GROUP ASSIGNMENT

Optimisation and Deep Learning

STUDENT NAME:	**Chan Jia Le (TP049952)** **Soong Gim Hoy (TP053242)** **Thong Kyn Hui (TP053489)** **Ragneshwary (TP053143)** **Gan Yee Sin (TP053268)**
INTAKE CODE:	**UC3F2011CS(DA)**
MODULE CODE:	**CT118-3-3-ODL**
ASSIGNMENT TITLE:	**Cardiovascular Disease (Heart Disease) Prediction**
LECTURER NAME:	**Dr. Poolan Marikannan Booma**
HAND IN DATE:	**11th May 2021**
HAND OUT DATE:	**23rd July 2021**

Table of Contents

1.0 Introduction

1.1 Cardiovascular Disease (Heart Disease)

Cardiovascular Disease is a general term for all types of disease that affects the blood vessels or heart including coronary heart disease which leads to stroke, heart attacks, etc. and cardiovascular disease is often labeled as heart disease. Additionally, all heart diseases are cardiovascular diseases (National Heart, Lung and Blood Institute, 2020) and throughout the research these 2 terms will be used interchangeably. Cardiovascular Disease has been labeled as one of the most fatal diseases in the world and is the leading cause of death for United States, killing one person every 36 seconds (Center for Disease Control and Prevention, 2020). Similarly, Heart Disease is also a major contributor to the fatalities in Malaysia, causing 30,598 deaths in 2017 which is 22.13% of the total deaths in Malaysia (Lourdunathan, 2018). Cardiovascular Diseases (heart diseases) can be diagnosed through types of tests like Electrocardiogram (ECG), Echocardiogram, Stress Test, etc. (Whitworth, 2020). The cardiovascular disease prediction can be conducted using Machine Learning Classification. Classification is the procedure of prediction a class of given data values or points, the classes are often named as target variable and prediction of either “Yes” or “No” for heart diseases are known as binary classification (Asiri, 2018). There are many algorithms that could be implemented for binary classification which includes Logistic Regression, k-Nearest Neighbors, Decision Trees, Support Vector Machine, etc. (Brownlee, 2020).

1.2 Data Selection

The dataset selected for the prediction of cardiovascular diseases (heart diseases) is collected from Kaggle. The data set is consisted of 8,165 rows with 13 columns. The observations included inside the heart disease prediction dataset has 4121 records of heart disease which contributes to around 51% of the data set. The ratio of the prediction value/ target variable is balanced. Furthermore, the attributes of the data set include important details that is as mentioned above like Cholesterol, Blood Pressure, etc. which has significance in the prediction of cardiovascular diseases. The data is consisted of categorical and continuous data, some categorical data is presented in non-numeric form. Additionally, the metadata of the dataset is included. The collected data is suitable for the instance of predicting cardiovascular diseases (heart diseases) as all the crucial attributes and metadata is included so the attributes can be better understood by the developers. Hence, enabling developers to answer the research question (prediction of cardiovascular diseases (heart disease)).

1.3 Model Selection

1.3.1 Artificial Neural Network (ANN) – Chan Jia Le (TP049952)

Artificial Neural Network are often just named as Neural Networks and it is a modeling technique that is inspired by the Neurons or Nervous Systems of a human being which learns from representative data that is narrates a decision process or a physical phenomenon (Sadiq, et al., 2019). The neural network is consisted of 3 layers, the input layer, hidden layer, and output layer. Inside those layers, there neurons which transmits through a connection named as synapses, each input is not treated with equal weight and the weight signifies the importance of the certain input (Hsu, 2020). Due to its ability to learn, the neural network could produce an output without the limitation of given input. Thus, compared to other algorithms Artificial Neural Network works better because the hidden layers in Neural allows for a learning capability which in turns makes prediction more accurate (Thomas, 2019). Hence, making it one of the best methods used for predictive analysis, in this case predicting heart diseases.

1.3.2 Logistic Regression – Ragneshwary (TP053143)

Logistic Regression is a supervised classification algorithm that is generally used to predict the probability of a certain target variable. There are few types of logistic regression which are binary or binomial, multinomial and ordinal. Logistic regression is very much like linear regression as it assumes that the data follows the linear function, but logistic regression models the data by using the sigmoid function (GeeksforGeeks, 2019). The sigmoid function is $\sigma(z) = \frac{1}{1+e^{-z}}$ where $\sigma(z)$ is the output between 0 and 1, e = natural log base and z is the input function (Nishadi, 2019). Binomial logistic regression can be used to predict if the patient has high chance of heart disease or not. It would produce two possible outcomes which is 0 or 1.

1.3.3 K-Nearest Neighbor (K-NN) Classifier – Thong Kyn Hui (TP053489)

The K-Nearest Neighbor (K-NN) classification algorithm is one of the many supervised classification algorithms. K-NN classifies object based on the similarity measures, which could be the distance functions. The K-NN algorithm is a non-parametric algorithm, no assumptions are required on the underlying data. (JavaTPoint, 2018) K-NN classification works by calculating the distance between a new input with all the observations in the dataset, and then the algorithm classifies the new input with its nearest neighbour. There are multiple calculation methods to calculate the distance between points in a graph. The most used one are the Euclidean Formula, $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$ or the Hamming Distance formula, $\sum_{i=1}^{k}|x_i - y_i|$.

(Sayad, 2018) The Euclidean Formula is used to calculate the distance measure for continuous variable only, whereas Hamming Formula is used for categorical variables. In this research of heart disease prediction, the target variable in the dataset is a Boolean variable, which is either 0 or 1. Therefore, Hamming Distance Formula will be applied to calculate the distance function of any new input and then classifies it to 0 or 1.

1.3.4 Random Forest Classification – Gan Yee Sin (TP053268)

Random forest classification is a supervised classification algorithm that applies the fundamental working of the decision tree algorithm. It refers to an advanced version of decision trees algorithm which merges multiple built trees together to get a more stable and accurate prediction by using the ensemble learning technique. (Section, 2020) Random Forest Classifier as the ensemble model of decision trees could reduce the chances of overfitting by averaging the prediction result. It obtains the final prediction result by getting the prediction from each of the built decision trees and then pick the best solution based on the means of voting on various tree prediction results. In addition, increasing the number of trees in the forest, the more precise the prediction result. (Donges, 2021) Therefore, Random Forest is one of the good classification algorithms that is widely used in predictive analysis projects. In this case, it is used for predicting heart disease.

1.3.5 Support Vector Machine – Soong Gim Hoy (TP053242)

Biological applications of the Support Vector Machine (SVM) such as classifying cancer genes from bone marrows have shown the usefulness of the algorithm in life sciences. (Noble, 2006) Excelling in classification tasks but versatile for regression as well, SVM will be a good suit for a binary classification task to predict the presence of heart disease. Applying SVM on the heart disease dataset will form a hyperplane to clearly classify data points and pinpoint existences of heart disease. (Gandhi, 2018) Different kernel functions will be used to find the best fit for applying SVM to detect heart diseases.

2.0 Findings and Discussion

2.1 Exploratory Data Analysis (EDA) & Data Preprocessing

2.1.1 Importing Libraries.

Table 2.1.1.1: Description of Python libraries used

Name	Description
pandas	Data analysis and manipulation tool. The library's focus is on the usage of the Dataframe object. It is also used for describing data.
sklearn	The main library for training machine learning models and model configuration. It is built on NumPy, SciPy and matplotlib. Model evaluation can also be done through this library.
NumPy	This library provides the foundation to the usage of multi-dimensional arrays in Python. Mathematical operations and implementations are made easy with this library.
Matplotlib	A library used to plot graphs. It is essential in plotting data points to perform EDA and model evaluation.
seaborn	Used mainly for data visualization. It is developed based on Matplotlib and has fewer syntax.

Famous Python libraries are used while completing this project. Table 2.1.1.1 above shows the python library names and description. Figure 2.1.1.1 below shows the code snippet to import the libraries.

```
import pandas as pd
from sklearn.model_selection import train_test_split
import sklearn
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Figure 2.1.1.1: Code snippet to import libraries

2.1.2 Data Loading

```
data_loc = ('C:/Users/jared/Downloads/ODL Raw Dataset.csv')
df = pd.read_csv(data_loc)
```

Figure 2.1.2.1 Loading data from directory (Code Snippet)

```
df.head()
```

	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
0	7355	58.0	0.0	175.0	80.0	11500.0	90.0	1.0	1.0	0.0	0.0	1.0	1.0
1	153	40.0	0.0	168.0	63.0	909.0	60.0	2.0	1.0	0.0	0.0	1.0	0.0
2	1	63.0	1.0	167.0	59.0	906.0	0.0	1.0	1.0	0.0	0.0	1.0	0.0
3	63	42.0	0.0	160.0	60.0	902.0	60.0	1.0	1.0	0.0	0.0	1.0	0.0
4	7972	63.0	0.0	156.0	101.0	220.0	110.0	1.0	2.0	0.0	0.0	0.0	1.0

```
df.shape
```

```
(8165, 13)
```

Figure 2.1.2.2 Data Head and Data Shape (Code Snippet & Output)

2.1.3 Data Description

No	Attributes/ Columns	Data Type	Description
1	id	Unique	Id number of the records
2	age	Continuous	Age of the individual in days
3	gender	Categorical	Gender of the individual (1: women, 2: men)
4	height	Continuous	Height of the individual in cm
5	weight	Continuous	Weight of the individual in kg
6	ap_hi	Continuous	Systolic Blood Pressure of the individual
7	ap_lo	Continuous	Diastolic Blood Pressure of the individual
8	cholesterol	Categorical	Cholesterol of the Individual (1: normal, 2: above normal, 3: well above normal)
9	gluc	Categorical	Glucose Level of the Individual (1: normal, 2: above normal, 3: well above normal)
10	smoke	Categorical	Subject Feature (Whether individual smoke) (0: False in smoking, 1: True in smoking)

11	alco	Categorical	Subject Feature (Whether individual drink alcohol beverage) (0: False in drink alcohol beverage, 1: True in drink alcohol beverage)
12	active	Categorical	Subject Feature (Whether individual is active) (0: False in being active, 1: True in being active)
13	cardio (target)	Target	The target variable of the dataset (0: False in cardiovascular disease, 1: True in cardiovascular disease)

Table 2.1.3.1 Description of the Attributes in Dataset

```
df1.isnull().sum()

id                  0
age                14
gender              3
height             24
weight             11
ap_hi              12
ap_lo              14
cholesterol         8
gluc                9
smoke               6
alco               10
active              4
cardio (target)   116
dtype: int64
```

Figure 2.1.3.2 Missing Value in Dataset for each Attribute (Code Snippet & Output)

```
df1.describe()
```

	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	g
count	8165.000000	8151.000000	8162.000000	8141.000000	8154.000000	8153.000000	8151.000000	8157.000000	8156.000
mean	4083.000000	53.791069	0.348321	164.397126	74.352784	128.250092	97.767759	1.370725	1.226
std	2357.176807	6.805494	0.476468	8.198798	14.584301	128.162879	204.844211	0.681764	0.570
min	1.000000	30.000000	0.000000	70.000000	30.000000	-100.000000	0.000000	1.000000	1.000
25%	2042.000000	49.000000	0.000000	159.000000	65.000000	120.000000	80.000000	1.000000	1.000
50%	4083.000000	54.000000	0.000000	165.000000	72.000000	120.000000	80.000000	1.000000	1.000
75%	6124.000000	59.000000	1.000000	170.000000	82.000000	140.000000	90.000000	2.000000	1.000
max	8165.000000	65.000000	1.000000	250.000000	200.000000	11500.000000	10000.000000	3.000000	3.000

Figure 2.1.3.3 Details (Mean, Mode, etc.) of each Attribute (Code Snippet & Output)

No	Attributes/ Columns	Measures	Value
1	Id	-	All details of this column are unique; hence no measurements could be selected
2	Age	Mode	56.0
		Mean	53.791069
		Min	30.0
		Max	65.0
		Standard Deviation	6.805494
		First Quartile	49.0
		Third Quartile	59.0
		Missing Value	14
3	Gender	Mode	0.0 (represents women)
		Missing Values	3
4	Height	Mode	165.0
		Mean	164.397126
		Min	70.0
		Max	250.0
		Standard Deviation	8.198798
		First Quartile	159.0
		Third Quartile	170.0
		Missing Value	0
5	Weight	Mode	65.0
		Mean	74.352784
		Min	30.0
		Max	200.0
		Standard Deviation	74.352784
		First Quartile	65.0
		Third Quartile	82.0
		Missing Value	11
6	ap_hi	Mode	120.0
		Mean	128.250092

		Min	-100
		Max	11500
		Standard Deviation	128.250092
		First Quartile	120.0
		Third Quartile	140.0
		Missing Value	12
7	ap_lo	Mode	80.0
		Mean	97.767759
		Min	0.0
		Max	10000.0
		Standard Deviation	204.844211
		First Quartile	80.0
		Third Quartile	90.0
		Missing Value	14
8	Cholesterol	Mode	1.0 (Normal level)
		Missing Values	8
9	Gluc	Mode	1.0 (Normal level)
		Missing Values	9
10	Smoke	Mode	0.0 (False)
		Missing Values	6
11	Alco	Mode	0.0 (False)
		Missing Values	10
12	Active	Mode	1.0 (True)
		Missing Values	4
13	cardio (target)	Mode	1.0 (True)
		Missing Values	116

Table 2.1.3.4 Tabulated Details (Mean, Mode, etc.) of each Attribute

2.1.4 Data Visualization

Age

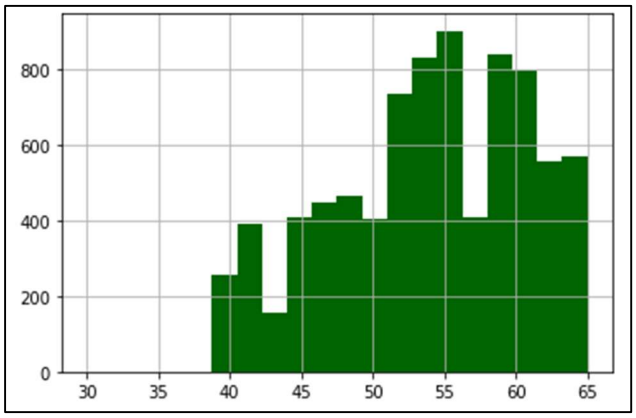

Figure 2.1.4.1 Histogram of Age (Output)

```
df['age'].hist(bins = 20,color = "DarkGreen")
```

Figure 2.1.4.2 Histogram of Age (Code Snippet)

Attribute:	Age
Graph type:	Histogram
Verdict:	The histogram reflects the distribution of age. The age group in the dataset are mostly all above 37 as reflected in the histogram with most of them in their early to mid-50s and late 50s to early 60s.

Table 2.1.4.3 Summary and Analysis of Visualization (Age)

Gender

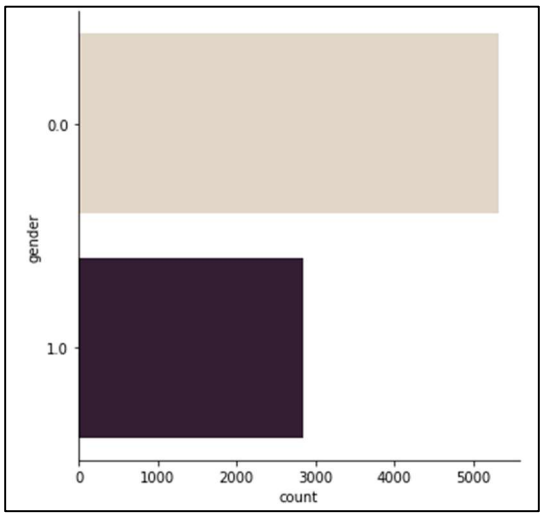

Figure 2.1.4.4 Bar Chart of Gender (Output)

```
sns.catplot(y="gender", kind = "count",palette="ch:.25", data=df)
```

Figure 2.1.4.5 Bar Chart of Gender (Code Snippet)

Attribute:	Gender
Graph type:	Bar Chart
Verdict:	The bar chart reflects the count of gender type in the dataset. The bar chart shows that most of the records are categorized under 0.0 which is women having more than 5000 records.

Table 2.1.4.6 Summary and Analysis of Visualization (Gender)

Weight

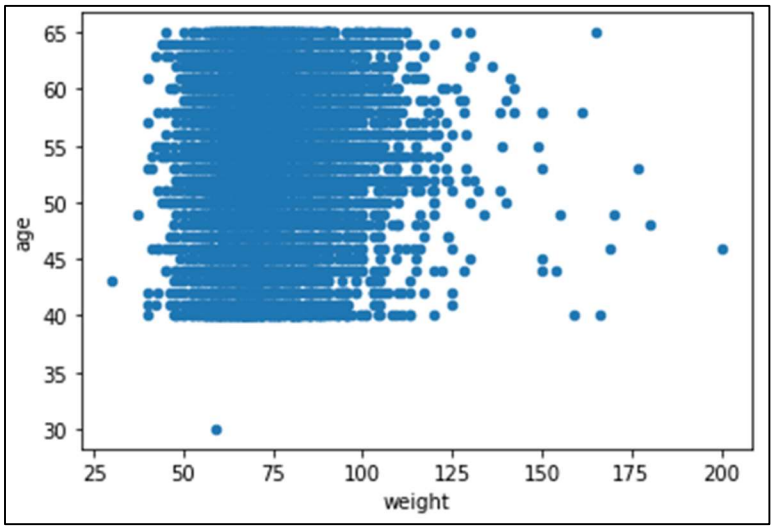

Figure 2.1.4.7 Scatter Plot of Weight (Output)

```
df.plot.scatter(x="weight",y='age')
```

Figure 2.1.4.8 Scatter Plot of Weight (Code Snippet)

Attribute:	Weight
Graph type:	Scatter Plot
Verdict:	The scatter plot reflects the distribution of weight based on age group. Surprisingly, the dataset has only one record that is aged 30. This observation though is an outlier is logical hence it should not be removed or modified. Additionally, the scatter plot reflects that majority of the weight is around 40 to 110 kg with higher values of weight located in the 40 to 58 age group.

Table 2.1.4.6 Summary and Analysis of Visualization (Weight)

Height

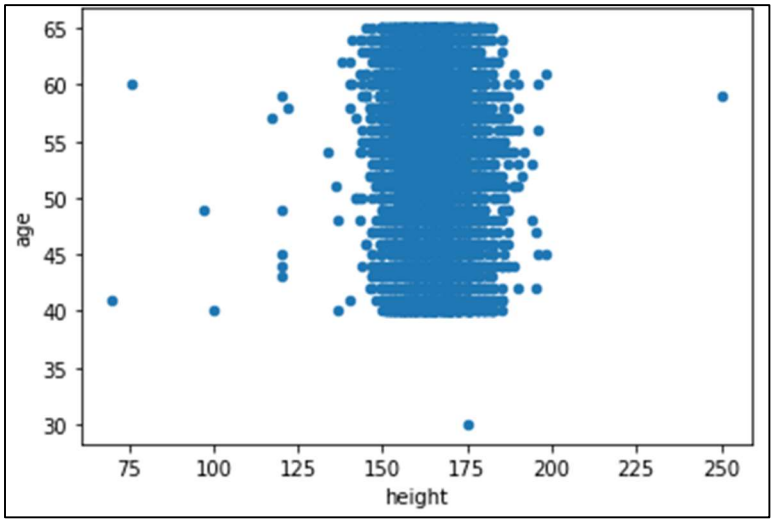

Figure 2.1.4.10 Scatter Plot of Height (Output)

```
df.plot.scatter(x="height",y='age')
```

Figure 2.1.4.11 Scatter Plot of Height (Code Snippet)

Attribute:	Height
Graph type:	Scatter Plot
Verdict:	The scatter plot above shows the distribution of height based on age. It has reflected that most of the height range is between 150 to 190 cm with some taller and shorter.

Table 2.1.4.12 Summary and Analysis of Visualization (Height)

Ap_hi

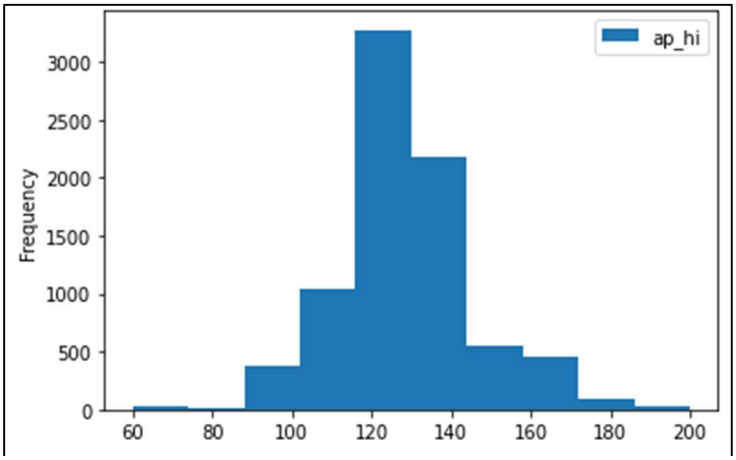

Figure 2.1.4.13 Histogram of Ap_hi (Systolic Blood Pressure) (Output)

```
df.plot.hist(y="ap_hi")
```

Figure 2.1.4.14 Histogram of Ap_hi (Systolic Blood Pressure) (Code Snippet)

Attribute:	ap_hi
Graph type:	Histogram
Verdict:	The histogram above shows the distribution of values in the attribute ap_hi. As reflected, most of the records have systolic blood pressure in the range of 120 to 140 while others are occupied by other values.

Table 2.1.4.15 Summary and Analysis of Visualization (Ap_hi)

Ap_lo

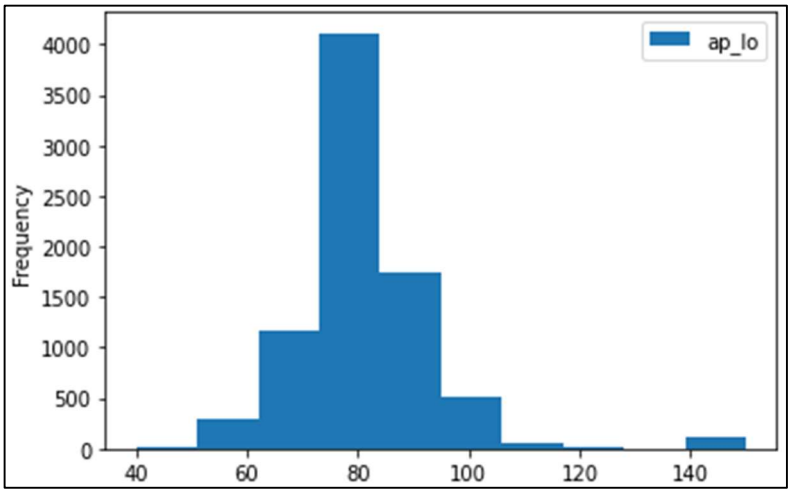

Figure 2.1.4.16 Histogram of Ap_lo (Diastolic Blood Pressure) (Output)

```
df.plot.hist(y="ap_lo")
```

Figure 2.1.4.17 Histogram of Ap_lo (Diastolic Blood Pressure) (Code Snippet)

Attribute:	ap_lo
Graph type:	Histogram
Verdict:	The histogram above shows the distribution of values in the attribute ap_lo. As reflected, most of the records have diastolic blood pressure in the range of 75 to 80 while others are occupied by other values. The histogram has reflected a normal distribution, however, above 140 in value, there are some outliers that could disrupt the data distribution. Hence, replacement or imputation should be made during pre-processing.

Table 2.1.4.18 Summary and Analysis of Visualization (Ap_lo)

Cholesterol

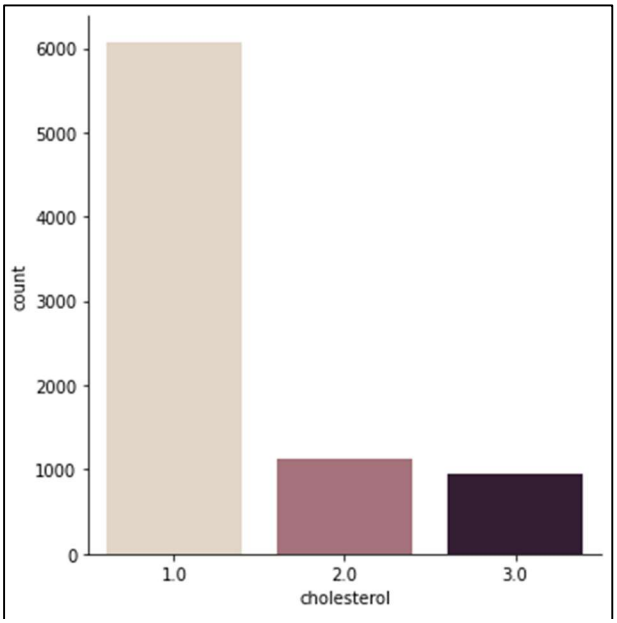

Figure 2.1.4.19 Bar Chart of Cholesterol (Output)

```
sns.catplot(x="cholesterol", kind = "count",palette="ch:.25", data=df1)
```

Figure 2.1.4.20 Bar Chart of Cholesterol (Code Snippet)

Attribute:	Cholesterol
Graph type:	Bar Chart
Verdict:	The bar chart above shows the count of different categories for attribute cholesterol. The bar chart is used due to the categorical nature of the cholesterol attribute. Additionally, as reflected in the bar chart, most of the records are categorized in 1.0 which is normal range while the categories 2.0 and 3.0 have similar count values.

Table 2.1.4.21 Summary and Analysis of Visualization (Cholesterol)

Gluc

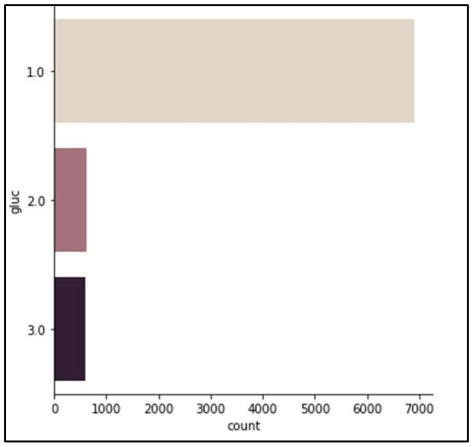

Figure 2.1.4.22 Bar Chart of Gluc (Glucose) (Output)

```
sns.catplot(y="gluc", kind = "count",palette="ch:.25", data=df)
```

Figure 2.1.4.23 Bar Chart of Gluc (Glucose) (Code Snippet)

Attribute:	Gluc
Graph type:	Bar Chart
Verdict:	The bar chart above shows the count of different categories for attribute gluc. Like the bar chart for cholesterol, majority of the record is categorized under 1.0 which is normal while others occupied by 2.0 and 3.0. The bar chart for cholesterol and glucose is extremely similar.

Table 2.1.4.24 Summary and Analysis of Visualization (Gluc)

Smoke

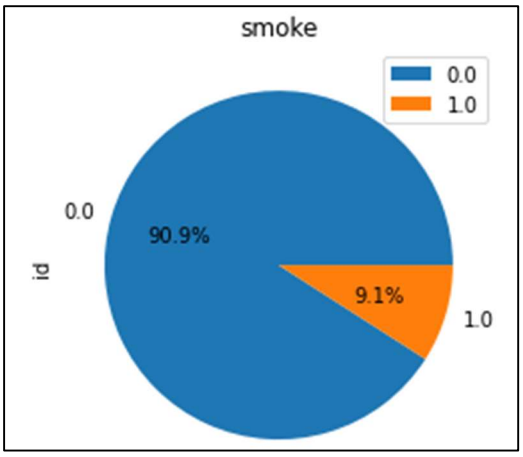

Figure 2.1.4.25 Pie Chart of Smoke (Output)

```
df.groupby(['smoke']).sum().plot(title = "smoke",kind= 'pie', y='id',autopct='%1.1f%%')
```

Figure 2.1.4.26 Pie Chart of Smoke (Output)

Attribute:	Smoke
Graph type:	Pie Chart
Verdict:	The pie chart above reflects the percentage of each categorical records of the attribute smoke. It shows that majority of the records do not smoke as most of the records are categorized at 0.0 (False). While only 9.1% of the observations does smoke.

Table 2.1.4.27 Summary and Analysis of Visualization (Smoke)

Alco

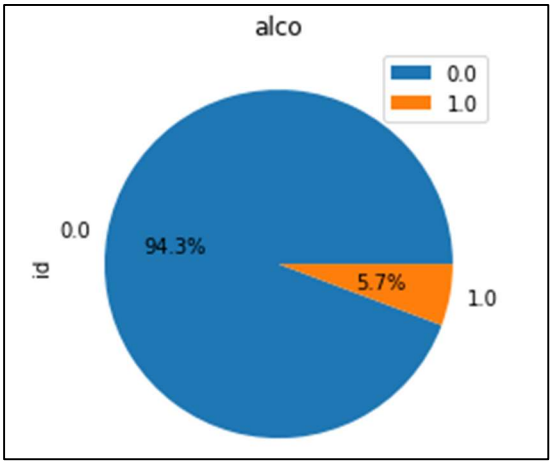

Figure 2.1.4.28 Pie Chart of Alco (Alcohol) (Output)

```
df1.groupby(['alco']).sum().plot(title = "alco",kind= 'pie', y='id',autopct='%1.1f%%')
```

Figure 2.1.4.29 Pie Chart of Alco (Alcohol) (Code Snippet)

Attribute:	Alco
Graph type:	Pie Chart
Verdict:	The pie chart above reflects the percentage of each categorical records of the attribute alco. Like the pie chart of smoke, most of the observations do not drink alcohol. In fact, over 94% of the observation do not consume alcohol while the other 5 to 6% does.

Table 2.1.4.30 Summary and Analysis of Visualization (Alco)

Active

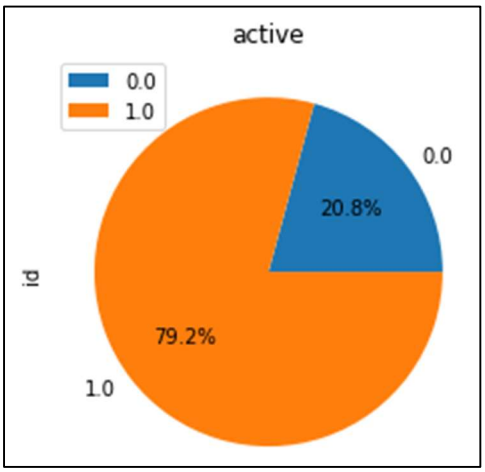

Figure 2.1.4.31 Pie Chart of Active (Output)

```
df.groupby(['active']).sum().plot(title = "active",kind= 'pie', y='id',autopct='%1.1f%%')
```

Figure 2.1.4.32 Pie Chart of Active (Code Snippet)

Attribute:	Active
Graph type:	Pie Chart
Verdict:	The pie chart above reflects the percentage of observations that are active or not. It is shown that 79.2% of the observations are active while the other 20.8% is not active. Hence, majority of the observation has an active and healthy lifestyle.

Table 2.1.4.33 Summary and Analysis of Visualization (Active)

Cardio (Target)

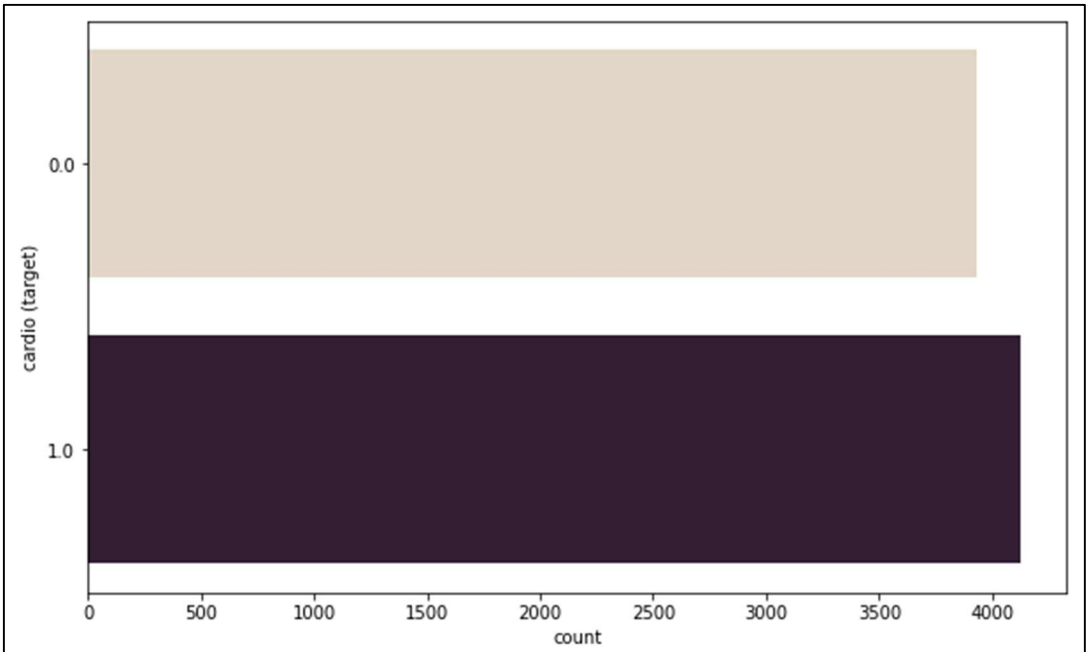

Figure 2.1.4.34 Bar Chart of Cardio (Target) (Output)

```
sns.catplot(y="gender", kind = "count",palette="ch:.25", data=df)
```

Figure 2.1.4.35 Bar Chart of Cardio (Target) (Code Snippet)

Attribute:	cardio (target)
Graph type:	Bar Chart
Verdict:	The bar chart reflects the count of the cardio (target) attribute. The target variable has a balanced outcome.

Table 2.1.4.36 Summary and Analysis of Visualization (Cardio (Target))

2.1.5 Data Pre-processing

In the data pre-processing stage, all the noise, missing or extreme values in the dataset will be handled either by imputing new values or removing the entire row. To check the total missing value present in the dataset, some python code will be used to count and sum the total missing values for each column. Additionally, the id of the dataset will be removed as it poses no significance in the classification model. The code snippet and results are as shown in figure below.

```
df.drop(['id'],axis=1)
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol
0	58.0	0.0	175.0	80.0	11500.0	90.0	1.0
1	40.0	0.0	168.0	63.0	909.0	60.0	2.0
2	63.0	1.0	167.0	59.0	906.0	0.0	1.0
3	42.0	0.0	160.0	60.0	902.0	60.0	1.0
4	63.0	0.0	156.0	101.0	220.0	110.0	1.0
...	...	...	...	...	...	...	...
8160	58.0	0.0	169.0	75.0	NaN	80.0	1.0
8161	50.0	1.0	165.0	70.0	NaN	80.0	1.0

```
[11] # Checking the missing values in each variables
     df.isnull().sum()

age                14
gender              3
height             24
weight             11
ap_hi              12
ap_lo              14
cholesterol         8
gluc                9
smoke               6
alco               10
active              4
cardio (target)   116
```

Figure 2.1.5.1 Dropping id column (Left) and rechecking missing value (Right)

Age

```
age = pd.isnull(df["age"])
df[age]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
16	NaN	1.0	165.0	73.0	125.0	90.0	1.0	1.0	0.0	0.0	0.0	0.0
41	NaN	0.0	156.0	63.0	100.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
62	NaN	0.0	153.0	73.0	120.0	80.0	2.0	1.0	0.0	0.0	1.0	0.0
77	NaN	0.0	172.0	55.0	110.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
92	NaN	0.0	156.0	65.0	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
107	NaN	0.0	168.0	69.0	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
116	NaN	1.0	178.0	78.0	120.0	80.0	1.0	1.0	0.0	0.0	0.0	0.0
131	NaN	0.0	158.0	73.0	110.0	70.0	1.0	2.0	0.0	0.0	1.0	0.0
143	NaN	0.0	159.0	97.0	120.0	80.0	1.0	3.0	0.0	0.0	1.0	0.0
155	NaN	1.0	168.0	72.0	120.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
167	NaN	1.0	171.0	74.0	120.0	80.0	1.0	1.0	1.0	1.0	1.0	0.0
5589	NaN	0.0	178.0	78.0	120.0	80.0	1.0	1.0	0.0	0.0	0.0	1.0
5607	NaN	0.0	156.0	91.0	115.0	70.0	3.0	1.0	0.0	0.0	0.0	1.0
7378	NaN	1.0	160.0	60.0	120.0	80.0	1.0	1.0	0.0	0.0	1.0	1.0

Figure 2.1.5.2 Rows with Missing values for Age Variable

There are 14 missing values in the age variable. These values will be replaced by the statistical measures such as mean, mode or median so that the imputed values will not affect the spread of the data. The imputation measures are selected based on the skewness of the variable. This skewness is visualized by plotting a box and whisker plot as shown below.

```
# Plotting Boxplot after imputing
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(df.age)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7f9025941210>
```

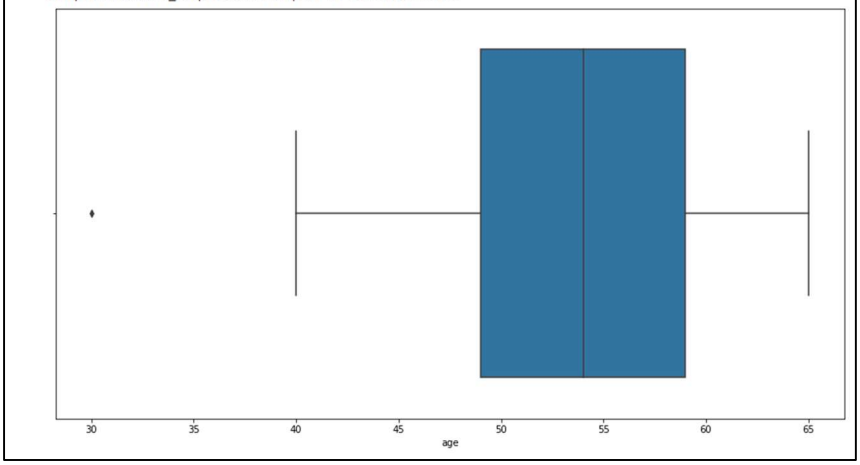

Figure 2.1.5.3 Box and Whisker Plot for Age

As per shown in the figure above, the age variable is not skewed and it follows a normal distribution. In this case, the missing values in the age variable will be imputed by using the mean value. The code snippet is as shown below.

```
# Imputing missing value of age variable with mean value
dataset['age'] = dataset['age'].fillna(dataset['age'].mean())
# Validating missing value after imputing
dataset.isnull().sum()
```

```
age                0
gender             3
height            24
weight            11
ap_hi             12
ap_lo             14
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
dtype: int64
```

Figure 2.1.5.4 Code Snippet to Impute Age Variable and Results

Gender

```
[218] gender_null = pd.isnull(df["gender"])
      df[gender_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
6450	65.0	NaN	154.0	86.0	140.0	90.0	1.0	1.0	0.0	0.0	1.0	1.0
7487	61.0	NaN	161.0	83.0	140.0	90.0	2.0	2.0	0.0	0.0	1.0	1.0
7731	65.0	NaN	170.0	69.0	110.0	70.0	3.0	3.0	0.0	0.0	0.0	1.0

Figure 2.1.5.5 Rows with Missing Values for Gender Variable

The gender variable is a categorical variable with only value "0" and "1". For categorical variables, the missing value will be replaced by the value which has the highest frequency in the dataset, which is also known as the mode. The mode of this column is visualized by plotting a bar chart.

```
# Plotting distribution graph for gender variable
sns.displot(dataset.gender)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f9028bfdf90>
```

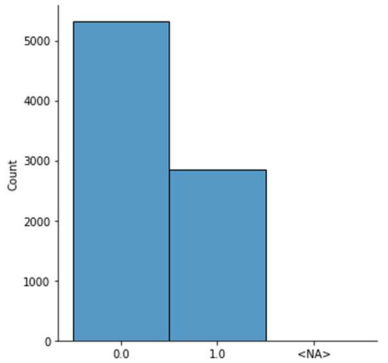

Figure 2.1.5.6 Distribution Graph for Gender Variable

From the bar chart above, the mode of gender is "0" which is women, therefore, the missing value will be imputed by mode value "0" so that it does not affect the accuracy of the dataset.

```
df["gender"].fillna('0', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height            24
weight            11
ap_hi             12
ap_lo             14
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.7 Code Snippet to Impute Gender Variable and Results

Height

```
height_null = pd.isnull(df["height"])
df[height_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
7	41.0	0	NaN	60.0	110.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
21	53.0	1	NaN	69.0	130.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
44	58.0	0	NaN	66.0	120.0	80.0	1.0	1.0	0.0	0.0	0.0	0.0
65	58.0	1	NaN	94.0	120.0	80.0	2.0	1.0	0.0	0.0	1.0	0.0
98	47.0	0	NaN	60.0	120.0	70.0	1.0	1.0	0.0	NaN	0.0	0.0
133	51.0	0	NaN	51.0	100.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
176	42.0	0	NaN	75.0	110.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
198	41.0	0	NaN	65.0	120.0	80.0	1.0	1.0	0.0	0.0	0.0	0.0
246	64.0	0	NaN	80.0	130.0	80.0	1.0	1.0	0.0	0.0	0.0	0.0
561	65.0	0	NaN	73.0	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
648	58.0	1	NaN	106.0	220.0	120.0	3.0	3.0	0.0	0.0	1.0	0.0
702	49.0	1	NaN	103.0	150.0	100.0	2.0	1.0	0.0	0.0	1.0	0.0
733	45.0	1	NaN	65.0	90.0	60.0	1.0	1.0	0.0	0.0	1.0	0.0
772	56.0	0	NaN	57.0	120.0	80.0	3.0	1.0	0.0	0.0	1.0	0.0
787	55.0	0	NaN	77.0	130.0	80.0	2.0	2.0	0.0	0.0	1.0	0.0
2462	49.0	0	NaN	65.0	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
2510	54.0	0	NaN	75.0	110.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
2581	61.0	0	NaN	58.0	150.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
6438	50.0	1	NaN	65.0	140.0	90.0	2.0	1.0	1.0	1.0	1.0	1.0
7425	58.0	0	NaN	69.0	NaN	90.0	2.0	1.0	0.0	0.0	1.0	1.0
7563	43.0	0	NaN	58.0	110.0	70.0	1.0	1.0	0.0	0.0	1.0	1.0
7659	52.0	0	NaN	131.0	110.0	80.0	1.0	1.0	0.0	0.0	1.0	1.0
7693	55.0	1	NaN	67.0	110.0	70.0	1.0	2.0	0.0	0.0	1.0	1.0
7808	43.0	0	NaN	79.0	150.0	90.0	1.0	1.0	0.0	0.0	1.0	1.0

Figure 2.1.5.8 Rows with Missing Values for Height Variable

```
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.height)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7fa2720bc950>
```

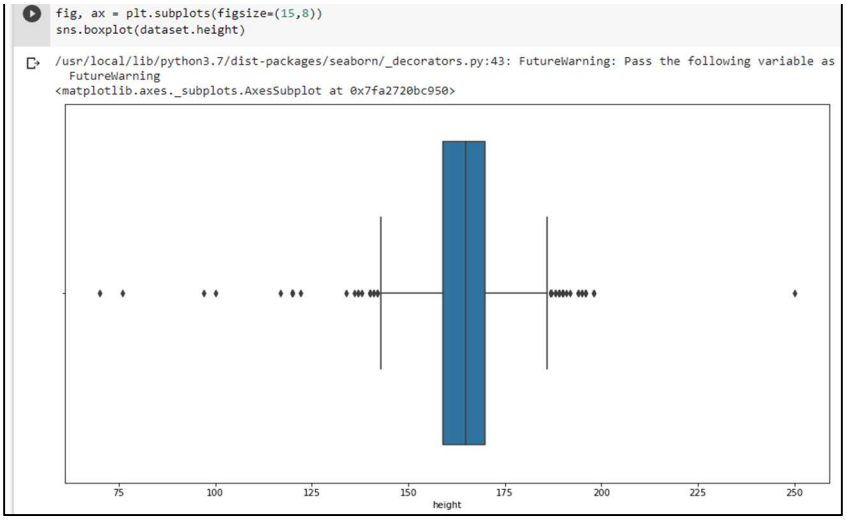

Figure 2.1.5.9 Box and Whisker Plot for Height Variable

As per shown in the figure above, the height variable is not skewed and it follows a normal distribution. In this case, the missing values in the height variable will be imputed by using the mean value. The code snippet is as shown below.

```
# Imputing missing value of height variable with mean value
dataset['height'] = dataset['height'].fillna(dataset['height'].mean())
# Validating missing value after imputing
dataset.isnull().sum()
```

```
age                0
gender             0
height             0
weight            11
ap_hi             12
ap_lo             14
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.10 Code Snippet to Impute Height Variable and Results

Weight

```
weight_null = pd.isnull(df["weight"])
df[weight_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
20	62.0	0	170.0	NaN	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
29	63.0	1	168.0	NaN	160.0	100.0	1.0	1.0	0.0	0.0	1.0	0.0
126	48.0	1	160.0	NaN	110.0	70.0	1.0	1.0	0.0	0.0	1.0	0.0
362	44.0	0	147.0	NaN	120.0	90.0	1.0	1.0	0.0	0.0	1.0	0.0
492	60.0	0	168.0	NaN	120.0	80.0	1.0	1.0	0.0	0.0	1.0	0.0
598	60.0	0	164.0	NaN	90.0	60.0	3.0	NaN	0.0	0.0	0.0	0.0
6460	60.0	0	155.0	NaN	130.0	80.0	3.0	3.0	0.0	0.0	0.0	1.0
7373	63.0	0	157.0	NaN	120.0	80.0	1.0	1.0	0.0	0.0	1.0	1.0
7510	56.0	0	165.0	NaN	120.0	70.0	1.0	1.0	0.0	0.0	1.0	1.0
7539	60.0	1	167.0	NaN	130.0	80.0	3.0	1.0	0.0	1.0	1.0	1.0
8061	46.0	0	158.0	NaN	110.0	70.0	1.0	3.0	0.0	0.0	1.0	NaN

Figure 2.1.5.11 Rows with Missing Values for Weight Variable

```
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.weight)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7fa27202c490>
```

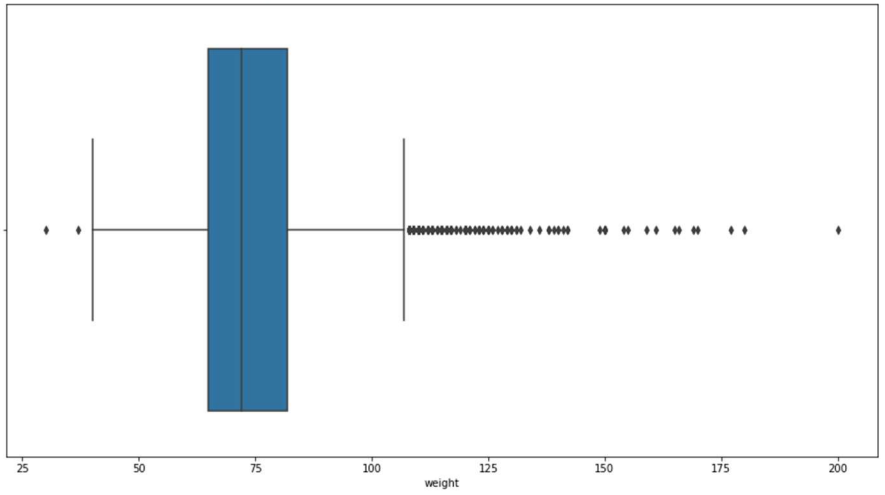

Figure 2.1.5.12 Box and Whisker Plot for Weight Variable

As per shown in the figure above, the weight variable is slightly skewed to the right, which shows a positive skewed variable. In this case, the missing values in the height variable will be imputed by using the median value. There are multiple outliers present in this variable. However, there are not considered are extreme value which do not make sense, therefore they are not removed. The code snippet is as shown below.

```
# Imputing missing value of weight variable with median value
dataset['weight'] = dataset['weight'].fillna(dataset['weight'].median())
# Validating missing value after imputing
dataset.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi             12
ap_lo             14
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.13 Code Snippet to Impute Weight Variable and Results

Ap_hi

```
aphi_null = pd.isnull(df["ap_hi"])
df[aphi_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
50	47.0	0	156.000000	60.0	NaN	80.0	1.0	1.0	0.0	0.0	1.0	0.0
135	46.0	1	172.000000	86.0	NaN	80.0	1.0	1.0	0.0	0.0	1.0	0.0
177	53.0	0	157.000000	94.0	NaN	80.0	2.0	1.0	0.0	0.0	1.0	0.0
255	46.0	0	166.000000	62.0	NaN	70.0	1.0	1.0	0.0	0.0	1.0	0.0
296	65.0	0	154.000000	79.0	NaN	80.0	1.0	1.0	0.0	0.0	1.0	0.0
489	44.0	0	163.000000	60.0	NaN	70.0	1.0	1.0	0.0	0.0	1.0	0.0
753	61.0	0	160.000000	72.0	NaN	80.0	1.0	1.0	0.0	0.0	1.0	0.0
2455	58.0	0	169.000000	75.0	NaN	80.0	1.0	1.0	0.0	0.0	1.0	0.0
2476	58.0	0	164.000000	97.0	NaN	90.0	2.0	2.0	1.0	0.0	1.0	0.0
2495	50.0	1	165.000000	70.0	NaN	80.0	1.0	1.0	0.0	0.0	0.0	0.0
7425	58.0	0	164.397126	69.0	NaN	90.0	2.0	1.0	0.0	0.0	1.0	1.0
7714	41.0	1	174.000000	74.0	NaN	90.0	1.0	2.0	0.0	0.0	1.0	1.0

Figure 2.1.5.14 Rows with Missing Values for ap_hi Variable

```
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.ap_hi)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7fa271f9fc10>
```

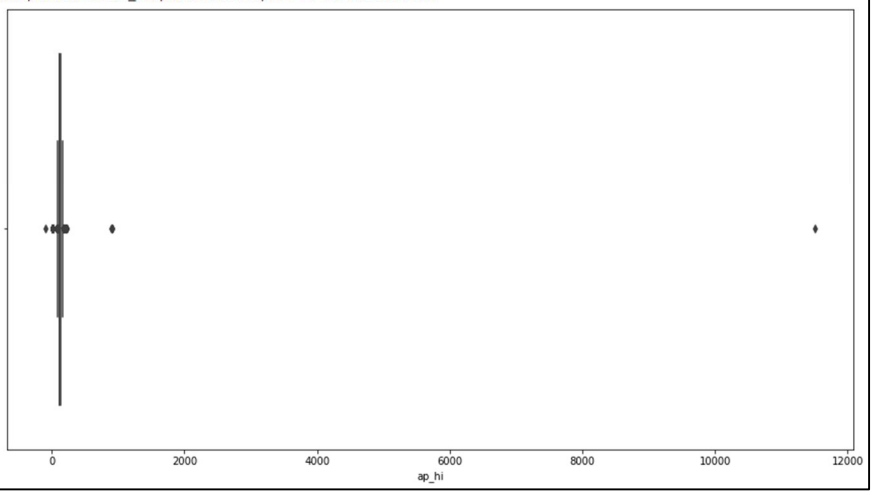

Figure 2.1.5.15 Box and Whisker Plot for ap_hi Variable Before Eliminating Outliers

The whisker plot above shows that there are extreme outliers. These values will be replaced before imputing the missing values.

```
dataset.loc[(dataset.ap_hi > 200 ),'ap_hi']= 200
dataset.loc[(dataset.ap_hi < 60 ),'ap_hi']= 60
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.ap_hi)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7f90323cdcd0>
```

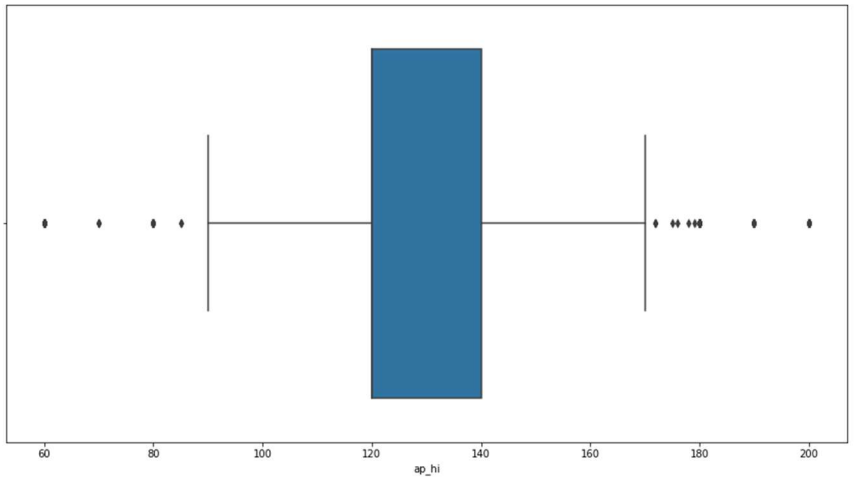

Figure 2.1.5.16 Box and Whisker Plot for ap_hi After Eliminating Outliers

As per shown in the figure above, the ap_hi variable is not skewed. In this case, the missing values in the height variable will be imputed by using the mean value. The code snippet is as shown below.

```
# Imputing missing value of ap_hi variable with mean value
dataset['ap_hi'] = dataset['ap_hi'].fillna(dataset['ap_hi'].mean())
# Validating missing value after imputing
dataset.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo             14
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.17 Code Snippet to Impute ap_hi Variable and Results.

Ap_lo

```
aplo_null = pd.isnull(df["ap_lo"])
df[aplo_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
90	54.0	0	164.0	57.0	120.0	NaN	2.0	1.0	0.0	0.0	1.0	0.0
179	56.0	0	164.0	50.0	100.0	NaN	1.0	1.0	1.0	0.0	1.0	0.0
439	55.0	0	163.0	60.0	120.0	NaN	1.0	1.0	0.0	0.0	0.0	0.0
630	49.0	1	173.0	86.0	120.0	NaN	1.0	1.0	0.0	0.0	0.0	0.0
639	41.0	0	156.0	59.0	120.0	NaN	1.0	1.0	0.0	0.0	0.0	0.0
2471	57.0	1	178.0	81.0	120.0	NaN	1.0	1.0	0.0	0.0	1.0	0.0
2592	52.0	1	177.0	73.0	120.0	NaN	1.0	1.0	1.0	1.0	1.0	0.0
6446	57.0	1	168.0	70.0	120.0	NaN	1.0	1.0	0.0	0.0	1.0	1.0
7374	58.0	0	158.0	68.0	120.0	NaN	1.0	1.0	0.0	0.0	1.0	1.0
7485	56.0	0	158.0	80.0	130.0	NaN	1.0	1.0	0.0	0.0	1.0	1.0
7529	49.0	1	162.0	80.0	140.0	NaN	1.0	1.0	0.0	0.0	1.0	1.0
7565	56.0	1	172.0	92.0	125.0	NaN	1.0	1.0	0.0	0.0	1.0	1.0
7645	41.0	1	169.0	69.0	120.0	NaN	1.0	1.0	0.0	0.0	0.0	1.0
7770	47.0	1	167.0	84.0	130.0	NaN	2.0	1.0	0.0	0.0	0.0	1.0

Figure 2.1.5.18 Rows with Missing Values for ap_lo Variable

```
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.ap_lo)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7f903235bed0>
```

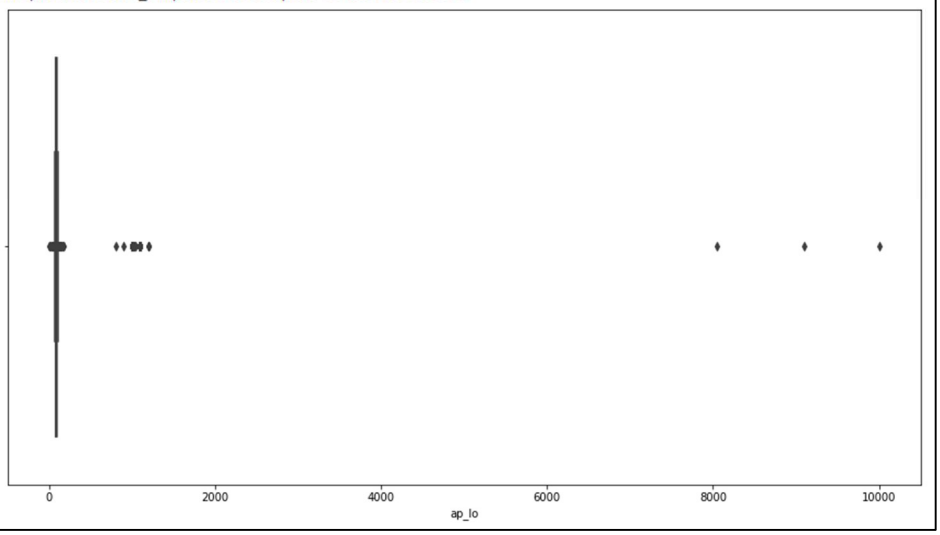

Figure 2.1.5.19 Box and Whisker Plot for ap_lo Before Eliminating Outliers

The whisker plot above shows that there are extreme outliers. These values will be replaced before imputing the missing values.

```
dataset.loc[(dataset.ap_lo > 150 ),'ap_lo']= 150
dataset.loc[(dataset.ap_lo < 40 ),'ap_lo']= 40
fig, ax = plt.subplots(figsize=(15,8))
sns.boxplot(dataset.ap_lo)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7f9031d10550>
```

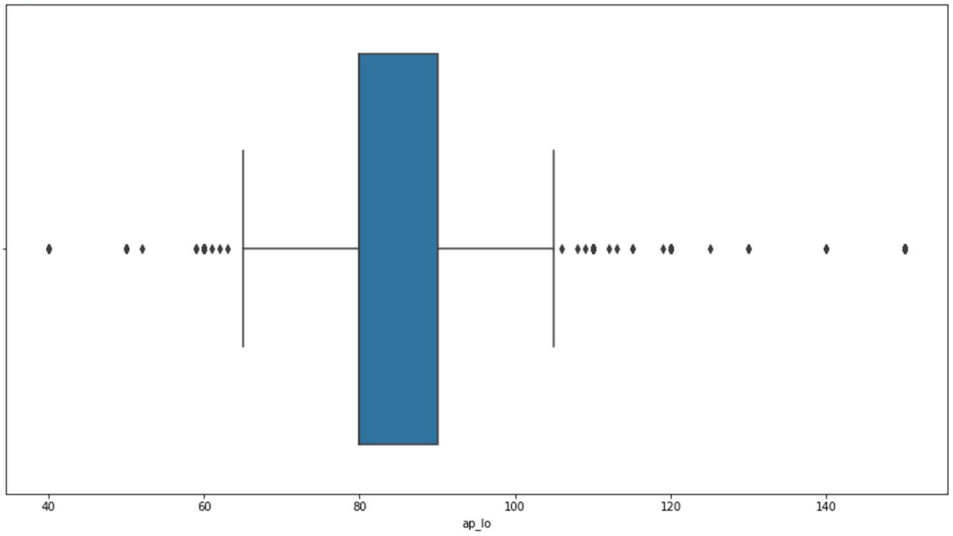

Figure 2.1.5.20 Box and Whisker Plot for ap_lo After Eliminating Outliers

As per shown in the figure above, the ap_hi variable is not skewed. In this case, the missing values in the height variable will be imputed by using the mean value. The code snippet is as shown below.

```
# Imputing missing value of ap_lo variable with mean value
dataset['ap_lo'] = dataset['ap_lo'].fillna(dataset['ap_lo'].mean())
# Validating missing value after imputing
dataset.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        8
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.21 Code Snippet to Impute ap_lo Variable and Results.

Cholesterol

```
cholesterol_null = pd.isnull(df["cholesterol"])
df[cholesterol_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
91	60.0	1.0	76.0	55.0	120.0	80.0	NaN	1.0	0.0	0.0	1.0	0.0
795	53.0	0.0	165.0	66.0	12.0	80.0	NaN	1.0	0.0	0.0	1.0	0.0
6455	57.0	1.0	168.0	103.0	140.0	100.0	NaN	1.0	0.0	0.0	0.0	1.0
7386	49.0	1.0	170.0	80.0	140.0	90.0	NaN	3.0	0.0	0.0	1.0	1.0
7680	65.0	0.0	162.0	88.0	140.0	90.0	NaN	1.0	0.0	0.0	1.0	1.0
8018	55.0	1.0	170.0	89.0	130.0	80.0	NaN	1.0	1.0	0.0	1.0	1.0
8107	55.0	0.0	152.0	82.0	120.0	80.0	NaN	1.0	0.0	NaN	1.0	NaN
8155	55.0	1.0	178.0	78.0	120.0	90.0	NaN	1.0	0.0	0.0	1.0	NaN

Figure 2.1.5.22 Rows with Missing Values for Cholesterol Variable

```
[204] # Plotting distribution graph for cholesterol variable
      sns.displot(df.cholesterol)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f9025e4d890>
```

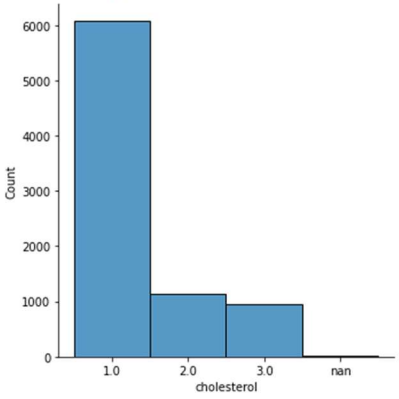

Figure 2.1.5.23 Distribution Graph of Cholesterol Variable

From the bar chart above, the mode of cholesterol is “1”, therefore, the missing value will be imputed by mode value “1” so that it does not affect the accuracy of the dataset.

```
df["cholesterol"].fillna('1', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               9
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.24 Code Snippet to Impute cholesterol Variable and Results.

Gluc

```
gluc_null = pd.isnull(df["gluc"])
df[gluc_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
100	44.0	0	154.0	56.0	120.0	70.0	1	NaN	0.0	0.0	1.0	0.0
134	51.0	0	170.0	65.0	120.0	80.0	1	NaN	0.0	0.0	1.0	0.0
200	51.0	1	176.0	98.0	120.0	80.0	1	NaN	0.0	0.0	1.0	0.0
371	57.0	0	176.0	120.0	150.0	90.0	1	NaN	0.0	0.0	0.0	0.0
598	60.0	0	164.0	72.0	90.0	60.0	3	NaN	0.0	0.0	0.0	0.0
7924	45.0	0	169.0	71.0	130.0	80.0	1	NaN	0.0	0.0	1.0	1.0
8029	59.0	1	180.0	105.0	150.0	90.0	2	NaN	0.0	0.0	1.0	1.0
8086	50.0	0	158.0	67.0	120.0	80.0	1	NaN	0.0	NaN	1.0	NaN
8119	57.0	0	158.0	90.0	120.0	80.0	2	NaN	0.0	0.0	1.0	NaN

Figure 2.1.5.25 Rows with Missing Values for Gluc Variable

```
[211] # Plotting distribution graph for gluc variable
      sns.displot(df.gluc)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f902607a990>
```

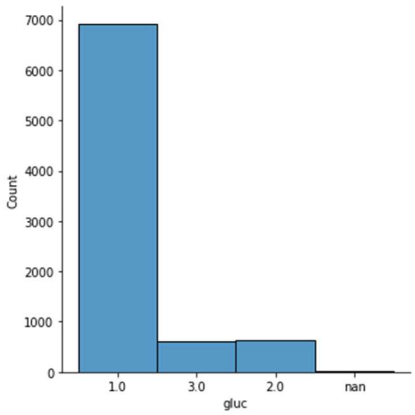

Figure 2.1.5.26 Distribution Graph of Gluc Variable

From the bar chart above, the mode of gluc is “1”, therefore, the missing value will be imputed by mode value “1” so that it does not affect the accuracy of the dataset.

```
df["gluc"].fillna('1', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               0
smoke              6
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.27 Code Snippet to Impute Gluc Variable and Results

Smoke

```
[240] smoke_null = pd.isnull(df["smoke"])
      df[smoke_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
181	63.0	0	156.0	55.0	110.0	60.0	1	1	NaN	0.0	1.0	0.0
562	45.0	1	173.0	99.0	130.0	80.0	2	1	NaN	0.0	1.0	0.0
721	56.0	1	178.0	105.0	120.0	80.0	1	1	NaN	0.0	1.0	0.0
749	58.0	0	168.0	80.0	120.0	70.0	1	1	NaN	0.0	0.0	0.0
7899	59.0	0	156.0	50.0	120.0	80.0	3	1	NaN	0.0	1.0	1.0
8012	56.0	0	149.0	56.0	80.0	60.0	1	1	NaN	0.0	1.0	1.0

Figure 2.1.5.28 Rows with Missing Values for Gluc Variable

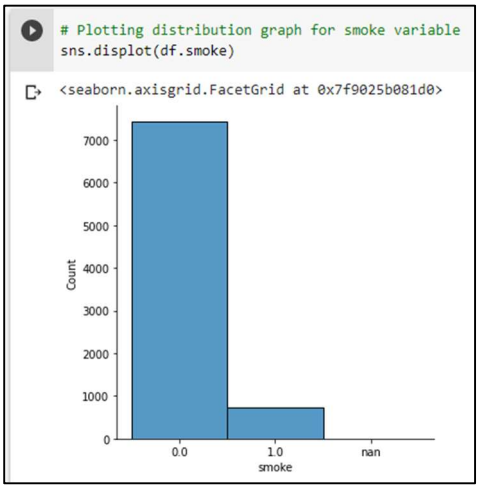

Figure 2.1.5.29 Distribution Graph of Gluc Variable

From the bar chart above, the mode of smoke is "0", therefore, the missing value will be imputed by mode value "0" so that it does not affect the accuracy of the dataset.

```
df["smoke"].fillna('0', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               0
smoke              0
alco              10
active             4
cardio (target)  116
```

Figure 2.1.5.30 Code Snippet to Impute Smoke Variable and Results

Alco

```
[241] alco_null = pd.isnull(df["alco"])
      df[alco_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	ctive	cardio (target)
98	47.0	0	164.397126	60.0	120.0	70.0	1	1	0.0	NaN	0.0	0.0
196	54.0	0	156.000000	60.0	120.0	90.0	1	1	0.0	NaN	1.0	0.0
284	54.0	1	170.000000	81.0	130.0	90.0	2	1	1.0	NaN	1.0	0.0
622	48.0	0	158.000000	62.0	110.0	70.0	1	1	0.0	NaN	1.0	0.0
7905	62.0	0	165.000000	69.0	130.0	80.0	1	1	0.0	NaN	1.0	1.0
7993	44.0	1	183.000000	100.0	170.0	110.0	2	1	1.0	NaN	1.0	1.0
8086	50.0	0	158.000000	67.0	120.0	80.0	1	1	0.0	NaN	1.0	NaN
8107	55.0	0	152.000000	82.0	120.0	80.0	1	1	0.0	NaN	1.0	NaN
8142	53.0	1	160.000000	65.0	130.0	80.0	1	1	1.0	NaN	1.0	NaN
8162	65.0	0	160.000000	59.0	120.0	80.0	1	1	0.0	NaN	0.0	NaN

Figure 2.1.5.31 Rows with Missing Values for Alco Variable

```
# Plotting distribution graph for alco variable
sns.displot(df.alco)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f9025b043d0>
```

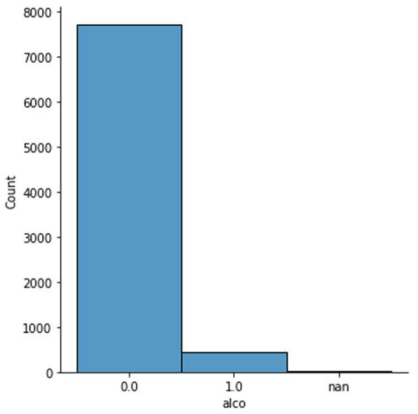

Figure 2.1.5.32 Distribution Graph of Smoke Variable

From the bar chart above, the mode of alco is “0”, therefore, the missing value will be imputed by mode value “0” so that it does not affect the accuracy of the dataset.

```
df["alco"].fillna('0', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               0
smoke              0
alco               0
active             4
cardio (target)  116
```

Figure 2.1.5.33 Code Snippet to Impute Alco Variable and Results

Active

```
active_null = pd.isnull(df["active"])
df[active_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
447	44.0	0	167.0	69.0	110.0	70.0	1	1	0.0	0.0	NaN	0.0
687	40.0	0	155.0	68.0	140.0	90.0	3	3	0.0	0.0	NaN	0.0
7960	60.0	1	166.0	75.0	140.0	80.0	1	2	1.0	0.0	NaN	1.0
8011	47.0	0	153.0	55.0	110.0	70.0	2	1	0.0	0.0	NaN	1.0

Figure 2.1.5.34 Rows with Missing Values for Active Variable

```
# Plotting distribution graph for active variable
sns.displot(df.active)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f9025a0c2d0>
```

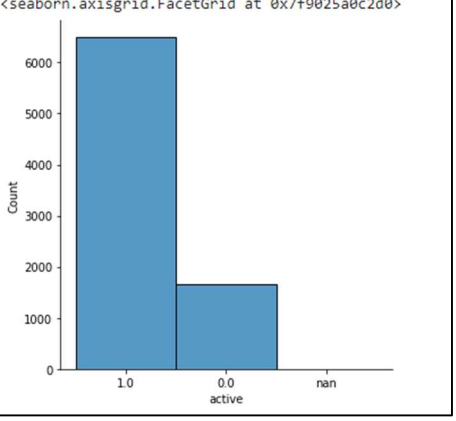

Figure 2.1.5.35 Distribution Graph of Active Variable

From the bar chart above, the mode of active is “1”, therefore, the missing value will be imputed by mode value “1” so that it does not affect the accuracy of the dataset.

```
df["active"].fillna('1', inplace = True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               0
smoke              0
alco               0
active             0
cardio (target)    116
```

Figure 2.1.5.36 Code Snippet to Impute Active Variable and Results

Cardio (Target)

```
target_null = pd.isnull(df["cardio (target)"])
df[target_null]
```

	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio (target)
8049	51.0	1	168.0	62.0	110.0	80.0	1	1	0	0	1	NaN
8050	48.0	0	156.0	56.0	100.0	60.0	1	1	0	0	0	NaN
8051	61.0	0	151.0	67.0	120.0	80.0	2	2	0	0	0	NaN
8052	61.0	0	157.0	93.0	130.0	80.0	3	1	0	0	1	NaN
8053	49.0	0	158.0	71.0	110.0	70.0	1	1	0	0	1	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...
8160	59.0	0	159.0	73.0	140.0	90.0	1	1	0	0	0	NaN
8161	52.0	0	150.0	87.0	126.0	72.0	1	1	0	0	1	NaN
8162	65.0	0	160.0	59.0	120.0	80.0	1	1	0	0	0	NaN
8163	51.0	1	167.0	84.0	130.0	90.0	1	1	0	0	1	NaN
8164	64.0	0	160.0	81.0	17.0	80.0	1	1	0	0	1	NaN

116 rows × 12 columns

Figure 2.1.5.37 Rows with Missing Values for Cardio (Target) Variable

Since the research will be developing models to predict the target variable, all the rows with missing target variable will be dropped and removed from the dataset. When all the missing

```
df.dropna(inplace=True)
df.isnull().sum()
```

```
age                0
gender             0
height             0
weight             0
ap_hi              0
ap_lo              0
cholesterol        0
gluc               0
smoke              0
alco               0
active             0
cardio (target)    0
```

Figure 2.1.5.38 Code Snippet to Drop All Rows with Missing Values for Target Variable

values and extreme outliers are imputed, the final pre-processed dataset has only 8049 rows and will be exported as “ODL_Preprocessed_Dataset” for machine learning model building.

```
df.shape
```

```
(8049, 12)
```

Figure 2.1.5.39 Data Frame Shape After Pre-processing

```
[ ] df.to_csv('/content/drive/MyDrive/Colab Notebooks/ODL/ODL_Preprocessed_Dataset.csv', header=True, index = False)
```

Figure 2.1.5.40 Exporting Pre-processed data to new csv.

2.2 Feature Selection

In the feature selection stage, the researcher will select set of features for further model building using correlation coefficient technique. Correlation coefficient for feature selection aids in finding the good independent variables that are highly correlated to the dependent variable (target). (Aman, 2020) A comparison table that stated the correlation coefficient values of the raw dataset and the cleaned dataset will be provided at below.

```
#calculate the correlation between variables using heatmap
corr = dataframe.corr()
plt.figure(figsize = (10, 10))
sns.heatmap(dataframe.corr(),annot = True, cmap = 'YlGnBu');
plt.figure(figsize = (6, 12))
```

Figure 2.2.1: Code Snippet of Correlation Coefficient (Raw Dataset)

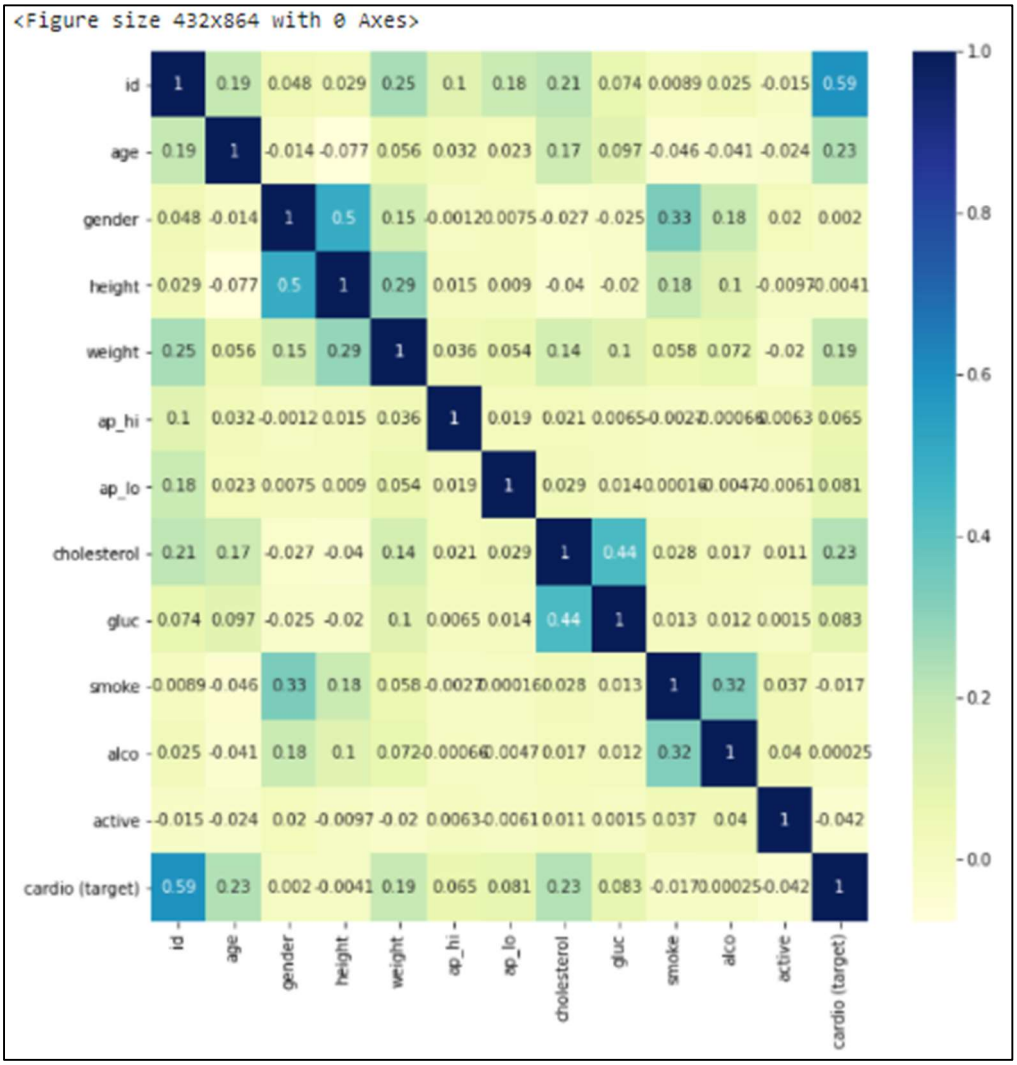

Figure 2.2.2: Correlation Coefficient Heatmap (Raw Dataset)

```
#calculate the correlation between variables using heatmap
corr = df.corr()
plt.figure(figsize = (8, 8))
sns.heatmap(df.corr(),annot = True, cmap = 'YlGnBu');
plt.figure(figsize = (6, 12))
```

Figure 2.2.3: Code Snippet of Correlation (Cleaned Dataset)

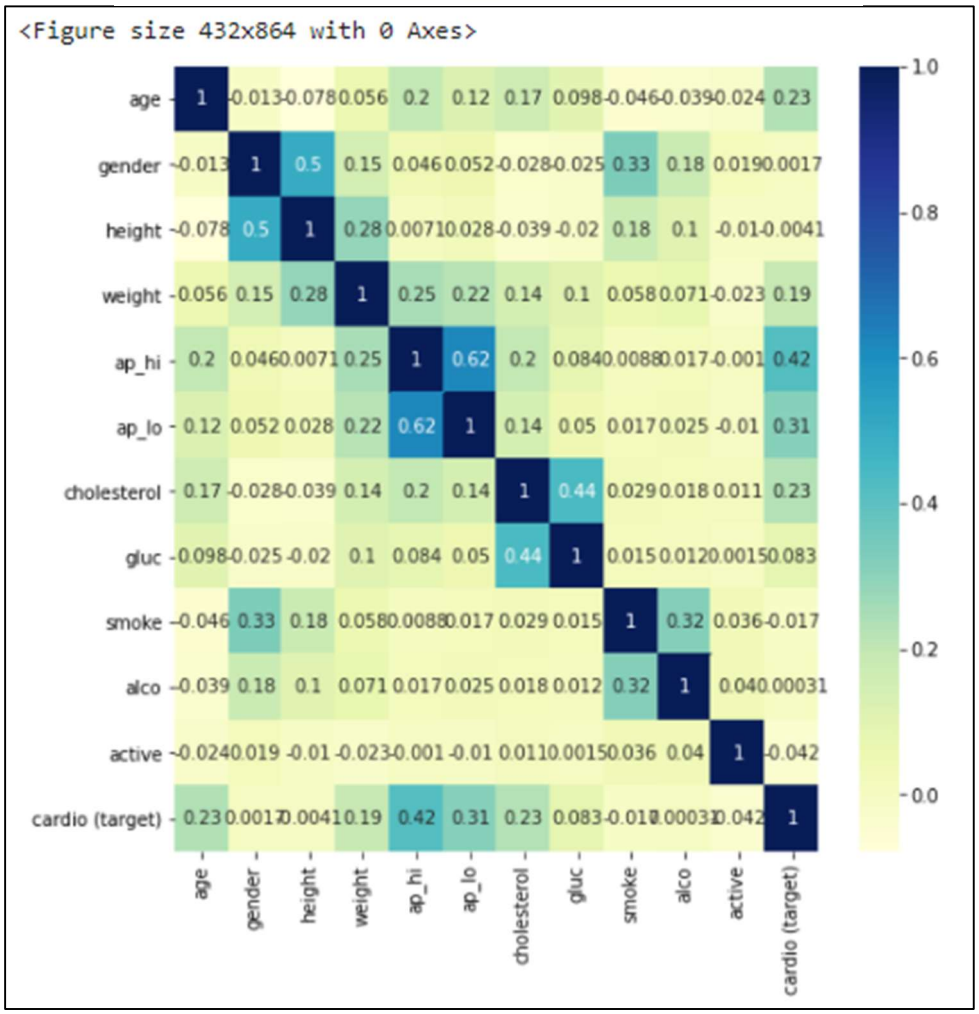

Figure 2.2.4: Correlation Coefficient Heatmap (Cleaned Dataset)

No	Attributes	Raw Dataset	Cleaned Dataset
1	cardio (target)	**1.00000**	**1.00000**
2	Id	**0.59000**	-
3	Age	**0.23000**	**0.23000**
4	Cholesterol	**0.23000**	**0.23000**
5	Weight	**0.19000**	**0.19000**

6	Gluc	0.08300	0.08300
7	ap_lo	0.08100	**0.31000**
8	ap_hi	0.06500	**0.42000**
9	Gender	0.02000	0.00170
10	Alco	0.00025	0.00031
11	Height	-0.00410	-0.00410
12	Smoke	-0.01700	-0.01700
13	Active	-0.04200	-0.04200

Table 2.2.1: Comparison Table for Correlation of Raw Dataset and Cleaned Dataset

As shown in Table 2.2.1, the result of correlation coefficient for the raw dataset includes an extra attribute 'id' which has been removed from the cleaned dataset as it does not provide any useful information for modelling. Based on the correlation coefficient results, the researcher decided to select the features with a threshold value higher than 0.18. The result of the features selection is slightly different due to an extra attribute in the raw dataset. Both the ap_lo and ap_hi attributes in the cleaned dataset possess high correlation with the target label but have low correlation value in the raw dataset. The attributes 'age', 'weight' and 'cholesterol' possess same high correlation value for the raw and cleaned dataset. The researcher adopts the selected independent variables from the cleaned dataset ultimately as the selected attribute 'id' in the raw dataset is a not important variable that provides no information to the machine learning modelling.

Based on the comparison above, it also further proved the importance of the data cleaning stage in data analysis projects as a redundant attribute would highly affect the features that would select to use in model training. The data is then split into train and test data before proceeding to model building using code in Figure 2.2.5 below.

```
x = data.loc[:,['ap_hi','ap_lo','age','cholesterol','weight']].values
y = data.loc[:,['cardio (target)']].values

x_train, x_test, y_train, y_test = train_test_split(x,y,test_size = 0.2)
```

Figure 2.2.5: Code snippet to split data into train and test data sets

2.3 Model Construction and Tunning

2.3.1 Artificial Neural Network (ANN) – Chan Jia Le (TP049952)

Meta Parameter Selection

During this section, the listing and explanation of each parameter and hyperparameters that are used to construct the Artificial Neural Network will be stated. The data that is used for constructing the Artificial Neural Network model will be from the feature selection section. The Artificial Neural Network is consisted of several parameters and hyperparameters as stated below:

Parameters	Description
Input Dimension	the input layer of the Artificial Neural Network, the value of input dimension will be the shape of X value. (Didugu, 2020)
Hidden Layers	The added hidden layers of Artificial Neural Network with a specified number of neurons. (Gad, 2018)
Neurons	Neurons or nodes are the processing units that mimics the human brain for the Artificial Neural Network. (Gad, 2018)
Activation	The activation function or transfer function is a functionality that buffers the data before being fed to the next layer. (Sharma, 2017)
Optimizer	Optimizers are ways or algorithms utilized to change the attributes of the Artificial Neural network to reduce losses. (Kumar, 2020)
Learning Rate	Learning rate is a hyperparameter that controls the amount of changes the model has in response to the estimated error when model weights are updated. (Brownlee, 2019)
Batch Size	Batch size is another hyperparameter that defined the number of samples to operate through before updating the model parameters. (Brownlee, 2019)
Epoch	Epoch is yet another hyperparameter that interprets the number of times the learning algorithm will operate through the training dataset (Brownlee, 2019)
Dropout Regularization	Dropout on neural network is regularization technique that randomly selects neurons to be ignored during training. (McNealis, 2020)

Table 2.3.1.1 Description of parameter and hyperparameters

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

#define the model
model = Sequential()


model.add(Dense(64, input_dim=5, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

#model creation
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

#model fitting
model.fit(X_train, y_train, epochs=10, batch_size=20,verbose=0) #verbose is to not
loss, acc = model.evaluate(X_test, y_test, verbose=0)
train_loss, train_acc = model.evaluate(X_train,y_train, verbose=0)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % acc)
```

Figure 2.3.1.2 Initial construction of Artificial Neural Network with predefined parameter and hyper parameter

```
Train Accuracy: 0.665
Test Accuracy: 0.655
```

Figure 2.3.1.3 Initial train and test accuracy of the constructed Artificial Neural Network

```
Model: "sequential_107"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_330 (Dense)            (None, 64)                384
_________________________________________________________________
dense_331 (Dense)            (None, 32)                2080
_________________________________________________________________
dense_332 (Dense)            (None, 1)                 33
=================================================================
Total params: 2,497
Trainable params: 2,497
Non-trainable params: 0
```

Figure 2.3.1.4 Model Summary

Figure 2.3.1.2 shows the initially constructed Artificial Neural Network, the first layer will be consisted of an input layer with 5 nodes and a hidden layer of 64 nodes. The second hidden layer will be consisted of 32 nodes and both activation function is relu. Furthermore, the output layer will only be 1 as it is binary classification and activation will be sigmoid due to the binary classification nature. The model will then be compiled with a binary cross entropy loss as it is binary classification, the Adam optimizer will be used for the ANN. Additionally, the model will fit through an epoch of 10 with batch size of 20 with training data X and y from the data splitting. The accuracy is as shown in the figure 2.3.1.3 which is 66.5% in train accuracy and 65.5% in test accuracy. The model summary is as shown in figure 2.3.1.4.

Model Tuning and Hyperparameter Tuning (Grid Search)

The initial constructed model only has an accuracy of 65.5%, which is not the best accuracy. Hence, the model will be gone through a series of model tunning and hyperparameter tunning with the utilization of grid search using sklearn's GridSearchCV.

Epoch & Batch Size Tunning

```
# Optimization of deep neural network
# identify best batch and epoch

import numpy
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

def create_model():
    model = Sequential()
    model.add(Dense(64, input_dim=5, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

#fix random seed for reproductivity
seed = 7
numpy.random.seed(seed)

model = KerasClassifier(build_fn=create_model,verbose=0)

batch_size = [10,20,40,60,80,100]
epochs = [10,50,100,150]
param_grid = dict(batch_size= batch_size,epochs=epochs)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)

print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Figure 2.3.1.5 Hyperparameter Tuning using Grid Search (Epoch & Batch Size)

```
Best: 0.729309 using {'batch_size': 10, 'epochs': 150}
0.612202 (0.056999) with: {'batch_size': 10, 'epochs': 10}
0.703836 (0.015841) with: {'batch_size': 10, 'epochs': 50}
0.701972 (0.032916) with: {'batch_size': 10, 'epochs': 100}
0.729309 (0.014596) with: {'batch_size': 10, 'epochs': 150}
0.568408 (0.025966) with: {'batch_size': 20, 'epochs': 10}
0.693745 (0.019418) with: {'batch_size': 20, 'epochs': 50}
0.720921 (0.015588) with: {'batch_size': 20, 'epochs': 100}
0.727755 (0.011367) with: {'batch_size': 20, 'epochs': 150}
0.597618 (0.047483) with: {'batch_size': 40, 'epochs': 10}
0.635805 (0.040102) with: {'batch_size': 40, 'epochs': 50}
0.654288 (0.051169) with: {'batch_size': 40, 'epochs': 100}
0.676658 (0.000962) with: {'batch_size': 40, 'epochs': 150}
0.580832 (0.046862) with: {'batch_size': 60, 'epochs': 10}
0.682715 (0.004558) with: {'batch_size': 60, 'epochs': 50}
0.679774 (0.046176) with: {'batch_size': 60, 'epochs': 100}
0.688466 (0.024305) with: {'batch_size': 60, 'epochs': 150}
0.609410 (0.024638) with: {'batch_size': 80, 'epochs': 10}
0.621212 (0.044072) with: {'batch_size': 80, 'epochs': 50}
0.660043 (0.028189) with: {'batch_size': 80, 'epochs': 100}
0.702129 (0.006753) with: {'batch_size': 80, 'epochs': 150}
0.579912 (0.036210) with: {'batch_size': 100, 'epochs': 10}
0.630994 (0.040586) with: {'batch_size': 100, 'epochs': 50}
0.695451 (0.011752) with: {'batch_size': 100, 'epochs': 100}
0.669529 (0.071059) with: {'batch_size': 100, 'epochs': 150}
```

Figure 2.3.1.6 Output of the Grid Search

Iteration	Accuracy	Iteration	Accuracy	Iteration	Accuracy
1	0.612202	9	0.597618	17	0.609410
2	0.703836	10	0.635805	18	0.621212
3	0.701972	11	0.654288	19	0.660043
4	**0.729309**	12	0.676658	20	0.702129
5	0.568408	13	0.580832	21	0.579912
6	0.693745	14	0.682715	22	0.630994
7	0.720921	15	0.679774	23	0.695451
8	0.727755	16	0.688466	24	0.669529

Results: Upon conducting grid search through the epoch and batch size hyperparameters. It is found that the best hyperparameter will be batch size = 10 and epochs = 150. The dimen sion of the data is 8049 rows x 6 columns. The 8049 will be divided by 10 into 805 batches and further runs an epoch of 150 which concludes to 805 x 150 = 120750 during the entire t raining process. The train accuracy has also been enhanced significantly, increasing from 6 6.5% to 72.9%.

Optimizer Tunning

```
import numpy
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

def create_model(optimizer='adam'):
    model = Sequential()
    model.add(Dense(64, input_dim=5, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# define the grid search parameters
optimizer = ['SGD', 'Adagrad', 'Adam']
param_grid = dict(optimizer=optimizer)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Figure 2.3.1.7 Model Tuning using Grid Search (Optimizer)

```
Best: 0.730861 using {'optimizer': 'Adam'}
0.714397 (0.005641) with: {'optimizer': 'SGD'}
0.727601 (0.018128) with: {'optimizer': 'Adagrad'}
0.730861 (0.013690) with: {'optimizer': 'Adam'}
```

Figure 2.3.1.8 Output of the Grid Search

Optimizer	Accuracy
Adam	**0.730861**
SGD	0.714397
Adagrad	0.727601

Results: The second optimization will be conducted on tunning the optimizer through grid search. The model will utilize the epoch of 150 and batch size of 10 deriving from the previous results. Furthermore, the grid search will transverse through the SGD, Adagrad and Adam optimizer to seek the best performer which is Adam optimizer with 73.1% accuracy.

Learning Rate Tunning

```
def create_model(learn_rate=0.01):
    model = Sequential()
    model.add(Dense(64, input_dim=5, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    optimizer = optimizers.Adam(lr=learn_rate)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

seed = 7
numpy.random.seed(seed)
# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# define the grid search parameters
learn_rate = [0.0001,0.0002,0.0003,0.001,0.002,0.003, 0.01, 0.1, 0.2, 0.3]
param_grid = dict(learn_rate=learn_rate)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Figure 2.3.1.9 Hyperparameter Tuning using Grid Search (Learning Rate)

```
Best: 0.726823 using {'learn_rate': 0.001}
0.679451 (0.029004) with: {'learn_rate': 0.0001}
0.686601 (0.018640) with: {'learn_rate': 0.0002}
0.703684 (0.015505) with: {'learn_rate': 0.0003}
0.726823 (0.008101) with: {'learn_rate': 0.001}
0.726667 (0.009136) with: {'learn_rate': 0.002}
0.696847 (0.041666) with: {'learn_rate': 0.003}
0.580834 (0.073755) with: {'learn_rate': 0.01}
0.496972 (0.014081) with: {'learn_rate': 0.1}
0.501321 (0.014343) with: {'learn_rate': 0.2}
0.503028 (0.014081) with: {'learn_rate': 0.3}
```

Figure 2.3.1.10 Output of the Grid Search

Learn Rate	Accuracy	Learn Rate	Accuracy
0.0001	0.679451	0.003	0.696847
0.0002	0.686601	0.01	0.580834
0.0003	0.703684	0.1	0.496972
0.001	**0.726823**	0.2	0.501321
0.002	0.726667	0.3	0.503028

Results: Upon the identification of the best optimizer for the research, which is the Adam o ptimizer, the researcher proceeds to perform hyperparameter tunning on the learning rate in the optimizer. different learning rate is initialized to perform the grid search. Upon complet

ion, the best resulting learning rate is 0.001 and the train accuracy is now 72.7% and has sli ghtly decreased. However, the researcher has decided to experiment further performing oth er tunning to examine the results of the model.

Weight Initialization Tunning

```
def create_model(init_mode='uniform'):
    # create model
    model = Sequential()
    model.add(Dense(64, input_dim=5,kernel_initializer=init_mode, activation='relu'))
    model.add(Dense(32,kernel_initializer=init_mode, activation='relu'))
    model.add(Dense(1,kernel_initializer=init_mode, activation='sigmoid'))
    optimizer = optimizers.Adam(lr=0.001)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

seed = 7
numpy.random.seed(seed)
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# define the grid search parameters
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_unifo
param_grid = dict(init_mode=init_mode)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Figure 2.3.1.11 Model Tuning using Grid Search (Weight Initialization)

```
Best: 0.729307 using {'init_mode': 'uniform'}
0.729307 (0.009045) with: {'init_mode': 'uniform'}
0.714865 (0.013321) with: {'init_mode': 'lecun_uniform'}
0.727600 (0.013082) with: {'init_mode': 'normal'}
0.512348 (0.007415) with: {'init_mode': 'zero'}
0.724493 (0.009810) with: {'init_mode': 'glorot_normal'}
0.720609 (0.012746) with: {'init_mode': 'glorot_uniform'}
0.714709 (0.006748) with: {'init_mode': 'he_normal'}
0.697623 (0.034521) with: {'init_mode': 'he_uniform'}
```

Figure 2.3.1.12 Output of the Grid Search

Weight Initialization	Accuracy	Weight Initialization	Accuracy
Uniform	**0.729307**	Glorot_normal	0.724493
Lecun_uniform	0.714865	Glorot_uniform	0.720609
Normal	0.727600	He_normal	0.714709
Zero	0.512348	He_uniform	0.697623

Results: The model is modified according to the best learning rate that is identified along with the best classifier, which is 0.001 and Adam optimizer, respectively. After that, the tunning will be performed on the weight initializer. Each weight initializer is defined before performing the grid search, and the transverse will begin to search the best param which in this case is the uniform weight initialization. However, the computation time has increased, and the accuracy has not improved, further investigations will be made to see which model tunning or hyperparameter tunning is necessary.

Dropout Rate and Weight Constraint Tunning

```
def create_model(dropout_rate=0.0, weight_constraint=0):
    # create model
    model = Sequential()
    model.add(Dropout(dropout_rate))
    model.add(Dense(64, input_dim=5, kernel_initializer='uniform', activation='relu', kernel
    model.add(Dropout(dropout_rate))
    model.add(Dense(32, kernel_initializer='uniform', activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
    optimizer = optimizers.Adam(learning_rate=0.001)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)

# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# define the grid search parameters
weight_constraint = [1, 2, 3, 4, 5]
dropout_rate = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
param_grid = dict(dropout_rate=dropout_rate, weight_constraint=weight_constraint)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Figure 2.3.1.13 Model Tuning using Grid Search (Dropout & Weight Constraints)

```
Best: 0.729308 using {'dropout_rate': 0.0, 'weight_constraint': 1}
0.729308 (0.010177) with: {'dropout_rate': 0.0, 'weight_constraint': 1}
0.719056 (0.009039) with: {'dropout_rate': 0.0, 'weight_constraint': 2}
0.718747 (0.007569) with: {'dropout_rate': 0.0, 'weight_constraint': 3}
0.723253 (0.021623) with: {'dropout_rate': 0.0, 'weight_constraint': 4}
0.727445 (0.014217) with: {'dropout_rate': 0.0, 'weight_constraint': 5}
0.574472 (0.026772) with: {'dropout_rate': 0.1, 'weight_constraint': 1}
0.542477 (0.029378) with: {'dropout_rate': 0.1, 'weight_constraint': 2}
0.540300 (0.011363) with: {'dropout_rate': 0.1, 'weight_constraint': 3}
0.541702 (0.015151) with: {'dropout_rate': 0.1, 'weight_constraint': 4}
0.528187 (0.012160) with: {'dropout_rate': 0.1, 'weight_constraint': 5}
0.487807 (0.006871) with: {'dropout_rate': 0.2, 'weight_constraint': 1}
0.523688 (0.047529) with: {'dropout_rate': 0.2, 'weight_constraint': 2}
0.488584 (0.005438) with: {'dropout_rate': 0.2, 'weight_constraint': 3}
0.490914 (0.005634) with: {'dropout_rate': 0.2, 'weight_constraint': 4}
0.489360 (0.007325) with: {'dropout_rate': 0.2, 'weight_constraint': 5}
0.487652 (0.007415) with: {'dropout_rate': 0.3, 'weight_constraint': 1}
0.487652 (0.007415) with: {'dropout_rate': 0.3, 'weight_constraint': 2}
0.487652 (0.007415) with: {'dropout_rate': 0.3, 'weight_constraint': 3}
0.487652 (0.007415) with: {'dropout_rate': 0.3, 'weight_constraint': 4}
0.487652 (0.007415) with: {'dropout_rate': 0.3, 'weight_constraint': 5}
```

Figure 2.3.1.14 Grid Search Output

Iteration	Accuracy	Iteration	Accuracy	Iteration	Accuracy
1	**0.729308**	8	0.540300	15	0.489360
2	0.719056	9	0.541702	16	0.489360
3	0.718747	10	0.528187	17	0.489360
4	0.723253	11	0.487807	18	0.489360
5	0.727445	12	0.523688	19	0.489360
6	0.574472	13	0.488584	……..	
7	0.542477	14	0.490914		

Results: Like the other processes, the model will proceed to inherit the best weight initialization algorithm and perform grid search on the new parameter. The model tunning for this phase will be dropout rate and weight constraint. Dropout rate is used to prevent overfitting. However, the grid search results indicate that the model performs best without any dropout regularization and with only 1 in weight constraint. Additionally, the best training accuracy which is 72.9% has no significant improvements.

Model Evaluation

The evaluation metrics chosen to measure the model performance will be the test scoring metrics and the precision, recall and F1 score. As mentioned in the above statements, some of the model or hyperparameter tuning has reflected slight decrease in accuracy. However, the researcher has decided to reconstruct 2 of the models to evaluate which tunning performed the best. The first constructed model, model 1 will only utilize the hyperparameter tuning of epoch size of 150 and batch size 10 along with the Adam optimizer as shown in figure 2.3.1.15. The other reconstructed model, model 2 will consist of all the tunning conducted including weight initialization, learning rate, etc. as reflected in figure 2.3.1.16.

```
model = Sequential()
model.add(Dense(64, input_dim=5, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])


model.fit(X_train, y_train, epochs=150, batch_size=10,verbose=0) #verbose is to no
y_pred = model.predict(X_test)
loss, acc = model.evaluate(X_test, y_test, verbose=0)
train_loss, train_acc = model.evaluate(X_train,y_train, verbose=0)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % acc)
print('Accuracy Score : ' + str(accuracy_score(y_test,y_pred.round())))
print('Precision Score : ' + str(precision_score(y_test,y_pred.round())))
print('Recall Score : ' + str(recall_score(y_test,y_pred.round())))
print('F1 Score : ' + str(f1_score(y_test,y_pred.round())))
from sklearn.metrics import confusion_matrix

cm = metrics.confusion_matrix(y_test,y_pred.round())
print(cm)
```

Figure 2.3.1.15 Model 1 with only epoch, batch size and optimizer configuration

```
model = Sequential()

model.add(Dense(64, input_dim=5, kernel_initializer='uniform', activation='relu', kerne
model.add(Dense(32, kernel_initializer='uniform', activation='relu'))
model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
optimizer = optimizers.Adam(learning_rate=0.001)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

#model fitting
model.fit(X_train, y_train, epochs=150, batch_size=10,verbose=0) #verbose is to not di
y_pred = model.predict(X_test)
loss, acc = model.evaluate(X_test, y_test, verbose=0)
train_loss, train_acc = model.evaluate(X_train,y_train, verbose=0)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % acc)
print('Accuracy Score : ' + str(accuracy_score(y_test,y_pred.round())))
print('Precision Score : ' + str(precision_score(y_test,y_pred.round())))
print('Recall Score : ' + str(recall_score(y_test,y_pred.round())))
print('F1 Score : ' + str(f1_score(y_test,y_pred.round())))
from sklearn.metrics import confusion_matrix

cm = metrics.confusion_matrix(y_test,y_pred.round())
print(cm)
```

Figure 2.3.1.16 Model 2 with all the optimized parameter and hyperparameter

```
Train Accuracy: 0.733
Test Accuracy: 0.712
Accuracy Score : 0.7118012422360248
Precision Score : 0.7271573604060914
Recall Score : 0.6970802919708029
F1 Score : 0.7118012422360249
[[573 215]
 [249 573]]
```

```
Train Accuracy: 0.731
Test Accuracy: 0.712
Accuracy Score : 0.7124223602484472
Precision Score : 0.760522496371553
Recall Score : 0.6374695863746959
F1 Score : 0.6935804103242885
[[623 165]
 [298 524]]
```

Figure 2.1.17 Results of Execution Left (Model 1) & Right (Model 2)

	Model 1	**Model 2**
Train Accuracy	0.733	0.731
Test Accuracy	0.712	0.712
Precision	0.727	0.761
Recall	0.697	0.637
F1 Score	0.712	0.694

Table 2.1.18 Results Comparison Table

The result of the model reconstruction is as shown in the table above. Both models performed similar, the model 1 performed slightly better on training accuracy leading an additional 0.002. The test accuracy, which is often label as model accuracy is the same between the 2 models. However, the precision, recall and f1 score reflects differently. Precision is the ratio between the True positive and the sum of true and negative positives, in this scenario, model 1 is correct about patient having cardiovascular disease around 72.7% of the time while model 2 has around 76.1%. Recall on the other hand is the measure of the model correctly identifying True Positives, it is also named as True Positive Rate (Huilgol, 2020). The model 1 has 69.7% in True Positive Rate and model 2 has a significantly lower True Positive rate of 63.7%. Because the models have ups and downs in different areas and the accuracy of the models is similar, the F1 score has become the most crucial evaluation metric. The F1 score is a metric used to seek balance between Precision and Recall and is a better measurement for seeking balance between Precision and Recall and is often the better metric in most real-life classifications (Huigol, 2019). The model 1 possesses higher F1 score while model 2 possesses a slightly lower score, hence reflecting that model 1 gives a better measure of those incorrectly classified cases. Hence, the Model 1 which is the model with optimization on epoch, batch size and Adam optimizer performed better in this research.

Summary

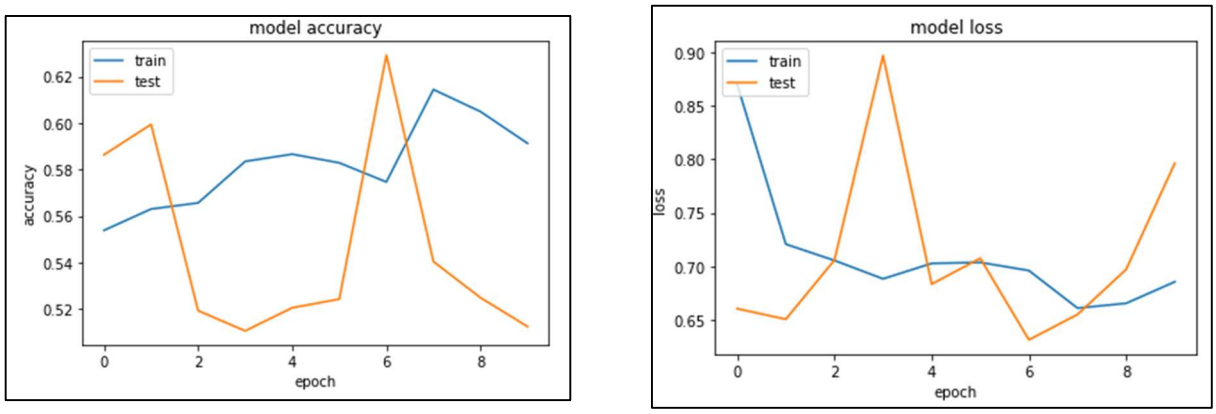

Figure 2.3.1.19 The Model Accuracy and Model Lost of the Initial Model

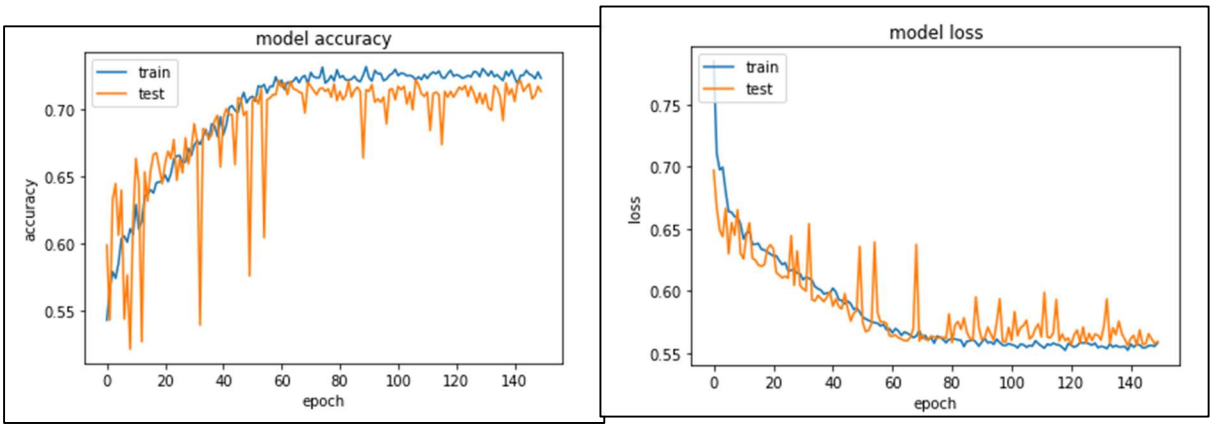

Figure 2.3.1.20 The Model Accuracy and Model Lost of the selected Model (Model 1)

The figure 2.3.1.19 shows the plot for the model accuracy and model loss for the initial model construction and the figure 2.3.1.20 reflects the results of the selected model. Summing up the results, initially when the model was constructed, the train and accuracy of the model is only 66.5% and 65.5%. After model tunning and hyperparameter tuning, the researcher has stumble upon 2 similar performing tunned models and has concluded that the model with the tunned hyperparameter epoch, batch size and optimizer performed the best. The model that is selected has train accuracy of 73.3% and test accuracy of 71.2%, which is a 5 to 7% increase in performance. Hence, the model tunning and hyperparameter tuning has significantly enhanced the result of the ANN model.

2.3.2 Logistic Regression – Ragneshwary (TP053143)

Meta Parameter Selection

The Logistic Regression have been constructed from the dataset that have gone through the process of exploratory data analysis and feature selection. The X value of the model is “ap_hi”, “ap_lo”, “age”, “cholesterol” and “weight” and the target or Y value is “cardio”. This feature has been selected as it has achieved the minimum threshold value that have been set. Logistic Regression does not have any critical hyperparameters to be tuned. The difference can be seen in performance with penalty, solver, or C. Below are the most familiar parameter for Logistic Regression:

Meta Parameter	Description	Example
Regularization (Penalty)	It helps by penalizing the regression coefficients that are high valued. It helps to avoid overfitting.	Penalty in [‘none’, ‘l1’, ‘l2’, ‘elasticnet’]
Solver	It helps to decide which algorithm that can be used to optimize the problem of the model.	Solver in [‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’]
C	It is the inverse strength of the regularization which usually be a positive float. Smaller values of C represent the stronger regularization. In default it is 1.0.	C in [100, 10, 1.0, 0.1, 0.01]

Table 2.3.2.1 Meta Parameter of Logistic Regression

```
#data partition
from sklearn.model_selection import train_test_split
X = dataset.loc[:,['ap_hi', 'ap_lo', 'age', 'cholesterol', 'weight']].values
y = dataset.loc[:,['cardio (target)']].values

#split data into train set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Figure 2.3.2.1 Base model of Logistic Regression data partition and data split

```
from sklearn.linear_model import LogisticRegression
classifier  = LogisticRegression(random_state=42)
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
Y_pred = classifier.predict(X_train)

from sklearn.metrics import confusion_matrix
cm_test = confusion_matrix(y_pred, y_test)
cm_test1 = confusion_matrix(Y_pred, y_train)

y_pred_train = classifier.predict(X_train)
Y_pred_train = classifier.predict(X_test)
cm_train = confusion_matrix(y_pred_train, y_train)
```

Figure 2.3.2.2 Base model of Logistic Regression

```
print("Classficiation report of test\n")
print(classification_report(y_test, y_pred))

print("Classficiation report of train\n")
print(classification_report(y_train, Y_pred))

output = classifier.score(X_test, y_test)
output1 = classifier.score(X_train,y_train)

print("Score for test:  ""{:.2f}".format(output*100),"%")
print("Score for train: ""{:.2f}".format(output1*100),"%")
```

Figure 2.3.2.3 Base model of Logistic Regression scores

The basic Logistic Regression was constructed starting by defining the X and y columns. Then the test and train set has been split with no random state. The Sklearn function of Logistic Regression with random state of 42 is defined and the model is fit. The target value for train and test is predicted. Classification report and score for train and test data is printed.

```
Score for test:  71.18 %
Score for train: 72.84 %
```

Figure 2.3.2.4 Base model of Logistic Regression score for test and train

By looking at the score of both test and train, it can be seen there is a significant difference between both where train is 72.84% and test is 71.18%. This reflects that there is a difference of 1.66% which can be considered as a error rate. The difference is not a lot and it is still acceptable. As the average score of the model is around 72%, the model is tuned to increase the score of models.

Hyperparameters Tuning using Grid Search

```
#grid serach
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
# define evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=42)
# define search space
space = dict()
space['solver'] = ['newton-cg', 'lbfgs', 'liblinear']
space['penalty'] = ['none', 'l1', 'l2', 'elasticnet']
space['C'] = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100]
# define search
search = GridSearchCV(classifier, space, scoring='accuracy', n_jobs=5, cv=cv)
# execute search
result = search.fit(X_train, y_train)
# summarize result
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)
```

Figure 2.3.2.5 Model tuning using Grid Search

The model that was trained before is using the default parameters that was determined by the Logistic Regression module. Even though the error rate is not high, the overall accuracy still can be improved. The model is improved by tuning the hyperparameters. Grid of parameters have been defined to find the best model and this is done by GridSearchCV and RepeatedStratifiedKFold that is used for evaluation. The dictionary does include all three parameters of Logistic Regression as stated in the table above. The best score and the best hyperparameters have been printed.

```
Best Score: 0.7294610859422546
Best Hyperparameters: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
```

Figure 2.3.2.6 Model tuning output

The best model's parameters would be as 'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'. It has the best score of 0.7295 which can be considered as 73%.

```
from sklearn.linear_model import LogisticRegression
classifier  = LogisticRegression(random_state=42,C=0.1, penalty= 'l1', solver= 'liblinear')
classifier.fit(X_train, y_train)


# Predicting the Test set results
y_pred = classifier.predict(X_test)
Y_pred = classifier.predict(X_train)

from sklearn.metrics import confusion_matrix
cm_test = confusion_matrix(y_pred, y_test)
cm_test1 = confusion_matrix(Y_pred, y_train)

y_pred_train = classifier.predict(X_train)
Y_pred_train = classifier.predict(X_test)
cm_train = confusion_matrix(y_pred_train, y_train)
```

Figure 2.3.2.7 Logistic Regression with model tuning

```
print("Classficiation report of test\n")
print(classification_report(y_test, y_pred))

print("Classficiation report of train\n")
print(classification_report(y_train, Y_pred))

output = classifier.score(X_test, y_test)
output1 = classifier.score(X_train,y_train)

print("Score for test:  ""{:.2f}".format(output*100),"%")
print("Score for train: ""{:.2f}".format(output1*100),"%")
```

Figure 2.3.2.8 Logistic Regression score

```
Score for test after model tuning:  71.74 %
Score for train after model tuning: 73.01 %
```

Figure 2.3.2.9 Logistic Regression score after model tuning for test and train

The base Logistic Regression model have been tuned by modifying the hyper parameters that was found using the grid search and the hyperparameters have been defined to train the model. The score for train set is 73.01% and test set is 71.74%. The comparison of model before and after model tuning is compared in the next section.

Model Evaluation

Train Set

Classficiation report of train

	precision	recall	f1-score	support
0.0	0.70	0.77	0.74	3151
1.0	0.76	0.68	0.72	3288
accuracy			0.73	6439
macro avg	0.73	0.73	0.73	6439
weighted avg	0.73	0.73	0.73	6439

Figure 2.3.2.10 Classification report for train set before model tuning

```
Score for train: 72.84 %
```

Figure 2.3.2.11 Train score before model tuning

Classficiation report of train after model tuning

	precision	recall	f1-score	support
0.0	0.70	0.78	0.74	3151
1.0	0.76	0.68	0.72	3288
accuracy			0.73	6439
macro avg	0.73	0.73	0.73	6439
weighted avg	0.73	0.73	0.73	6439

Figure 2.3.2.12 Classification report for train set after model tuning

```
Score for train after model tuning: 73.01 %
```

Figure 2.3.2.13 Train score after model tuning

Train Set	Accuracy	Precision	Recall	F1-Score
Before model tuning	0.73	0.70	0.77	0.74
After model tuning	0.73	0.70	0.78	0.74

Table 2.3.2.2 Summary of classification report for train set

As can be seen above the train set score before model tuning is 72.84% and after 73.01%. The improvement of the score is not a lot but there is a difference of 0.17% which can be considered as the score is almost similar. In terms of Precision, both train set before and after model tuning has the same core of 0.70 which means only 70% is the accuracy of positive predictions. As for Recall, surprisingly there is a increase from 0.77 to 0.78 which shows the fraction of instances that are positive that was correctly identified. The difference is not huge, due to that it can be considered similar before and after model tuning. The F1 score reflects the mean of recall and precision which 1.0 is considered the best F1 Score. The train set before and after model tune have the value of 0.74 which is considered more efficient than average score of F1 score. The train set before and after model tune does not have any significant difference.

Test Set

Classficiation report of test

	precision	recall	f1-score	support
0.0	0.68	0.77	0.72	777
1.0	0.75	0.66	0.70	833
accuracy			0.71	1610
macro avg	0.72	0.71	0.71	1610
weighted avg	0.72	0.71	0.71	1610

Figure 2.3.2.14 Classification report for test set before model tuning

```
Score for test:  71.18 %
```

Figure 2.3.2.15 Test score before model tuning

Classficiation report of test after model tuning

	precision	recall	f1-score	support
0.0	0.68	0.78	0.73	777
1.0	0.76	0.66	0.71	833
accuracy			0.72	1610
macro avg	0.72	0.72	0.72	1610
weighted avg	0.72	0.72	0.72	1610

Figure 2.3.2.16 Classification report for test set after model tuning

```
Score for test after model tuning:  71.74 %
```

Figure 2.3.2.17 Test score after model tuning

Test Set	Accuracy	Precision	Recall	F1-Score
Before model tuning	0.71	0.68	0.77	0.72
After model tuning	0.72	0.68	0.78	0.73

Table 2.3.2.3 Summary of classification report for test set

As for test set, surprisingly there is a increase in the accuracy after model tuning which is before model tuning the accuracy is 71.18% and after model tuning is 71.74%, there is a difference of 0.56% which is not a huge gap. In terms of Precision, both test set before and after model tuning has 0.68 which is the accuracy of positive predictions. As for Recall, surprisingly there is a increase from 0.77 to 0.78 which shows the fraction of instances that are positive that was correctly identified. The difference is not huge, due to that it can be considered similar before and after model tuning. The F1 score reflects the mean of recall and precision which 1.0 is considered the best F1 Score. The test set before model tune have 0.72 and after model tune have the value of 0.73 which is both considered almost similar and it is more efficient than average score of F1 score.

Confusion Matrix

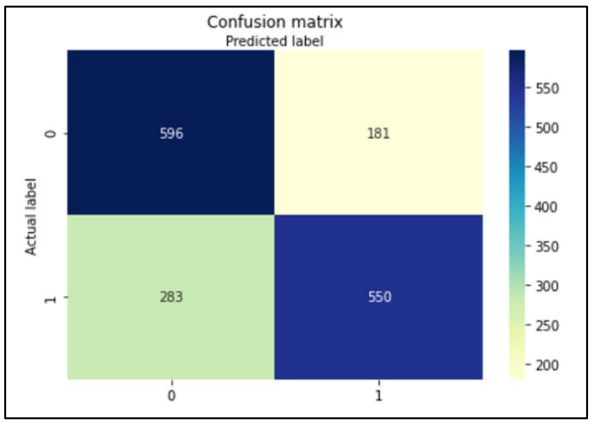

Figure 2.3.2.18 Confusion Matrix before model tuning

	Negative	Positive
Negative	TN (596)	FP (181)
Positive	FN (283)	TP (550)

Table 2.3.2.4 Confusion Matrix before model tuning

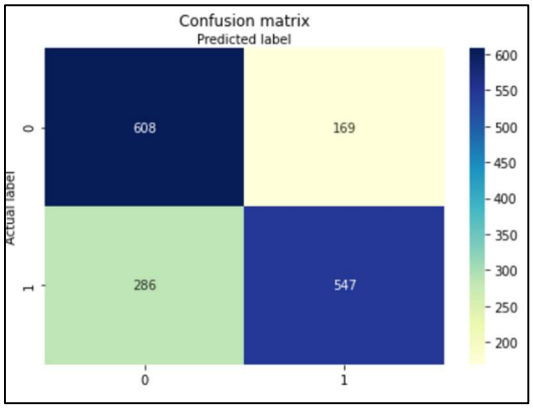

Figure 2.3.2.19 Confusion Matrix before model tuning

	Negative	Positive
Negative	TN (608)	FP (169)
Positive	FN (286)	TP (547)

Table 2.3.2.5 Confusion Matrix before model tuning

As for per the confusion matrix before and after model tunning for Logistic Regression, it can be seen there is a significant increase in the true negative which proves that the prediction of people not having cardiovascular disease is predicted correctly. There is also a slight drop in true positive where the prediction of people having cardiovascular disease is predicted correctly has dropped.

In conclusion, the better fit for the dataset is the Logistic Regression after model tuning. This is because there is a significant difference and improvement to the model, maybe not from accuracy the difference is small, but there is a difference when comparing the classification report and the confusion matrix. The model is able to predict much better after model is tuned.

2.3.3 K-Nearest Neighbor (K-NN) Classifier – Thong Kyn Hui (TP053489)

Meta Parameter Selection

During this section, the listing and explanation of each parameter and hyperparameters that are used to construct the K-Nearest Neighbor (KNN) Classification model will be stated. The KNN algorithm consists of several parameters as stated below:

Parameters	Description
N_Neighbors	The total number of neighbors used to classify the dataset into several groups. Each groups share similar characteristic and distinct from other groups. Default value is "5". (Fraj, 2017)
Algorithm	The algorithm used to construct the nearest neighbors. Example of algorithms available are Ball Tree, KD Tree, Brute-Force Search, or "auto" which decide the most appropriate algorithm based on the dataset. (Martulandi, 2019)
Leaf Size	Leaf size of Ball Tree and KD Tree which affect the speed of construction and memory required to store the tree. Default value is "30".
Metric	The distance metric used to construct the tree. The default metric is "minkowski" with p value "2" which is equivalent for standard Euclidean metric. (P.Jones, 2020)
Metric_Params	Additional keyword arguments for the tree construction. Default value is "none".
N_Jobs	The number of parallel jobs running concurrently to construct the tree. Default value is "none".
Weights	The weight function which is used for prediction. This value could be uniform, distance or callable. Uniform indicates that all points in each neighbor are weighted equally and distance weight points by considering the distance.

Table 2.3.3.1 Parameter List of K-Nearest Neighbor (KNN) Classifier (ScikitLearn, 2018)

Initially, the KNN classifier model is imported from the sklearn library. Then, dataset is passed to the KNN classifier model for training and testing. The accuracy of both training and testing set is computed without any hyperparameter and model tuning. The accuracy is shown in figure below.

```
[8] from sklearn import neighbors
    knn = neighbors.KNeighborsClassifier()
    knn.fit(X_train, y_train)
    print("Accuracy of training set is ", knn.score(X_train, y_train))
    print("Accuracy of testing set is ", knn.score(X_test, y_test))
    knn=knn.predict(X_test)

    Accuracy of training set is  0.7746544494486721
    Accuracy of testing set is  0.6956521739130435
```

```
[11] from sklearn.metrics import confusion_matrix, classification_report,
       plot_roc_curve,plot_precision_recall_curve
     print(classification_report(y_test,knn))

                   precision    recall  f1-score   support

              0.0       0.69      0.70      0.70       804
              1.0       0.70      0.69      0.69       806

         accuracy                           0.70      1610
        macro avg       0.70      0.70      0.70      1610
     weighted avg       0.70      0.70      0.70      1610
```

Figure 2.3.3.1 Accuracy of KNN Classifier Training and Testing Set

Figure 2.3.3.1 above shows that the accuracy of training set is approximately 0.7747, which is much higher compared to that of testing set at approximately 0.6957. These figures indicate that the KNN classifier is an overfitting model, whereby the model learned and trained too much based on the training test, or maybe the model picked up all the outliers and noise in the dataset. As a result, the classifier did not perform well during the testing. This issue could be resolved in the model tuning stage by using cross-validation process. The precision and recall values are not ideal as well. Improvements are expected after the parameters tuning stage.

Parameters Tuning using Randomized Search

```
[13] error = []
     for i in range(1,60):
       knn = neighbors.KNeighborsClassifier(n_neighbors=i)
       knn.fit(X_train, y_train)
       pred_i = knn.predict(X_test)
       error.append(np.mean(pred_i != y_test))
```

```
plt.figure(figsize=(12,8))
plt.plot(range(1,60), error, color = 'red', linestyle = 'dashed', marker = 'o',
         markerfacecolor = 'blue', markersize = 10)
plt.title('Mean Error Rate for N Value')
plt.xlabel('N Value')
plt.ylabel('Mean Error Rate')
```

```
Text(0, 0.5, 'Mean Error Rate')
```

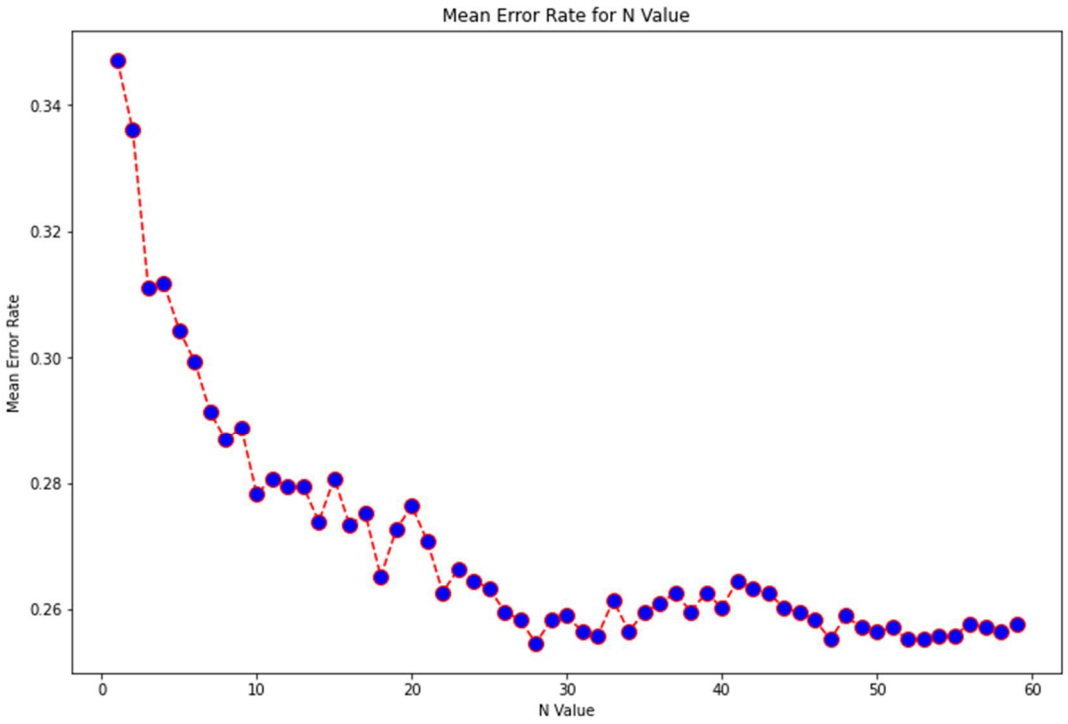

Figure 2.3.3.2 N_Neighbors Value Tuning

Figure 2.3.3.2 above shows the code snippet to compute the mean error of predicting for N value of ranging from 1 to 60. The graph plotted shows that the Mean Error Rate is inversely proportional to the n_neighbors value. The error rate is lowest starting from N value 50.

```
[55] from sklearn.model_selection import RandomizedSearchCV

     k = np.random.randint(50,60,5)
     algorithm = ['auto', 'ball_tree', 'kd_tree', 'brute']
     weights = ['uniform', 'distance']

     params = {'n_neighbors' : k,
               'algorithm' : algorithm,
               'weights' : weights
               }

     knn_random_search = RandomizedSearchCV(knn, params, n_iter=5, cv=5, n_jobs=-1, verbose =0)
     knn_random_search.fit(X_train, y_train)

     RandomizedSearchCV(cv=5, error_score=nan,
                        estimator=KNeighborsClassifier(algorithm='auto',
                                                       leaf_size=30,
                                                       metric='minkowski',
                                                       metric_params=None,
                                                       n_jobs=None, n_neighbors=59,
                                                       p=2, weights='uniform'),
                        iid='deprecated', n_iter=5, n_jobs=-1,
                        param_distributions={'algorithm': ['auto', 'ball_tree',
                                                           'kd_tree', 'brute'],
                                             'n_neighbors': array([51, 54, 53, 56, 58]),
                                             'weights': ['uniform', 'distance']},
                        pre_dispatch='2*n_jobs', random_state=None, refit=True,
                        return_train_score=False, scoring=None, verbose=0)
```

Figure 2.3.3.3 Randomized Search Parameter Tuning

The code snippet as shown in figure 2.3.3.3 import the RandomizedSearchCV for parameter tuning purposes. The parameters that will be tuned are the n_neighbors, algorithm, and the weights. Five n_neighbors values will be randomly selected from the range of 50 to 60. The algorithm value that will be tested are auto, Ball Tree, KD Tree and Brute Force, whereas the weights will be either uniform or distance. The RandomizedSearchCV will be apply random pairing of different value for each parameter and obtain the best mix and matches which produce the highest accuracy. (ScikitLearn, 2018) The RandomizedSearchCV will be performed for three times to obtain the best performing accuracy. The results obtained are shown in the table below.

Test 1

Weights	N_ Neighbors	Algorithm	Test 1 Score	Test 2 Score	Test 3 Score	Test 4 Score	Test 5 Score	**Mean Score**
Distance	56	KD Tree	0.7049	0.7065	0.6832	0.6917	0.6985	**0.6970**
Distance	53	Ball Tree	0.7049	0.7049	0.6793	0.6925	0.6961	**0.6956**
Uniform	54	Brute	0.7065	0.7220	0.7057	0.7026	0.7210	**0.7116**
Uniform	54	Auto	0.7080	0.7197	0.7049	0.7057	0.7241	**0.7125**
Distance	58	KD Tree	0.7065	0.7057	0.6832	0.6917	0.6969	**0.6968**

Test 2

Weights	N_ Neighbors	Algorithm	Test 1 Score	Test 2 Score	Test 3 Score	Test 4 Score	Test 5 Score	**Mean Score**
Uniform	50	Ball Tree	0.7026	0.7197	0.7049	0.7049	0.7202	**0.7105**
Uniform	59	KD Tree	0.7127	0.7189	0.7057	0.7041	0.7226	**0.7121**
Distance	57	Auto	0.7034	0.7065	0.6883	0.6902	0.6961	**0.6959**
Uniform	58	Auto	0.7065	0.7212	0.7057	0.7026	0.7264	**0.7125**
Distance	50	Ball Tree	0.7041	0.7065	0.6785	0.6871	0.6985	**0.6949**

Test 3

Weights	N_ Neighbors	Algorithm	Test 1 Score	Test 2 Score	Test 3 Score	Test 4 Score	Test 5 Score	**Mean Score**
Distance	50	Auto	0.7065	0.7065	0.6809	0.6917	0.6985	**0.6968**
Distance	52	Auto	0.7049	0.7072	0.6770	0.6902	0.6977	**0.6954**
Distance	50	Auto	0.7049	0.7057	0.6777	0.6888	0.6977	**0.6949**
Distance	52	Brute	0.7049	0.7072	0.6785	0.6902	0.6977	**0.6957**
Uniform	58	Brute	0.7034	0.7236	0.7057	0.6995	0.7202	**0.7105**

After running the RandomizedSearchCV for three times, the combination of uniform weight, n_neighbors value 54 and auto algorithm value performed the best, with average score 0.7125.

Model Evaluation

```
[57] print(knn_random_search.best_params_)
     knn_pre=knn_random_search.predict(X_test)
     print("Accuracy of training set ", knn_random_search.score(X_train, y_train))
     print("Accuracy of testing set", knn_random_search.score(X_test, y_test))

     {'weights': 'uniform', 'n_neighbors': 54, 'algorithm': 'auto'}
     Accuracy of training set  0.7232489517005746
     Accuracy of testing set 0.7440993788819876
```

Figure 2.3.3.4 Accuracy of KNN Classifier Training and Testing Set After Tuning

Figure 2.3.3.4 shows the accuracy of the KNN classifier after RandomizedSearchCV parameter tuning. The accuracy of the training set is at 0.7232 and the testing test is at 0.7440. The overfitting issue that was encountered earlier was resolved after the parameter tuning. The accuracy of the testing set has also improved from previously 0.6957 to 0.7440.

```
[ ] from sklearn.metrics import confusion_matrix, classification_report,
      plot_roc_curve,plot_precision_recall_curve
    print(classification_report(y_test,knn_pre))

                  precision    recall  f1-score   support

             0.0       0.71      0.80      0.76       804
             1.0       0.78      0.68      0.72       806

        accuracy                           0.74      1610
       macro avg       0.74      0.74      0.74      1610
    weighted avg       0.74      0.74      0.74      1610
```

Figure 2.3.3.5 Precision, Recall and F1-Score of KNN Classifier After Tuning

As per shown in figure 2.3.3.5 above, the precision and recall value for both target variable value “0” and “1” have shown some significant improvement compared to before parameter tuning. The precision value is interpreted by calculating the correctly predicted value over total number of predicted values. For value “0”, out of 100 predicted values “0”, only 71 of them are correctly predicted. Whereas for value “1”, out of 100 predicted value “1”, 78 of them are correctly predicted. The recall value is interpreted by calculating the correctly predicted value over total number of actual values. Among 100 patients with “0” target value, the model could correctly predict “0” for 80 of them, whereas for value “1”, the model could correctly predict 68 of them. The overall F1-Score for value “0” has improved 6%, and 3% for value “1”.

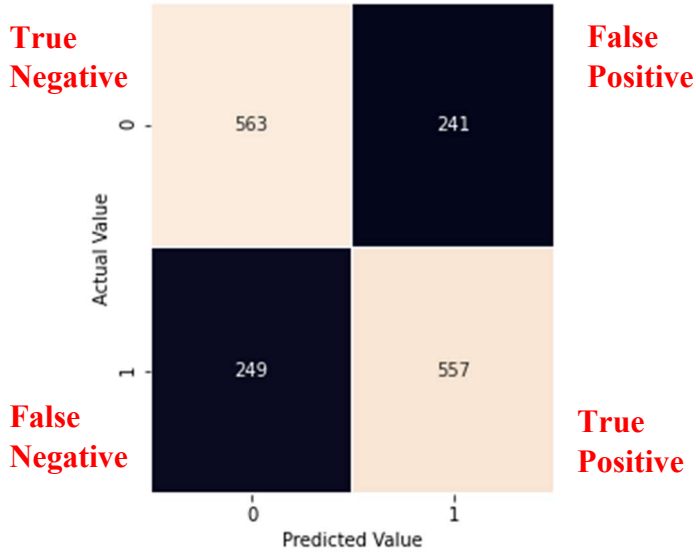

Figure 2.3.3.6 Confusion Matrix for KNN Classifier Before RandomizedSearchCV Tuning

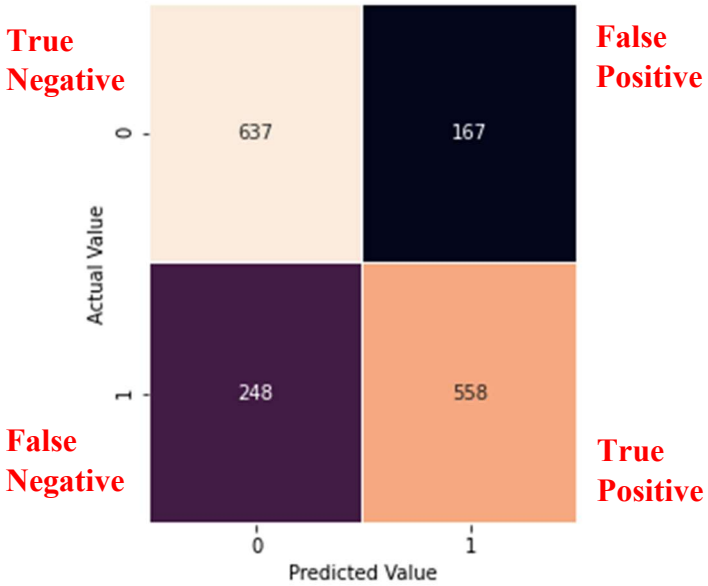

Figure 2.3.3.7 Confusion Matrix for KNN Classifier After RandomizedSearchCV Tuning

Figure 2.3.3.6 shows the confusion matrix of the KNN classifier prior parameter tuning and figure 2.3.3.7 shows the one after tuning. As a comparison, the amount of true negative, increased significantly from 563 to 637 after tuning. The amount of false positive also shows a decrease from 241 to 167 after tuning.

Based on the predicting accuracy, precision value, recall value and the confusion matrix, the RandomizedSearchCV parameter tuning has shown improvement in the KNN classifier. The overfitting issue was resolved as well. The tuned KNN classifier now has a predicting accuracy of 74.40%, which is a 5% increase in performance comparing with before. Hence, hyperparameter tuning has significantly enhanced the result of the KNN classifier model.

2.3.4 Random Forest Classifier (RF) - Gan Yee Sin (TP053268)

Meta Parameter Selection

The Random Forest classifier is trained with the data selected from feature selection section. Few important random forest hyperparameters that would increase the Random Forest model predictive power were chosen for further hyperparameter tuning and are stated in the below Table 2.3.4.1 with descriptions.

Parameters	Description
max_depth	Represent the maximum depth of tree or the longest path between tree's leaf node and the root node. (Saxena, 2020)
min_samples_split	Represent minimum observations that considered for an internal tree node to further split. Maximizing the number of samples split could avoid the tree-based algorithm from overfitting as the number of splits is reduced. (Saxena, 2020)
max_features	Represent the maximum independent variables observations needed to find the best split for trees. It is important to specify the maximum features number as it gives information on how many the features will be considered for splitting in an individual tree. Increase the features number will directly affect the processing speed. (Plapinger, 2017)
n_estimator	Represent the number of trees that builds for prediction in a forest. Increase the number of estimators could increase the model performance but at the same time decelerate the computation time. In addition, increase number of estimators could lower the chances for the tree-based algorithms from overfitting. (Donges, 2019)

Table 2.3.4.1: Random Forest Classifier Hyperparameters Selection

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

random = RandomForestClassifier(random_state = 42)
random.fit(X_train, y_train)
Y_pred = random.predict(X_train)
y_pred = random.predict(X_test)
```

Figure 2.3.4.1: Code Snippet of Random Forest Classifier with Default Parameters

Figure 2.3.4.1 shows the code snippet of the Random Forest classifier trained with default parameters and a random state value of 42. The random state value is to ensure the results get

will exactly the same multiple times. Y_pred object indicates the machine learning model prediction for the train set while y_pred object is for the test set.

```
target_names = ['class 0', 'class 1']
print('Train Set Classification Report of Random Forest:\n')
print(classification_report(y_train, Y_pred, target_names=target_names))
print('Test Set Classification Report of Random Forest:\n')
print(classification_report(y_test, y_pred, target_names=target_names))
cm = confusion_matrix(y_test, y_pred)
conf_matrix = pd.DataFrame(data = cm, columns = ['Predicted: 0', 'Predicted: 1'], index = ['Actual: 0', 'Actual: 1'])
plt.figure(figsize = (8, 5))
sns.heatmap(conf_matrix, annot = True, fmt = 'd', cmap = 'GnBu')
```

```
Train Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.91      0.95      0.93      3139
     class 1       0.95      0.91      0.93      3300

    accuracy                           0.93      6439
   macro avg       0.93      0.93      0.93      6439
weighted avg       0.93      0.93      0.93      6439

Test Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.69      0.68      0.68       789
     class 1       0.70      0.70      0.70       821

    accuracy                           0.69      1610
   macro avg       0.69      0.69      0.69      1610
weighted avg       0.69      0.69      0.69      1610
```

Figure 2.3.4.2: Random Forest Classification Model Performance (before Tuned)

Figure 2.3.4.2 shows the model performance of both the train set and test set. The Random Forest model is overfitting as it displayed a large difference between the result of the train set and the test set. The model accuracy for the train set is above 90% while the model accuracy for the test set is 69%. The model accuracy of the test set only has 69% which means the model has 31% chances of wrongly classify the observations, the error rate is considered as high for a classification model. Besides, the F1-score for test set is not ideal as well. Therefore, the researcher decided to further investigate by tuning the Random Forest hyperparameters to examine whether it would be able to improve the predictive accuracy and resolve the model overfitting issue.

Hyperparameters Tuning using Grid Search

In this section, the researcher will tune the four chosen hyperparameters as stated in Table 2.3.4.1 with GridSearchCV. To find out the optimal hyperparameters, each of the chosen parameters will be tuned with random pre-defined value.

```
from sklearn.model_selection import GridSearchCV

# Number of trees in random forest
n_estimators = np.arange(100, 900, 200)
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [6, 8, 10, 12, 14]
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10, 15]
# Create the random grid
param_grid = {'n_estimators': n_estimators,
              'max_features': max_features,
              'max_depth' : max_depth,
              'min_samples_split': min_samples_split,
              }

# Use the random grid to search for best hyperparameters
# First create the base model to tune
rf = RandomForestClassifier()
# Random search of parameters, using 3 fold cross validation,
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid, cv = 3, n_jobs = -1, verbose = 2)
# Fit the random search model
grid_search.fit(X_train, y_train)
grid_search.best_params_
```

Figure 2.3.4.3: Random Forest Classifier Hyperparameters Tuning using Grid Search

The number of trees considered in the forest was set using the Python Numpy .arrage() function by defining the start interval, stop interval and the step size for each increment of trees. Based on Figure 2.3.4.3, the number of trees was set from 100 to 900 with the value of 200 as an increment for each step of training, for the purpose to examine the best number of trees in the forest for this project dataset. The maximum features required for every split was set with auto and the square root of features. In addition, the interval for maximum depth of trees and the minimum number of observations required in splitting the tree node was set randomly to experiment the optimal value that fit the Random Forest classifier. The pre-defined hyperparameters grid was sampled and perform the K-fold cross validation. The GridSearchCV will be experimented with three times to obtain the optimal model accuracy. The Model Experiment 3 having similar model accuracy for train and test set, which means the overfitting issues was resolved by the cross-validation method. The results will be shown in Table 2.3.4.2 below.

Table 2.3.4.2: Random Forest Grid Search Hyperparameters Tuning Experiments

Grid Search Model Experiment 1:

```
{'max_depth': 6,
 'max_features': 'auto',
 'min_samples_split': 2,
 'n_estimators': 700}
```

```
Accuracy of train set:  0.7439043329709583
Accuracy of train set:  0.737888198757764
Train Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.71      0.81      0.75      3139
     class 1       0.79      0.68      0.73      3300

    accuracy                           0.74      6439
   macro avg       0.75      0.75      0.74      6439
weighted avg       0.75      0.74      0.74      6439

Test Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.71      0.80      0.75       789
     class 1       0.78      0.68      0.73       821

    accuracy                           0.74      1610
   macro avg       0.74      0.74      0.74      1610
weighted avg       0.74      0.74      0.74      1610
```

Grid Search Model Experiment 2:

```
{'max_depth': 6,
 'max_features': 'auto',
 'min_samples_split': 15,
 'n_estimators': 500}
```

```
Accuracy of train set:  0.7421959931666408
Accuracy of train set:  0.739751552795031
Train Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.71      0.81      0.75      3139
     class 1       0.79      0.68      0.73      3300

    accuracy                           0.74      6439
   macro avg       0.75      0.74      0.74      6439
weighted avg       0.75      0.74      0.74      6439

Test Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.71      0.80      0.75       789
     class 1       0.78      0.68      0.73       821

    accuracy                           0.74      1610
   macro avg       0.74      0.74      0.74      1610
weighted avg       0.74      0.74      0.74      1610
```

Grid Search Model Experiment 3:

```
{'max_depth': 8,
 'max_features': 'sqrt',
 'min_samples_split': 15,
 'n_estimators': 500}
```

```
Accuracy of train set:  0.7563286224569032
Accuracy of train set:  0.7453416149068323
Train Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.72      0.82      0.77      3139
     class 1       0.80      0.70      0.75      3300

    accuracy                           0.76      6439
   macro avg       0.76      0.76      0.76      6439
weighted avg       0.76      0.76      0.76      6439

Test Set Classification Report of Random Forest:

              precision    recall  f1-score   support

     class 0       0.72      0.80      0.75       789
     class 1       0.78      0.69      0.74       821

    accuracy                           0.75      1610
   macro avg       0.75      0.75      0.74      1610
weighted avg       0.75      0.75      0.74      1610
```

Figure 2.3.4.4: Model Performance after Hyperparameters Tuning

Based on the results obtained from hyperparameters tuning, Experiment 3 shows relatively optimal model performance with the combination of max_depth 8, max_features square root, min_samples_spit 15 and n_estimators 500.

Model Evaluation

In this project, the negative cases will be class 0 (has no cardiovascular disease) while the positive cases will be class 1 (cardiovascular disease). The Random Forest classifier (before Tuned) will be known as Model I while Random Forest Classifier (after Tuned) for Experiment 3 will be known as Model II. The results of test set will be used to measure Random Forest model performance of class 0 and class 1.

	Model I				**Model II (Model Experiment 3)**			
	Accuracy	**Recall**	**Precision**	**F1-Score**	**Accuracy**	**Recall**	**Precision**	**F1-Score**
Class 0	0.69	0.68	0.69	0.68	0.75	0.80	0.72	0.75
Class 1		0.70	0.70	0.70		0.69	0.78	0.74
The model accuracy of Model II has significantly improved from Model I (69% to 75%). The result of the recall, precision, and F1-Score of Model II is better than Model I except for recall of Class 1. For recall metric for Class 1, Model I performed with 1% better than Model II. Conversely, the true positive rate for Class 0 of Model II is 80% whereas the true positive rate of Model I is 68%. This can be further defined that Model II able to correctly predict Class 0 for 80 patients out of 100 while Model I able to correctly predict Class 0 for 68 patients out of 100. Model II performed better precision for both Class 0 and Class 1. For precision of Class 1, Model II performed better than Model I in correctly predicted the positive prediction. Model II is about 78% in correctly predicted the positive prediction while Model I is about 70% in correctly predicted the positive prediction. The F1-score of Model II is higher than Model I, which means Model II gives a better measure for the wrongly classified cases, the false negative and false positive.								

Table 2.3.4.3: Model Performance of Class 0 and Class 1 (Test Set)

Confusion matrix refers to the evaluation tool of machine learning model which aid in supporting and cross-checking with the model accuracy, precision, recall metric. Table 2.3.4.4 will demonstrate the four prediction ways of confusion matrix with the dependent variable (cardio) of this project.

False Positive (FP)	The predicted result has cardiovascular disease, but the actual has no cardiovascular disease.
False Negative (FN)	The predicted result has no cardiovascular disease, but the actual has cardiovascular disease.

True Positive (TP)	The predicted result has cardiovascular disease, and the actual has cardiovascular disease.
True Negative (TN)	The predicted result has no cardiovascular disease, and the actual has no cardiovascular disease.

Table 2.3.4.4: Confusion Matrix of Cardiovascular Disease (heart disease) Prediction

Model I Confusion Matrix:

TN = 538	FP = 251
FN = 247	TP = 574

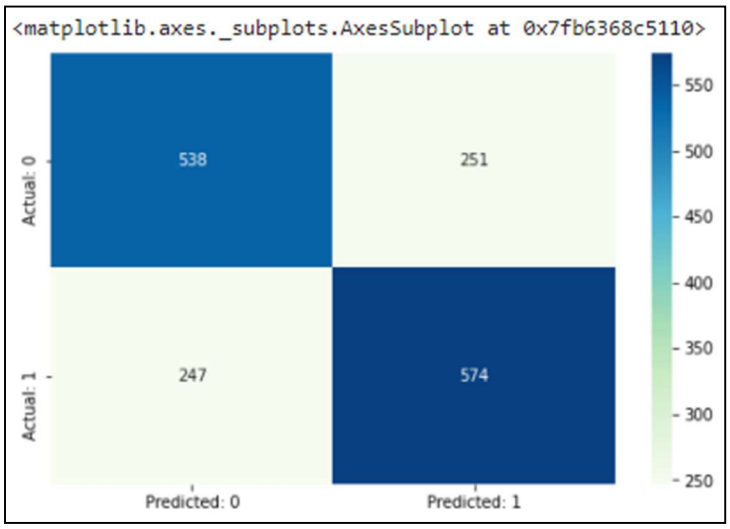

Figure 2.3.4.5: Model I Confusion Matrix Heatmap

Model II Confusion Matrix:

TN = 630	FP = 159
FN = 251	TP = 570

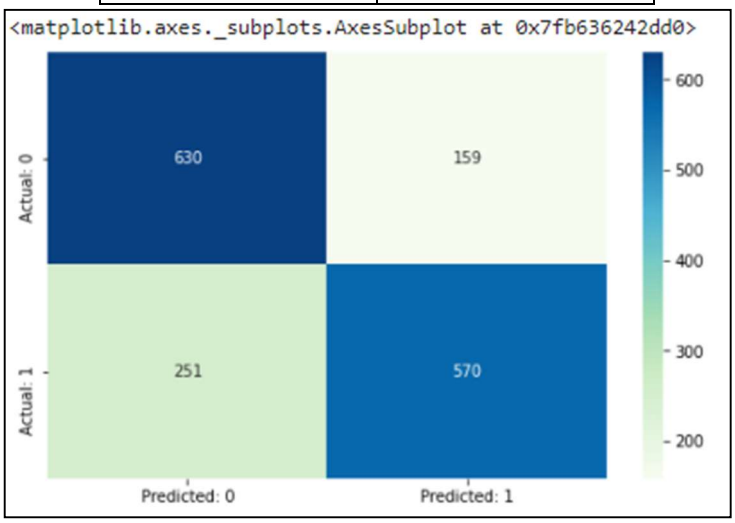

Figure 2.3.4.6: Model II Confusion Matrix Heatmap

For Model I, the total of true predicted observations has 1,112 while the wrong predicted observations have 498. For Model II, the total of true predicted observations has 1,200 while

the wrong predicted observations have 410. Model II has better model performance than Model I as there is a significant increase in the number of observations for true predictions (True Negative and True Positive) and decrease in the number of observations for false predictions (False Negative and False Positive).

The tuned Random Forest Classifier (Model Experiment 3) has enhanced the model performance; hence, it is known as the optimal model in this part of the experiment. To highlight, Model II shows better predict power for Class 0 (has no cardiovascular) than class 1 (has cardiovascular). This can be proved by the improvement number of observations of True Negative and False Positive based on the confusion matrix from Model I to Model II.

2.3.5 Support Vector Machine (SVM) – Soong Gim Hoy (TP053242)

Meta Parameter Selection

Table 2.3.5.1: Description of SVM model parameters

Parameters	**Description**
kernel	Transformation of low dimensional input space into higher dimensional space. Different kernel functions are used to observe which produces the best fit. Kernels used are linear, Gaussian Radial Basis Function, and sigmoid. (Liu, 2020)
gamma	The maximum acceptable error margin between support vectors and the optimal hyperplane. The higher the value, the more data points further from the optimal hyperplane are considered. The lower the value, the more influence closer data points to optimal hyperplane have. Support vectors are the closest data points to the optimal hyperplane. This parameter is only applicable for sigmoid and RBF kernel functions (Liu, 2020)
C	This parameter controls the amount of error considered bearable. The higher the value is set; more data points are needed to be classified as correct. The lower the value, the more lenient the model is towards errors. (Liu, 2020)

Columns ‘ap_hi’, ‘ap_lo’, ‘cholesterol’, ‘age’, and ‘weight’ are used to train the model along with the target variable ‘cardio (target)’. Table 2.3.5.1 above shows the selection of meta parameters and their descriptions. Figure 2.3.5.1 below shows the code snippet with the chosen meta parameters to find the best model using grid search.

```
C = [0.01,0.1,1]
gamma = [1,0.1,0.01]

parameters = [{'C': C,
               'kernel':['linear']},
              {'C': C,
              'gamma': gamma,
              'kernel': ['rbf']},
              {'C': C,
              'gamma': gamma,
              'kernel': ['sigmoid']}]

grid = GridSearchCV(SVC(),parameters,refit=True,verbose = 3,n_jobs = 1)
grid.fit(x_train, y_train.ravel())
```

Figure 2.3.5.1: Code snippet of chosen meta parameters

Model Tuning

Figure 2.3.5.2 below shows the result of the first Grid Search done to find the best hyperparameters. The best setting was 0.01 C value with the linear kernel function. The best accuracy obtained was 72%.

```
Best parameters set found on development set:

{'C': 0.01, 'kernel': 'linear'}

Grid scores on development set:

0.726 (+/-0.018) for {'C': 0.01, 'kernel': 'linear'}
0.726 (+/-0.019) for {'C': 0.1, 'kernel': 'linear'}
0.725 (+/-0.019) for {'C': 1, 'kernel': 'linear'}
0.511 (+/-0.001) for {'C': 0.01, 'gamma': 1, 'kernel': 'rbf'}
0.511 (+/-0.001) for {'C': 0.01, 'gamma': 0.1, 'kernel': 'rbf'}
0.671 (+/-0.028) for {'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'}
0.515 (+/-0.002) for {'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}
0.625 (+/-0.018) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
0.714 (+/-0.029) for {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}
0.626 (+/-0.017) for {'C': 1, 'gamma': 1, 'kernel': 'rbf'}
0.684 (+/-0.026) for {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
0.719 (+/-0.026) for {'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}
0.511 (+/-0.001) for {'C': 0.01, 'gamma': 1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 0.01, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 0.01, 'gamma': 0.01, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 0.1, 'gamma': 1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 0.1, 'gamma': 0.01, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 1, 'gamma': 1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.511 (+/-0.001) for {'C': 1, 'gamma': 0.01, 'kernel': 'sigmoid'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

         0.0       0.69      0.78      0.73       780
         1.0       0.76      0.67      0.71       830

    accuracy                           0.72      1610
   macro avg       0.73      0.72      0.72      1610
weighted avg       0.73      0.72      0.72      1610
```

Figure 2.3.5.2: Scores for each choices of hyperparameter and best model

The features were then standardized using the StandardScaler() function to fine tune the model.

Figure 2.3.5.3 below shows the code snippet to perform feature standardizing.

```
scalar = StandardScaler()
x_train_sc = scalar.fit_transform(x_train)
x_test_sc = scalar.fit_transform(x_test)
```

Figure 2.3.5.3: Code snippet of feature standardizing

Figure 2.3.5.4 below shows the difference before and after feature standardizing.

```
Before Standardizing
[[135.  85.  42.   1.  94.]
 [ 90.  60.  41.   1.  67.]
 [120.  80.  49.   1.  59.]
 ...
 [100.  70.  40.   1.  56.]
 [120.  80.  47.   1.  65.]
 [130.  80.  54.   2.  94.]]

After Standardizing
[[ 0.48996788  0.21180976 -1.72342854 -0.54030057  1.33171105]
 [-2.12213941 -1.812417   -1.87040685 -0.54030057 -0.50039937]
 [-0.38073455 -0.19303559 -0.69458037 -0.54030057 -1.04324689]
 ...
 [-1.54167112 -1.00272629 -2.01738517 -0.54030057 -1.24681472]
 [-0.38073455 -0.19303559 -0.98853699 -0.54030057 -0.63611125]
 [ 0.19973374 -0.19303559  0.04031118  0.93385002  1.33171105]]
```

Figure 2.3.5.4: Difference between features before and after standardizing

The Grid Search is then run again, and the results are as follow in Figure 2.3.5.5 below. The best hyperparameters are now 0.1 gamma, 1.0 C value with the RBF kernel function. Although accuracy did not increase as much, the overall average score has increased. This proves that standardizing the features is beneficial for the SVM model.

```
Best parameters set found on development set:

{'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}

Grid scores on development set:

0.724 (+/-0.018) for {'C': 0.01, 'kernel': 'linear'}
0.725 (+/-0.019) for {'C': 0.1, 'kernel': 'linear'}
0.725 (+/-0.019) for {'C': 1, 'kernel': 'linear'}
0.674 (+/-0.027) for {'C': 0.01, 'gamma': 1, 'kernel': 'rbf'}
0.732 (+/-0.019) for {'C': 0.01, 'gamma': 0.1, 'kernel': 'rbf'}
0.720 (+/-0.018) for {'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'}
0.727 (+/-0.025) for {'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}
0.732 (+/-0.021) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
0.724 (+/-0.016) for {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}
0.722 (+/-0.026) for {'C': 1, 'gamma': 1, 'kernel': 'rbf'}
0.732 (+/-0.018) for {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
0.729 (+/-0.016) for {'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}
0.629 (+/-0.007) for {'C': 0.01, 'gamma': 1, 'kernel': 'sigmoid'}
0.722 (+/-0.015) for {'C': 0.01, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.714 (+/-0.020) for {'C': 0.01, 'gamma': 0.01, 'kernel': 'sigmoid'}
0.601 (+/-0.010) for {'C': 0.1, 'gamma': 1, 'kernel': 'sigmoid'}
0.714 (+/-0.011) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.722 (+/-0.014) for {'C': 0.1, 'gamma': 0.01, 'kernel': 'sigmoid'}
0.601 (+/-0.013) for {'C': 1, 'gamma': 1, 'kernel': 'sigmoid'}
0.655 (+/-0.032) for {'C': 1, 'gamma': 0.1, 'kernel': 'sigmoid'}
0.724 (+/-0.017) for {'C': 1, 'gamma': 0.01, 'kernel': 'sigmoid'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

         0.0       0.68      0.80      0.74       780
         1.0       0.77      0.65      0.71       830

    accuracy                           0.72      1610
   macro avg       0.73      0.72      0.72      1610
weighted avg       0.73      0.72      0.72      1610
```

Figure 2.3.5.5: Scores for each choices of hyperparameters and best model using standardized features

At the end, it is decided that the model chosen to be the RBF kernel function with C value of 1 with feature standardizing. This is since a higher C-value means that the model is more sensitive to changes in the dataset then the original best linear kernel function with only 0.01 C-value. The low gamma value also represents that the SVM model has a lower leniency to errors.

Model Evaluation

Figure 2.3.5.6 below shows the classification report of the final model. The accuracy of the model is 72% meaning that the model should be classifying the accurate answer 72% of the time. The difference between the precision for class 0 and 1 show that the model may be more accurate in classifying true positive cases of heart diseases over false positives. This may be because there are more '1' classes in the whole dataset and more weights are present to filter out false positives. The high recall achieved by the model for '0' class shows that the model can classify the absence of heart diseases better than the presence of heart diseases. The higher false negatives which cause the lower recall score is because there are more '1' classes in the training target variable. The overall f1-score shows that the model is better at classifying '0' class then '1' class. Figure 2.3.5.7 below shows the confusion matrix of the final model. The confusion matrix shows that the model has many false negatives then false positives. This can be dangerous as people with heart diseases may be predicted as having no heart diseases.

	precision	recall	f1-score	support
0.0	0.68	0.80	0.74	780
1.0	0.77	0.65	0.71	830
accuracy			0.72	1610
macro avg	0.73	0.72	0.72	1610
weighted avg	0.73	0.72	0.72	1610

Figure 2.3.5.6: Classification report of final model

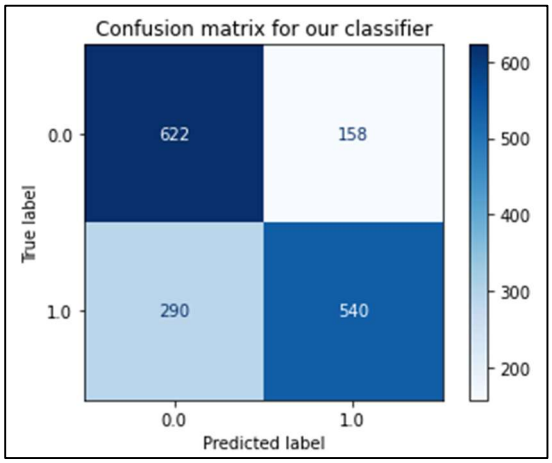

Figure 2.3.5.7: Confusion matrix of final model

3.0 Conclusion

Table 3.1: Comparison of F1-score of all models

Model	F1-score	Precision	Recall
Artificial Neural Network	0.71	0.73	0.70
Logistic Regression	0.73	0.68	**0.77**
K-Nearest Neighbour	0.74	0.74	0.74
Random Forest	**0.75**	**0.75**	0.75
Support Vector Machine	0.72	0.73	0.72

After fine tuning and optimisation, all models achieved a better score then the initial version. Table 3.1 above shows the comparison of important evaluation metrics from each model. Each model has their advantages and disadvantages in classifying cardiovascular disease. Overall, the random forest classifier has the best accuracy and precision. Logistic Regression, however, have higher recall, which subjects to lesser false negatives predicted. All the models have achieved accuracy from 70% to 75% which is considered better than average score. The accuracy of a model does depend on the type of the dataset and the characteristic of the dataset. As for this dataset, that is the optimum score that can be achieved using these four models.

Before modelling, initial exploration has been done on the dataset to understand the characteristic and the distribution of the dataset. This helped to find what kind of modification that is needed to be done on the dataset to ease the process of modelling. Correlation between the feature is also found to know the relationship between the features and feature that has high correlation value with the target has been selected that would help to achieve a good accuracy in modelling.

All the models that have been tuned have been compared in terms of classification report to see the difference of the model before and after model tuning. The difference has been explained on based on its precision, accuracy, F1 score and recall. In conclusion, this project has introduced the fundamentals of optimisation and deep learning to researchers. Each step to come to the final model has brought insights and knowledge for building better machine learning models in the future.

4.0 References

1. Aman, 2020. *Feature Selection Techniques in Machine Learning.* [Online]
   Available at: https://www.analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/
   [Accessed 18 July 2021].

2. Asiri, S., 2018. *Machine Learning Classifiers.* [Online]
   Available at: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623
   [Accessed 21 May 2021].

3. Brownlee, J., 2019. *Difference Between a Batch and an Epoch in a Neural Network.* [Online]
   Available at: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
   [Accessed 8 July 2021].

4. Brownlee, J., 2019. *Understand the impact of Learning Rate on Neural Netowrk Performance.* [Online]
   Available at: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/
   [Accessed 13 July 2021].

5. Brownlee, J., 2020. *4 Types of Classification Tasks in Machine Learning.* [Online]
   Available at: https://machinelearningmastery.com/types-of-classification-in-machine-learning/
   [Accessed 1 June 2021].

6. Center for Disease Control and Prevention, 2020. *Heart Disease Facts.* [Online]
   Available at: https://www.cdc.gov/heartdisease/facts.htm#:~:text=Heart%20disease%20is%20the%20leading,1%20in%20every%204%20deaths.
   [Accessed 27 May 2021].

7. Didugu, C., 2020. *Ultimate Guide to Input shape and Model complexity in Neural Netowrks.* [Online]
   Available at: https://towardsdatascience.com/ultimate-guide-to-input-shape-and-model-

complexity-in-neural-networks-ae665c728f4b
[Accessed 5 July 2021].

8. Donges, N., 2019. *A complete guide to the random forest algorithm.* [Online]
Available at: https://builtin.com/data-science/random-forest-algorithm
[Accessed 16 July 2021].

9. Donges, N., 2021. *A COMPLETE GUIDE TO THE RANDOM FOREST ALGORITHM.* [Online]
Available at: https://builtin.com/data-science/random-forest-algorithm
[Accessed 3 July 2021].

10. Fraj, M. B., 2017. *In Depth: Parameter tuning for KNN.* [Online]
Available at: https://medium.com/@mohtedibf/in-depth-parameter-tuning-for-knn-4c0de485baf6
[Accessed 20 July 2021].

11. Gad, A., 2018. *Beginners Ask "How Many Hidden Layers/Neurons to Use in Artificial Neural Networks".* [Online]
Available at: https://towardsdatascience.com/beginners-ask-how-many-hidden-layers-neurons-to-use-in-artificial-neural-networks-51466afa0d3e
[Accessed 12 July 2021].

12. Gandhi, R., 2018. *Support Vector Machine - Introduction to Machine Learning Algorithm.* [Online]
Available at: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
[Accessed 1 July 2021].

13. GeeksforGeeks, 2019. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/understanding-logistic-regression/
[Accessed 5 June 2021].

14. Hsu, H., 2020. *How Do Neural Network Systems Work?.* [Online]
Available at: https://medium.com/chmcore/how-do-neural-network-systems-work-dbe1bc0c4226
[Accessed 27 May 2021].

15. Huigol, P., 2019. *Accuracy vs F1 score.* [Online]
Available at: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2
[Accessed 15 July 2021].

16. Huilgol, P., 2020. *Precision vs Recall - An Intuitive Guide for Every Machine Learning Person.* [Online]
Available at: https://www.analyticsvidhya.com/blog/2020/09/precision-recall-machine-learning/
[Accessed 15 July 2021].

17. JavaTPoint, 2018. *K-Nearest Neighbor(KNN) Algorithm for Machine Learning.* [Online]
Available at: https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
[Accessed 7 June 2021].

18. Kumar, S., 2020. *Overview of Various Optimizers in Neural Networks.* [Online]
Available at: https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5#:~:text=Optimizers%20are%20algorithms%20or%20methods,problems%20by%20minimizing%20the%20function.
[Accessed 14 July 2021].

19. Liu, C., 2020. *SVM Hyperparameter Tuning using GridSearchCV.* [Online]
Available at: https://www.vebuso.com/2020/03/svm-hyperparameter-tuning-using-gridsearchcv/
[Accessed 17 July 2021].

20. Lourdunathan, P., 2018. *8 breakthrough treatments for heart disease,* Kuala Lumpur: Free Malaysia Today .

21. Martulandi, A., 2019. *K-Nearest Neighbors in Python + Hyperparameters Tuning.* [Online]
Available at: https://medium.datadriveninvestor.com/k-nearest-neighbors-in-python-hyperparameters-tuning-716734bc557f
[Accessed 20 July 2021].

22. McNealis, N., 2020. *A Simple Introducton to Dropout Regularization (With Code!).* [Online]

Available at: https://medium.com/analytics-vidhya/a-simple-introduction-to-dropout-regularization-with-code-5279489dda1e
[Accessed 14 July 2021].

23. National Heart, Lung and Blood Institute, 2020. *Know the differences, cardiovascular disease, heart disease, coronary heart disease.* [Online]
Available at: https://www.nhlbi.nih.gov/sites/default/files/media/docs/Fact_Sheet_Know_Diff_Design.508_pdf.pdf
[Accessed 30 June 2021].

24. Nishadi, A. S. T., 2019. Predicting Heart Diseases In Logistic Regression Of. *International Journal of Advanced Research and Publications,* p. 6.

25. Noble, W. S., 2006. What is a support vector machine?. *Nature Biotechnology,* Issue 24, pp. 1565-1567.

26. P.Jones, F., 2020. *K-Nearest Neighbors.* [Online]
Available at: https://www.datasklr.com/select-classification-methods/k-nearest-neighbors
[Accessed 20 July 2021].

27. Plapinger, T., 2017. *Tuning a Random Forest Classifier.* [Online]
Available at: https://medium.com/@taplapinger/tuning-a-random-forest-classifier-1b252d1dde92
[Accessed 16 July 2021].

28. Sadiq, R., Rodrigues, M. J. & Mian, H. R., 2019. Empirical Models to Predict Disinfection By-Products (DBPs) in Drinking Water: An Updated Review. *Encyclopedia of Environmental Health,* 1(2), pp. 324-338.

29. Saxena, S., 2020. *A Beginner's Guide to Random Forest Hyperparameter Tuning.* [Online]
Available at: https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/
[Accessed 16 July 2021].

30. Sayad, S., 2018. *K Nearest Neighbors - Classification.* [Online]
Available at: https://www.saedsayad.com/k_nearest_neighbors.htm
[Accessed 7 June 2021].

31. ScikitLearn, 2018. *KNeighborsClassifier.* [Online]
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
[Accessed 20 July 2021].

32. ScikitLearn, 2018. *RandomizedSearchCV.* [Online]
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
[Accessed 20 July 2021].

33. Section, 2020. *Introduction to Random Forest in Machine Learning.* [Online]
Available at: https://www.section.io/engineering-education/introduction-to-random-forest-in-machine-learning/
[Accessed 5 July 2021].

34. Sharma, S., 2017. *Activation Functions in Neural Networks.* [Online]
Available at: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
[Accessed 12 July 2021].

35. Thomas, M., 2019. *Neural Networks: Advantages and Applications.* [Online]
Available at: https://www.marktechpost.com/2019/04/18/introduction-to-neural-networks-advantages-and-applications/
[Accessed 2 June 2021].

36. Whitworth, G., 2020. *Everything You Need to Know About Heart Disease.* [Online]
Available at: https://www.healthline.com/health/heart-disease
[Accessed 27 May 2021].

5.0 Appendix

5.1 Work Breakdown Structure

	Soong Gim Hoy	Thong Kyn Hui	Chan Jia Le	Gan Yee Sin	Ragneshwary
1.0 Introduction	20%	20%	20%	20%	20%
2.0 Findings & Discussion	20%	20%	20%	20%	20%
3.0 Conclusion	20%	20%	20%	20%	20%
Signature	Phillip	Ben	Chan	Gan	Ragnesh

5.2 Word Count

The word count is excluded Abstract, Table, Figure and Citation

Chapters	Word Count
Introduction	1090 words
Main Body	3842 words
Conclusion	275 words
Total	**5207 words**