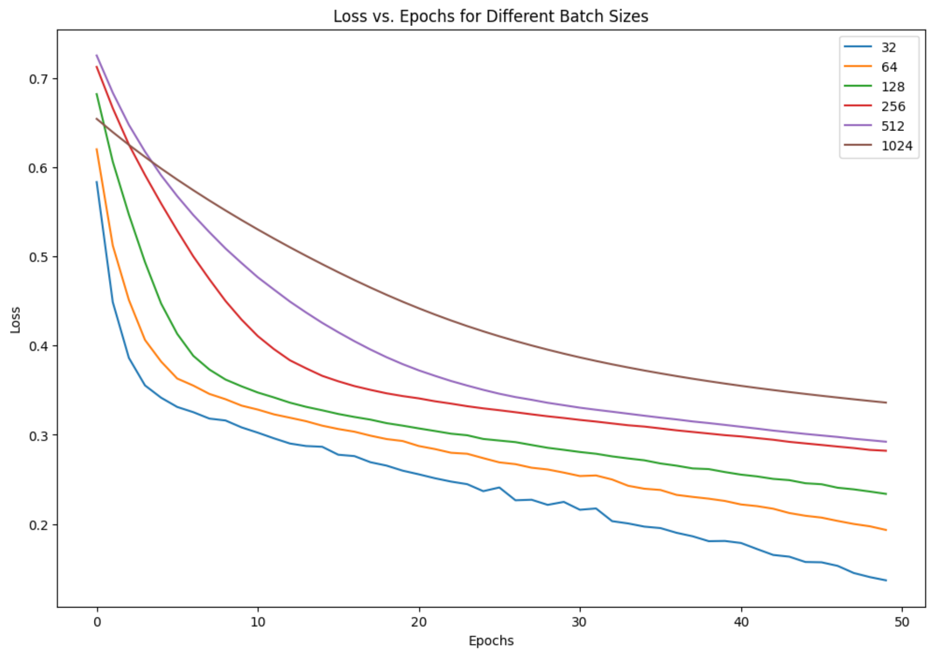
-Optimizer: At first we planned to choose Adam, but we still need to test the model on different optimizers to find out if it is true or not. We can see that in the below graph, adam and rmsprop results are very close but after many run tests, we see that adam is more stable than rmsprop.



- Learning rate: The learning rate is a hyperparameter that determines the step size or the rate at which the optimizer adjusts the model's parameters during training.Learning rate controls how quickly the model is adapted to the problem. We find out that the learning rate is around 0.001 to 0.005. With lower values, it takes longer epoch to converge, but with the higher one, it is not stable, which can ignore the global optimum to choose local optimum (you can see that in the graph below).



-Batch size: Batch size determines how many samples are processed together in parallel before the model's parameters are updated. To find good batch size, we try on difference value, and the batch size around 32 is the best one:

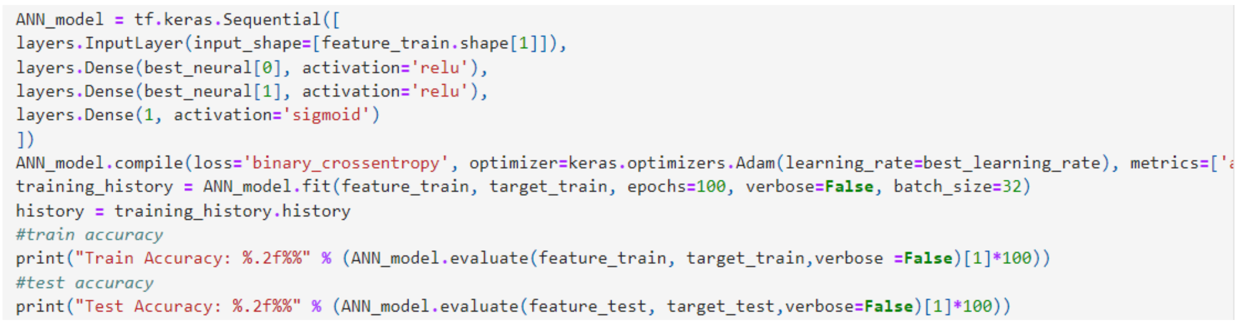


-loss: Because our final target is whether that person has heart disease or not, which is binary(1 or 0), so we chose binary_crossentropy for this model.
-Number of layers: After testing, we find out that with a low number of layers like 0 or 1, the result is not stable but with the high one (4,5 or more) it takes so much time to run and some time is worse than lower number. So we decided to choose 2 hidden layers for this model.

- Number of neurons in each layer: We used grid search for this hyperparameter, along with learning rate

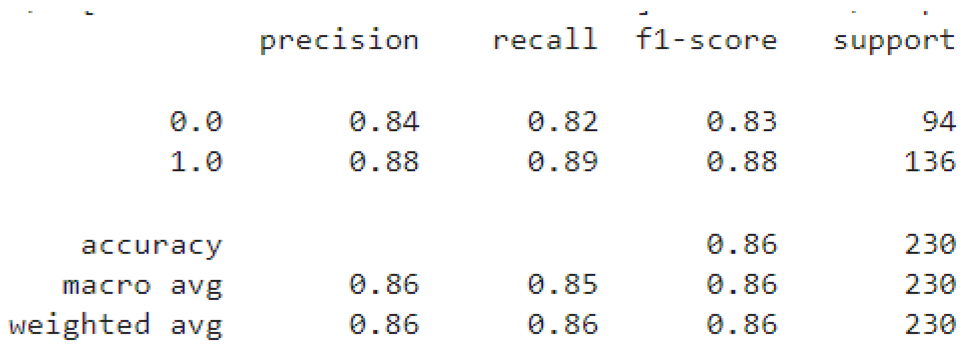
After using Grid search, those hyperparameters are chosen for the model:

Optimizer :Adam
learning rate: 0.001
Batch size: 32
epochs: 100
hidden layer 1: 15 neural
hidden layer 2: 15 neural
loss : binary cross entropy
This could change when running the code again but these hyperparameters appear most and give the best result.



```
ANN_model = tf.keras.Sequential([
layers.InputLayer(input_shape=[feature_train.shape[1]]),
layers.Dense(best_neural[0], activation='relu'),
layers.Dense(best_neural[1], activation='relu'),
layers.Dense(1, activation='sigmoid')
])
ANN_model.compile(loss='binary_crossentropy', optimizer=keras.optimizers.Adam(learning_rate=best_learning_rate), metrics=['a
training_history = ANN_model.fit(feature_train, target_train, epochs=100, verbose=False, batch_size=32)
history = training_history.history
#train accuracy
print("Train Accuracy: %.2f%%" % (ANN_model.evaluate(feature_train, target_train,verbose =False)[1]*100))
#test accuracy
print("Test Accuracy: %.2f%%" % (ANN_model.evaluate(feature_test, target_test,verbose=False)[1]*100))
```

Train Accuracy: 89.39%
Test Accuracy: 86.09%



```
              precision    recall  f1-score   support

         0.0       0.84      0.82      0.83        94
         1.0       0.88      0.89      0.88       136

    accuracy                           0.86       230
   macro avg       0.86      0.85      0.86       230
weighted avg       0.86      0.86      0.86       230
```

5. Random Forest

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

5.1. Algorithm of Random Forest for Classification

1. Input: training data D
2. Learning: grow K trees as follows:
    a) Generate a training set D_i by sampling with replacement from D.
    b) Learn the i^{th} tree from D_i:
        i. At each node:
            - Select randomly a subset S of attributes.
            - Split the node into subtrees according to S.
        ii. Grow this tree up to its largest size without pruning.
3. Output: the ensemble of trees i^{th} .

* To make a prediction at a new point for classification:

$$\widehat{y(x)}^{-} = majority\ vote\ \{\widehat{y(x)}\}.$$

Where $\widehat{y(x)}$ is the class prediction of the i^{th} random-forest tree.

* Evaluation: Some traditional classification metrics that can be used to evaluate the algorithm are precision, recall, f1-score, accuracy, and confusion matrix.

5.2. Pre-processing

For Random Forest, it is not necessary to scale the values of the features. This is primarily due to the utilization of decision trees within the algorithm. Decision trees are inherently insensitive to the scale of the input data. But since we are going to use KNN imputer, numerical variables are scaled like the preprocessing step in EDA.

In contrast to other algorithms, for the Random Forest Classifier, a decision was made to use label encoding instead of one-hot encoding for categorical features such as 'ChestPainType', 'RestingECG', and 'ST_Slope'. This decision was based on comparing the differences in encoding methods. By utilizing label encoding, which assigns a unique numerical label to each category, we streamline the preprocessing phase and ensure compatibility with the Random Forest Classifier algorithm.

5.3. Fine-tune the algorithm

To optimize the Random Forest classifier, a two-step approach is employed.

First, a randomized search is conducted to find the best hyperparameters by exploring different combinations of parameters such as the number of trees, maximum features, maximum depth, minimum samples split, minimum samples leaf, and bootstrap. The model with the highest accuracy is selected from this search.

```python
# Define hyperparameter grids
random_grid = {
    'n_estimators': [int(x) for x in np.linspace(start=200, stop=2000, num=10)],
    'max_features': ['sqrt', 'log2', None],
    'max_depth': [int(x) for x in np.linspace(10, 110, num=11)] + [None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}
```

The selected model is then evaluated on the test dataset to calculate its accuracy. Additionally, the top five parameter combinations with the highest mean test scores are extracted for further analysis.

In the second step, a grid search is performed using the unique parameter values identified from the top combinations. The GridSearchCV function exhaustively searches all possible combinations and selects the best model based on cross-validation results.

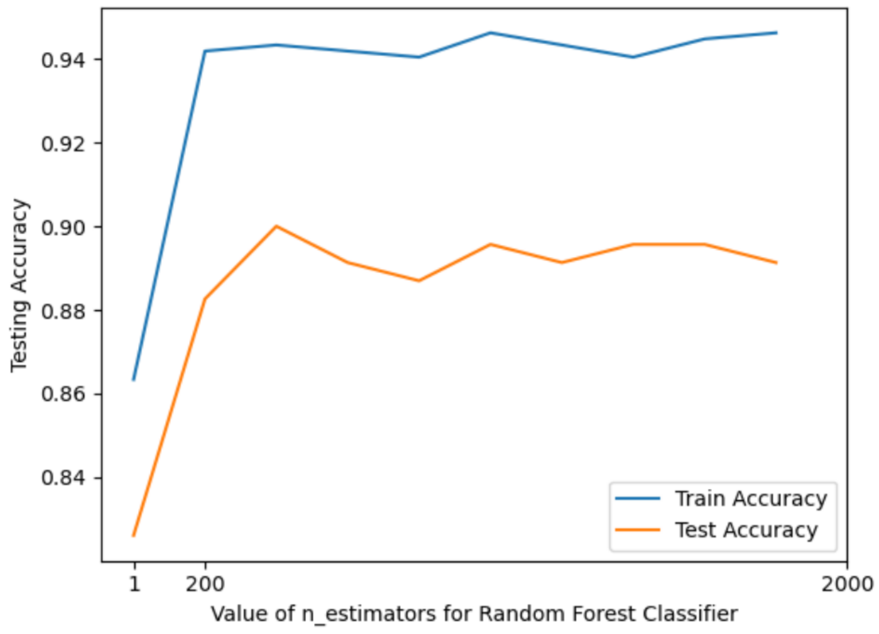
The best parameters:

```
{'bootstrap': True, 'max_depth': 100, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 1
400}
```

After performing a grid search to identify the optimal hyperparameters for our machine learning model, we continued the training process using the selected parameters. To further investigate the impact of the number of estimators on the model's performance, we trained the model with varying numbers of estimators, ranging from 1 to 2000 with a step size of 200.

The results revealed interesting trends and trade-offs. Initially, as the number of estimators increased, there was a noticeable improvement in the model's performance.

However, after a certain point, the performance gains diminished, and the model reached a plateau. This finding suggests that increasing the number of estimators beyond a certain threshold does not significantly enhance the model's predictive power. By analyzing the relationship between the number of estimators and the evaluation metric(s), we were able to determine an optimal number of estimators that strikes a balance between model complexity and performance. These insights provide valuable guidance for selecting the appropriate number of estimators to maximize performance while minimizing computational resources.



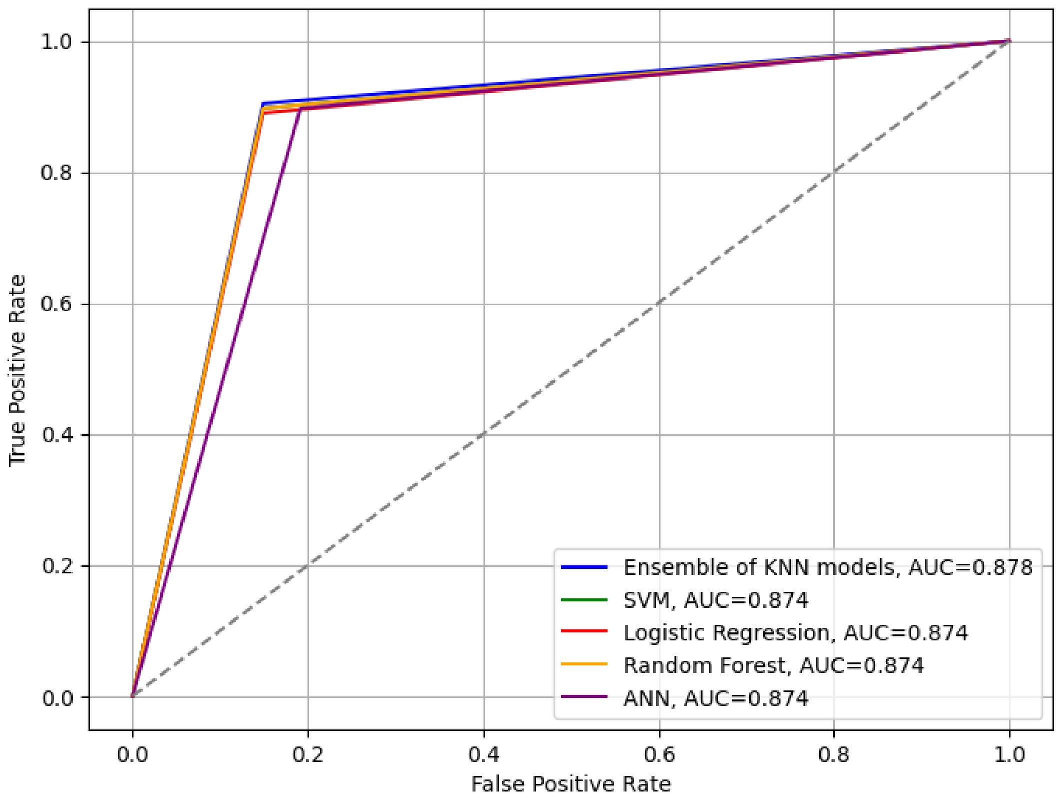
```
              precision    recall  f1-score   support

         0.0       0.86      0.85      0.86        94
         1.0       0.90      0.90      0.90       136

    accuracy                           0.88       230
   macro avg       0.88      0.88      0.88       230
weighted avg       0.88      0.88      0.88       230
```

IV. Practical result

By using scikit-learn library, we have implement all models and above methods and got the following table results and ROC curve for different models

Model	Accuracy	Macro Avg Precision	Macro Avg Recall	Macro Avg F1-score
Logistic Regression	0.87	0.87	0.87	0.87
Support Vector Machine	0.88	0.87	0.87	0.87
Ensemble of K - Nearest Neighbors models	0.89	0.89	0.89	0.89
Artificial Neural Network	0.86	0.86	0.85	0.86
Random Forest	0.87	0.87	0.87	0.87

Result of different models in term of different metrics



ROC Curve for different models

V. Conclusion and Future Improvement

1. Conclusion

These metrics provide an evaluation of the performance of each model. The accuracy represents the overall accuracy of the model in predicting the target variable. The macro average precision, recall, and F1-score provide an average measure of precision, recall, and F1-score across all classes.

The time taken and memory consumption represent the computational resources required by each model during training or inference. These metrics can be useful in determining the efficiency and scalability of the models.

Based on these metrics, it can be observed that the ensemble of K-Nearest Neighbors models has the highest accuracy, macro average precision, macro average recall, and macro average F1-score among the models listed. It also has relatively low time taken and memory consumption, indicating efficient performance.

On the other hand, the Random Forest model shows high time taken and memory consumption, which may indicate a more resource-intensive model. However, it still achieves competitive performance in terms of accuracy and other metrics.

2. Future Improvement

To improve the performance of models in predicting heart disease, several future improvements can be implemented. These include feature engineering to create more informative input, hyperparameter tuning to optimize model configurations, ensemble methods to combine multiple models, data augmentation to increase the training set size, model regularization to prevent overfitting, exploring different types of models, utilizing cross-validation techniques for evaluation, ensuring data quality, collaborating with domain experts, incorporating external data sources, and striking a balance between model performance and practical considerations.

These improvements aim to enhance the accuracy and reliability of heart disease predictions, ultimately assisting in early detection and management of cardiovascular diseases.

References

https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction?fbclid=IwAR18zxKwk4SyNDW-GJm6iEXwkBwITIgoMHboURYsH3yrFiCj1EqJHHlZ6eM

https://www.kaggle.com/code/tanmay111999/heart-failure-prediction-cv-score-90-5-models#Feature-Engineering

https://www.analyticsvidhya.com/blog/2020/07/knnimputer-a-robust-way-to-impute-missing-values-using-scikit-learn/