**CPM252 FOUNDATIONS AND PROGRAMMING FOR DATA ANALYTICS**

SEMESTER 1

ACADEMIC YEAR 2023/2024

UNIVERSITI SAINS MALAYSIA

# PROJECT: MACHINE LEARNING MODEL FOR HEART DISEASE DETECTION

**PREPARED BY:**


| NO. | NAME | MATRICS. NO |
|---|---|---|
| 1 | CHONG WEN QIN | 163375 |
| 2 | MUHAMMAD FIRDAUS BIN MOHD.ALI | 159141 |
| 3 | SANTTOSH A/L G MUNIYANDY | 159193 |


**PREPARED FOR:**

TS. DR. CHEW XINYING

## ABSTRACT

In this project, we aim to perform predictions on a dataset about heart disease using three machine learning algorithms which are Naive Bayes, K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). The main goal of this project is to predict the potential of heart disease of a person based on a few features that can be collected at home or in a low-developed area where healthy supplies are very limited. Therefore, to determine the most suitable model algorithm, we perform Naive Bayes, K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) in Jupyter Notebook by using Python. Then, we compare the three algorithms across various perspectives to determine the champion model in performing predictions. Finally, we conclude that Naive Bayes is the champion model due to its effectiveness.

## CONTENTS

## INTRODUCTION

### Machine Learning

A world where computers are not limited to only obtain and process the rich information, but actually learn, optimize, derive predictions and share the knowledge of the end-product is the remarkability and wondrous essence of machine learning. A subset of artificial intelligence, the fundamental process of machine learning is to enable computers to independently learn from the provided data without being explicitly programmed by human sources. These principles differ significantly from traditional computation in which the algorithms of machine learning explicitly or require the physical forces of humans to calculate or to solve the occurring problems. On the other hand, machine learning is a well-known automated process that enables computers to solve problems without hands-on activity. The most common machine learning methods used are supervised and unsupervised learning in which there are different types of learning algorithms under these methods such as decision tree learning, k-nearest neighbors, naive-bayes, etc.

Supervised learning is solely based on training from labeled data which is the input data is accompanied by theoretical output. It used the training dataset to train the model by producing the desired output which as mentioned is provided as labeled data. This method allows the algorithm to learn and compare with the actual output, which then optimizes itself and adjusts until the error has been minimized. The common use of supervised learning methods is to use historical data and predict statistically for future events. Within supervised machine learning, it can be separated into two types which are classification and regression. Classification is a process of predicting a categorical class where it assigns the test dataset into specific categories. From the learning stage, it recognizes the trends or patterns, in which it helps to classify the unlabeled data. Regression predicts a continuous numerical value, where the fundamentals are to understand the relationship of dependent and independent variables. The output may contain numerical value within a desired range to show the probability of an event occurring.

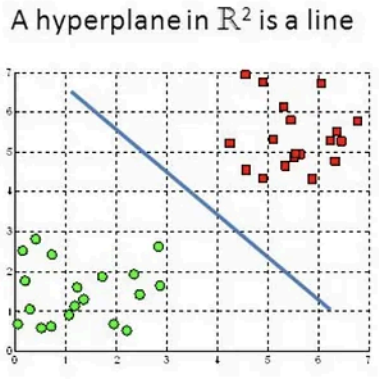
Unsupervised learning learns the data with unlabeled data which lacks target output and given the opportunity to derive insights without the guidance given as supervised learning. It has to independently create inferences since they are not provided with fixed output to train with, thus they must find the patterns on their own. Clustering is one the most popular used types of

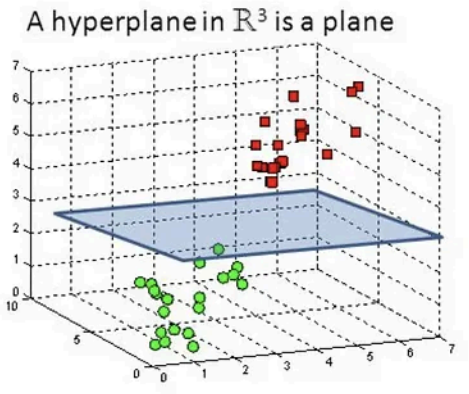
unsupervised learning. Clustering involves zero prior knowledge of data classes and the cluster algorithm attempts to familiarize data points based on traits, characteristics or features. This aids greatly in obtaining hidden patterns and trends.

Support Vector Machines (SVM)

Branching from the supervised machine learning algorithms, Support Vector Machines (SVM) are commonly used for classification and regression methodologies. In the 90's, Support Vector Machines was developed by Vladimir Vapnik and Cortes. Support Vector Machines became an important model in the field for its speciality in solving binary classification.

The working concept for Support Vector Machines (SVM) is that it finds the best line or called a decision boundary in the two dimensions in which it separates the data points according to their respective classes. It is often called a 'hyperplane' where it serves as a straight line in two dimensions or in higher dimensions, it takes different forms of plane to distinguish the classes. The idea of easily distinguishing the classes of the data points, it maximizes its margin which is the distance between the hyperplane and the closest data points to the hyperplane. Usually by detecting the strategic positions of the data points, it defines the most optimal position for the hyperplane to maximize the margin.



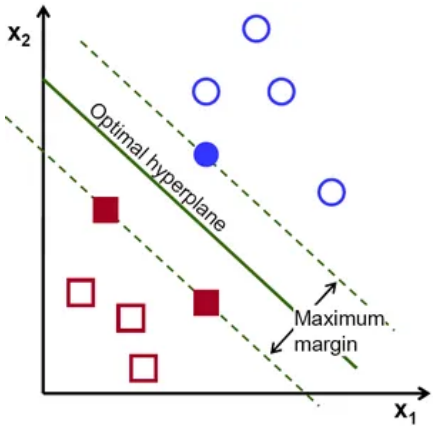



*Figure 1.1 Hyperplane*

Support Vector Machines exhibit various types to handle multiple types of approach ranging from linear to non-linear separable data. SVM uses a kernel function to map the space of data points and establish the relationships. The different types of kernel functions are linear, polynomial or radial basis function (RBF). In the SVM, there is a kernel space where we have to input a specific mathematical equation to map the hyperplane in the space. This hyperplane, as mentioned before, will classify the data points by maximizing the margin. Linear SVM is often used for classifying linearly separable data. This is a simple concept of dividing the classes of the data point by a straight line in the space. Nonlinear SVM is a difficult scenario as it is not as simple to make a single line and classify the data points. In certain scenarios, the data points needed to be classified by kernel functions such as polynomial kernel, RBF kernel and sigmoid kernel. This method heavily aids in capturing the complex relationships and leverages the SVM to higher dimensional spaces.

Support Vector Machines hold the power in handling high dimensional spaces. The term high dimensional means the number of features is higher in number than the number of data points which outputs complex relationships. For the cases of limited data, SVM works perfectly the same without any downgrade or defect to ensure the hyperplane works well even when data is limited. In the real world application, the datasets often lie in complexity, SVM adapts to this issue by handling the nonlinear data by using kernel functions. SVM is also prone to overfitting as it uses the idea of maximizing the margin which helps generalize the data.

There are also limitations to Support Vector Machines (SVMs), in which the first is that it is directly proportional to the size of datasets. For instance if the dataset is large, it would take a very long time to train and the memory would require computationally large spaces. SVMs also have options to customize its parameters such as the kernel functions, however this brings to disadvantage as we would need to be very careful and precise in choosing the type of parameters.

Naive Bayes Classifier

Naive Bayes Classifier is a supervised machine learning algorithm used for classification. It is called Naive due to its assumption that all the features are independent or unrelated to any of the other features in the data set while Bayes refers to an 18th century statistician and philosopher, Reverend Thomas Bayes who formulated Bayes' Theorem. In other words, Naive Bayes works based on the principle of Bayes' Theorem with an independence assumption among features which means that the features are equally important and contribute equally to the prediction. The independence of features allows the algorithm to make predictions quickly and accurately. For application, Naive Bayes Classifier is usually used for spam filtering, text classification, sentiment analysis, mental diagnosis and credit scoring.

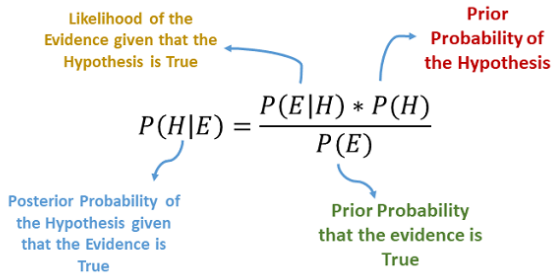



*Figure 1.2 Bayes' Theorem*

There are three types of Naive Bayes Classifier which are Gaussian, Bernoulli and Multinomial Naive Bayes. Gaussian Naive Bayes works with continuous data and assumes the features follow the Gaussian distribution which can also be called the Normal distribution. Bernoulli Naive Bayes works with Binomial variables, that is variables with two values only, such as 0 and 1 or true and false. Multinomial Naive Bayes works with discrete variables such as frequency counts and assumes the features follow Multinomial distribution.

Naive Bayes is known for its low complexity. It is easier to implement and efficient. It is effective to manage high-dimensional data with a large number of features. It also performs well with limited training data especially when the assumption of independence holds.

However, Naive Bayes Classifier may be subject to zero frequency when a categorical variable in test data does not exist in training data. When this happens, the model will assign zero probability and will be unable to make predictions. To avoid this, Laplace smoothing techniques are needed. Another disadvantage of this algorithm is the unrealistic assumption. This is because it is difficult to get a set of data with features which are completely independent in our real life and this may lead to incorrect classifications.

K-Nearest Neighbours (KNN)

K-Nearest Neighbours is an algorithm that uses the majority vote as a factor to determine the outcome of the new data. This machine learning algorithm stores all of the available cases from the data provided into the machine learning and uses it to classify the new data later. It is usually called a lazy-learner because it does not actually learn. Instead, it predicts the result based on the similarity of the previous data. KNN is a non-parametric supervised learning method first developed by Evelyn Fix and Joseph Hodges in 1951, and later expanded by Thomas Cover. (IBM, 2023)

KNN works by creating a vector in a multi-dimensional space by using each data point in the dataset, where each dimension corresponds to a feature. Next, machine learning takes the new data into the created space. It will calculate the distance with each neighbour, which is called k value, finding the nearest one based on the k value and determine the result by comparing it with the majority of k's outcomes. If the k value is even, there is a chance the result is going to be an error because both outcomes are the same. It is better to use an odd k-value to avoid the error. KNN calculates the distance by using the Euclidean distance. The formula is shown below :

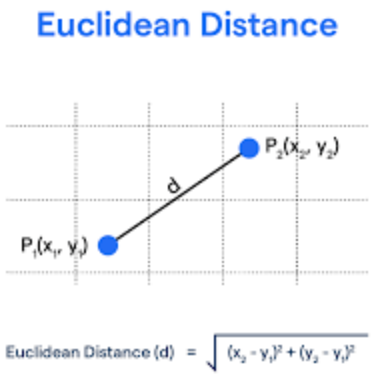



*Figure 1.3 Euclidean Distance*

*Source : Botpenguin.com*

To determine the k value, statistically just square root the number of the dataset. However, in real situations, this method is not really good. The only way is to experiment with various numbers of k around the calculated k value. Large k value can make the model too generalized while small k value can lead to the noise.

## OBJECTIVE

The objective of this project is to create a machine learning model that can predict the potential of heart disease of a person by collecting various factors that can be collected at home or in a low-developed area.

## JUSTIFICATION OF CHOICE

The reason for this objective choice is nowadays, there are still some people that live in the rural area. Usually, their surroundings have limited healthcare. The medical apparatus is very scarce or even not available in those areas. This model will be convenient for doctors and nurses to get a clue of some symptoms that lead to heart diseases. Doctors can respond fast to the potential patient and provide a comprehensive checkup and react accordingly. Since the heart is an important organ in human's body, early detection for heart disease is crucial for ensuring a better life.

The chosen machine learning algorithm is Naive Bayes (NB), K-Nearest Neighbours (KNN) and Support Vector Machine (SVM). All of the machine learning models chosen are classified as supervised and categorical. This is because the data's result (target) is either 1 (presence of heart disease) or 0 (no presence of heart disease). So, it is categorical and there is not something in the middle. Most of the data have their label, for example, there are 4 labels available for *cp* feature, which allow us to do training of models. Therefore, the suitable one is the supervised algorithm. Naive Bayes model is the first model that was chosen. Since the relation among any of the features are unknown, this model is suitable for this project. The second model is K-Nearest Neighbours. Since some of the health symptoms will result in the same outcome, this model can be used for this project. Lastly, the Support Vector Machine model will be useful because the features sometimes are grouped together, creating a clear support vector and better result.

## STEPS TO BUILD THE MACHINE LEARNING MODELS

Step 1: Import Libraries

First, we import the library needed for this project into Jupyter Notebook. In this project, we import *Pandas* and *Numpy* to read the dataset. The model is also imported from scikit learn module.

```
In [1]: #import libraries
        import pandas as pd
        import numpy as np
        import numpy.random as npr

        #import models from scikit learn module:
        #for data splitting
        from sklearn.model_selection import train_test_split
        #for data standardization
        from sklearn.preprocessing import StandardScaler
        #for Naive Bayes
        from sklearn.naive_bayes import BernoulliNB
        #for K-Nearest Neighbors (KNN)
        from sklearn.neighbors import KNeighborsClassifier
        #for Support Vector Machine (SVM)
        from sklearn import svm
        #for evaluation of accuracy
        from sklearn import metrics
        from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, accuracy_score, f1_score
```

*Figure 4.1 Import libraries*

Step 2: Data Preparation

Then, the dataset, namely "*heart_disease.csv*", is imported and loaded by using *Pandas*. By default, the first 5 rows and the last 5 rows are shown, indicating that the dataset is successfully loaded.

```
In [2]: cd C:/Users/CHONG WEN QIN/Downloads

        C:\Users\CHONG WEN QIN\Downloads

In [3]: # Reading the dataset in a dataframe using Pandas
        df = pd.read_csv("heart_disease.csv")
        df
```

Out[3]:

| | 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

*Figure 4.2 Read the Dataset*

*df.describe()* is used to generate the descriptive statistics of the data. It displays the count, mean, standard deviation, minimum, quartiles and maximum of the dataset.

In [4]:
```
#Get summary of numerical variables
df.describe()
```

Out[4]:

| | 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 4.3 Descriptive Statistics of Dataset*

After that, we use *df.columns* to access all the features' names of the data.

In [5]:
```
# Reading the features of dataset
df.columns
```

Out[5]:
```
Index(['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
       'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target'],
      dtype='object')
```

*Figure 4.4 Features*

We also use *df.dtypes* to determine and observe the data types of all features of the dataset.

```
In [6]: # Check the data types of each features
        df.dtypes

Out[6]: age          int64
        sex          int64
        cp           int64
        trestbps     int64
        chol         int64
        fbs          int64
        restecg      int64
        thalach      int64
        exang        int64
        oldpeak    float64
        slope        int64
        ca           int64
        thal         int64
        target       int64
        dtype: object
```

*Figure 4.5 Data Types of Each Features*

Then, we use *df.apply(lambda x: sum(x.isnull()), axis = 0)* to check if there is any missing value in the dataset. Clearly, there is no missing value in this dataset and therefore, we can skip the steps of replacing or removing the missing value.

```
In [7]: # Check if there is null in the dataset
        df.apply(lambda x: sum(x.isnull()), axis = 0)

Out[7]: 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 4.6 Check Missing Values*

*df.duplicated().sum()* is applied to check if there is duplicate data in the dataset. From the figure below, we know that there is duplicate data.

```
In [8]: # Check if there is duplicate in the dataset
        df.duplicated().sum()

Out[8]: 1
```

*Figure 4.7 Check Duplicate Data*

Therefore, we use *df.drop_duplicates(inplcae = True)* to remove the duplicate data. Then, we observe the dataset again. There are 302 rows out of 303 rows left after removing the duplicate data.

```
In [9]: # Drop the duplicates
        df.drop_duplicates(inplace = True)
        df
```

Out[9]:

| | 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 4.8 Drop Duplicate Data*

Then, we use *df.describe()* to generate and observe the descriptive statistics of the dataset again.

```
In [10]: df.describe()
```

Out[10]:

| | age | sex | cp | trestbps | chol | fbs | restecg | thalach | exang | oldpeak | slope | ca | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 302.00000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000000 | 302.000 |
| mean | 54.42053 | 0.682119 | 0.963576 | 131.602649 | 246.500000 | 0.149007 | 0.526490 | 149.569536 | 0.327815 | 1.043046 | 1.397351 | 0.718543 | 2.314 |
| std | 9.04797 | 0.466426 | 1.032044 | 17.563394 | 51.753489 | 0.356686 | 0.526027 | 22.903527 | 0.470196 | 1.161452 | 0.616274 | 1.006748 | 0.613 |
| min | 29.00000 | 0.000000 | 0.000000 | 94.000000 | 126.000000 | 0.000000 | 0.000000 | 71.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000 |
| 25% | 48.00000 | 0.000000 | 0.000000 | 120.000000 | 211.000000 | 0.000000 | 0.000000 | 133.250000 | 0.000000 | 0.000000 | 1.000000 | 0.000000 | 2.000 |
| 50% | 55.50000 | 1.000000 | 1.000000 | 130.000000 | 240.500000 | 0.000000 | 1.000000 | 152.500000 | 0.000000 | 0.800000 | 1.000000 | 0.000000 | 2.000 |
| 75% | 61.00000 | 1.000000 | 2.000000 | 140.000000 | 274.750000 | 0.000000 | 1.000000 | 166.000000 | 1.000000 | 1.600000 | 2.000000 | 1.000000 | 3.000 |
| max | 77.00000 | 1.000000 | 3.000000 | 200.000000 | 564.000000 | 1.000000 | 2.000000 | 202.000000 | 1.000000 | 6.200000 | 2.000000 | 4.000000 | 3.000 |

*Figure 4.9 Descriptive Statistics of Dataset*

Next, *df['target'].value_counts()* is applied. From the output, it is clear that the proportion of positive (1) and negative (0) results for heart disease is good.

```
In [11]: df['target'].value_counts()
Out[11]: target
         1    164
         0    138
         Name: count, dtype: int64
```

*Figure 4.10 Count the Proportion of Target*

Step 3: Features Selection

We choose our features based on our objective. Since our objective is to allow the potential patients to do a quick test, we choose the features that can be easily observed by using a simple machine.

```
In [12]: X = df[['cp','trestbps','chol','fbs','restecg']]
         Y = df['target']
```

*Figure 4.11 Select Features*

Step 4: Data Splitting

Then, we apply *train_test_split()* to split the data into 3 divisions which are 60% train set and 40% test set, 70% train set and 30% test set and 80% train set and 20% test set in order to determine the most appropriate data split.

```
In [13]: # Splitting data into 60% train set and 40% test set
         X_train60,X_test40,Y_train60,Y_test40 = train_test_split(X,Y,random_state = 0,test_size = 0.4)
         # Splitting data into 70% train set and 30% test set
         X_train70,X_test30,Y_train70,Y_test30 = train_test_split(X,Y,random_state = 0,test_size = 0.3)
         # Splitting data into 80% train set and 20% test set
         X_train80,X_test20,Y_train80,Y_test20 = train_test_split(X,Y,random_state = 0,test_size = 0.2)
```

*Figure 4.12 Split the Data*

Step 5: Data Standardization

*StandardScaler()*, *fit_transform()* and *transform()* are used to standardize the data.

```
In [14]: # Data Standardization
         sd = StandardScaler()
         X_train60 = sd.fit_transform(X_train60)
         X_test40 = sd.transform(X_test40)
         X_train70 = sd.fit_transform(X_train70)
         X_test30 = sd.transform(X_test30)
         X_train80 = sd.fit_transform(X_train80)
         X_test20 = sd.transform(X_test20)
```

*Figure 4.13 Standardize the Data*

```
In [15]: print(X_train60.shape)
         print(X_test40.shape)
         print(X_train70.shape)
         print(X_test30.shape)
         print(X_train80.shape)
         print(X_test20.shape)

(181, 5)
(121, 5)
(211, 5)
(91, 5)
(241, 5)
(61, 5)
```

```
In [16]: print(Y_train60.shape)
         print(Y_test40.shape)
         print(Y_train70.shape)
         print(Y_test30.shape)
         print(Y_train80.shape)
         print(Y_test20.shape)

(181,)
(121,)
(211,)
(91,)
(241,)
(61,)
```

*Figure 4.14 Proportion of Each Data Split*

Now, from the output, we know that we have 181 rows and 121 rows for 60% train set and 40% test set; 211 rows and 91 rows for 70% train set and 30% test set; 241 rows and 61 rows for 80% train set and 20% test set.

Step 6: Build the Machine Learning Model

1. Naive Bayes

   i. Train the model for each division of data.

```
In [17]: # Fit the data into Naive Bayes Model
         NB_model60 = BernoulliNB()
         NB_model70 = BernoulliNB()
         NB_model80 = BernoulliNB()

         NB_model60.fit(X_train60,Y_train60)
         NB_model70.fit(X_train70,Y_train70)
         NB_model80.fit(X_train80,Y_train80)
Out[17]: BernoulliNB
         BernoulliNB()
```

*Figure 4.15 Build Naive Bayes Model*

   ii. Predict the test set.

```
In [18]: # Predict the test set results
         NB_pred40=NB_model60.predict(X_test40)
         NB_pred30=NB_model70.predict(X_test30)
         NB_pred20=NB_model80.predict(X_test20)
```

*Figure 4.16 Predict the Test Set Results*

2. K-Nearest Neighbors (KNN)

   i. Find the best value of K.

   We choose the best K by comparing the average accuracies of the model for each value of k in the range of 1 to 30 across different test and train splits to find the K with the highest average accuracy.

```
In [21]: # Find the best k-value
splits = [(X_train60,X_test40,Y_train60,Y_test40),(X_train70,X_test30,Y_train70,Y_test30),(X_train80,X_test20,Y_train80,Y_test20)]
k_values = range(1,30)
avg_accuracy_per_k = []

for k in k_values:
    accuracies = []

    for X_train, X_test, Y_train, Y_test in splits:
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, Y_train)

        Y_pred = knn.predict(X_test)

        accuracy = np.mean(Y_pred == Y_test)
        accuracies.append(accuracy)

    avg_accuracy = np.mean(accuracies)
    avg_accuracy_per_k.append((k, avg_accuracy))

best_k, best_avg_accuracy = max(avg_accuracy_per_k, key=lambda x: x[1])

print('Best k value:', best_k, 'with average accuracy:', best_avg_accuracy)
```

```
Best k value: 18 with average accuracy: 0.7712555700633197
```

*Figure 4.17 Determine the best k-value*

ii. Train the model for each division of data by using K = 18.

```
In [22]: # Fit the train set into the model using k = 18
knn_model60 = KNeighborsClassifier(n_neighbors = 18)
knn_model70 = KNeighborsClassifier(n_neighbors = 18)
knn_model80 = KNeighborsClassifier(n_neighbors = 18)

knn_model60.fit(X_train60, Y_train60)
knn_model70.fit(X_train70, Y_train70)
knn_model80.fit(X_train80, Y_train80)
```

```
Out[22]: KNeighborsClassifier
         KNeighborsClassifier(n_neighbors=18)
```

*Figure 4.18 Build KNN Model*

iii. Predict the test set.

```
In [23]: # Predict the test set results
knn_pred40 = knn_model60.predict(X_test40)
knn_pred30 = knn_model70.predict(X_test30)
knn_pred20 = knn_model80.predict(X_test20)
```

*Figure 4.19 Predict the Test Set Results*

3. Support Vector Machine (SVM)

i. Train the model for each division of data.

```
In [26]: # Fit the data into SVM Model
         SVM_model60 = svm.SVC()
         SVM_model70 = svm.SVC()
         SVM_model80 = svm.SVC()

         SVM_model60.fit(X_train60,Y_train60)
         SVM_model70.fit(X_train70,Y_train70)
         SVM_model80.fit(X_train80,Y_train80)

Out[26]: SVC
         SVC()
```

*Figure 4.20 Build SVM Model*

ii. Predict the test set.

```
In [27]: # Predict the test set results
         SVM_pred40 = SVM_model60.predict(X_test40)
         SVM_pred30 = SVM_model70.predict(X_test30)
         SVM_pred20 = SVM_model80.predict(X_test20)
```

*Figure 4.21 Predict the Test Set Results*

## Step 7: Model Evaluation

1. Naive Bayes

i. Accuracy, recall, precision, F1 score and confusion matrix are calculated.

```
In [19]: # Evaluate the accuracy
         print("60% Train and 40% Test")
         print('Accuracy:', accuracy_score(Y_test40, NB_pred40))
         print('Recall', recall_score(Y_test40, NB_pred40, average = "weighted"))
         print ('Precision:', precision_score(Y_test40, NB_pred40, average="weighted"))
         print ('F1 score:', f1_score(Y_test40, NB_pred40, average = "weighted"))
         NB_confusion40 = confusion_matrix(Y_test40, NB_pred40)
         print('Confusion matrix:')
         print(NB_confusion40)

         print("\n70% Train and 30% Test")
         print('Accuracy:', accuracy_score(Y_test30, NB_pred30))
         print('Recall', recall_score(Y_test30, NB_pred30, average = "weighted"))
         print ('Precision:', precision_score(Y_test30, NB_pred30, average = "weighted"))
         print ('F1 score:', f1_score(Y_test30, NB_pred30, average = "weighted"))
         NB_confusion30 = confusion_matrix(Y_test30, NB_pred30)
         print('Confusion matrix:')
         print(NB_confusion30)

         print("\n80% Train and 20% Test")
         print('Accuracy:', accuracy_score(Y_test20, NB_pred20))
         print('Recall', recall_score(Y_test20, NB_pred20, average = "weighted"))
         print ('Precision:', precision_score(Y_test20, NB_pred20, average = "weighted"))
         print ('F1 score:', f1_score(Y_test20, NB_pred20, average = "weighted"))
         NB_confusion20 = confusion_matrix(Y_test20, NB_pred20)
         print('Confusion matrix:')
         print(NB_confusion20)

60% Train and 40% Test
Accuracy: 0.7851239669421488
Recall 0.7851239669421488
Precision: 0.7853772838098857
F1 score: 0.7850358548454875
Confusion matrix:
[[46 14]
 [12 49]]

70% Train and 30% Test
Accuracy: 0.7912087912087912
Recall 0.7912087912087912
Precision: 0.7912991264055095
F1 score: 0.7911583402703374
Confusion matrix:
[[35 10]
 [ 9 37]]

80% Train and 20% Test
Accuracy: 0.8032786885245902
Recall 0.8032786885245902
Precision: 0.8032786885245902
F1 score: 0.8032786885245902
Confusion matrix:
[[21  6]
 [ 6 28]]
```

*Figure 4.22 Naive Bayes Model Evaluation*

ii. Check overfitting for each division of data.

```
In [20]: # Check for overfit
NB_trainpred60 = NB_model60.predict(X_train60)
NB_trainpred70 = NB_model70.predict(X_train70)
NB_trainpred80 = NB_model80.predict(X_train80)

print("Overfitting Checking for Naive Bayes")
print("\n60% Train and 40% Test")
print('Accuracy for test set :', accuracy_score(Y_test40, NB_pred40))
print('Accuracy for train set :', accuracy_score(Y_train60, NB_trainpred60))

print("\n70% Train and 30% Test")
print('Accuracy for test set :', accuracy_score(Y_test30, NB_pred30))
print('Accuracy for train set :', accuracy_score(Y_train70, NB_trainpred70))

print("\n80% Train and 20% Test")
print('Accuracy for test set :', accuracy_score(Y_test20, NB_pred20))
print('Accuracy for train set :', accuracy_score(Y_train80, NB_trainpred80))
```

```
Overfitting Checking for Naive Bayes

60% Train and 40% Test
Accuracy for test set : 0.7851239669421488
Accuracy for train set : 0.7403314917127072

70% Train and 30% Test
Accuracy for test set : 0.7912087912087912
Accuracy for train set : 0.7440758293838863

80% Train and 20% Test
Accuracy for test set : 0.8032786885245902
Accuracy for train set : 0.7468879668049793
```

*Figure 4.23 Overfit Checking*

2. K-Nearest Neighbors (KNN)

i. Accuracy, recall, precision, F1 score and confusion matrix are calculated.

```
In [24]: # Evaluate the accuracy
print("60% Train and 40% Test")
print('Accuracy:', accuracy_score(Y_test40, knn_pred40))
print('Recall:', recall_score(Y_test40, knn_pred40, average = "weighted"))
print('Precision:', precision_score(Y_test40, knn_pred40, average = "weighted"))
print ('F1 score:', f1_score(Y_test40, knn_pred40, average = "weighted"))
knn_confusion40 = confusion_matrix(Y_test40, knn_pred40)
print('Confusion matrix:')
print(knn_confusion40)

print("\n70% Train and 30% Test")
print('Accuracy:', accuracy_score(Y_test30, knn_pred30))
print('Recall:', recall_score(Y_test30, knn_pred30, average = "weighted"))
print('Precision:', precision_score(Y_test30, knn_pred30, average = "weighted"))
print ('F1 score:', f1_score(Y_test30, knn_pred30, average = "weighted"))
knn_confusion30 = confusion_matrix(Y_test30, knn_pred30)
print('Confusion matrix:')
print(knn_confusion30)

print("\n80% Train and 20% Test")
print('Accuracy:', accuracy_score(Y_test20, knn_pred20))
print('Recall:', recall_score(Y_test20, knn_pred20, average = "weighted"))
print('Precision:', precision_score(Y_test20, knn_pred20, average = "weighted"))
print ('F1 score:', f1_score(Y_test20, knn_pred20, average = "weighted"))
knn_confusion20 = confusion_matrix(Y_test20, knn_pred20)
print('Confusion matrix:')
print(knn_confusion20)
```

```
60% Train and 40% Test
Accuracy: 0.7520661157024794
Recall: 0.7520661157024794
Precision: 0.752409526951005
F1 score: 0.7520322449532584
Confusion matrix:
[[46 14]
 [16 45]]

70% Train and 30% Test
Accuracy: 0.7912087912087912
Recall: 0.7912087912087912
Precision: 0.7923301188607311
F1 score: 0.7909056460780599
Confusion matrix:
[[34 11]
 [ 8 38]]

80% Train and 20% Test
Accuracy: 0.7704918032786885
Recall: 0.7704918032786885
Precision: 0.7733889202939513
F1 score: 0.7711127670144063
Confusion matrix:
[[21  6]
 [ 8 26]]
```

*Figure 4.24 KNN Model Evaluation*

ii. Check overfitting for each division of data.

```
In [25]: # Check for overfit
knn_trainpred60 = knn_model60.predict(X_train60)
knn_trainpred70 = knn_model70.predict(X_train70)
knn_trainpred80 = knn_model80.predict(X_train80)

print("Overfitting Checking for K-Nearest Neighbors")
print("60% Train and 40% Test")
print('Accuracy for test set :', accuracy_score(Y_test40, knn_pred40))
print('Accuracy for train set :', accuracy_score(Y_train60, knn_trainpred60))

print("\n70% Train and 30% Test")
print('Accuracy for test set :', accuracy_score(Y_test30, knn_pred30))
print('Accuracy for train set :', accuracy_score(Y_train70, knn_trainpred70))

print("\n80% Train and 20% Test")
print('Accuracy for test set :', accuracy_score(Y_test20, knn_pred20))
print('Accuracy for train set :', accuracy_score(Y_train80, knn_trainpred80))
```

```
Overfitting Checking for K-Nearest Neighbors
60% Train and 40% Test
Accuracy for test set : 0.7520661157024794
Accuracy for train set : 0.7348066298342542

70% Train and 30% Test
Accuracy for test set : 0.7912087912087912
Accuracy for train set : 0.7440758293838863

80% Train and 20% Test
Accuracy for test set : 0.7704918032786885
Accuracy for train set : 0.7510373443983402
```

*Figure 4.25 Overfit Checking*

3. Support Vector Machine (SVM)

i. Accuracy, recall, precision, F1 score and confusion matrix are calculated.

```
In [28]: # Evaluate the accuracy
print("60% Train and 40% Test")
print('Accuracy:', accuracy_score(Y_test40, SVM_pred40))
print('Recall:', recall_score(Y_test40, SVM_pred40, average = "weighted"))
print('Precision:', precision_score(Y_test40, SVM_pred40, average = "weighted"))
print ('F1 score:', f1_score(Y_test40, SVM_pred40, average = "weighted"))
SVM_confusion40 = confusion_matrix(Y_test40, SVM_pred40)
print('Confusion matrix:')
print(SVM_confusion40)

print("\n70% Train and 30% Test")
print('Accuracy:', accuracy_score(Y_test30, SVM_pred30))
print('Recall:', recall_score(Y_test30, SVM_pred30, average = "weighted"))
print('Precision:', precision_score(Y_test30, SVM_pred30, average = "weighted"))
print ('F1 score:', f1_score(Y_test30, SVM_pred30, average = "weighted"))
SVM_confusion30 = confusion_matrix(Y_test30, SVM_pred30)
print('Confusion matrix:')
print(SVM_confusion30)

print("\n80% Train and 20% Test")
print('Accuracy:', accuracy_score(Y_test20, SVM_pred20))
print('Recall:', recall_score(Y_test20, SVM_pred20, average = "weighted"))
print('Precision:', precision_score(Y_test20, SVM_pred20, average = "weighted"))
print ('F1 score:', f1_score(Y_test20, SVM_pred20, average = "weighted"))
SVM_confusion20 = confusion_matrix(Y_test20, SVM_pred20)
print('Confusion matrix:')
print(SVM_confusion20)
```

```
60% Train and 40% Test
Accuracy: 0.7603305785123967
Recall: 0.7603305785123967
Precision: 0.7636372808908376
F1 score: 0.7594098480738081
Confusion matrix:
[[42 18]
 [11 50]]

70% Train and 30% Test
Accuracy: 0.7912087912087912
Recall: 0.7912087912087912
Precision: 0.7912991264055095
F1 score: 0.7911583402703374
Confusion matrix:
[[35 10]
 [ 9 37]]

80% Train and 20% Test
Accuracy: 0.8032786885245902
Recall: 0.8032786885245902
Precision: 0.8032786885245902
F1 score: 0.8032786885245902
Confusion matrix:
[[21  6]
 [ 6 28]]
```

*Figure 4.26 SVM Model Evaluation*

ii. Check overfitting for each division of data.

```
In [29]: # Check for overfit
SVM_trainpred60 = SVM_model60.predict(X_train60)
SVM_trainpred70 = SVM_model70.predict(X_train70)
SVM_trainpred80 = SVM_model80.predict(X_train80)

print("Overfitting Checking for Support Vector Machine")
print("60% Train and 40% Test")
print('Accuracy for test set :', accuracy_score(Y_test40, SVM_pred40))
print('Accuracy for train set :', accuracy_score(Y_train60, SVM_trainpred60))

print("\n70% Train and 30% Test")
print('Accuracy for test set :', accuracy_score(Y_test30, SVM_pred30))
print('Accuracy for train set :', accuracy_score(Y_train70, SVM_trainpred70))

print("\n80% Train and 20% Test")
print('Accuracy for test set :', accuracy_score(Y_test20, SVM_pred20))
print('Accuracy for train set :', accuracy_score(Y_train80, SVM_trainpred80))
```

```
Overfitting Checking for Support Vector Machine
60% Train and 40% Test
Accuracy for test set : 0.7603305785123967
Accuracy for train set : 0.7790055248618785

70% Train and 30% Test
Accuracy for test set : 0.7912087912087912
Accuracy for train set : 0.7582938388625592

80% Train and 20% Test
Accuracy for test set : 0.8032786885245902
Accuracy for train set : 0.7634854771784232
```

*Figure 4.27 Overfit Checking*

## COMPARISON AND RECOMMENDATION

In this project, we used three different machine learning algorithms to determine the champion model to predict and detect heart disease which are Naive Bayes, K-Nearest Neighbours (KNN) and Support Vector Machine (SVM).

First of all, among these three model algorithms, KNN has the longest code and thus consumes the longest time to predict the testing set results. This is because we need to find the most appropriate k-value through the average accuracy of the model across 3 divisions of data by looping before training the data using KNN. For Naive Bayes and SVM, we just need to fit the data into the model and start training the model, which requires less time and less steps to predict the test set results.

Next, from the model evaluation, we can observe and compare the accuracy, precision, recall and F1 score of the models.

Naive Bayes:

| | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
|---|---|---|---|---|
| **40% Test Set** | 0.785124 | 0.785377 | 0.785124 | 0.785036 |
| **30% Test Set** | 0.791209 | 0.791299 | 0.791209 | 0.791158 |
| **20% Test Set** | 0.803279 | 0.803279 | 0.803279 | 0.803279 |

*Table 5.1 Naive Bayes Model Evaluation*

KNN:

| | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
|---|---|---|---|---|
| **40% Test Set** | 0.752066 | 0.752410 | 0.752066 | 0.752032 |
| **30% Test Set** | 0.791209 | 0.792330 | 0.791209 | 0.790906 |
| **20% Test Set** | 0.770492 | 0.773389 | 0.770492 | 0.771113 |

*Table 5.2 KNN Model Evaluation*

SVM:

| | Accuracy | Precision | Recall | F1 Score |
|---|---|---|---|---|
| **40% Test Set** | 0.760331 | 0.763637 | 0.760331 | 0.759410 |
| **30% Test Set** | 0.791209 | 0.791299 | 0.791209 | 0.791158 |
| **20% Test Set** | 0.803279 | 0.803279 | 0.803279 | 0.803279 |

*Table 5.3 SVM Model Evaluation*

We compare the accuracy of the models first. When we use 60% train set 40% test set, we found that Naive Bayes has the highest accuracy which is 0.785124. When we use 70% train set 30% test set, we found that Naive Bayes, KNN and SVM have the same accuracy which is 0.791209. When we use 80% train set 20% test set, we found that Naive Bayes and SVM with the same accuracy (0.803279) are more accurate when compared to KNN. Therefore, in general, although the accuracies between the three model algorithms are quite close, Naive Bayes is more accurate at predicting all divisions of data. The higher accuracy performed by Naive Bayes makes it a better model to achieve the objective of this project.

Then, we compare the precision of the models. When we use 60% train set 40% test set, we found that Naive Bayes has the highest precision which is 0.785377. When we use 70% train set 30% test set, we found that KNN has the highest precision which is 0.792330. When we use 80% train set 20% test set, we found that Naive Bayes and SVM have the same precision (0.803279) which is higher than KNN. In this case, although three of the model algorithms have high and quite similar precision, we still can conclude that the precision of Naive Bayes is slightly higher among the 3 algorithms.

Next, we do not compare recall since all the values of recall are the same as accuracy. Therefore, we will proceed to the comparison of F1 score of the models. When we use 60% train set 40% test set, we found that Naive Bayes has the highest F1 score which is 0.785036. When we use 70% train set 30% test set and 80% train set 20% test set, we found that Naive Bayes and SVM have the same higher F1 score which are 0.791158 and 0.803279 respectively. From here, although the results are very similar among Naive Bayes and SVM, we still can conclude that

Naive Bayes has higher F1 Score which indicates that Naive Bayes can identify the positive results and negative results better. Therefore, we can once again conclude that Naive Bayes is a better algorithm to predict and detect heart disease.

In short, according to the comparison above, we would recommend Naive Bayes as the champion model since it is consistent, has shorter code, requires shorter time to perform prediction as well as has higher accuracy, precision and F1 score.

## RESULTS AND DISCUSSION

In this project, the dataset "*heart_disease.csv*" is loaded and it can be observed that there are a total of 303 rows and 14 columns (*features*). There is duplicate data in the dataset and therefore, we drop the duplicate data. As a result, 302 rows out of 303 rows left.

Moving on to the features selection, we choose the features based on our main objective which is to predict the potential of heart disease of a person by collecting various factors that can be collected easily. Therefore, in our opinion, the most suitable features for this project are *cp* (chest pain type), *trestbps* (resting blood pressure), *chol* (serum cholesterol), *fbs* (fasting blood sugar) and *restecg* (resting electrocardiographic results). Those features can be obtained even with a simple machine that is portable and easy-to-use. The features selection is important to build a good machine learning model.

Then, after selecting the features and splitting the data, data standardization is carried out. This is an important step to rescale the data of the features to a standard range so that they have a mean of 0 and a standard deviation of 1. This step is to let all the features have a similar scale, allowing the features to be treated equally by the algorithm during training and to avoid bias. Data standardization can also enhance the performance of certain model algorithms, especially those models that involve distance measures, like K-Nearest Neighbors (KNN). After that, we start building machine learning algorithms.

| | **60% train set 40% test set** | **70% train set 30% test set** | **80% train set 20% test set** |
|---|---|---|---|
| **Naive Bayes** | 0.785124 | 0.791209 | 0.803279 |
| **KNN** | 0.752066 | 0.791209 | 0.770492 |
| **SVM** | 0.760331 | 0.791209 | 0.803279 |

*Table 6.1 Accuracy of Each Model across Each Division of Data*

Table above shows the accuracy of each model across each division of data. From the table above, we can observe that Naive Bayes has the highest accuracy among 3 different machine learning algorithms.

According to the table shown, we can observe that when Naive Bayes predicted 20% test set using 80% training, the accuracy is the highest (0.803279), which is the same as the accuracy of SVM on predicting the 20% test set. This shows that the models can perform consistently on 80% train set 20% test set. On the other hand, we can observe that the worst performing model algorithm is KNN with the lowest accuracy of 0.752066 when predicting the 40% test set using 60% train set. However, the difference between the highest and lowest accuracy is very small, which is 0.051213 only. This is probably because we have standardized the data before we train the model and predict the test results.

| | | **60% train set 40% test set** | **70% train set 30% test set** | **80% train set 20% test set** |
|---|---|---|---|---|
| **Naive Bayes** | **Train** | 0.740331 | 0.744076 | 0.746888 |
| | **Test** | 0.785124 | 0.791209 | 0.803279 |
| **KNN** | **Train** | 0.734807 | 0.744076 | 0.751037 |
| | **Test** | 0.752066 | 0.791209 | 0.770492 |
| **SVM** | **Train** | 0.779006 | 0.758294 | 0.763485 |
| | **Test** | 0.760331 | 0.791209 | 0.803279 |

*Table 6.2 Comparison between the Accuracy of Train Set and Test Set*

Table above shows the comparison between the accuracy of the train set and test set. Based on the table above, we would like to check the overfitting of each model on each division of data. Overfitting usually happens when the model algorithm learns the training data too well and leads to poor performance on the test set. Therefore, checking for overfitting is a crucial step to maintain the accuracy of every model and prevent biases. From the table shown above, we can observe that for Naive Bayes, KNN and SVM, the differences between the accuracy of train sets and test sets are small, which indicates that overfitting does not occur. However, we should also be careful since when the SVM model trained by 60% train set is used to perform predictions, the accuracy of the train set is slightly higher than the accuracy of the test set, which is likely to lead to overfitting.

In this dataset, we also found that among each division of train and test set, 80% train set 20% test set gives the highest and most consistent accuracy when it is used to train the model. As evidence, Naive Bayes and SVM get the same results for the accuracy, precision, recall and F1 score when 80% train set is used. Therefore, we can conclude that for "*heart_disease.csv*", 80-20 split is the most appropriate to be used to train the model.

In short, by comparison, we recommend Naive Bayes as our champion model since it has the most consistent and highest accuracy among all 3 machine learning algorithms. Furthermore, the code of Naive Bayes is shorter if compared to KNN, therefore is easier and consumes a shorter time to build it. Besides that, Naive Bayes is clearly not an overfit model if compared to SVM which may overfit in 60% train set. Thus, in general, Naive Bayes is the most effective model among all 3 model algorithms in predicting and detecting heart disease.

## CONCLUDING REMARKS

In this report, we have chosen three machine learning models which are Support Vector Machines, K-Nearest Neighbors and Naive Bayes, in which our main objective is to enable the detection of heart disease in restricted conditions (features). By choosing the influential features which are the type of chest pain (*cp*), the resting blood pressure (*trestbps*), cholesterol (*chol*), fasting blood sugar (*fbs*) and resting electrocardiographic results (*restecg*), we are able to demonstrate the strengths and limitations for this process. In summary, the highest score among the machine learning models are Naive Bayes followed by Support Vector Machines and K-Nearest Neighbors. Support Vector Machine uses kernel space by maximizing its margin to separate classes of data points meanwhile Naive Bayes acts as the independence of assumption for features where all the features are not correlated and finally K-Nearest Neighbors assigns classes from determining the nearest neighbors data points.

From the final results and reports, it is shown that Support Vector Machines and Naive Bayes are more competitive in terms of the metrics scores than K-Nearest Neighbors. It was evaluated that K-Nearest Neighbors requires more handiwork to determine the suitable ‘k’ parameter which heavily affects the performance scores. Even though Naive Bayes has shown a spectacular performance, it is logical to take rational thinking as we are focused on medical subjects. As Naive Bayes takes the features to be independently correlated, it is known that it is highly dangerous to adapt into medical matter. On the other perspective, Naive Bayes and SVMs are computationally efficient and quicker in processing the data than K-Nearest Neighbors. The reason why is because, as we mentioned, the limitation faced from determining the suitable parameter ‘k’, and we also similarly face issues with SVMs and Naive Bayes. In terms of SVMs, we faced the issue of choosing the right kernel function which in return of the wrong option will result in longer training time. As to this, we had to properly experiment and understand the kernel options. For Naive Bayes, we had to understand the complexity of feature independence which might not hold the same amount of truth in every event.

For this specific task of finding the possibility of having heart disease, we are more focused to obtain the highest metrics score among the models, thus the final choice of model is Naive Bayes. Support Vector Machines would be the second choice as it produces almost similar

scores to Naive Bayes, however it would need more optimization for the preprocessing parts of machine learning to avoid wrong parameter tuning.

## LESSON LEARNED FROM THE PROJECT

Choosing an appropriate machine learning model takes much consideration and factors such as type of scenario, relationships of features with target, type of features and much more. We should not blindly choose models solely based on comfort. Model tuning and evaluation of the model should always be prioritized and take consideration for issues that commonly occur such as overfitting or underfitting so that the model can be used without biased purpose. The understanding of the traits of the dataset should be prioritized before starting machine learning. The importance of data preprocessing to identify presence of missing data, duplicate data or outliers helps significantly for dataset distribution. In the hope that our report and analysis have strengthened the view of different techniques and methods, we explored the ways to enhance the accuracy and overcome the limitation for the models.

## CONCLUSION

In conclusion, for our main objective to find heart disease from home or places which have a restricted amount of health supplies, we analyzed and studied the strengths and weaknesses of the three machine learning models which are Support Vector Machines, Naive Bayes and K-Nearest Neighbors. We found that the best among three machine learning models is Naive Bayes. We observed that Naive Bayes produces an accuracy of where it tops the K-Nearest Neighbors and Support Vector Machines. Our insights provides several findings on areas such as improving or adding more techniques to improve our dataset and similarly to handle specific dataset for a specific objective.

**REFERENCES**


MonkeyLearn (2023), An Introduction To Machine Learning: https://monkeylearn.com/machine-learning/.

Lisa Tagliaferri (1 June 2022), An Introduction To Machine Learning: https://www.digitalocean.com/community/tutorials/an-introduction-to-machine-learning/

IBM (2023), What is Supervised Learning: https://www.ibm.com/topics/supervised-learning#:~:text=the%20next%20step-,What%20is%20supervised%20learning%3F,data%20or%20predict%20outcomes%20accurately.

Kumar, N. (2019, January 14). *Naive Bayes Classifiers - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/naive-bayes-classifiers/

*What is Naïve Bayes | IBM*. (n.d.). Www.ibm.com. https://www.ibm.com/topics/naive-bayes#:~:text=The%20Na%C3%AFve%20Bayes%20classifier%20is

What is the k-nearest neighbors algorithm? | IBM. (2023). Ibm.com. https://www.ibm.com/topics/knn#:~:text=Evelyn%20Fix%20and%20Joseph%20Hodges,%E2%80%9CNearest%20Neighbor%20Pattern%20Classification.%E2%80%9D

Fred Tabsharani (2023), support vector machine (SVM). https://www.techtarget.com/whatis/definition/support-vector-machine-SVM

Vijay Kanade ( 29 September 2022), What is a Support Vector Machine? Working, Types, and Examples. https://www.spiceworks.com/tech/big-data/articles/what-is-support-vector-machine/#:~:text=A%20support%20vector%20machine%20(SVM)%20is%20a%20machine%20learning%20algorithm,classes%2C%20labels%2C%20or%20outputs.

Rohith Gandhi (8 Jun 2018), Support Vector Machine- Introduction to Machine Learning Algorithms). https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Scikitlearn (2023), Support Vector Machines https://scikit-learn.org/stable/modules/svm.html

**TASK DIVISION & ROLES**

| No. | Name | Roles |
|---|---|---|
| 1 | CHONG WEN QIN | - Steps to build model algorithms<br>- Comparison and Recommendation<br>- Results and Discussion |
| 2 | MUHAMMAD FIRDAUS BIN MOHD.ALI | - Introduction<br>- Objectives<br>- Justification of Choice |
| 3 | SANTTOSH A/L G MUNIYANDY | - Concluding Remarks<br>- Lesson Learned<br>- Conclusion |