CPC152 FOUNDATIONS AND PROGRAMMING FOR DATA ANALYTICS

SEMESTER 2 2022/2023

SCHOOL OF COMPUTER SCIENCES

PROJECT

PREPARED BY: GROUP 8

NAME	MATRIC NUMBER
AINA` ZAWANI BINTI ZAIDAN	163719
CHOONG YEW CHUNG	163991
LI, XINFENG	160640
MUHAMMAD IZHAM BIN HARIS	163955
NURIN FARAH IZZATI BINTI MOHD RUSDI	160406
TAN YI PIN	163382

LECTURER:

Ts DR CHEW XINYING

SUBMISSION DATE:

25TH JUNE 2023

TABLE OF CONTENTS

1.0 ABSTRACT

Machine learning is a subset of artificial intelligence (AI) and computer science that centres on using data and algorithms to simulate how humans acquire knowledge, slowly increasing the accuracy of the system. The rapidly expanding discipline of data science includes this AI technique as a key element. Additionally, machine learning is implemented through algorithms trained using statistical techniques to produce classifications or predictions and to find important insights in data mining projects. To clarify, machine learning algorithms are programs that are able to discover hidden patterns in data, forecast results, and enhance performance based on past experience. The algorithms can be categorised into three types which are supervised; unsupervised; and reinforcement learning algorithms. In this project we will be using three popular machine learning algorithms: K-Nearest neighbour (KNN), support vector machine (SVM), and decision tree. All three are of the supervised learning type where the first two are further grouped under the classification subset while the decision tree is under regression. Section 2 will further introduce these algorithms. Section 3 will have our experiment's objectives, which is to: accurately predict heart disease patients, identify which traits are most important for correctly predicting cardiac disease, and identify the most efficient machine learning method for the heart disease data set. We will justify our decision in choosing KNN, SVM, and decision tree in Section 4. Furthermore, these three algorithms will be implemented on a dataset about heart disease with the steps to build each algorithm explained in Section 5. Section 6 will discuss the comparisons between the three algorithms and the recommendations. The discussion of the results of our experiment will be done in Section 7. Finally, our paper will end with Section 8: the concluding remarks. From the project, we found that KNN is the best model suited for the heart disease dataset, with high accuracy and precision needed to correctly predict patients with the heart disease. Overall, we narrowed down 5 features that are crucial in identifying victims of heart illness: 'ca', 'cp', 'exang', 'thal', 'thalach'.

2.0 INTRODUCTION TO 3 MACHINE LEARNING ALGORITHMS

This section will introduce three machine learning algorithms. The first subsection will explain about K-Nearest neighbour (KNN). The next will be an introduction to the support vector machine (SVM) learning algorithm. Subsection 2.3 will describe the last algorithm which is the decision tree.

2.1 KNN (K-Nearest Neighbours)

K-Nearest neighbours, simply pronounced as KNN, is one of the simplest supervised machine learning algorithms. It is used for regression and classification but is mainly used for classification problems. This means the algorithm classifies a given data point according to how its neighbours are classified. KNN categorises new cases using a similarity metric after storing all previously classified cases. Alternatively, KNN is known as a lazy learner algorithm since it saves the training dataset rather than learning from it immediately. Instead, it uses the dataset to perform an action when classifying data. During the training phase, the algorithm simply stores the dataset and subsequently classifies new data into a category that is quite close to the new data. Distance functions are calculated to measure the distance between the data points. Based on the requirements, the function can be either Minkowski, Manhattan, Hamming or Euclidean distance.

This algorithm has a parameter, '*k*', which denotes the number of nearest neighbours to take into account while voting by majority. A procedure called parameter tuning is used to choose a proper value of *k* and is impertinent for better accuracy. There are two ways to choose the parameter. First is by finding the square of *n*, where it refers to the total number of data points. Second is by choosing *k* as an odd number, which helps to avoid uncertainty between two classes of data. It is important to note a very small value of *k* can make the data noisy and will cause outliers in the algorithm. On the other hand, a large value of *k* can lead to issues in processing and in terms of resources.

Furthermore, this model is implemented when the data is categorised, noise free (data is not corrupted and is structured to enable machines to understand and interpret it accurately), and when it is a small dataset because the algorithm is a lazy learner. Data scientists use the KNN algorithm because it is based on feature similarity which makes it simple to determine the category or class of a given dataset.

2.2 SVM (Support Vector Machine)

Support Vector Machine or SVM, is a powerful and versatile machine learning algorithm used for both classification and regression tasks. SVM was initially designed for binary classification, aiming to find an optimal decision boundary that separates two classes in the feature space. The key idea behind SVM is to maximize the margin between the decision boundary and the closest data points from each class, known as support vectors. By maximizing the margin, SVM aims to achieve better generalization and robustness to new, unseen data.

SVM aims to find an optimal hyperplane that separates different classes in the feature space, maximizing the margin between the classes. This hyperplane is selected based on a subset of training data called support vectors, which are the data points closest to the decision boundary. SVM can handle both linearly separable and non-linearly separable data by applying the kernel trick, which transforms the input features into a higher-dimensional space. SVM can handle both linearly separable and non-linearly separable data by applying the kernel trick, which transforms the input features into a higher-dimensional space. The algorithm seeks to find the optimal hyperplane that minimizes classification errors while maximizing the margin. SVM is known for its ability to handle high-dimensional data and effectively handle outliers. It is widely used in various domains, including image classification, text categorization, bioinformatics, and finance. SVM offers a flexible and robust approach to classification and regression tasks, making it a popular choice in machine learning applications.

2.3 Decision Tree

In machine learning, the decision tree is one of the most popular algorithms that can be used for both classification and also regression tasks. It is a prediction model that learns from the data a hierarchy of decision-making guidelines. The tree has a root node, branches, nodes inside the branches, and leaf nodes at the ends of the branches. A feature or attribute is represented by a node, and each possible value or consequence of the attribute or feature is represented by a branch. The root node represents the entire dataset. The process of constructing it involves dividing the training data into smaller subsets based on attribute values. This is done recursively until a stopping point is reached, which could be the maximum depth of the tree, or the minimum number of samples needed to split a node. The goal is to create subsets of the data that are as homogeneous as possible with respect to the target variable.

The algorithm uses different metrics like information gain, Gini impurity, or entropy to figure out the most suitable feature to split the data at each node. These metrics are useful in assessing the quality of a split and determining how pure or impure the subsets are. Based on the various values of this feature, the dataset is split into groups. This process of splitting is done over and over again on each group until a stopping point is reached, like a maximum tree depth or a minimum number of instances. At this point, the projected class name or number value is given to the last nodes, which are called leaf nodes. To make predictions, new cases are examined by going through the tree and following the branches based on the values of their features until a leaf node is reached. The method can be pruned to reduce overfitting by eliminating nodes or branches that are not needed. Overall, the decision tree algorithm makes a model that can be understood and used to make predictions. This model shows how links between features and the goal variable can be used to make accurate predictions about data that has not yet been seen.

In summary, a decision tree in machine learning is a predictive model that learns a hierarchical structure of decision rules from data, allowing for the classification or regression of new instances based on their feature values.

3.0 OBJECTIVES OF EXPERIMENT

In this project, we set three objectives to achieve:

- To **effectively predict patients with heart disease** by using machine learning algorithms.
- To determine which characteristics or factors are most crucial for accurately predicting heart disease.
- To determine which machine learning technique is the most effective for the heart disease data set.

4.0 JUSTIFICATION OF CHOICES

The justification for choosing the three machine learning algorithms which are K-Nearest Neighbors, Decision Tree and Support Vector Machine is because the selected dataset (heart disease) is a supervised dataset and the target variable in the dataset is binary data.

4.1 KNN (K-Nearest Neighbors)

K-Nearest Neighbors (KNN) was chosen because it only requires a K value as the parameter that refers to the number of nearest neighbours to include in the majority of the voting process and distance metric between two data points is defined for the similarity. Choosing the right value of K has a great effect on better accuracy. Popular method to get the distance metric between two data points is the Euclidean distance method. These hyperparameters make K-Nearest Neighbors simple to implement due to its simplicity and accuracy.

Lazy learner, a characteristic of K-Nearest Neighbors, makes it a simple algorithm because it does not learn discriminative function from the training data but "memorises" the training dataset instead. This means that it does not require an explicit training phase. The model simply stores the training instances and their corresponding labels. This can be advantageous when dealing with streaming data or when the dataset is large and dynamic.

Besides, K-Nearest Neighbors is flexible in handling different feature types such as numerical variables and categorical variables. K-Nearest Neighbors offers flexibility in feature selection and distance metric choices, making it adaptable to various heart disease datasets. Heart disease analysis involves considering a wide range of features such as age, sex, resting blood pressure, serum cholesterol levels and more. K-Nearest Neighbors can handle both numerical and categorical features without the need for extensive preprocessing. This flexibility allows a wide range of datasets without the need for extensive data processing or transformation. KNN allows for the selection of different distance metrics such as Euclidean or Manhattan distance. This adaptability allows programmers to align the distance metric with the characteristics of the heart disease dataset, incorporating domain knowledge into the analysis. Choosing a suitable distance

metric is critical since it influences how similarities are measured and can impact the algorithm's performance.

Moreover, KNN naturally handles multiclass classification cases by considering the majority class among the k-nearest neighbours. It can effectively differentiate between different classes or categories, making it valuable for predicting outcomes in multiclass scenarios. This is particularly useful when dealing with datasets that involve multiple classes or distinct categories. This capability is particularly relevant in heart disease analysis, as it allows for the identification of different heart disease categories or stages of severity. K-Nearest Neighbors can perform well when provided with enough representative data. As the number of instances grows, K-Nearest Neighbors has a higher chance of capturing the underlying patterns and relationships within the heart disease dataset. Sufficient representation across different classes or categories enables KNN to make more reliable classifications.

In summary, K-Nearest Neighbors's simplicity of implementation, flexibility in feature and distance choices, natural handling of multiclass cases, and performance with enough representative data make it a valuable choice for heart disease dataset machine learning.

4.2 SVM (Support Vector Machine)

Next, Support Vector Machine (SVM) is responsible for finding the decision box to separate different classes and maximise the margin. SVM is particularly effective in high-dimensional spaces, where the number of features is large compared to the number of instances. In the case of heart disease machine learning, the dataset contains numerous medical measurements and patient attributes. It utilises a decision boundary (hyperplane) to separate different classes in the feature space. By finding an optimal hyperplane, Support Vector Machine can effectively handle this heart disease complex datasets which have a large number of features, making it suitable for tasks such as image recognition or text classification.

Moreover, SVM allows for the use of different kernel functions, such as linear, polynomial, or Radial Basis Function (RBF) Kernels. Kernels transform the input data into a higher-dimensional space, where it may be easier to find a linear decision boundary. This flexibility allows the algorithm to handle complex nonlinear relationships between features without explicitly defining the transformations, making it suitable for heart disease datasets with nonlinear separability. First, the linear kernel is the simplest form of the kernel function, and it represents the original feature space. It works well when the heart disease dataset exhibits linear separability, where a straight line or hyperplane can effectively separate the classes. The linear kernel is computationally efficient and often used when the data does not require complex transformations. Second, the polynomial kernel allows for non-linear transformations of the data. It maps the original features into a higher-dimensional space using polynomial functions. The polynomial kernel is useful when the relationship between features in the heart disease dataset exhibits non-linear patterns. By introducing non-linear terms, it can capture complex interactions between variables. Last, the Radial Basis Function Kernel is one of the most used kernel functions in

Support vector machine. It transforms the data into an infinite-dimensional space using Gaussian functions. The Radial Basis Function Kernel is suitable for datasets where the boundaries between classes are not clearly defined or have intricate shapes. It can capture complex and non-linear relationships between features, making it a powerful choice for heart disease datasets with intricate patterns.

Furthermore, SVM performs well with small to medium-sized datasets. It exhibits good generalisation capabilities. It can avoid overfitting, where the model memorises the training data instead of learning underlying patterns. Overfitting is a common concern in machine learning, especially with limited data. Support vector machine addresses this issue by maximising the margin between the support vectors, which are the most critical instances for classification. This margin maximisation helps to create a robust decision boundary that generalises well to unseen data. By avoiding overfitting, SVM can make accurate predictions on new instances from the heart disease dataset. This characteristic is particularly beneficial when working with heart disease datasets where obtaining sufficient amount of data can be challenging.

Overall, the effectiveness of Support Vector Machines in high-dimensional spaces, its robustness to outliers, flexibility in kernel selection, performance with small to medium-sized datasets and ability to avoid overfitting make it a justified choice for heart disease dataset machine learning.

4.3 Decision Tree

Then, Decision Tree (DT) is a graphical representation of all the possible solutions to a decision based on certain conditions. Tree models where the target variable can take a finite set of values are called classification trees and target variables can take continuous values (numbers) are called regression trees. It can handle both numerical and categorical data.

Firstly, Decision tree is easy to visualise and interpret. Decision tree provides a transparent and interpretable model for heart disease prediction. The structure of a decision tree consists of a series of decision nodes and leaf nodes, where each decision node represents a feature and a split criterion. The tree-like structure of decision trees represents a series of if-else rules that mimic human decision-making. Each node in the tree corresponds to a decision based on a feature, leading to a branch that determines the outcome. This transparency makes it easier for users to understand and interpret the model's predictions, which can be important in domains where interpretability is crucial. The decision rules learned by the tree can provide valuable insights into the risk factors and symptoms associated with heart disease.

Decision tree can effectively handle nonlinear relationships and capture complex interactions between features, which is essential for heart disease machine learning. The heart disease dataset contains complex interactions between risk factors and symptoms that do not follow linear patterns. By recursively splitting the data based on different features and their thresholds, decision trees can partition the feature space into regions with homogeneous classes.

This ability to divide the feature space into non-linear decision boundaries allows decision trees to model intricate relationships between features, making them suitable for tasks where such relationships exist such as effectively identifying combinations of risk factors that contribute to heart disease.

This algorithm can handle various types of data, including discrete and continuous values. Unlike some classifiers that require extensive data preprocessing, decision trees can work directly with raw data. Continuous values can be converted into categorical values by setting appropriate thresholds, allowing the algorithm to split the data based on these categories.

Furthermore, the heart disease dataset may have missing values, which can be problematic for some algorithms. Decision tree can handle missing values by simply considering them as a separate category during the splitting process. The decision tree can treat missing values as a distinct category during the splitting process. Instead of excluding instances with missing values from the analysis, decision trees create a separate branch or node to handle these instances. Alternatively, decision tree can also perform missing value imputation during the training phase. Instead of treating missing values as a separate category, Decision Tree can estimate or impute the missing values based on the available data. Various imputation techniques can be used, such as mean imputation, median imputation, or imputation based on statistical modelling. This allows decision trees to utilise complete instances for learning and decision-making.

Plus, decision tree can be used for both classification and regression tasks, making them more versatile compared to some other algorithms. While some classifiers are specifically designed for one type of task, decision trees can handle both scenarios effectively. Besides, ID3, C4.5, and CART are decision tree algorithms that offer different advantages and considerations for heart disease machine learning. ID3 is simple and interpretable but limited in handling continuous features and missing values. C4.5 extends ID3 to handle mixed attribute types and missing values, while also providing pruning capabilities. CART further extends decision trees to regression tasks, making it suitable for heart disease datasets requiring both classification and regression analyses by using the Gini coefficient.

In other words, the interpretability and easy to visualise, ability to handle nonlinearity and feature interactions, flexibility in terms of data types, handling missing values, and supporting both classification and regression tasks make decision tree a justified choice for heart disease dataset machine learning.

5.0 STEPS TO BUILD MACHINE LEARNING PROBLEM

Step 1: Importing data

Importing the necessary libraries for reading the data.

```
In [62]: import numpy as np
         import numpy.random as npr
         import matplotlib.pyplot as plt
         import pandas as pd
```

Figure 5.0.1 Import libraries.

Set the directory of the file:

```
In [63]: cd C:/Users/yewch/Python Lecture/CPC152 Assignment 2
         C:\Users\yewch\Python Lecture\CPC152 Assignment 2
```

Figure 5.0.2 Set file directory.

Read the dataset that has been downloaded into the directory and use panda and display the dataset.

```
In [64]: df = pd.read_csv("heart disease.csv")
         df
```

Out[64]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

303 rows × 14 columns

```
In [66]: # get summary of numerical variables
         df.describe()
```

Out[66]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.00
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.31
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.61
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.00
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.00
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.00
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.00

Figure 5.0.3 Read dataset and display dataset.

Step 2: Data Cleaning

Check if there are any null values in the dataset, if yes, replace with mean of the dataset, if not leave it be.

As shown below, there are no null values in the dataset, do nothing.

```
In [67]: #check the number of null in the dataset
         df.apply(lambda x: sum(x.isnull()),axis=0)

Out[67]: age         0
         sex         0
         cp          0
         trestbps    0
         chol        0
         fbs         0
         restecg     0
         thalach     0
         exang       0
         oldpeak     0
         slope       0
         ca          0
         thal        0
         target      0
         dtype: int64
```

Figure 5.0.4 No null values.

Next, check if there are any duplicated values in the dataset. In the dataset “heart disease.csv”, we have found out one duplicated row in this dataset.

```
In [69]: duplicatedRows = df[df.duplicated()]
         duplicatedRows
```

Out[69]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
164	38	1	2	138	175	0	1	173	0	0.0	2	4	2	1

Figure 5.0.5 Duplicated dataset.

We drop off the duplicated row in the dataset, ensuring there is no same data overlapping affecting the results.

```
In [70]: # drop the duplicated row of the dataset

         df.drop_duplicates()
```

Out[70]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

302 rows × 14 columns

Figure 5.0.6 Drop duplicated row.

Step 3: Splitting the dataset

Import the necessary library for splitting the dataset into test set and train set.

```
In [71]: # Splitting the Dataset into 80% train set 20% test set
         from sklearn.model_selection import train_test_split


         # get the locations
         X = df.iloc[:, :-1]
         y = df.iloc[:, -1]

         # split the dataset
         X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.2, random_state=0)
```

Figure 5.0.7 Importing library for splitting dataset.

First, we test out the basic fitting for 80% train set and 20% test set. Display out the train set and test set to check if the splitting is correct.

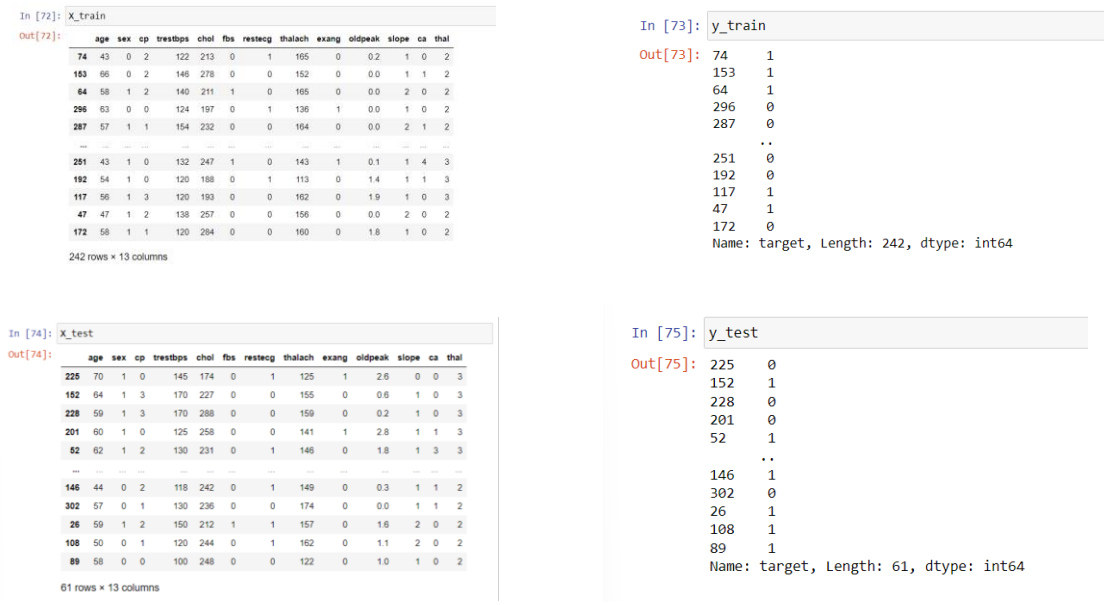


Figure 5.0.8 Display 80% train set and 20% test set.

5.1 KNN (K-Nearest Neighbours)

Performing basic fitting with different test ratios

Do basic fitting for the 80% train set and 20% test set, which includes all the features and make the prediction.

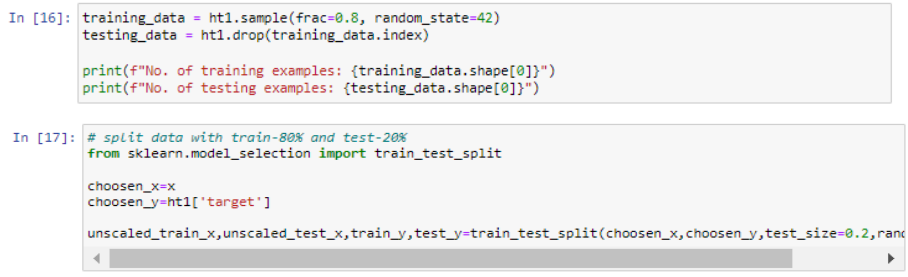


Figure 5.1.1 Basic fitting with all features.

Standard Scaling

We perform standard scaling to ensure that the distance calculations, kernel functions, regularization parameter, and feature influence are all balanced and fair.

```
In [18]: from sklearn.preprocessing import StandardScaler

         scaler=StandardScaler()
         train_x=scaler.fit_transform(unscaled_train_x)
         test_x=scaler.fit_transform(unscaled_test_x)

         print(pd.DataFrame(ht1, columns=ht1.columns))
         print(train_x)
         print(test_x)
```

Figure 5.1.2 KNN standard scaling.

K values

Find the k value using the loop to get the best k value.

```
In [19]: from sklearn.neighbors import KNeighborsClassifier
         from sklearn import metrics

         #Try running from k=1 through 25 and record testing accuracy
         k_range = range(1,26)
         scores = {}
         scores_list = []
         for k in k_range:
                 knn = KNeighborsClassifier(n_neighbors=k) #default is 5
                 knn.fit(train_x,train_y)
                 pred_y=knn.predict(test_x)
                 scores[k] = metrics.accuracy_score(test_y,pred_y)
                 scores_list.append(metrics.accuracy_score(test_y,pred_y))

In [20]: #show score of k
         scores

In [21]: %matplotlib inline
         import matplotlib.pyplot as plt

         #plot the relationship between K and the testing accuracy
         plt.plot(k_range,scores_list)
         plt.xlabel('Value of K for KNN')
         plt.ylabel('Testing Accuracy')

Out[21]: Text(0, 0.5, 'Testing Accuracy')
```

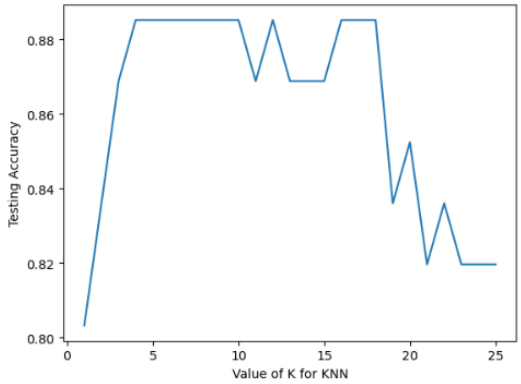


Figure 5.1.3 Value of *k*.

Predict the target

Predict the target using [8:2] ratio with k = 5 and using all features.

```
In [22]: #predict 'target'
knn=KNeighborsClassifier(n_neighbors=5)
knn.fit(train_x,train_y)
knnPredict=knn.predict(test_x)
knnPredict

Out[22]: array([0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
                0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1],
               dtype=int64)

In [23]: from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, ac

print ('Accuracy:', accuracy_score(test_y, knnPredict))
print ('Recall:', recall_score(test_y, knnPredict, average="weighted"))
print ('Precision:', precision_score(test_y, knnPredict, average="weighted"))
confusion= confusion_matrix(test_y, knnPredict)
print('Confusion matrix:')
print(confusion)

Accuracy: 0.8852459016393442
Recall: 0.8852459016393442
Precision: 0.8858452317997532
Confusion matrix:
[[26  3]
 [ 4 28]]
```

Figure 5.1.4 [8:2] ratio with *k* = 5.

Predict the target using [7:3] ratio with k = 8 and using all features.

```
In [31]: #predict 'target'
knn=KNeighborsClassifier(n_neighbors=8)
knn.fit(train_x,train_y)
knnPredict=knn.predict(test_x)
knnPredict

Out[31]: array([0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
                0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1,
                1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1,
                1, 1, 1], dtype=int64)

In [32]: from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, ac

print ('Accuracy:', accuracy_score(test_y, knnPredict))
print ('Recall:', recall_score(test_y, knnPredict, average="weighted"))
print ('Precision:', precision_score(test_y, knnPredict, average="weighted"))
confusion = confusion_matrix(test_y, knnPredict)
print('Confusion matrix:')
print(confusion)

Accuracy: 0.8791208791208791
Recall: 0.8791208791208791
Precision: 0.8790994371482177
Confusion matrix:
[[36  6]
 [ 5 44]]
```

Figure 5.1.5 [7:3] ratio with *k* = 8.

Predict the target using [6:4] ratio with k = 11 and using all features.

```
In [40]: #predict 'target'
knn=KNeighborsClassifier(n_neighbors=11)
knn.fit(train_x,train_y)
knnPredict=knn.predict(test_x)
knnPredict

Out[40]: array([0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0,
                0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,
                1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,
                1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1,
                1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0], dtype=int64)

In [41]: from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, ac

print ('Accuracy:', accuracy_score(test_y, knnPredict))
print ('Recall:', recall_score(test_y, knnPredict, average="weighted"))
print ('Precision:', precision_score(test_y, knnPredict, average="weighted"))
confusion = confusion_matrix(test_y, knnPredict)
print('Confusion matrix:')
print(confusion)

Accuracy: 0.8677685950413223
Recall: 0.8677685950413223
Precision: 0.8675497613781864
Confusion matrix:
[[43  9]
 [ 7 62]]
```

Figure 5.1.6 [6:4] ratio with k = 11.

Prediction with Feature Selection

```
In [11]: #import library
         import pandas as pd
         import numpy as np
         from sklearn.feature_selection import SelectKBest
         from sklearn.feature_selection import chi2

In [12]: from sklearn.ensemble import ExtraTreesClassifier
         import matplotlib.pyplot as plt

         model = ExtraTreesClassifier()
         model.fit(x,y)

         #use inbuilt class feature_importances of tree based classifiers
         print(model.feature_importances_)

In [13]: #plot graph of feature importances for better visualization
         feat_importances = pd.Series(model.feature_importances_, index=x.columns)
         feat_importances.nlargest(5).plot(kind='barh')

         plt.show()
```

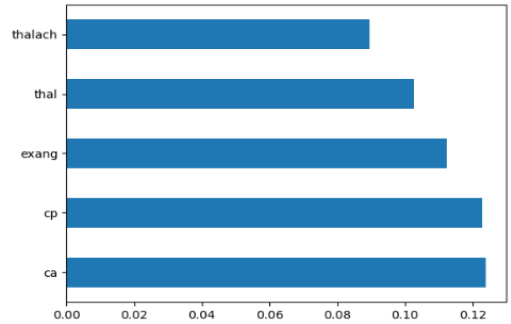


Figure 5.1.7 Prediction with feature selection

Using 5 best features [8:2]:

```
In [43]: features = ['thal','exang','ca','cp','thalach']
         x = ht1.loc[:, features]
         y = ht1.loc[:, ['target']]

         choosen_x=x
         choosen_y=ht1['target']
         unscaled_train_x,unscaled_test_x,train_y,test_y=train_test_split(choosen_x,choosen_y,test_size=0.2,ran

         from sklearn.preprocessing import StandardScaler

         scaler=StandardScaler()
         train_x=scaler.fit_transform(unscaled_train_x)
         test_x=scaler.fit_transform(unscaled_test_x)

In [46]: #predict 'target'
         knn=KNeighborsClassifier(n_neighbors=23)
         knn.fit(train_x,train_y)
         knnPredict=knn.predict(test_x)
         knnPredict

Out[46]: array([0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1,
                0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
                1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1], dtype=int64)

In [47]: from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, ac

         print ('Accuracy:', accuracy_score(test_y, knnPredict))
         print ('Recall:', recall_score(test_y, knnPredict, average="weighted"))
         print ('Precision:', precision_score(test_y, knnPredict, average="weighted"))
         confusion = confusion_matrix(test_y, knnPredict)
         print('Confusion matrix:')
         print(confusion)

         Accuracy: 0.8852459016393442
         Recall: 0.8852459016393442
         Precision: 0.8877935312361541
         Confusion matrix:
         [[22  5]
          [ 2 32]]
```

Figure 5.1.8 5 best features [8:2]

Using 4 best features [8:2]:

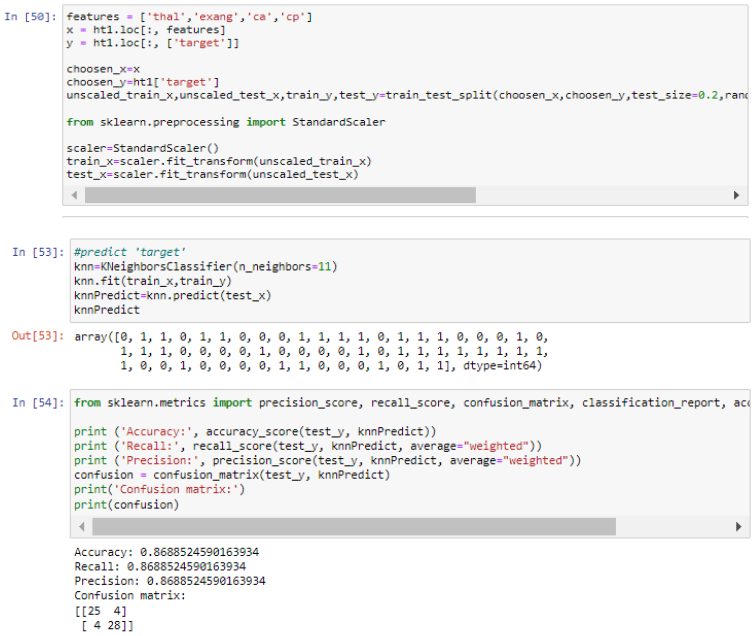


Figure 5.1.9 4 best features [8:2]

Using 3 best features [8:2]:

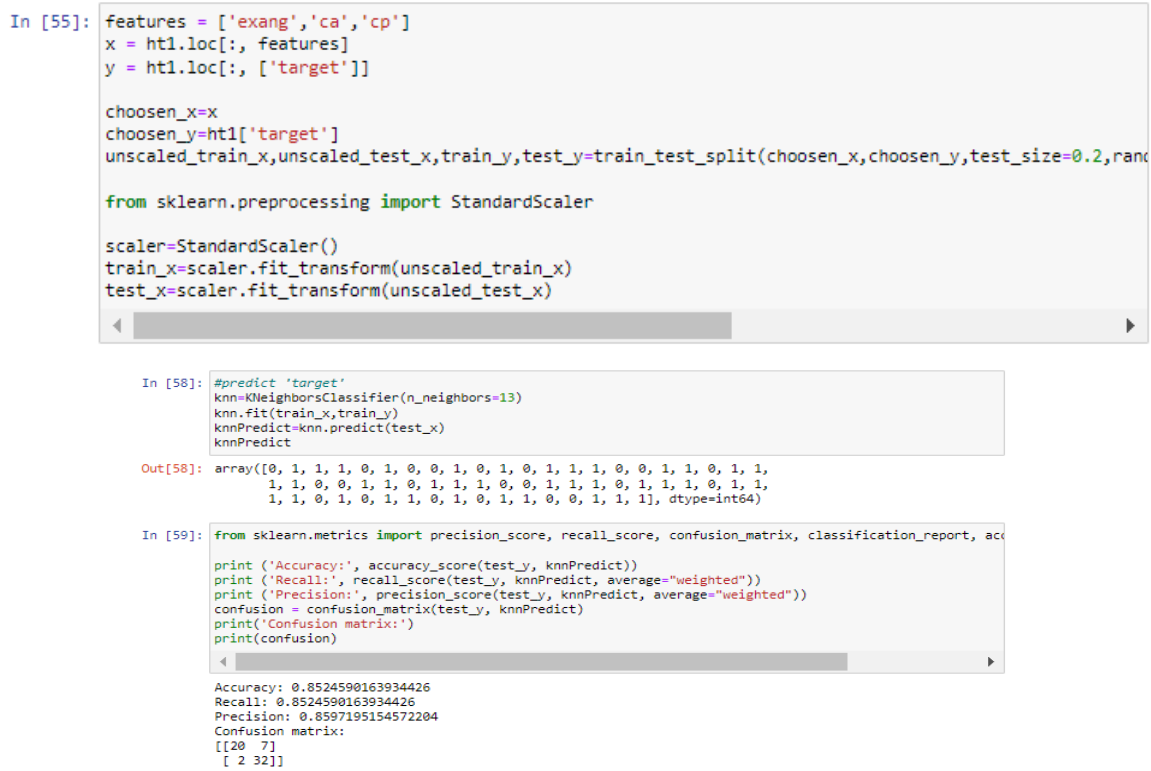


Figure 5.1.10 3 best features [8:2]

5.2 SVM (Support Vector Machine)

<u>Standard Scaling</u>

We perform standard scaling to ensure that the distance calculations, kernel functions, regularization parameter, and feature influence are all balanced and fair.

```
In [76]: #perform standard scaling
         from sklearn.preprocessing import StandardScaler

         scaler = StandardScaler()
         # Fit the scaler on the training data
         scaler.fit(X_train)

         # Transform the training and test sets
         X_train_scaled = scaler.transform(X_train)
         X_test_scaled = scaler.transform(X_test)
```

Figure 5.2.1 SVM standard scaling.

Performing basic fitting with different test ratios

Do basic fitting for the 80% train set and 20% test set, which includes all the features and make the prediction.

```
In [77]: #import models from sckit learn module:
         from sklearn import svm
         from sklearn import metrics

In [78]: model = svm.SVC()

In [79]: model.fit(X_train_scaled,y_train)

In [80]: predictions =  model.predict(X_test_scaled)
         predictions

Out[80]: array([0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0,
                1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1], dtype=int64)
```

Figure 5.2.2 Basic fitting 80% train set and 20% test set.

Display how well the machine learning model performs in terms of accuracy, recall, precision, and overall classification performance so that we can make informed decisions about model improvement or deployment.

```
In [81]: from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, accuracy_score, f1_score

         print ('Accuracy:', accuracy_score(y_test, predictions))
         print ('Recall:', recall_score(y_test, predictions, average="weighted"))
         print ('Precision:', precision_score(y_test, predictions, average="weighted"))
         confusion = confusion_matrix(y_test, predictions)
         print('Confusion matrix:')
         print(confusion)

         Accuracy: 0.8688524590163934
         Recall: 0.8688524590163934
         Precision: 0.873504145252654
         Confusion matrix:
         [[21  6]
          [ 2 32]]
```

Figure 5.2.3 Overall SVM performance.

Splitting the train set and test set into 70% and 30%, perform standard scaling and perform basic fitting with all features included for the predictive outcome.

```
In [82]: # Splitting the Dataset into 70% train set and 30% test set

         # get the locations
         X = df.iloc[:, :-1]
         y = df.iloc[:, -1]

         # split the dataset
         X_train2, X_test2, y_train2, y_test2 = train_test_split(
             X, y, test_size=0.3, random_state=0)
```

```
In [87]: #perform standard scaling

         scaler = StandardScaler()
         # Fit the scaler on the training data
         scaler.fit(X_train2)

         # Transform the training and test sets
         X_train_scaled2 = scaler.transform(X_train2)
         X_test_scaled2 = scaler.transform(X_test2)
```

```
In [88]: model.fit(X_train_scaled2,y_train2)
```

```
In [89]: predictions2 = model.predict(X_test_scaled2)
         predictions2
```

```
Out[89]: array([0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0,
                1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1,
                1, 1, 0], dtype=int64)
```

```
In [90]:
         print ('Accuracy:', accuracy_score(y_test2, predictions2))
         print ('Recall:', recall_score(y_test2, predictions2, average="weighted"))
         print ('Precision:', precision_score(y_test2, predictions2, average="weighted"))
         confusion = confusion_matrix(y_test2, predictions2)
         print('Confusion matrix:')
         print(confusion)
```

```
Accuracy: 0.8131868131868132
Recall: 0.8131868131868132
Precision: 0.8248430141287284
Confusion matrix:
[[31 13]
 [ 4 43]]
```

Figure 5.2.4 Basic scaling 70% train set, 60% test set.

Next, we continue with splitting the test set into 60% train set 40% test set and repeat the above steps, standard scaling and make predictions.

```
In [91]: # Splitting the Dataset into 60% train set and 40% test set

         # get the locations
         X = df.iloc[:, :-1]
         y = df.iloc[:, -1]

         # split the dataset
         X_train3, X_test3, y_train3, y_test3 = train_test_split(
             X, y, test_size=0.4, random_state=0)
```

```
In [96]: #perform standard scaling

         scaler = StandardScaler()
         # Fit the scaler on the training data
         scaler.fit(X_train3)

         # Transform the training and test sets
         X_train_scaled3 = scaler.transform(X_train3)
         X_test_scaled3 = scaler.transform(X_test3)
```

```
In [97]: model.fit(X_train_scaled3,y_train3)
```

```
In [98]: predictions3 = model.predict(X_test_scaled3)
         predictions3

Out[98]: array([0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0,
                1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1,
                1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1,
                0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1], dtype=int64)

In [99]: print ('Accuracy:', accuracy_score(y_test3, predictions3))
         print ('Recall:', recall_score(y_test3, predictions3, average="weighted"))
         print ('Precision:', precision_score(y_test3, predictions3, average="weighted"
         confusion = confusion_matrix(y_test3, predictions3)
         print('Confusion matrix:')
         print(confusion)

         Accuracy: 0.8114754098360656
         Recall: 0.8114754098360656
         Precision: 0.8209182543198937
         Confusion matrix:
         [[42 17]
          [ 6 57]]
```

Figure 5.2.5 Basic scaling 60% train set, 40% test set.

Prediction with Feature Selection

For feature selection, we will be using the Extra Tree Classifier method to determine which features affect the results in accordance.

Import the library for Extra Trees Classifier

```
In [100]: from sklearn.feature_selection import RFE
```

Figure 5.2.6 Import library for Extra Trees Classifier.

Ensure to locate the features and target correctly.

```
In [101]: X = df.iloc[:,0:13] #features
          y = df.iloc[:,-1] # target column / label --> target

In [102]: from sklearn.ensemble import ExtraTreesClassifier
          import matplotlib.pyplot as plt

          model = ExtraTreesClassifier()
          model.fit(X,y)
          print(model.feature_importances_)

          [0.07077515 0.05357731 0.1200831  0.0613566  0.06021249 0.01955509
           0.03753972 0.08599223 0.10463324 0.08488018 0.06661331 0.12956523
           0.10521634]
```

Figure 5.2.7 Locate features and targets.

Plot out the graph with the 10 largest relations to the target we are predicting.

```
In [103]: #plotting of graph to show and have better visualization

          feat_importances = pd.Series(model.feature_importances_,index=X.columns)
          feat_importances.nlargest(10).plot(kind='barh')
          plt.show()
```

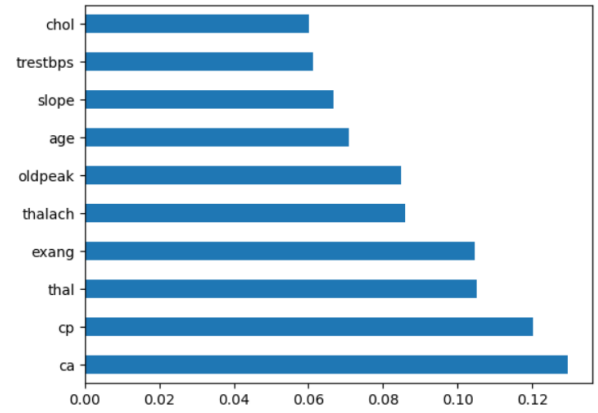


Figure 5.2.8 Graph of 10 largest target relations.

1st Attempt with SVM (5 features) [8:2]

Select the 5 features with the highest score using the location of the column of each feature.

```
In [108]: model = svm.SVC()

In [136]: model.fit(X_train_scaled[:, [2,11,12,7,8]], y_train)

In [138]: predictions_1 = model.predict(X_test_scaled[:, [2,11,12,7,8]])
          predictions_1

Out[138]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1], dtype=int64)

In [139]: print ('Accuracy:', accuracy_score(y_test, predictions_1))
          print ('Recall:', recall_score(y_test, predictions_1, average="weighted"))
          print ('Precision:', precision_score(y_test, predictions_1, average="weighted"
          confusion = confusion_matrix(y_test, predictions_1)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.8688524590163934
          Recall: 0.8688524590163934
          Precision: 0.869471766848816
          Confusion matrix:
          [[22  5]
           [ 3 31]]
```

Figure 5.2.9 Select 5 features with highest score [8:2].

1st Attempt with SVM (5 features) [7:3]

Repeat the same steps.

```
In [140]: model.fit(X_train_scaled2[:, [2,11,12,7,8]], y_train2)
```

```
In [141]: predictions_2 = model.predict(X_test_scaled2[:, [2,11,12,7,8]])
          predictions_2

Out[141]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
                 1, 0, 0], dtype=int64)

In [142]: print ('Accuracy:', accuracy_score(y_test2, predictions_2))
          print ('Recall:', recall_score(y_test2, predictions_2, average="weighted"))
          print ('Precision:', precision_score(y_test2, predictions_2, average="weighted
          confusion = confusion_matrix(y_test2, predictions_2)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.8241758241758241
          Recall: 0.8241758241758241
          Precision: 0.8335886335886337
          Confusion matrix:
          [[32 12]
           [ 4 43]]
```

Figure 5.2.10 Select 5 features with highest score [7:3].

1st Attempt with SVM (5 features) [6:4]

```
In [143]: model.fit(X_train_scaled3[:, [2,11,12,7,8]], y_train3)

In [144]: predictions_3 = model.predict(X_test_scaled3[:, [2,11,12,7,8]])
          predictions_3

Out[144]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1,
                 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
                 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,
                 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], dtype=int64)

In [145]: print ('Accuracy:', accuracy_score(y_test3, predictions_3))
          print ('Recall:', recall_score(y_test3, predictions_3, average="weighted"))
          print ('Precision:', precision_score(y_test3, predictions_3, average="weighted
          confusion = confusion_matrix(y_test3, predictions_3)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.8032786885245902
          Recall: 0.8032786885245902
          Precision: 0.8106573417599692
          Confusion matrix:
          [[42 17]
           [ 7 56]]
```

Figure 5.2.11 Select 5 features with highest score [6:4].

2nd Attempt with SVM (4 features) [8:2]

```
In [125]: model.fit(X_train_scaled[:, [2,8,11,12]], y_train)
```

```
In [126]: predictions_4 = model.predict(X_test_scaled[:, [2,8,11,12]])
          predictions_4

Out[126]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1], dtype=int64)

In [127]: print ('Accuracy:', accuracy_score(y_test, predictions_4))
          print ('Recall:', recall_score(y_test, predictions_4, average="weighted"))
          print ('Precision:', precision_score(y_test, predictions_4, average="weighted"
          confusion = confusion_matrix(y_test, predictions_4)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.8360655737704918
          Recall: 0.8360655737704918
          Precision: 0.8395918520463667
          Confusion matrix:
          [[20  7]
           [ 3 31]]
```

Figure 5.2.12 Select 4 features with highest score [8:2].

2nd Attempt with SVM (4 features) [7:3]

```
In [128]: model.fit(X_train_scaled2[:, [2,8,11,12]], y_train2)

In [129]: predictions_5 = model.predict(X_test_scaled2[:, [2,8,11,12]])
          predictions_5

Out[129]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1,
                 1, 1, 0], dtype=int64)

In [130]: print ('Accuracy:', accuracy_score(y_test2, predictions_5))
          print ('Recall:', recall_score(y_test2, predictions_5, average="weighted"))
          print ('Precision:', precision_score(y_test2, predictions_5, average="weighted
          confusion = confusion_matrix(y_test2, predictions_5)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.8241758241758241
          Recall: 0.8241758241758241
          Precision: 0.8395422946506538
          Confusion matrix:
          [[31 13]
           [ 3 44]]
```

Figure 5.2.13 Select 4 features with highest score [7:3].

2nd Attempt with SVM (4 features) [6:4]

```
In [131]: model.fit(X_train_scaled3[:, [2,8,11,12]], y_train3)
```

```
In [132]: predictions_6 = model.predict(X_test_scaled3[:, [2,8,11,12]])
          predictions_6

Out[132]: array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
                 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
                 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1,
                 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,
                 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], dtype=int64)

In [133]:
          print ('Accuracy:', accuracy_score(y_test3, predictions_6))
          print ('Recall:', recall_score(y_test3, predictions_6, average="weighted"))
          print ('Precision:', precision_score(y_test3, predictions_6, average="weighted"))
          confusion = confusion_matrix(y_test3, predictions_6)
          print('Confusion matrix:')
          print(confusion)

          Accuracy: 0.819672131147541
          Recall: 0.819672131147541
          Precision: 0.8276007461147495
          Confusion matrix:
          [[43 16]
           [ 6 57]]
```

Figure 5.2.14 Select 4 features with highest score [6:4].

5.3 Decision Tree

Selection of features

First, the data is visualised as a heat map and in the target row we select the five features with the highest correlation coefficients. cp, thalach, exang, oldpeak, ca.

```
In [111]: import matplotlib.pyplot as plt
          import seaborn as sb
          corr=df.corr()
          X=corr.index
          plt.figure(figsize=(20,20))
          g=sb.heatmap(corr, annot=True, cmap="RdYlGn")
          plt.show()
```

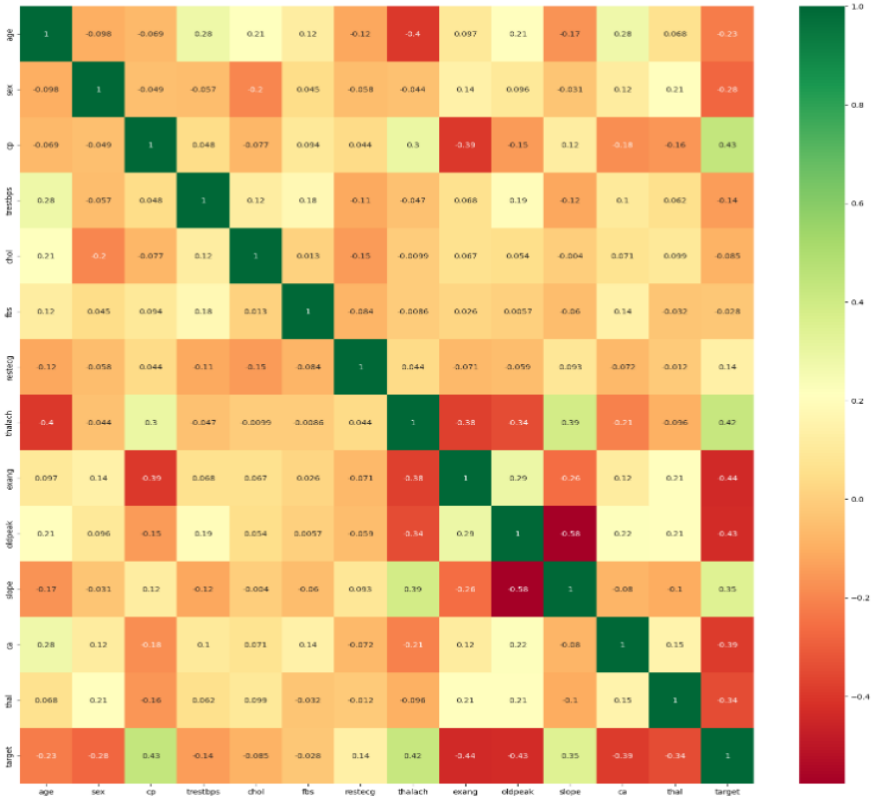


Figure 5.3.1 Heat map of five features with highest correlation.

Splitting

Next, splitting the data, with 80% as the training set and 20% as the test set.

```
In [55]: from sklearn.model_selection import train_test_split

         X = df.iloc[:, :-1]
         y = df.iloc[:, -1]

         X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.2, random_state=42)

In [56]: train=X_train.assign(target=y_train)
         test=X_test.assign(target=y_test)
```

Figure 5.3.2 Splitting 80% train set, 20% test set.

Similarly, modify test_size to 0.3 or 0.4, corresponding to 70% and 60% as the training set. To eliminate errors in the dataset after splitting, we set random_state to a fixed number 42.

Determining the parameters of decision tree

The best parameters can be determined quickly using random search, but using grid search is more stable and has a higher accuracy score.

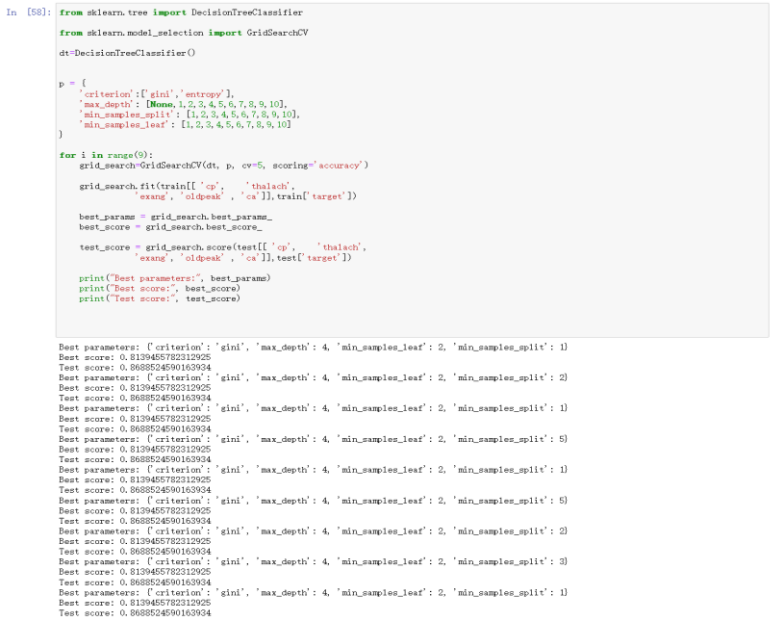
Random search (80:20):

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import RandomizedSearchCV

from scipy.stats import randint

for i in range(10):
    dt=DecisionTreeClassifier()

    p= {
        'criterion':['gini','entropy'],
        'max_depth': randint(1, 10),
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10),
    }


    random_search = RandomizedSearchCV(estimator=dt, param_distributions=p, cv=4, n_iter=50)

    random_search.fit(train[[ 'cp',     'thalach',
           'exang', 'oldpeak' , 'ca']],train['target'])

    print("Best Parameters: ", random_search.best_params_)
    print("Best Score: ", random_search.best_score_)
```

```
Best Parameters:  {'criterion': 'gini', 'max_depth': 8, 'min_samples_leaf': 9, 'min_samples_split': 7}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 9, 'min_samples_split': 4}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best Score:  0.8014344262295082
Best Parameters:  {'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 8, 'min_samples_split': 4}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 9, 'min_samples_split': 5}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 9, 'min_samples_split': 14}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 9, 'min_samples_split': 3}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 9, 'min_samples_split': 18}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'entropy', 'max_depth': 9, 'min_samples_leaf': 9, 'min_samples_split': 6}
Best Score:  0.7974726775956285
Best Parameters:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 3, 'min_samples_split': 15}
Best Score:  0.8014344262295082
```

Figure 5.3.3 Random search (80:20).

Grid search (80:20):



```
In [58]: from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import GridSearchCV

dt=DecisionTreeClassifier()


p = {
    'criterion':['gini','entropy'],
    'max_depth': [None,1,2,3,4,5,6,7,8,9,10],
    'min_samples_split': [1,2,3,4,5,6,7,8,9,10],
    'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]
}

for i in range(9):
    grid_search=GridSearchCV(dt, p, cv=5, scoring='accuracy')

    grid_search.fit(train[[ 'cp',    'thalach',
                'exang', 'oldpeak' , 'ca']],train['target'])

    best_params = grid_search.best_params_
    best_score = grid_search.best_score_

    test_score = grid_search.score(test[[ 'cp',    'thalach',
                'exang', 'oldpeak' , 'ca']],test['target'])

    print("Best parameters:", best_params)
    print("Best score:", best_score)
    print("Test score:", test_score)
```

```
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 5}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 5}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 3}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.8139455782312925
Test score: 0.8688524590163934
```

Figure 5.3.4 Grid search (80:20).

We choose grid search and use the most frequent best parameter to set our decision tree.

Grid search (70:30):

```
for i in range(9):
    grid_search=GridSearchCV(dt, p, cv=5, scoring='accuracy')

    grid_search.fit(train[[ 'cp',    'thalach',
                'exang', 'oldpeak' , 'ca']],train['target'])

    best_params = grid_search.best_params_
    best_score = grid_search.best_score_

    test_score = grid_search.score(test[[ 'cp',    'thalach',
                'exang', 'oldpeak' , 'ca']],test['target'])

    print("Best parameters:", best_params)
    print("Best score:", best_score)
    print("Test score:", test_score)
```

```
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 10}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 9}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 9}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 10}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 10}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 9}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 10}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 10}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 9}
Best score: 0.8161683277962348
Test score: 0.8351648351648352
```

Figure 5.3.5 Grid search (70:30).

Grid search (60:40):

```
In [64]: for i in range(9):
             grid_search=GridSearchCV(dt, p, cv=5, scoring='accuracy')

             grid_search.fit(train[[ 'cp',    'thalach',
                        'exang', 'oldpeak' , 'ca']],train['target'])

             best_params = grid_search.best_params_
             best_score = grid_search.best_score_

             test_score = grid_search.score(test[[ 'cp',    'thalach',
                        'exang', 'oldpeak' , 'ca']],test['target'])

             print("Best parameters:", best_params)
             print("Best score:", best_score)
             print("Test score:", test_score)


Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 4, 'min_samples_split': 5}
Best score: 0.7791291291291291
Test score: 0.8688524590163934
Best parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 4, 'min_samples_split': 4}
Best score: 0.7791291291291291
Test score: 0.8688524590163934
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.7843843843843844
Test score: 0.8770491803278688
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best score: 0.7843843843843844
Test score: 0.8770491803278688
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 2}
Best score: 0.7788288288288289
Test score: 0.860655737704918
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 1}
Best score: 0.7788288288288289
Test score: 0.860655737704918
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best score: 0.7843843843843844
Test score: 0.8770491803278688
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 1}
Best score: 0.7843843843843844
Test score: 0.8770491803278688
Best parameters: {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 1}
Best score: 0.7788288288288289
Test score: 0.860655737704918
```

Figure 5.3.6 Grid search (60:40).

Training model and prediction

80% as train set:

```
In [59]: from sklearn.tree import DecisionTreeClassifier
         import timeit


         model=DecisionTreeClassifier(criterion='gini',max_depth= 4, min_samples_leaf= 2, min_samples_split= 1)


         model.fit(train[['cp',    'thalach',
                'exang', 'oldpeak' , 'ca']],train['target'])
         start=timeit.default_timer()

         predictions=model.predict(test[['cp',    'thalach',
                'exang', 'oldpeak' , 'ca']])

         process=timeit.default_timer()-start

         test=test.assign(Prediction=predictions)
         from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, classification_report

         print ('Accuracy:', accuracy_score(test['target'], predictions))
         print ('Recall:', recall_score(test['target'], predictions, average="weighted"))
         print ('Precision:', precision_score(test['target'], predictions, average="weighted"))
         print ('F1 Score:', f1_score(test['target'], predictions, average="weighted"))

         confusion = confusion_matrix(test['target'], predictions)
         print('Confusion matrix:')
         print(confusion)

         print('processing time per entity:')
         print(process/len(test))

Accuracy: 0.8688524590163934
Recall: 0.8688524590163934
Precision: 0.8700310725383049
F1 Score: 0.8684976225959832
Confusion matrix:
[[24  5]
 [ 3 29]]
processing time per entity:
3.844590155793293e-05
```

Figure 5.3.7 80% train set and prediction.

70% as train set:

```
from sklearn.tree import DecisionTreeClassifier
import timeit


model=DecisionTreeClassifier(criterion='entropy',max_depth= 4, min_samples_leaf= 1, min_samples_split= 9)


model.fit(train[['cp',    'thalach',
       'exang', 'oldpeak' , 'ca']],train['target'])
start=timeit.default_timer()

predictions=model.predict(test[['cp',    'thalach',
       'exang', 'oldpeak' , 'ca']])

process2=timeit.default_timer()-start

test=test.assign(Prediction=predictions)
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, classification_report

print ('Accuracy:', accuracy_score(test['target'], predictions))
print ('Recall:', recall_score(test['target'], predictions, average="weighted"))
print ('Precision:', precision_score(test['target'], predictions, average="weighted"))
print ('F1 Score:', f1_score(test['target'], predictions, average="weighted"))

confusion = confusion_matrix(test['target'], predictions)
print('Confusion matrix:')
print(confusion)

print('processing time per entity:')
print(process2/len(test))
```

```
Accuracy: 0.8351648351648352
Recall: 0.8351648351648352
Precision: 0.835688121402407
F1 Score: 0.835325317253028
Confusion matrix:
[[34  7]
 [ 8 42]]
processing time per entity:
2.7989011029533755e-05
```

Figure 5.3.8 70% train set and prediction.

60% as train set:

```
from sklearn.tree import DecisionTreeClassifier
import timeit


model=DecisionTreeClassifier(criterion='gini',max_depth= 4, min_samples_leaf= 1, min_samples_split= 1)


model.fit(train[['cp',    'thalach',
       'exang', 'oldpeak' , 'ca']],train['target'])
start=timeit.default_timer()

predictions=model.predict(test[['cp',    'thalach',
       'exang', 'oldpeak' , 'ca']])

process3=timeit.default_timer()-start

test=test.assign(Prediction=predictions)
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, classification_report

print ('Accuracy:', accuracy_score(test['target'], predictions))
print ('Recall:', recall_score(test['target'], predictions, average="weighted"))
print ('Precision:', precision_score(test['target'], predictions, average="weighted"))
print ('F1 Score:', f1_score(test['target'], predictions, average="weighted"))

confusion = confusion_matrix(test['target'], predictions)
print('Confusion matrix:')
print(confusion)

print('processing time per entity:')
print(process3/len(test))
```

```
Accuracy: 0.860655737704918
Recall: 0.860655737704918
Precision: 0.8614347168933845
F1 Score: 0.8595394304834666
Confusion matrix:
[[41 11]
 [ 6 64]]
processing time per entity:
1.9526229510522738e-05
```

Figure 5.3.9 60% train set and prediction.

6.0 COMPARISON AND RECOMMENDATION

6.1 Effect of Features on the Model

In the KNN model, we found that the accuracy score came to 0.8852 when using all parameters for the training data, the same as when using the best five parameters. The precision score, although high at 0.8858, is lower than the 0.8877 achieved when using the best five parameters.

Although the difference is not significant due to the small test set data, using the best five parameters is more advantageous than using all parameters for training**,** which may lead to increased model complexity, useless feature interference and overfitting.

When fewer features were used in the training set, the accuracy scores were all lower than the best five features.

When we discuss the ratio of the training set to the test set, it is clear that the accuracy score of 0.8852 for the first model (80% for train set) is higher than that of 0.8351 for the second model (70% for train set) and 0.8677 for the third model (60% for train set).

This can also be reflected in the SVM model and decision tree model. In the decision tree model, the results showed that the first model (80%) of the decision tree achieved a maximum accuracy score of 0.8689 and a precision score of over 0.87. The accuracy score of the other two decision tree models was 0.8351 and 0.8606. Although the speed of the third model (60%) is much faster (not much difference in scores), because the first model has too little data in the test set, the prediction error has a greater impact on the score, so we think the first model (80%) has to be more accurate due to the third model in decision tree.

In summary, we believe that the best model with the best five features for the data set, and with 80% and 20% for the training and test sets, respectively, is better than others.

6.2 Performance

For the prediction of heart disease, life is always more valuable than money, so our focus is on finding all possible patients, and some misdiagnosis, those who are not ill are diagnosed as ill, can be tolerated, so we prefer to focus on recall and accuracy.

Of these three models, the highest recall score was the best model in KNN with a recall score of 0.8852, while the best models in decision tree and SVM had equal recall scores of 0.8689.

However, KNN is the slowest of the three models.

Time of prediction:

KNN:

```
In [72]: #predict 'target'
knn=KNeighborsClassifier(n_neighbors=23)
knn.fit(train_x,train_y)

import timeit
start=timeit.default_timer()

knnPredict=knn.predict(test_x)

process=timeit.default_timer()-start

print('Time of prediction:',process )

Time of prediction: 0.0034204999974463135
```

Figure 6.2.1 KNN time prediction.

SVM:

```
: import timeit
start=timeit.default_timer()

predictions =  model.predict(X_test_scaled)

process=timeit.default_timer()-start

print('Time of prediction:',process )

Time of prediction: 0.0026331999979447573
```

Figure 6.2.2 SVM time prediction.

Decision Tree:

```
In [74]:
start=timeit.default_timer()

predictions=model.predict(test[['cp',     'thalach',
       'exang', 'oldpeak' , 'ca']])

process=timeit.default_timer()-start

print('Time of prediction:',process )

Time of prediction: 0.002999600001203362
```

Figure 6.2.3 Decision Tree time prediction.

In the above code we can see that the fastest model is the SVM with a prediction time of only 0.0026, followed by the decision tree with a prediction time close to 0.0030, while the KNN is close to 0.0034.

KNN is nearly 30% slower than SVM.

7.0 RESULTS AND DISCUSSION

7.1 Results

Result of KNN:

Table 7-1-1 KNN results.

	Best 5 features (Percentage of training set: 80%)	All features (Percentage of training set: 80%)	All features (Percentage of training set: 70%)	All features (Percentage of training set: 60%)
Accuracy	0.8852459016393442	0.8852459016393442	0.8351648351648352	0.8181818181818182
Recall	0.8852459016393442	0.8852459016393442	0.8351648351648352	0.8181818181818182
Precision	0.8877935312361541	0.8858452317997532	0.8363131121751812	0.8174553756745537
Confusion matrix	[[22 5] [2 32]]	[[26 3] [4 28]]	[[28 10] [5 48]]	[[38 12] [10 61]]

Result of SVM:

Table 7-1-2 SVM results.

	Best 5 features (Percentage of training set: 80%)	All features (Percentage of training set: 80%)	Best 5 features (Percentage of training set: 70%)	Best 5 features (Percentage of training set: 60%)
Accuracy	0.8688524590163934	0.8688524590163934	0.8241758241758241	0.8032786885245902
Recall	0.8688524590163934	0.8688524590163934	0.8241758241758241	0.8032786885245902
Precision	0.8694717668488816	0.8735041452526654	0.8335886335886337	0.8106573417599692
Confusion matrix	[[22 5] [3 31]]	[[21 6] [2 32]]	[[32 12] [4 43]]	[[42 17] [7 56]]

Result of Decision Tree:

Table 7-1-3 Decision Tree results.

	Best 5 features (Percentage of training set: 80%)	All features (Percentage of training set: 80%)	Best 5 features (Percentage of training set: 70%)	Best 5 features (Percentage of training set: 60%)
Accuracy	0.8688524590163934	0.8360655737704918	0.8351648351648352	0.8606557377704918
Recall	0.8688524590163934	0.8360655737704918	0.8351648351648352	0.8606557377704918
Precision	0.8700310725383049	0.8582966813274691	0.8356881214024407	0.8614347168933845
Confusion matrix	[[24 5] [3 29]]	[[19 9] [1 32]]	[[34 7] [8 42]]	[[41 11] [6 64]]

Result of comparison
(Percentage of training set: 80%):

Table 7-1-4 Comparison results.

	Best	Second	Last
Accuracy	KNN: 0.8852459016393442	SVM: 0.8688524590163934	Decision Tree: 0.8688524590163934
Prediction time	SVM: 0.00263319999794475	Decision Tree: 0.00299960000120336	KNN: 0.00342049999744631

7.2 Discussion

7.2.1 To effectively predict patients with heart disease by using machine learning algorithms.

As mentioned earlier, since our aim was to effectively predict heart disease, and we consider life above all else, our analysis favoured recall scores and accuracy scores (the higher the recall and accuracy scores, the more significant the effect). We then divided the data we obtained into a training set and a test set. We explored the proportions of the training and test sets. We then selected the five most relevant features after testing. After repeated experiments, (we focused on evaluating recall and accuracy), we selected the most appropriate algorithm to build the best model.

Table 7-2-1-1 KNN as the most suitable algorithm.

Best model in the most suitable algorithm (KNN)	Best 5 features (Percentage of training set: 80%)
Accuracy	0.8852459016393442
Recall	0.8852459016393442
Precision	0.8877935312361541
Confusion matrix	[[22 5] [2 32]]

7.2.2 Identification of important features

In the process of creating the model we used two different methods to compare the importance of various features.One way is to use '.feature_importances_' to get the importance values of every feature.

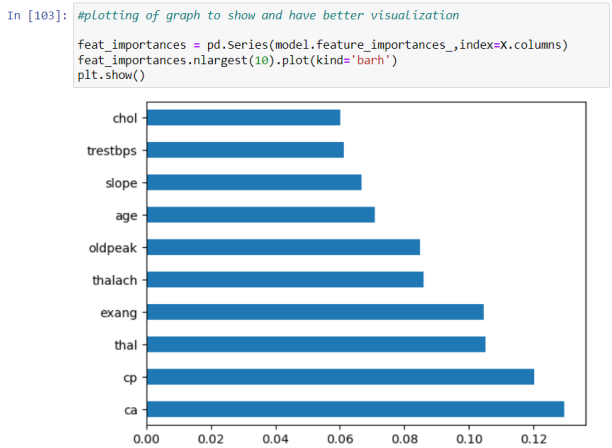


Figure 7.2.1.1 Graph of values of important features.

Another way is to get the correlation coefficient of each feature by '.corr()', the closer the value is to 1 or -1, the more relevant it is.

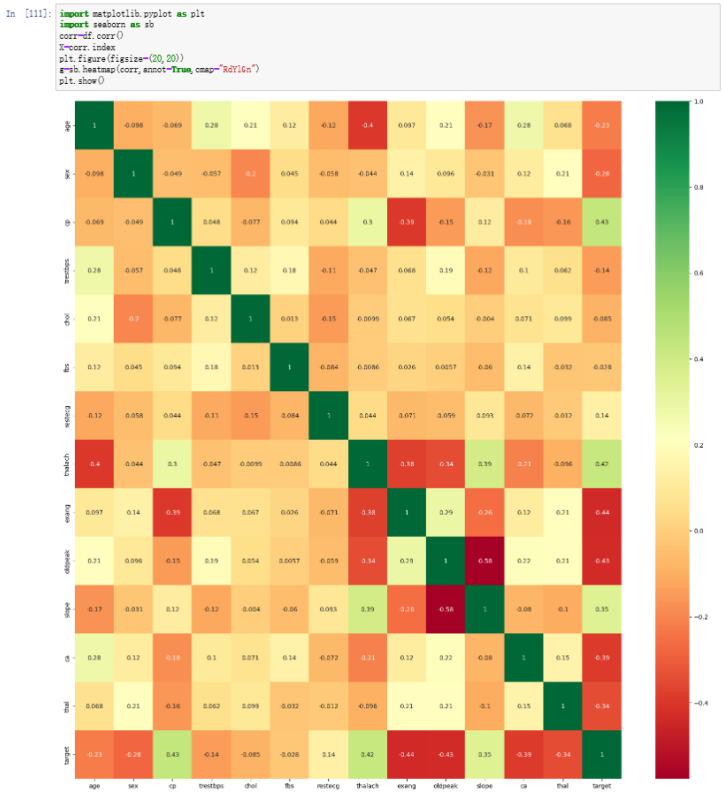


Figure 7.2.1.2 Heat map of important features.

We tested this by increasing the number of the kind of features in order of ranking, and the results showed that the model performed when the first five best features were selected over all the features selected.

Table 7-2-1-2 Performance of KNN for first five best features.

Example in KNN	Best 3 features (Percentage of training set: 80%)	Best 4 features (Percentage of training set: 80%)	Best 5 features (Percentage of training set: 80%)	All features (Percentage of training set: 80%)
Accuracy	0.8524590163934426	0.8688524590163934	0.8852459016393442	0.8852459016393442
Recall	0.8524590163934426	0.8688524590163934	0.8852459016393442	0.8852459016393442
Precision	0.8597195154572204	0.8688524590163934	0.8877935312361541	0.8858452317997532

So, we decided that these five best features ('ca', 'cp', 'exang', 'thal', 'thalach') were the most crucial.

7.2.3 The champion model

In modelling the three algorithms above, we tried to eliminate the errors arising from the different data distributions by changing the value of random_state. We found that only the model with an 80%:20% ratio was more stable and had the highest average score.

As the decision tree algorithm is more sensitive to fine values, there is a phenomenon that a 60%:40% model sometimes achieves a higher score in the decision tree model.

In particular, as the SVM algorithm is better at working with high-dimensional data, there are instances where a training model with all parameters outperforms a model with the best five optimal parameters.

In the case of the KNN algorithm, the main criticism lies in the long running time required due to its more complex computational process.

The choice between accuracy scores, which are critical for predicting disease and directly related to patients' lives, and run speed is a difficult one. Operational speed, on the other hand, is even more important in emergency diagnoses.

In the end we decided that accuracy was more important in this case, so we decided that the KNN algorithm (80% as the training set) was our champion model.

8.0 CONCLUDING REMARKS

All three machine learning algorithms, each have their own strengths and weaknesses. KNN demonstrated good performance in terms of accuracy and simplicity. It was effective in capturing local patterns and handling non-linear decision boundaries.

SVM showed strong performance in both linear and non-linear classification tasks. It effectively handled high-dimensional data and demonstrated good generalization ability. However, selecting appropriate hyperparameters is crucial for optimal results.

Decision Tree exhibited good interpretability, as its decisions could be easily visualized and understood. It handled both numerical and categorical data well and was robust against outliers. Decision Trees were effective in capturing complex relationships and interactions between features. However, they were prone to overfitting, especially with deep and complex trees, and required regularization techniques like pruning or ensemble methods to improve generalization. It is much more suitable for large datasets

Based on the analysis conducted, it has been observed that the features 'ca', 'cp', 'exang', 'thal', and 'thalach' have the most significant influence in predicting whether a patient has heart disease or not. These features demonstrate strong predictive power and contribute significantly to the classification outcome. The importance of these features suggests that they contain valuable information for distinguishing between individuals with and without heart disease.

Our goal is to ensure that the model has a high recall rate, which measures the proportion of actual positive cases (patients with heart disease) that are correctly identified. By prioritizing recall, we aim to minimize false negatives and avoid missing any individuals who require medical attention.

Additionally, we emphasize accuracy, which measures the overall correctness of the model's predictions by considering both true positives and true negatives. Achieving high accuracy helps in ensuring that the majority of the model's classifications align with the ground truth, reducing the risk of both false positives and false negatives.

Overall, the choice of algorithm depends on the specific characteristics of the dataset and the desired outcome. KNN is a simple and interpretable algorithm suitable for smaller datasets, while SVM can handle both linear and non-linear problems but may be computationally expensive for larger datasets. Decision Trees offer interpretability and feature importance insights but require careful pruning to avoid overfitting. In this case, we can conclude that KNN is a better option than SVM and Decision Tree.

9.0 LESSON LEARNED FROM PROJECT

Through this group project, we learned a lot of things including the experience in team working and also the knowledge in machine learning. In terms of team working, clear communication and defining roles and responsibilities within the team are very important. It is important to open a group chat or any channels of communication and ensure that each group member understands their role in the group project. Hence, when we first received our group member list from the e-learning portal, all of us could not wait to find all our group members by inviting each other to join the group chat on Whatsapp through email. When any group member encounters difficulties or problems, we can find the solution together. We found that channels of communications are very important to enable all group members to share their ideas, insights, and approaches to address various aspects of the project.

Next, task allocation and coordination are very important. Dividing the group project into smaller tasks or subtasks and assigning them to team members based on their strengths and expertise are important in letting our group project done smoothly. To achieve this target, we list out all the subtasks and let our group members choose the part they wish to handle and learn. It is important to ensure that there is clarity regarding the dependencies between tasks and coordinate the workflow to ensure smooth progress. We open a google docs for our group project so that everyone is clear with how our project is going on and able to trace the progress of every group member. Regularly assess the progress of each team member and offer support if needed.

Besides, one of the lessons learned from the machine learning project is understanding the dataset. Exploring the dataset and its characteristics helps in identifying relevant features, understanding their relationships, and help us to set the correct and clear objective of our project. The characteristics of the heart disease dataset, such as the number of instances, feature types, and missing values, should be taken into account when applying machine learning algorithms. Each algorithm may have different sensitivities to these characteristics, and adapting the approach accordingly can lead to improved model performance.

Furthermore, we learn the pros and cons of each machine learning algorithm. The choice of machine learning algorithms plays a crucial role in the success of the machine learning project. Each algorithm has its strengths and limitations, through our group discussion we are able to choose the top 3 suitable algorithms to complete our project. Implementing Decision Tree, Support Vector Machine, and K-Nearest Neighbors algorithms provides an opportunity to compare and evaluate different approaches to solve the heart disease dataset classification problem. It teaches us about the strengths and limitations of each algorithm and helps in understanding which algorithm performs better for the specific task.

In summary, by working on this group project we learned how to apply good team work to maximise our working efficiency and create a good working environment. Other than that, we learn how to work with K-Nearest Neighbors, Support Vector Machines, and Decision Tree algorithms.

10.0 CONCLUSION

In summary, machine learning algorithms, including K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Decision Trees, have demonstrated their usefulness in a range of data classification and regression tasks. Each algorithm possesses unique advantages and limitations, rendering them suitable for different situations.

KNN, a non-parametric algorithm, relies on identifying the nearest neighbours to a given data point. It is straightforward to implement and performs well with small to medium-sized datasets. KNN excels when dealing with intricate and nonlinear decision boundaries. However, its effectiveness may decline when confronted with high-dimensional data, and it can be computationally demanding during prediction.

SVM, a robust algorithm, constructs optimal hyperplanes to separate data into distinct classes. It handles high-dimensional data effectively and accommodates complex decision boundaries. SVM's regularisation parameters enable control over the trade-off between model complexity and generalisation. Nevertheless, SVM can be sensitive to the selection of the kernel function and requires careful hyperparameter tuning.

Decision Trees, versatile algorithms, establish a hierarchical structure of decision nodes to classify or regress data based on feature values. They are easily interpretable and can handle both categorical and numerical data. Decision Trees perform admirably with large datasets and exhibit resilience against outliers. However, without proper pruning, they can overfit the training data and are sensitive to minor variations in the input data.

K-Nearest Neighbors (KNN) is often preferred over Support Vector Machine (SVM) and Decision Trees for several reasons. Firstly, KNN offers a simplicity that is highly valued in many scenarios. It is a straightforward algorithm to understand and implement, making it accessible to both beginners and experts alike. By finding the nearest neighbors based on distance metrics, KNN can classify or predict new data points based on the majority vote or averaging of the values of its neighbors. This intuitive approach makes it easier to grasp the core concept behind KNN compared to the intricate workings of SVM and Decision Trees.

Another advantage of KNN is its non-parametric nature. Unlike SVM and Decision Trees, KNN does not rely on making assumptions about the underlying data distribution. This flexibility enables KNN to handle complex relationships between features without imposing strict constraints. Whether the data follows a linear or non-linear pattern, KNN can adapt and capture the inherent structure. Moreover, KNN is suitable for both classification and regression tasks, making it versatile in various problem domains. It can handle multi-class classification without requiring additional modifications and can perform regression by averaging the values of the K nearest neighbors. This adaptability makes KNN an appealing choice for a wide range of applications.

Additionally, KNN demonstrates robustness to outliers, which is an important consideration in real-world datasets. Since KNN does not assume a particular data distribution, outliers have minimal impact on the algorithm. Outliers are unlikely to significantly alter the nearest neighbors' majority, allowing KNN to maintain its predictive accuracy in the presence of outlier points. Furthermore, KNN provides interpretability by examining the nearest neighbors. Understanding the neighbors that contribute to a particular prediction or classification can offer

valuable insights and aid in explaining the decision-making process to stakeholders or domain experts.

K-Nearest Neighbors (KNN) is particularly well-suited for predicting patients with heart disease due to its high accuracy, recall, and precision in such cases. When it comes to healthcare applications, correctly identifying individuals with heart disease is crucial for timely interventions and appropriate medical treatment. KNN has shown excellent performance in this domain, delivering reliable predictions that aid in accurate diagnosis and risk assessment. By leveraging the power of proximity-based classification, KNN can effectively identify patients with heart disease based on their similarity to known cases in the training data. This approach enables KNN to capture subtle patterns and relationships among patients' features, leading to accurate predictions and reduced false negatives. The ability of KNN to achieve high accuracy, recall, and precision makes it a valuable tool in healthcare settings, assisting medical professionals in making informed decisions for patient care.

In the context of predicting heart disease, certain features play a crucial role in identifying affected patients, and KNN can effectively utilize them. Five such features, 'ca', 'cp', 'exang', 'thal', and 'thalach,' have been recognized as significant indicators of heart disease. The feature 'ca' represents the number of major vessels colored by fluoroscopy, 'cp' denotes chest pain type, 'exang' indicates exercise-induced angina, 'thal' signifies a thalassemia blood disorder, and 'thalach' represents the maximum heart rate achieved during exercise. These features capture important physiological and clinical factors related to heart disease and its diagnosis. By considering the proximity and relationships of these crucial features, KNN can accurately classify patients as either having heart disease or being healthy, providing valuable insights to guide medical interventions and treatment plans. The ability of KNN to leverage these informative features and utilize their inherent patterns enhances its effectiveness in the prediction and identification of patients with heart disease.

In summary, KNN's high accuracy, recall, and precision make it a compelling choice for predicting patients with heart disease. Leveraging the power of proximity-based classification, KNN can effectively identify individuals with heart disease based on their similarity to known cases, leading to accurate predictions and reduced false negatives. Additionally, the inclusion of crucial features such as 'ca', 'cp', 'exang', 'thal', and 'thalach' further enhances the predictive capabilities of KNN, enabling it to capture and utilize the relevant physiological and clinical factors associated with heart disease. This combination of accuracy, recall, precision, and feature importance makes KNN a valuable tool in healthcare applications, assisting in the early identification and management of patients with heart disease.

11.0 REFERENCES

Decision Tree - GeeksforGeeks. (2017, October 16). GeeksforGeeks. *https://www.geeksforgeeks.org/decision-tree/*

Gandhi, R. (2022, November 14). Support Vector Machine — Introduction to Machine Learning Algorithms. *Medium*. https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

K-Nearest Neighbor (KNN) Algorithm for Machine Learning - Javatpoint. (n.d.). *www.javatpoint.com*. *https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning*

Machine Learning Algorithms - Javatpoint. (n.d.). www.javatpoint.com. https://www.javatpoint.com/machine-learning-algorithms

Ray, S. (2023). Learn How to Use Support Vector Machines (SVM) for Data Science. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/#How_to_Implement_SVM_in_Python_and_R

Simplilearn. (2018, June 6). *KNN Algorithm In Machine Learning | KNN Algorithm Using Python | K Nearest Neighbor | Simplilearn* [Video]. YouTube. https://www.youtube.com/watch?v=4HKqjENq9OU

What is a Decision Tree | IBM. (n.d.). *https://www.ibm.com/topics/decision-trees*

Yadav, P. (2019, September 23). Decision Tree in Machine Learning. Medium. *https://towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96*