Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)
DU012(A)

UNIVERSITI TUNKU ABDUL RAHMAN

Faculty of Information and Communication Technology

UCCD2063 ARTIFICIAL INTELLIGENCE TECHNIQUES

ACADEMIC YEAR: 2024

2024 June Trimester

Group 50

Student Name	Kong Zi Lin	Steffi Yim Kar Mun	Tan Chee Yin
Student ID	2200584	2200192	2203330
Contribution	30%	30%	40%
Signature	zilin	[illegible]	[illegible]

Chapter 1: Introduction

1.1 Introduction

Health, while seemingly straightforward in concept, remains one of the most challenging goals for individuals and communities worldwide. Achieving and maintaining good health requires a comprehensive approach, taking various factors into account which may greatly influence our well-being. From lifestyle choices to environmental conditions, the pursuit of health is a complex endeavour that demands attention to detail and a commitment to making informed decisions. To achieve one of the keys to life happiness, one would have to watch their daily meal intake, perform daily exercise, get sufficient rest, stay hydrated, manage stress level and mental health, have regular health check-ups, and avoid harmful habits such as smoking and excessive alcohol consumption. And that is only the tip of the iceberg, there are still many more steps and precautions that we need to take if we wish to secure our well-being.

Although this may sound like a lot of tasks that we need to complete regularly throughout our life, it is still achievable through careful planning and efforts. However, it is still easier said than done as life does not revolve around health. In the era of technology, people's lifestyles are busier than ever. Works and Responsibilities piling up from left to right, not a single person is free from this curse. Even at the young age of adolescence, individuals are already feeling the pressure of academic demands, extracurricular activities, and social obligations. If children’s lifestyles are influenced to this extent, it is unimaginable what adults must endure. As a result, many are forced to prioritise convenience over optimal health, succumbing to a lifestyle dominated by fast food, lack of movement, and even insomnia which may lead to long-term health consequences that can be difficult to reverse.

Of course, this should not deter us from taking the necessary steps to maintain our health while managing the demands of our lives. Recognizing the challenges people face, health departments and organisations worldwide have made it their mission to assist and guide everyone in their journey towards achieving good health. With the number of people dismissing health as a minor aspect of life, increasing at a high rate, more and more initiatives are being undertaken, from providing resources, education and awareness in order to control this growing trend.

Luckily, thanks to our current technology advancement, health care departments can receive sufficient help in facing this obstacle. From making accurate diagnosis, better analysis of patient and disease records, greater access to health care to better and accurate tests. Sure, cure is very important to rid viruses or diseases from humans, returning them back to their optimal health conditions, prevention is equally important. As mentioned before, technology has granted us the opportunity to analyse patients and sickness records to produce significant insights such as their relationships and patterns, allowing us to avoid going down the same path that would result in deteriorating health, or even worse, death.

According to the World Health Organization, cardiovascular diseases (CVDs), a group of disorders of the heart and blood vessels, are the leading cause of death globally, reaching an estimate of 17.9 million people's lives every year. Out of all the deaths, 85% of them were caused by heart attack and strokes in which a blockage prevents blood from flowing to the heart or brain. It is widely believed that the cause of such events is due to unhealthy lifestyles and bad environment.

Therefore, this is the global issue that our project aims to address, identifying the risk factors that contribute to the emergence of CVD. By utilising machine learning within the realm of Artificial Intelligence (AI), our project seeks to analyse vast amounts of data to uncover patterns and correlations that may not be immediately apparent through traditional methods. Using the dataset provided, our project aspires to create an accurate prediction model by examining features which include the demographic of individuals and their lifestyles' habits that map to risks of cardiovascular diseases ranked from low to high.

With this approach, our project will demonstrate the significance of integrating advanced technologies like Artificial Intelligence into various industries, particularly healthcare. This represents a paradigm shift in how we approach the prevention and diagnosis of cardiovascular diseases. By leveraging AI, we can move from a reactive to a proactive model of healthcare, where potential health issues are identified and addressed before they become critical. Valuable insights regarding factors of cardiovascular diseases can be identified and help humanity from falling into the same path that eventually leads one to a devastating life or death.

1.2 Objective

The primary objective of this project is to develop a machine learning model that accurately predicts an individual's risk of cardiovascular disease (CVD). This prediction will be based on a comprehensive analysis of various features related to the individual's daily lifestyle and demographic information. Key features include gender, age, weight, height, family history of cardiovascular disease, alcohol intake, food and water consumption, smoking habits, and levels of physical activity.

Moving on, this project also aims to enhance diagnostic accuracy. By predicting CVD risk with precision, the model will assist healthcare professionals in making more informed decisions and identifying patients at high risk for early intervention. This proactive approach could lead to timely medical action and better health outcomes for individuals.

Another key objective of this project is to determine and promote preventative steps for CVDs. By uncovering correlations between lifestyle factors and cardiovascular risk, the model will provide valuable insights into how lifestyle modifications can reduce CVD prevalence and improve overall health. This knowledge has the potential to guide public health recommendations and personal health choices.

Furthermore, the project aims to advance research into cardiovascular diseases. The findings from the model can drive deeper investigations into the biological and environmental factors contributing to these conditions. This could lead to new strategies for prevention and treatment, contributing to the broader scientific understanding of cardiovascular health.

Chapter 2: Methods

2.1 Data Description

The ‘dataset.csv’ dataset contains 18 attributes, namely 17 categorical and numerical data and 1 targeted output. There are a total of 2100 entries in the dataset. The 17 existing attributes consist of both numerical and categorical data. The 8 numerical data in this dataset are ‘Age’, ‘Height(cm)’, ‘Weight(kg)’, ‘Vege_day’, ‘Meals_day’, ‘Water_intake(L)’, ‘Exercise’, and ‘Income’. The 9 categorical data found in this dataset are ‘Gender’, ‘Family_history’, ‘Alcohol’, ‘Junk_food’, ‘Snack’, ‘Smoking’, ‘Transportation’, ‘TV’, and ‘Discipline’. The targeted output in the dataset is ‘Cardiovascular_risk(y)’. The descriptions of each attribute are as below:

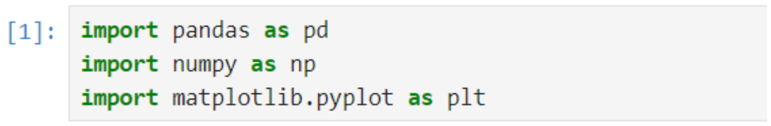
Table 2.1.1: Data Description for Each Feature

Features	**Description**
Gender	The gender of the individuals, either ‘Male’ or ‘Female’.
Age	The age of the individuals.
Height (cm)	The height of the individuals, in centimetres.
Weight (kg)	The weight of the individuals, in kilograms.
Family_history	The family history of each individual, regarding cardiovascular disease, labelled as ‘yes’ or ‘no’.
Alcohol	The alcohol intake of the individual, classified as ‘none’, ‘low’, ‘medium’, or ‘high’.
Junk_food	The junk food intake of the individual, labelled with ‘yes’ or ‘no’.
Vege_day	The number of times the individual has vegetables in meals per day.
Meals_day	The number of meals the individual has per day.

Snack	The frequency of snacks intake of the individual, classified as ‘No’, ‘Sometimes’, ‘Frequently’ or ‘Always’.
Smoking	The clarification of whether the individual smokes or not, labelled with ‘yes’ or ‘no’.
Water_intake (L)	The water intake of the individual per day, in litres.
Transportation	The mode of transportation the individual uses, classified as ‘car’, ‘bus’, ‘walk’, ‘motorcycle’, ‘bicycle’.
Exercise	The number of times the individual exercises.
TV	The frequency of TV watchtime of the individual, labelled as ‘rare’, ‘moderate’, and ‘often’.
Income	The income of each individual.
Discipline	The determination of whether the individual has discipline or not, classified as ‘yes’ or ‘no’.
Cardiovascular_risk(y)	The risk of each individual contracting cardiovascular disease(s), measured with ‘high’, ‘medium’, or ‘low’.

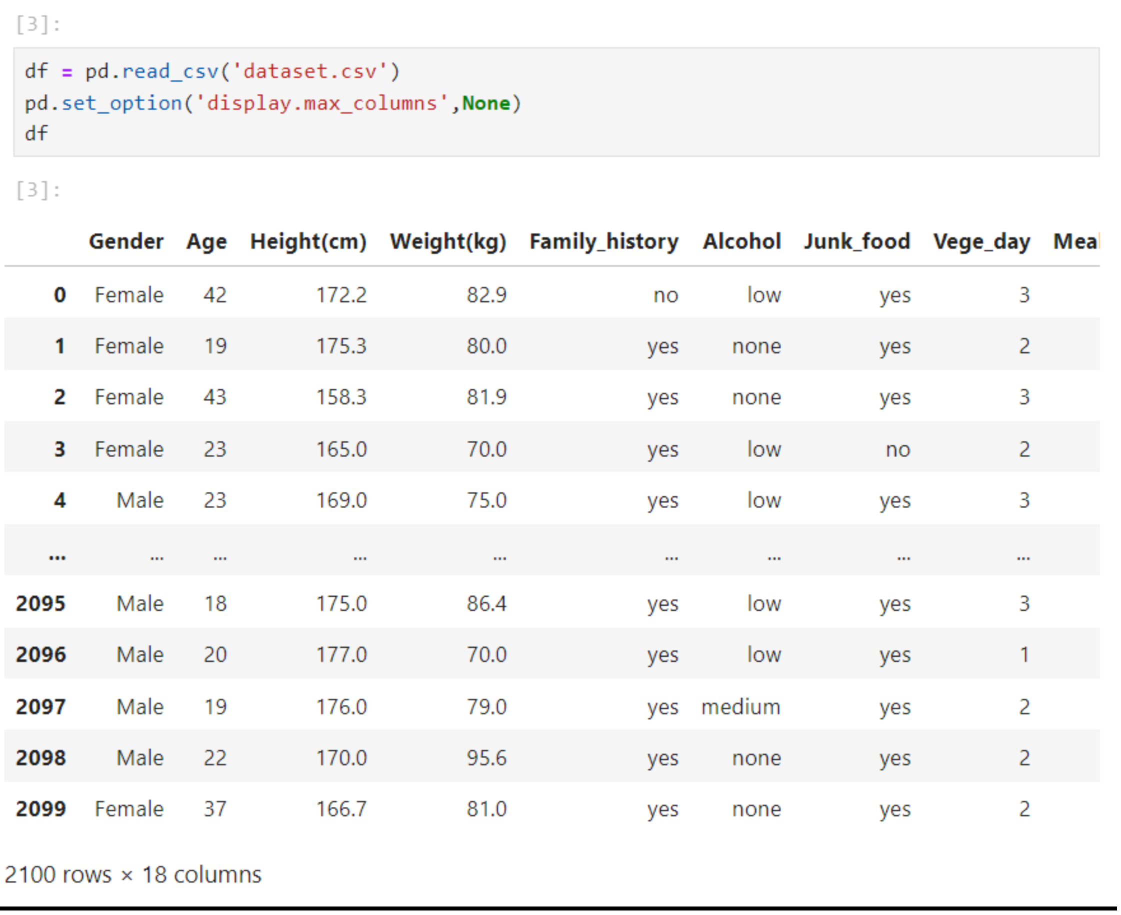
2.2 Data Exploration and Visualization

First, the relevant libraries needed are imported. Then the ‘dataset.csv’ dataset is loaded so that the first four and the last five entries are displayed as below:



```
[1]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
```

Figure 2.2.0 Importing Libraries



```
[3]:
df = pd.read_csv('dataset.csv')
pd.set_option('display.max_columns',None)
df
```

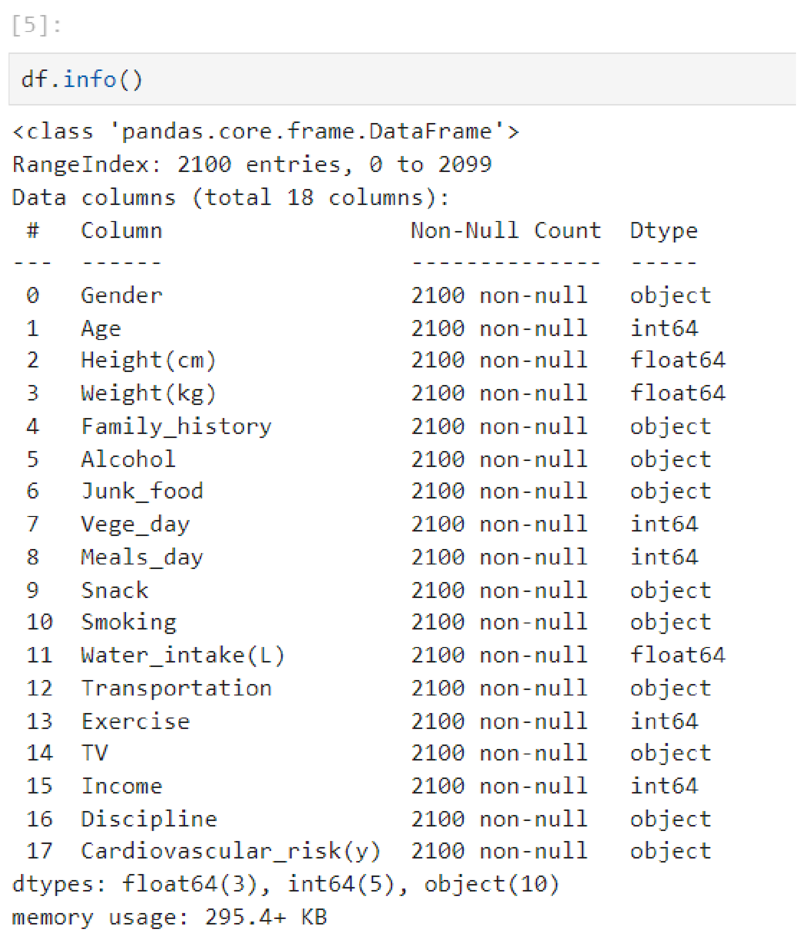
```
[3]:
```

	Gender	Age	Height(cm)	Weight(kg)	Family_history	Alcohol	Junk_food	Vege_day	Mea
0	Female	42	172.2	82.9	no	low	yes	3	
1	Female	19	175.3	80.0	yes	none	yes	2	
2	Female	43	158.3	81.9	yes	none	yes	3	
3	Female	23	165.0	70.0	yes	low	no	2	
4	Male	23	169.0	75.0	yes	low	yes	3	
...	...	...	...	...	...	...	...	...	
2095	Male	18	175.0	86.4	yes	low	yes	3	
2096	Male	20	177.0	70.0	yes	low	yes	1	
2097	Male	19	176.0	79.0	yes	medium	yes	2	
2098	Male	22	170.0	95.6	yes	none	yes	2	
2099	Female	37	166.7	81.0	yes	none	yes	2	

2100 rows × 18 columns

Figure 2.2.1 Data Importation

From Figure 2.2.1, it is known that there are a total of 2100 entries that consist of 18 attributes or columns from the dataset.

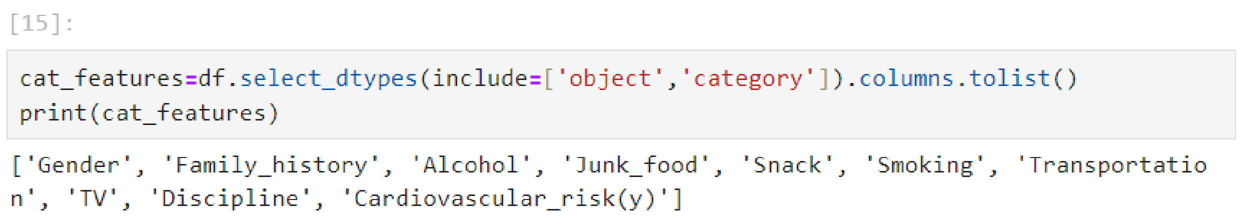


```
[5]:
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2100 entries, 0 to 2099
Data columns (total 18 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   Gender                  2100 non-null   object 
 1   Age                     2100 non-null   int64  
 2   Height(cm)              2100 non-null   float64
 3   Weight(kg)              2100 non-null   float64
 4   Family_history          2100 non-null   object 
 5   Alcohol                 2100 non-null   object 
 6   Junk_food               2100 non-null   object 
 7   Vege_day                2100 non-null   int64  
 8   Meals_day               2100 non-null   int64  
 9   Snack                   2100 non-null   object 
 10  Smoking                 2100 non-null   object 
 11  Water_intake(L)         2100 non-null   float64
 12  Transportation          2100 non-null   object 
 13  Exercise                2100 non-null   int64  
 14  TV                      2100 non-null   object 
 15  Income                  2100 non-null   int64  
 16  Discipline              2100 non-null   object 
 17  Cardiovascular_risk(y)  2100 non-null   object 
dtypes: float64(3), int64(5), object(10)
memory usage: 295.4+ KB
```

Figure 2.2.2 Data Information

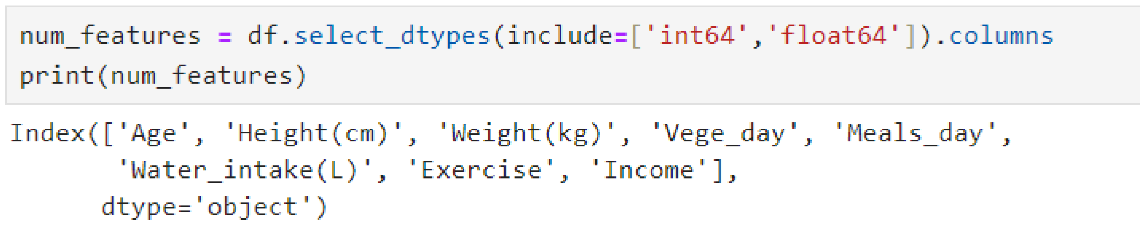
After displaying the overall data information of the dataset as shown in Figure 2.2.2, a total of 8 numerical data and 10 categorical data can be found. Another method was also used to identify both categorical data and numerical data as shown below:



```
[15]:
cat_features=df.select_dtypes(include=['object','category']).columns.tolist()
print(cat_features)
```

```
['Gender', 'Family_history', 'Alcohol', 'Junk_food', 'Snack', 'Smoking', 'Transportatio
n', 'TV', 'Discipline', 'Cardiovascular_risk(y)']
```

Figure 2.2.3 Categorical Data Identification

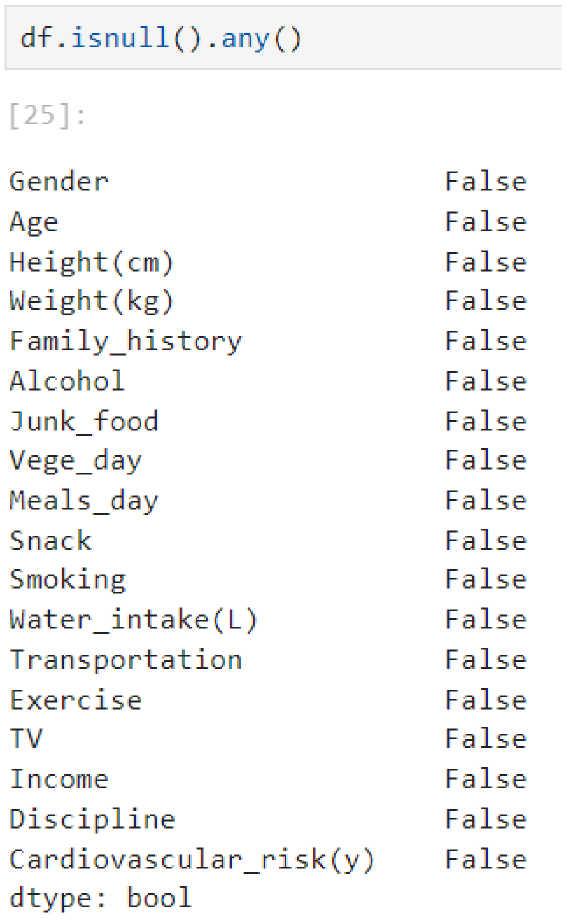


```
num_features = df.select_dtypes(include=['int64','float64']).columns
print(num_features)
```

```
Index(['Age', 'Height(cm)', 'Weight(kg)', 'Vege_day', 'Meals_day',
       'Water_intake(L)', 'Exercise', 'Income'],
      dtype='object')
```

Figure 2.2.4 Numerical Data Identification

Next, the ‘df.isnull().any()’ is used to identify any missing values in the dataset. Since ‘False’ is returned for each column of the dataset, this means there are no null or missing values in any attribute as shown in Figure 2.2.5,



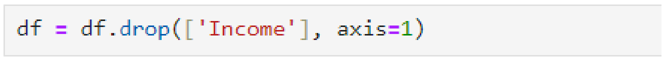
```
df.isnull().any()
```

```
[25]:

Gender                     False
Age                        False
Height(cm)                 False
Weight(kg)                 False
Family_history             False
Alcohol                    False
Junk_food                  False
Vege_day                   False
Meals_day                  False
Snack                      False
Smoking                    False
Water_intake(L)            False
Transportation             False
Exercise                   False
TV                         False
Income                     False
Discipline                 False
Cardiovascular_risk(y)     False
dtype: bool
```

Figure 2.2.5 Identification of Missing Values

2.3 Data Preprocessing

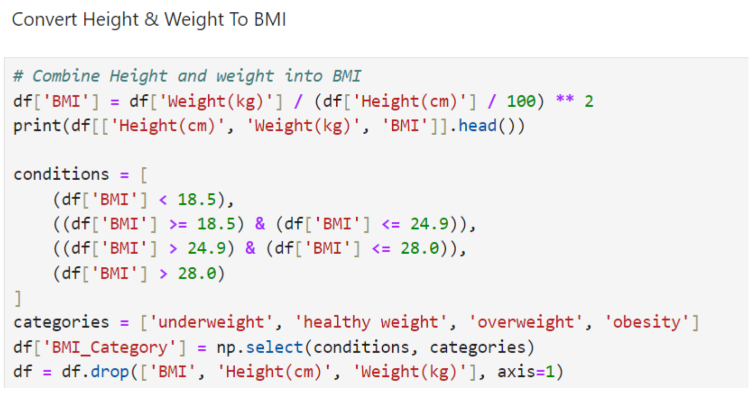
The first step that we took in data preprocessing is to remove the ‘Income’ column from the df dataframe as it was deemed irrelevant to predicting Cardiovascular_risk(y), as shown in Figure 2.3.1. Without removing said column, noise would be introduced to the dataframe which can affect the models during training, increasing complexity to the models.



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

Figure 2.3.1 Dropping ‘Income’ Column

Next, the Height and Weight columns in the data frame will be used to calculate the Body Mass Index (BMI) of each entry because BMI is deemed to be more useful and informative to the models to be trained, as shown in Figure 2.3.2. After computing the BMI, the values are then categorised as ‘underweight’, ‘healthy weight’, ‘overweight’, and ‘obesity’ in a newly created ‘BMI Category’ column. The original BMI, Height(cm), and Weight(kg) columns are dropped after the transformation, leaving just the BMI_Category for further analysis.



Convert Height & Weight To BMI

```python
# Combine Height and weight into BMI
df['BMI'] = df['Weight(kg)'] / (df['Height(cm)'] / 100) ** 2
print(df[['Height(cm)', 'Weight(kg)', 'BMI']].head())

conditions = [
    (df['BMI'] < 18.5),
    ((df['BMI'] >= 18.5) & (df['BMI'] <= 24.9)),
    ((df['BMI'] > 24.9) & (df['BMI'] <= 28.0)),
    (df['BMI'] > 28.0)
]
categories = ['underweight', 'healthy weight', 'overweight', 'obesity']
df['BMI_Category'] = np.select(conditions, categories)
df = df.drop(['BMI', 'Height(cm)', 'Weight(kg)'], axis=1)
```

Figure 2.3.2 Computing BMI categories

A bar chart is later plotted to show the distribution of each category in the BMI_category column, as shown in Figure 2.3.3.

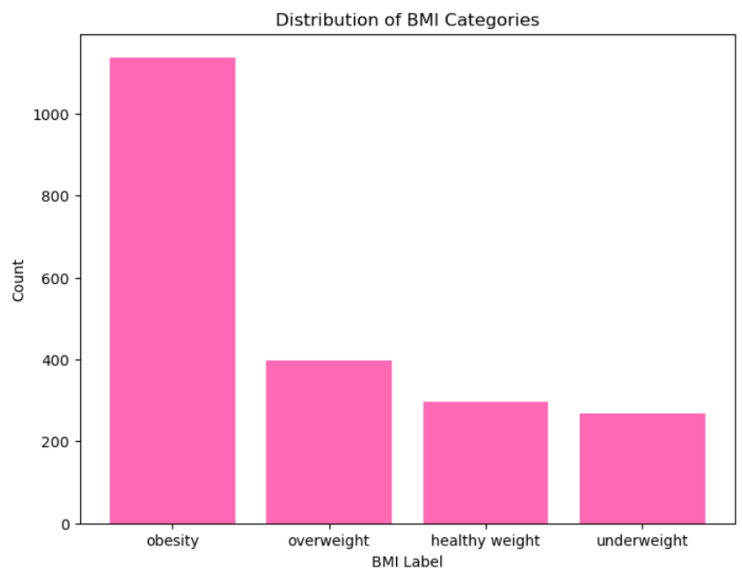
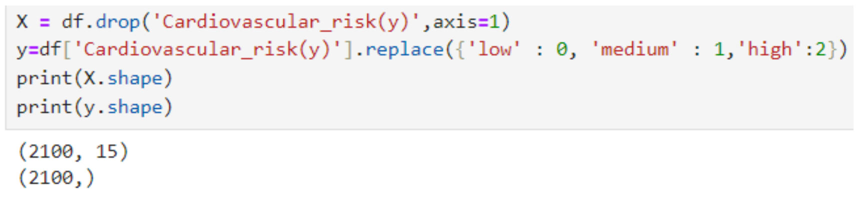


Figure 2.3.3 Barchart of BMI Categories

Next X and y variables will be created to hold all columns of the dataset except for Cardiovascular_risk(y) and only Cardiovascular_risk(y) respectively. The target column Cardiovascular_risk(y) is encoded into numeric values (low = 0, medium = 1, high = 2). This conversion is done to prepare the labels for model training, as shown in Figure 2.3.4.

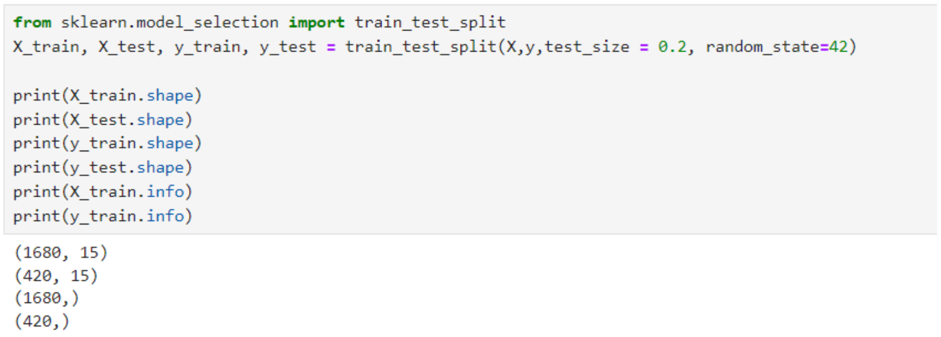


```
X = df.drop('Cardiovascular_risk(y)',axis=1)
y=df['Cardiovascular_risk(y)'].replace({'low' : 0, 'medium' : 1,'high':2})
print(X.shape)
print(y.shape)
```

```
(2100, 15)
(2100,)
```

Figure 2.3.4 Separating X and y

Next, the dataset is split into training and test sets, with 80% of the data used for training and 20% for testing. The random_state=42 ensures reproducibility by fixing the random shuffling of data, as shown in Figure 2.3.5.

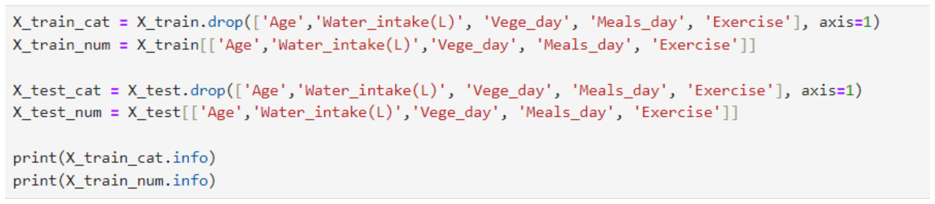


```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.2, random_state=42)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
print(X_train.info)
print(y_train.info)
```

```
(1680, 15)
(420, 15)
(1680,)
(420,)
```

Figure 2.3.5 Splitting training and testing sets

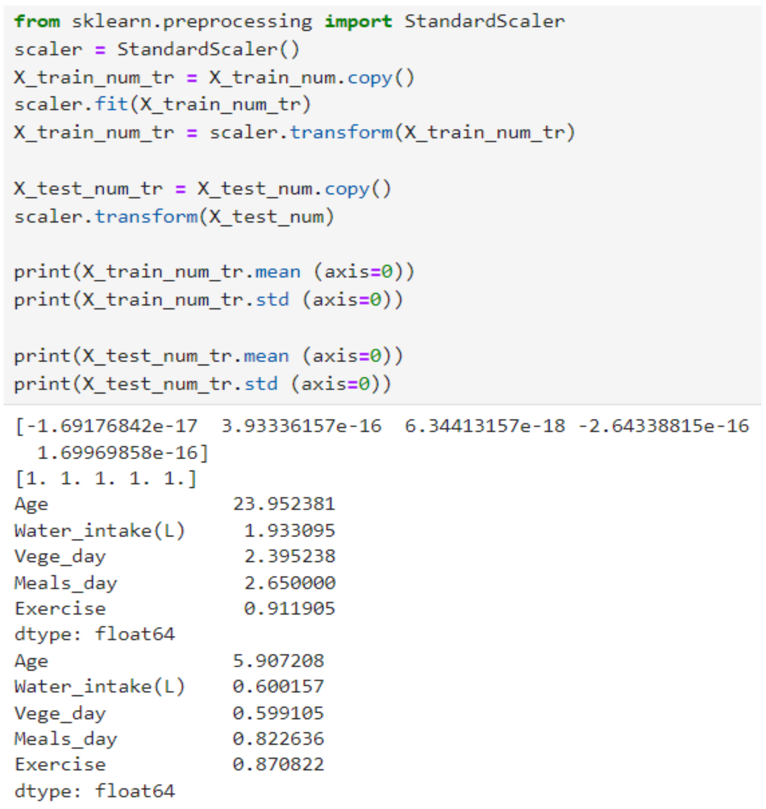
Next, the features are separated into categorical (X_train_cat) and numerical (X_train_num) groups for separate preprocessing steps. This is to help apply different transformations like scaling for numerical data and one-hot encoding for categorical data found in the dataset, as shown in Figure 2.3.6.



```
X_train_cat = X_train.drop(['Age','Water_intake(L)', 'Vege_day', 'Meals_day', 'Exercise'], axis=1)
X_train_num = X_train[['Age','Water_intake(L)','Vege_day', 'Meals_day', 'Exercise']]

X_test_cat = X_test.drop(['Age','Water_intake(L)', 'Vege_day', 'Meals_day', 'Exercise'], axis=1)
X_test_num = X_test[['Age','Water_intake(L)','Vege_day', 'Meals_day', 'Exercise']]

print(X_train_cat.info)
print(X_train_num.info)
```

Figure 2.3.6 Separating numerical and categorical features

Standardisation ensures that features with larger values do not dominate the learning process. The fit_transform method is applied on the training data, and transform is used on the test data, ensuring the test data is scaled using the same parameters as the training data, as shown in Figure 2.3.7.

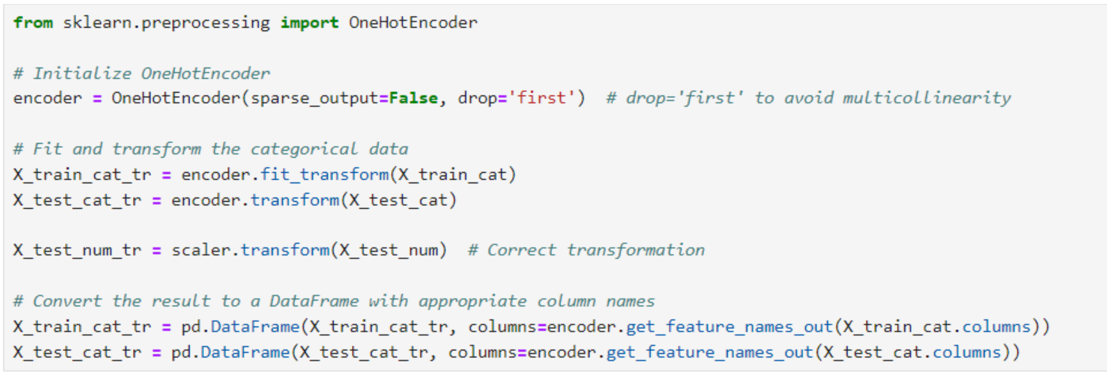


```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_num_tr = X_train_num.copy()
scaler.fit(X_train_num_tr)
X_train_num_tr = scaler.transform(X_train_num_tr)

X_test_num_tr = X_test_num.copy()
scaler.transform(X_test_num)

print(X_train_num_tr.mean (axis=0))
print(X_train_num_tr.std (axis=0))

print(X_test_num_tr.mean (axis=0))
print(X_test_num_tr.std (axis=0))
```

```
[-1.69176842e-17  3.93336157e-16  6.34413157e-18 -2.64338815e-16
  1.69969858e-16]
[1. 1. 1. 1. 1.]
Age                 23.952381
Water_intake(L)      1.933095
Vege_day             2.395238
Meals_day            2.650000
Exercise             0.911905
dtype: float64
Age                 5.907208
Water_intake(L)     0.600157
Vege_day            0.599105
Meals_day           0.822636
Exercise            0.870822
dtype: float64
```

Figure 2.3.7 Standardization of data

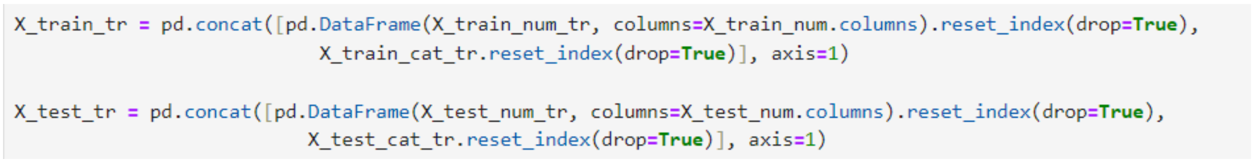
Then, OneHotEncoder encodes the categorical variables into binary variables. The drop='first' argument avoids multicollinearity by dropping the first category. The categorical columns are then transformed into a matrix of binary variables, as shown in Figure 2.3.8.



```
from sklearn.preprocessing import OneHotEncoder

# Initialize OneHotEncoder
encoder = OneHotEncoder(sparse_output=False, drop='first')  # drop='first' to avoid multicollinearity

# Fit and transform the categorical data
X_train_cat_tr = encoder.fit_transform(X_train_cat)
X_test_cat_tr = encoder.transform(X_test_cat)

X_test_num_tr = scaler.transform(X_test_num)  # Correct transformation

# Convert the result to a DataFrame with appropriate column names
X_train_cat_tr = pd.DataFrame(X_train_cat_tr, columns=encoder.get_feature_names_out(X_train_cat.columns))
X_test_cat_tr = pd.DataFrame(X_test_cat_tr, columns=encoder.get_feature_names_out(X_test_cat.columns))
```

Figure 2.3.8 Encoding categorical data

After preprocessing the numerical and categorical features separately, they are combined back into a single DataFrame for both the training (X_train_tr) and test (X_test_tr) sets, as shown in Figure 2.3.9.



```
X_train_tr = pd.concat([pd.DataFrame(X_train_num_tr, columns=X_train_num.columns).reset_index(drop=True),
                           X_train_cat_tr.reset_index(drop=True)], axis=1)

X_test_tr = pd.concat([pd.DataFrame(X_test_num_tr, columns=X_test_num.columns).reset_index(drop=True),
                          X_test_cat_tr.reset_index(drop=True)], axis=1)
```

Figure 2.3.9 Concatenation of features

2.4 Model Selection

2.4.1 K-Nearest Neighbours (KNN) Model

Reasons to Pick This Model

1. Simplicity and Interpretability

KNN is an extremely user-friendly technique that uses the majority class of a data point's closest neighbours in the feature space to classify it. This makes it simple for non-technical audiences or stakeholders to comprehend, apply, and explain. KNN's decision-making process is clear, in contrast to intricate models like neural networks or ensemble approaches. It only counts the number of nearby data points that belong to each class and assigns the new data point to the class that is most prevalent. In domains where interpretability is critical, such as healthcare or finance, where comprehension of the rationale behind forecasts is just as significant as precision, this simplicity is helpful.

2. Non-Parametric Nature

Being a non-parametric method, which means it doesn't make firm assumptions about the underlying data distribution, is one of KNN's main advantages. Numerous other methods, like SVM with linear kernels or linear regression, presume that the data follows specific distributions (like Gaussian or linear), which could not be the case for datasets seen in real-world settings.In this situation, when the data might not follow any particular distribution, KNN can adjust to many forms and patterns without necessitating predetermined hypotheses. Because of this, it is robust when used with complex datasets that don't easily fit into the parameters of parametric models. KNN can capture associations that may be nonlinear, which is very essential when working with heterogeneous health or behavioural data.

3. Versatility with Distance Metrics

The versatility of KNN stems from its capacity to gauge the degree of similarity between data points using various distance measures. KNN employs the Euclidean distance by default, but depending on the type of data, it can also be adjusted to use the Manhattan distance, Cosine similarity, or a different measure. For example, optimising the distance measure might improve performance if the data points are dispersed unevenly or densely within the feature space. It is possible to evaluate a variety of distance metrics (such as Manhattan or Euclidean) to make sure the model accurately depicts the connections between the data points. KNN has a major advantage over models with fixed distance calculations since it can experiment with different distance measures.

4. Effective for Mixed Data Types

The dataset includes both categorical (such as gender, family history) and numerical (such as age, water intake) characteristics. When properly preprocessed, such as by scaling for numerical features and encoding for categorical variables, KNN can be applied with great flexibility to such mixed datasets. All characteristics are guaranteed to contribute equally to the distance calculation by scaling the numerical features (e.g., using StandardScaler), preventing any one feature from significantly impacting the results. After the data is converted, KNN can efficiently handle both kinds of variables, identifying trends and parallels even in cases where some features are categorical and others are numerical. Because of its adaptability, KNN is a great choice for datasets including a variety of data kinds, preventing any feature type from being overlooked during the classification process.

How The Model Works

Being a lazy learning algorithm, KNN does not create an explicit model while it is being trained. Rather, when making predictions, it utilises the complete training dataset that it has learned by memory. Training involves no complex calculations or parameter estimations. KNN then stores the annotated training data points and gets ready for more predictions.

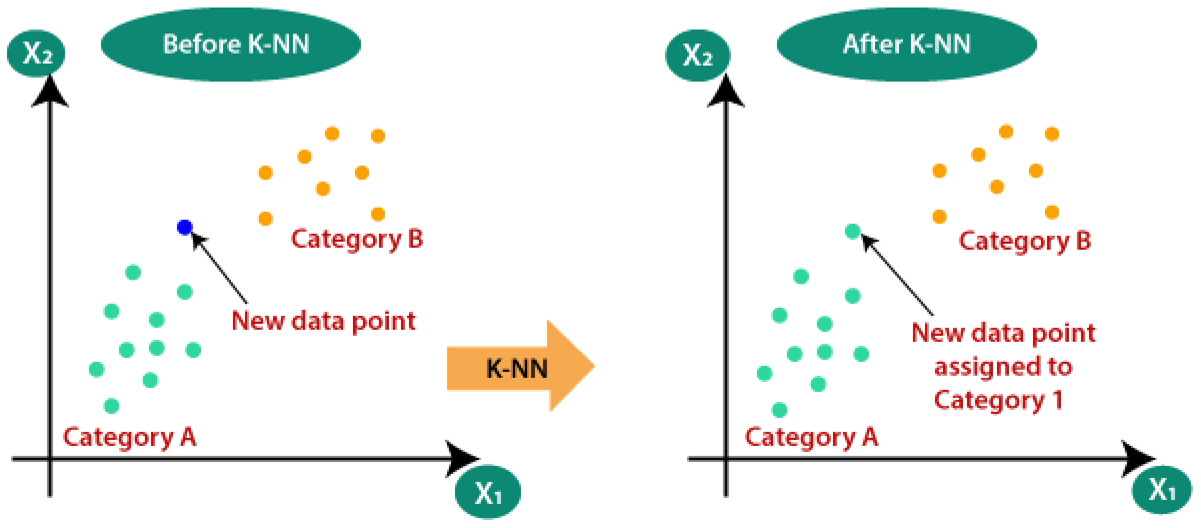
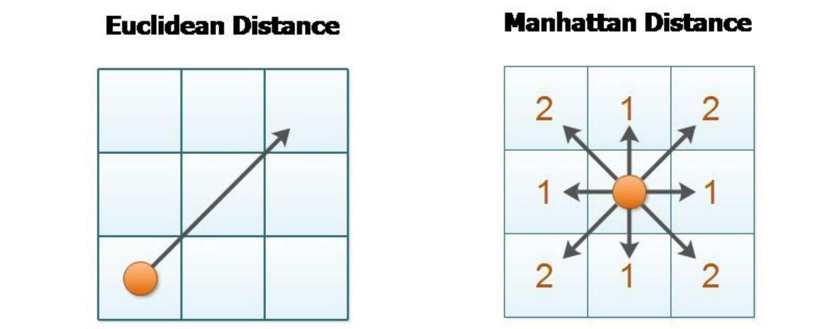


Figure 2.4.1-a: KNN Model

KNN determines the separation between a newly added, unlabeled data point and every other point in the training set. Depending on the dataset and problem type, common distance measures include Manhattan distance, Euclidean distance, and others.

With k being a user-defined value, the technique finds the new data point's k-nearest neighbours. How "closeness" is measured depends on the distance metric selected. For example, the Manhattan distance algorithm computes the sum of absolute differences across dimensions, whereas the Euclidean distance algorithm computes the straight-line distance between two points in the feature space.



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad |x_1 - x_2| + |y_1 - y_2|$$

Figure 2.4.1-b: Euclidean Distance and Manhattan Distance

Following the identification of the closest neighbours, KNN uses a majority vote technique to apply a class label to the newly discovered data point. The class with the most votes is allocated to the new point, and each of the k-nearest neighbours "votes" for the class label it belongs to. This enables KNN to use the labels of neighbouring, comparable points in the training data to classify the new data point. For instance, if k=3, the algorithm chooses the three closest neighbours, and the projected class is determined by assigning the label with the majority—that is, two out of the three votes. The KNN model's performance depends critically on the value of k. Overfitting could happen if k is too little since the model would become excessively sensitive to noise. However, if k is excessively big, the model might oversimplify the categorization, which would result in underfitting. Usually, the best value for k is chosen through the process of cross-validation.

In certain variants, the KNN algorithm employs weighted voting, in which neighbours that are closer together are assigned a higher weight than those who are farther away. This method weights the class labels of the closest neighbours based on how close they are to the new data point. A point that is very close to the new data point, for instance, can have a greater impact on the final class label than a point that is farther away. Following the voting process, the new data point is categorised according to the k-nearest neighbours' majority class. The model then outputs this projected class as the data point's prediction.

2.4.2 Random Forest

Reasons to pick this model:

1. Robust Performance:

Random Forest is renowned for its strong performance and robustness. It combines the predictions of multiple decision trees, each built on different random subsets of the data. This ensemble approach helps mitigate the risk of overfitting, as the model averages the predictions of individual trees, which reduces the impact of noise and variability in the data. As a result, Random Forest often delivers reliable predictions and maintains high accuracy across various datasets.

2. Handling Complex Data:

Random Forest excels in handling complex data with many features and interactions. It is capable of managing high-dimensional datasets and capturing non-linear relationships between features. The algorithm’s ability to perform feature selection internally also helps in identifying the most important variables, which improves the model's interpretability and performance.

3. Versatility:

Random Forest is a versatile algorithm that can be used for both classification and regression tasks. It can handle various types of data, including numeric and categorical variables. Additionally, it is effective in dealing with missing values and maintaining performance even with incomplete data.

4. Feature Importance:

One of the advantages of Random Forest is its ability to provide insights into feature importance. By evaluating how much each feature contributes to the decision-making process, Random Forest helps in understanding the relative importance of different variables. This feature is valuable for feature selection and for gaining insights into the underlying data patterns.

5. Ease of Use:

Random Forest is relatively easy to use and requires minimal parameter tuning compared to some other algorithms. Its robustness and the fact that it performs well out-of-the-box make it a popular choice for many machine learning tasks, especially when dealing with real-world data where parameter tuning might be challenging.

How the model works:

Random Forest is an ensemble learning method that utilises bagging (bootstrap aggregating) to build a collection of decision trees. The process begins by creating multiple decision trees, each trained on a different random subset of the data through bootstrapping, which involves sampling the training data with replacement. This ensures that each tree is exposed to different variations of the dataset. Additionally, during the training of each tree, a random subset of features is considered at each split, reducing correlation between the trees and enhancing model diversity.

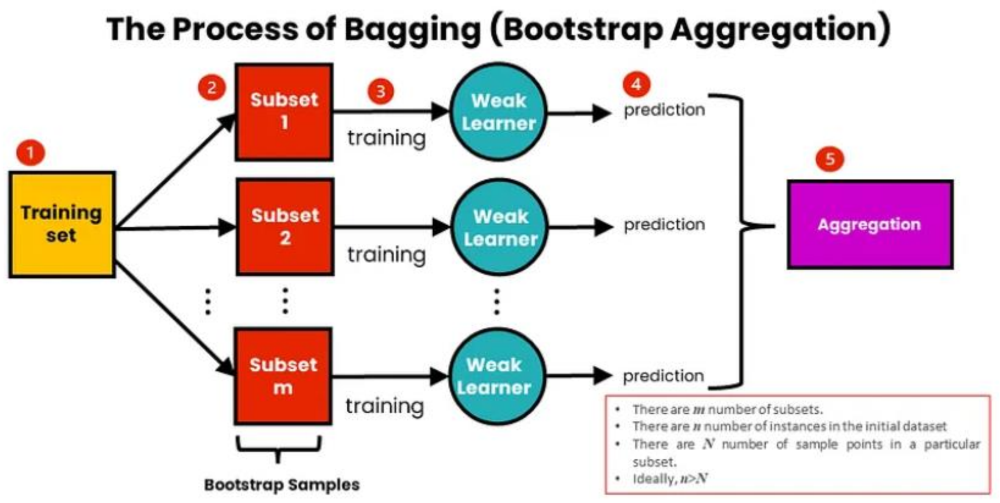

The Process of Bagging (Bootstrap Aggregation)
1
Training set
2
Subset 1
Subset 2
Subset m
Bootstrap Samples
3
training
training
training
Weak Learner
Weak Learner
Weak Learner
4
prediction
prediction
prediction
5
Aggregation
There are m number of subsets.
There are n number of instances in the initial dataset
There are N number of sample points in a particular subset.
Ideally, $n>N$

Figure 2.4.2-a:

Each decision tree in the Random Forest is constructed independently and grows to its maximum depth or until it meets a predefined stopping criterion. Once all the trees are built, the Random

Forest model aggregates their predictions to generate a final result. For classification tasks, the model employs a majority voting approach, where the class label that receives the most votes from the individual trees is selected as the final prediction. For regression tasks, the model averages the predictions from all the trees to produce a final value.

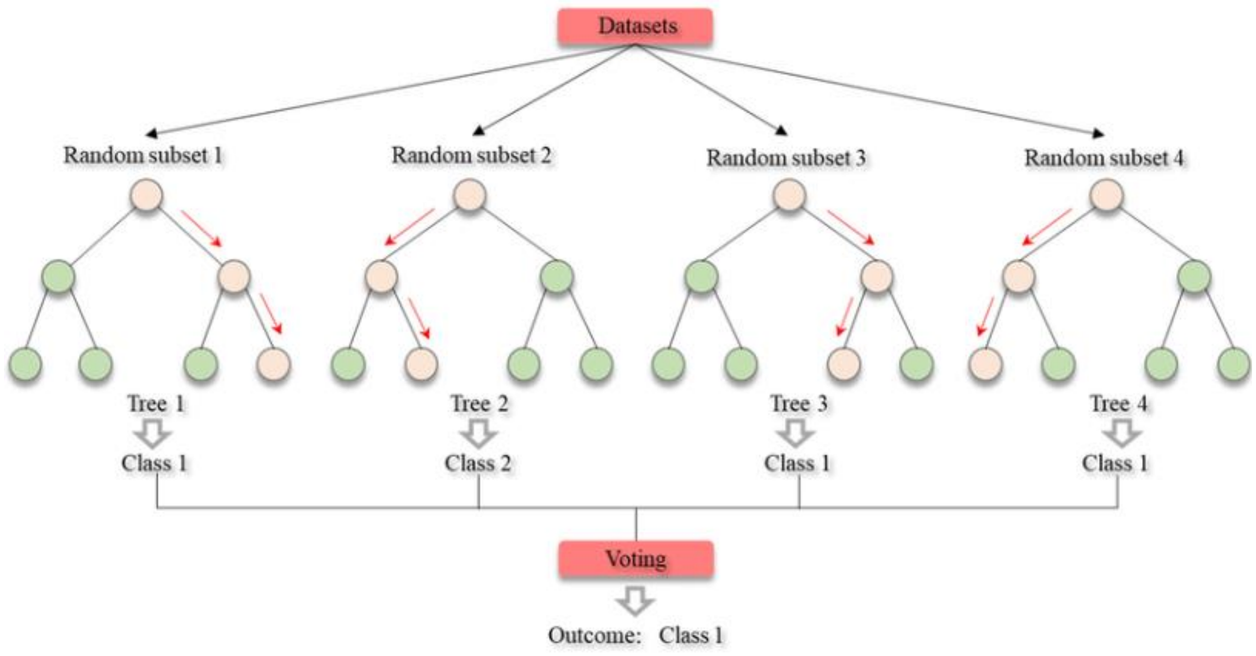


Figure 2.4.2-b:

2.4.3 Gradient Boosting Classifier (GBC) Model

Reasons to pick this model:

1. Ability to Reduce Overfitting Through Regularization:

Gradient Boosting consists of several hyperparameters, like *learning_rate*, *n_estimators*, *max_depth*, and *subsample*. These hyperparameters enable fine-tuning to prevent overfitting, a common issue with certain models, from occurring. By controlling the contribution of each tree in the ensemble through *learning_rate* and *n_estimators* hyperparameters, an efficient model that generalises well to new complex data, like this dataset, where small adjustments can affect the cardiovascular risk prediction. It is better than models like decision trees or logistic regression, which can experience overfitting when dealing with noisy data.

2. Capturing Non-Linear Relationships:

GBC is a great ensemble technique that captures non-linear relationships between features and the target. Given that health-related factors like 'Exercise', 'Smoking', and 'Cardiovascular_risk' may have non-linear interactions, GBC's ability to model complex patterns makes it suitable for this type of dataset.

3. Built-In Feature Selection and Importance:

Gradient Boosting models inherently perform feature selection during the training process, focusing on the most important features and ignoring irrelevant ones. In this dataset, where some features may be more predictive of cardiovascular risk than others, like 'Family_history' versus 'TV', GBC automatically prioritises the key predictors, leading to better performance without extensive manual feature engineering.

How the model works:

Gradient Boosting Classifier (GBC) is a great machine learning algorithm that builds an ensemble of weak learners, typically decision trees, in a sequential manner. Unlike other ensemble methods where trees are built independently, GBC builds trees iteratively, with each new tree focusing on the errors made by the previous ones. This approach allows GBC to progressively improve its performance by correcting mistakes, making it highly effective for tasks where capturing complex patterns is essential. By combining multiple weak learners, GBC creates a robust model capable of capturing both linear and non-linear relationships in the data, making it a versatile choice for many types of problems, including classification tasks like cardiovascular risk prediction.

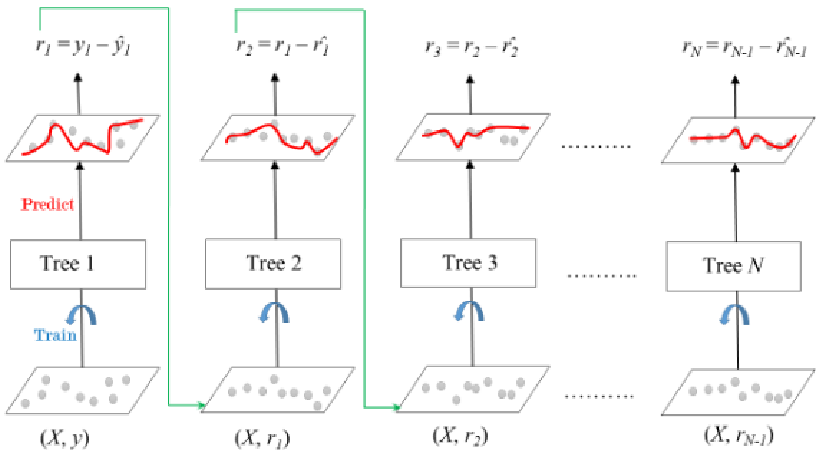


Figure 2.4.3-a: How Gradient Boosting Classifier Works

A key strength of Gradient Boosting is its ability to control overfitting through regularisation. Hyperparameters such as learning_rate, n_estimators, and max_depth help in regulating the complexity of the model and the contribution of each tree, ensuring that the model doesn't become overly fitted to the training data. Regularisation methods like shrinkage and subsampling provide fine-grained control over the model's complexity. This makes GBC particularly suitable for datasets that contain noise or have an imbalance in class distribution.

2.4.4 XGBoost (Extreme Gradient Boosting)

Reasons to pick this model:

1. High Performance:

XGBoost is known for its state-of-the-art Accuracy, often able to obtain superior accuracy compared to other machine learning algorithms due it its ability to handle complex patterns and interactions between features. As a result, many competitors in machine learning competitions such as Kaggle is able to secure victory, or at least obtain a stunning end result all thanks to its strong predictive performance, which further proved XGBoost's effectiveness in constructing a strong predictive machine learning model.

2. Regularisation and Hyperparameter Tuning:

XGBoost offers extensive hyperparameter tuning options, including learning_rate, n_estimators, max_depth, min_child_weight, subsample, and gamma that can be adjusted to acquire a better machine learning model performance. On top of that, it also has built-in regularisation options, such as L1 and L2 regularisation, which adds penalties for overly complex trees and helps keep the model simple. So, through its hyperparameters fine-tuning features, we can greatly prevent overfitting as it can make the model less sensitive to noise, preventing it from overfitting to the training data. Furthermore, it can also help the model generalise better to unseen data and allow the model to learn patterns more gradually and accurately. Often, this also allows the model to greatly adapt to different types of datasets, especially those with high-dimensional data, missing values, or imbalances. All in all, the ability that allows users to perform hyperparameters fine-tuning in XGBoost not only supports a model to better fit the specific characteristics of our data, but also improves both its accuracy and robustness, ensuring a strong and reliable prediction no matter the data conditions.

3. Efficiency and Speed:

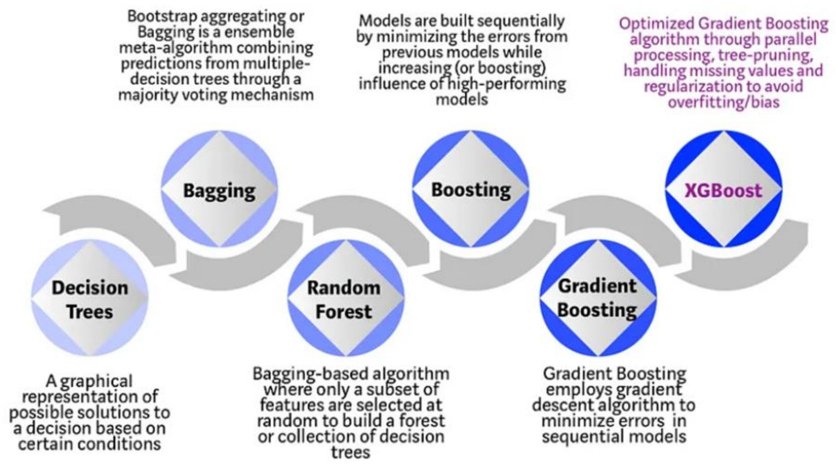
XGBoost is designed with computational efficiency in mind, offering several features that make it faster than many other algorithms. One key feature is parallel processing, which utilises hardware resources to speed up model training by running operations simultaneously. Additionally, XGBoost is optimised to handle sparse data and efficiently manage missing values, making it ideal for real-world datasets that often contain incomplete or imbalanced data. These optimizations ensure that XGBoost delivers both high performance and faster training times, even with large or complex datasets.

4. **Scalability and Flexibility:**

Moreover, XGBoost is highly scalable, making it suitable for both small datasets and large-scale machine learning tasks. It efficiently handles millions of data points and high-dimensional datasets, which makes it a popular choice for diverse machine learning applications. Additionally, XGBoost supports various tasks, such as classification, regression, and ranking, enhancing its versatility. Hence, its flexibility in managing different types of data and tasks encountered in a wide range of machine learning problems is one of the main reasons we selected XGBoost as it ensures an excellent fit for our dataset, which is discovered to be a multiclass classification problem.

How does it work?

XGBoost (Extreme Gradient Boosting) is a highly efficient and versatile machine learning algorithm used for both regression and classification tasks. As an ensemble learning method, XGBoost combines multiple weak learners, typically decision trees, to create a robust and accurate predictive model. Although it shares similarities with Random Forest, which also uses decision trees, XGBoost differs significantly in its approach.



Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism
Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models
Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias
Bagging
Boosting
XGBoost
Decision Trees
Random Forest
Gradient Boosting
A graphical representation of possible solutions to a decision based on certain conditions
Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees
Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

Figure 2.4.4-a: XGBoost Description

The primary distinction lies in their ensemble techniques. Random Forest uses bagging (bootstrap aggregating), where multiple trees are built independently on random subsets of the data and then averaged.

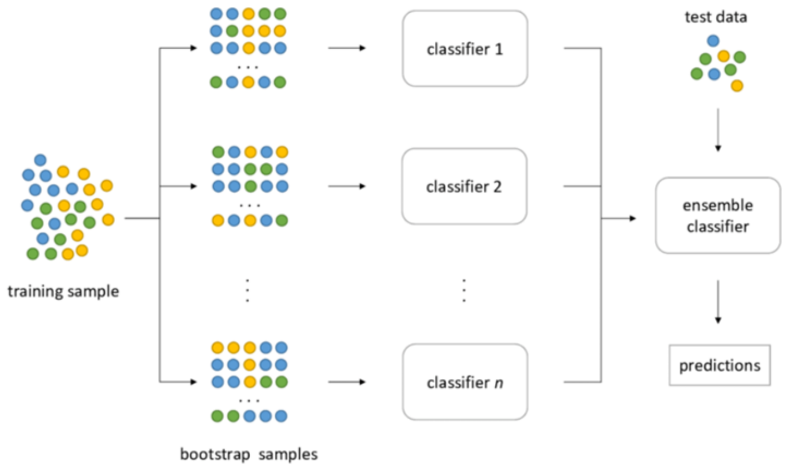


Figure 2.4.4-b:Bootstrap Aggregating

XGBoost, in contrast, employs gradient boosting, where new models are built sequentially to predict the residuals or errors of the previous models, which are then combined to make the final prediction.

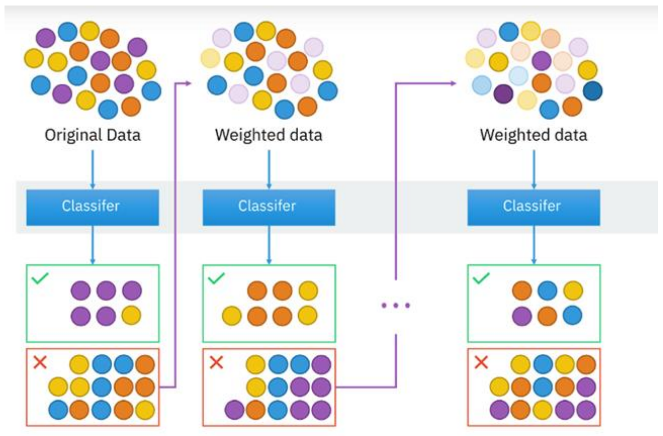


Figure 2.4.4-b: How XGBoost works

The XGBoost algorithm operates through an iterative process. It starts with an initial model that makes a baseline prediction, such as the mean value for regression or a uniform probability for classification which is our current case.

Then, the process continues by calculating the residuals, or the differences between the model's current predictions and the actual values.

Residual = $y - y'$

Where:

- y is the actual value.
- y' is the predicted value.

These residuals highlight the areas where the model's predictions are lacking. XGBoost then trains a new decision tree specifically to predict these residuals. Unlike in Random Forest, where each tree is built independently, the trees in XGBoost are added one after another. Each new tree adjusts the model's predictions to better match the true values, thus improving accuracy incrementally.

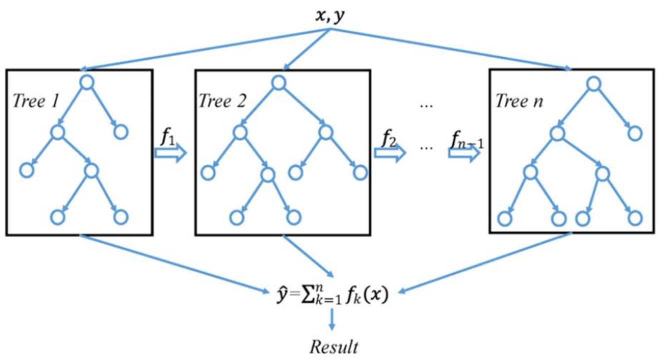


Figure 2.4.4-c:

Once a new tree is trained, its predictions are combined with those from the existing trees to update the overall model. This update process utilizes the concept of gradient descent, with the new tree's contribution scaled by a learning rate. The learning rate controls how much impact the new tree has on the final prediction, helping to prevent overfitting.

$y_{new} = y_{old} + \text{learning rate} \times f(x)$

Where:

- f(x) is the prediction from the new tree.
- y_{new} is the current new tree to be trained
- y_{old} is the previous old tree trained

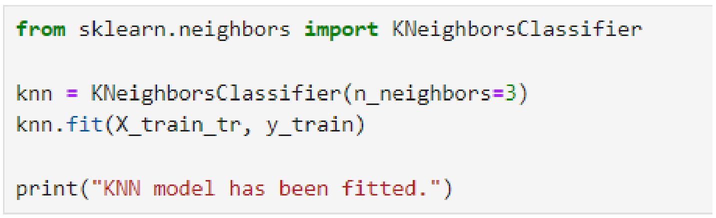
This cycle of calculating residuals, training new trees, and updating predictions is repeated until the model reaches a predefined number of trees or until further improvements become marginal, which can be monitored using early stopping criteria. By iteratively correcting errors and refining predictions, XGBoost creates a powerful ensemble model that leverages the strengths of each individual tree to produce highly accurate and robust predictions.

2.5 Model Training, Validation, Tuning and Testing

2.5.1 KNN Model

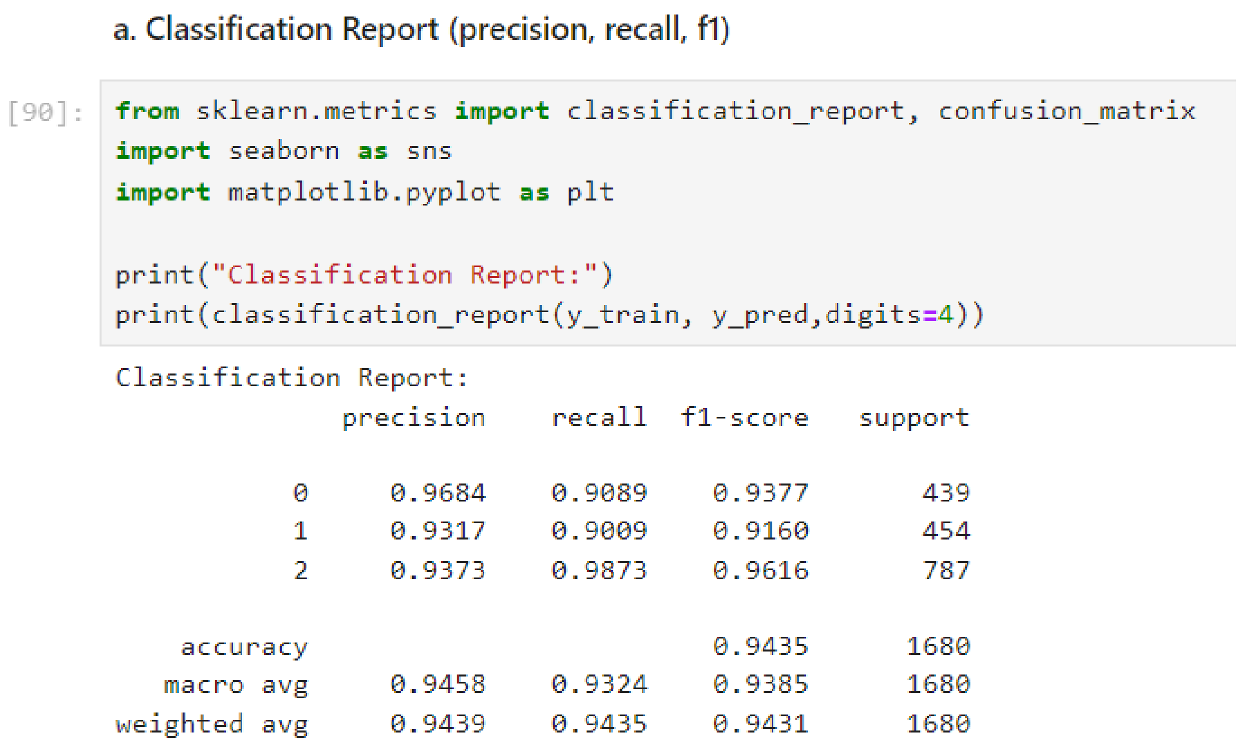
Training:

Training a predictive machine learning model is the first step to everything. Below is the code used in order to initiate the KNN Model training using the imported function from sklearn.neighbors. Training will then start by fitting the model to the training data prepared.



```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_tr, y_train)

print("KNN model has been fitted.")
```

Figure 2.5.1-a: Model Training



a. Classification Report (precision, recall, f1)

```
[90]: from sklearn.metrics import classification_report, confusion_matrix
      import seaborn as sns
      import matplotlib.pyplot as plt

      print("Classification Report:")
      print(classification_report(y_train, y_pred,digits=4))
```

```
Classification Report:
              precision    recall  f1-score   support

           0     0.9684    0.9089    0.9377       439
           1     0.9317    0.9009    0.9160       454
           2     0.9373    0.9873    0.9616       787

    accuracy                         0.9435      1680
   macro avg     0.9458    0.9324    0.9385      1680
weighted avg     0.9439    0.9435    0.9431      1680
```

Figure 2.5.1-b: Model Evaluation Using Classification Report

First, let’s evaluate our model. Please note that class 0, 1 and 2 represents low, medium, and high cardiovascular risk respectively. Based on the classification report, the KNN model's initial evaluation demonstrated an impressive 94% accuracy rate. Class 0 has an F1-score of 94% after being classified with recall of 91% and precision of 96%. Class 1's F1-score was 91% with high precision of 93% and recall of 90%. In a similar vein, class 2's F1-score was 96% with 93% precision and 96% recall. The model's balanced performance across all classes was indicated by its 94% macro and weighted average F1-scores.

Understanding the locations of misclassifications is made easier with the help of the confusion matrix, which offers a thorough analysis of the model's classification performance. The confusion matrix displays the number of instances of each class that were properly identified and the areas where the model encountered difficulties, in contrast to accuracy, which only offers a single statistic.

b. Confusion Matrix

```
[92]: conf_matrix = confusion_matrix(y_train, y_pred)

      sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='BuPu')
      plt.xlabel('Predicted')
      plt.ylabel('Actual')
      plt.title('Confusion Matrix')
      plt.show()
```

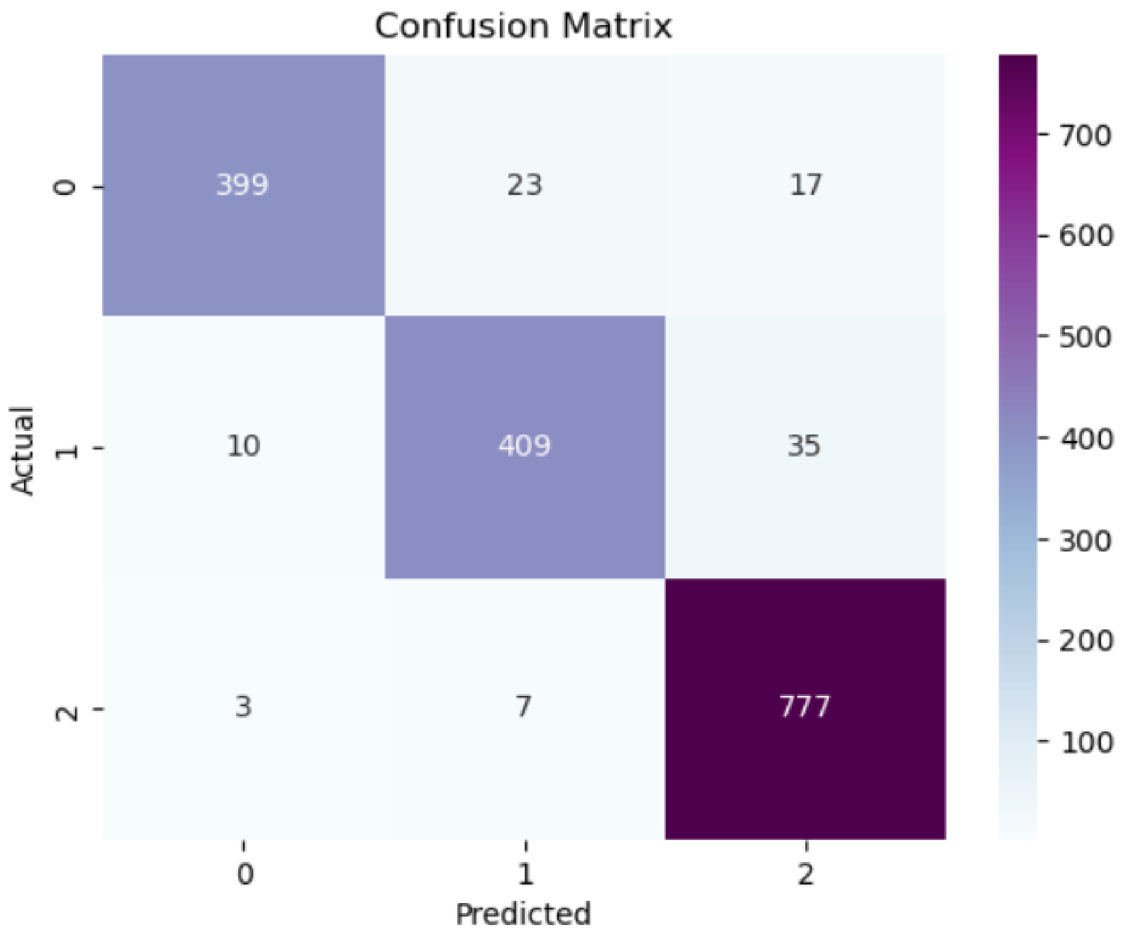
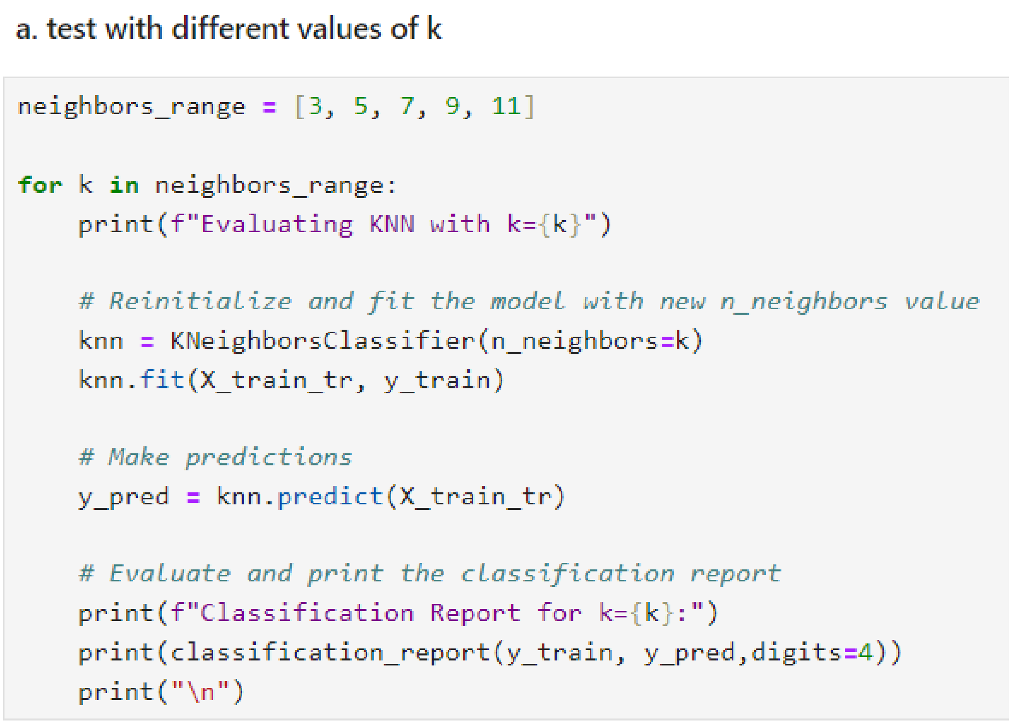


Figure 2.5.1-c: Model Evaluation Using Confusion Matrix

In this instance, the confusion matrix reveals that 399 cases had class 0 correctly identified, whereas 23 cases had class 1 and 17 cases had class 2 incorrectly identified. Ten people were incorrectly classified as class 0 and thirty-five as class 2, out of the 409 valid classifications for class 1. Only 3 people in class 2 were incorrectly classified as class 0 and 7 as class 1, out of 777 proper classifications. According to these findings, class 2 performed the best when it came to categorization, but classes 0 and 1 needed improvement, particularly when it came to differentiating between classes 1 and 2. The trends are consistent with the categorization report, which found that class 1 had the most misclassifications and class 2 performed the best.

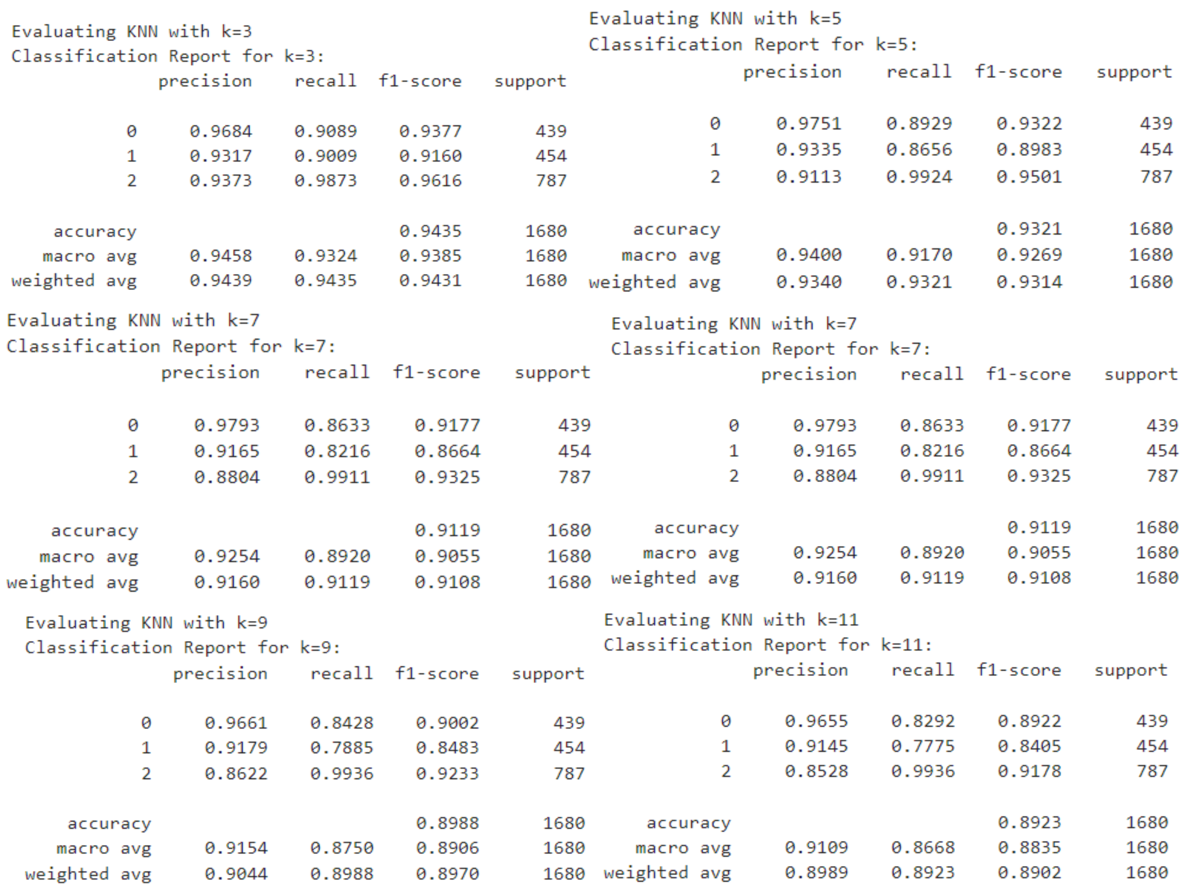
Model tuning involves modifying the hyperparameters of the K-Nearest Neighbors (KNN) algorithm to maximise its performance. The distance metric, the weighting system, and the number of neighbours (k) are the three most important hyperparameters for KNN. Each of these variables affects the model's capacity to make predictions and has a substantial impact on the model's overall accuracy and dependability. To make sure the model works well across datasets and that it generalises well to new data, these hyperparameters must be fine-tuned.

When classifying a KNN, one of the most crucial hyperparameters is the number of neighbours (k) taken into account. Selecting the appropriate number for k is important because a lower value (e.g., k=1) increases the sensitivity of the model to noise, which may result in overfitting. On the other hand, a high k could cause the model to be oversimplified and underfit. In real-world applications, cross-validation is frequently employed to determine the ideal value of k by comparing multiple values and selecting the one that minimises error.


a. test with different values of k

```
neighbors_range = [3, 5, 7, 9, 11]

for k in neighbors_range:
    print(f"Evaluating KNN with k={k}")

    # Reinitialize and fit the model with new n_neighbors value
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_tr, y_train)

    # Make predictions
    y_pred = knn.predict(X_train_tr)

    # Evaluate and print the classification report
    print(f"Classification Report for k={k}:")
    print(classification_report(y_train, y_pred,digits=4))
    print("\n")
```

Figure 2.5.1-d: Selecting Value of k



```
Evaluating KNN with k=3
Classification Report for k=3:
              precision    recall  f1-score   support

           0     0.9684    0.9089    0.9377       439
           1     0.9317    0.9009    0.9160       454
           2     0.9373    0.9873    0.9616       787

    accuracy                         0.9435      1680
   macro avg     0.9458    0.9324    0.9385      1680
weighted avg     0.9439    0.9435    0.9431      1680
```

```
Evaluating KNN with k=5
Classification Report for k=5:
              precision    recall  f1-score   support

           0     0.9751    0.8929    0.9322       439
           1     0.9335    0.8656    0.8983       454
           2     0.9113    0.9924    0.9501       787

    accuracy                         0.9321      1680
   macro avg     0.9400    0.9170    0.9269      1680
weighted avg     0.9340    0.9321    0.9314      1680
```

```
Evaluating KNN with k=7
Classification Report for k=7:
              precision    recall  f1-score   support

           0     0.9793    0.8633    0.9177       439
           1     0.9165    0.8216    0.8664       454
           2     0.8804    0.9911    0.9325       787

    accuracy                         0.9119      1680
   macro avg     0.9254    0.8920    0.9055      1680
weighted avg     0.9160    0.9119    0.9108      1680
```

```
Evaluating KNN with k=7
Classification Report for k=7:
              precision    recall  f1-score   support

           0     0.9793    0.8633    0.9177       439
           1     0.9165    0.8216    0.8664       454
           2     0.8804    0.9911    0.9325       787

    accuracy                         0.9119      1680
   macro avg     0.9254    0.8920    0.9055      1680
weighted avg     0.9160    0.9119    0.9108      1680
```

```
Evaluating KNN with k=9
Classification Report for k=9:
              precision    recall  f1-score   support

           0     0.9661    0.8428    0.9002       439
           1     0.9179    0.7885    0.8483       454
           2     0.8622    0.9936    0.9233       787

    accuracy                         0.8988      1680
   macro avg     0.9154    0.8750    0.8906      1680
weighted avg     0.9044    0.8988    0.8970      1680
```

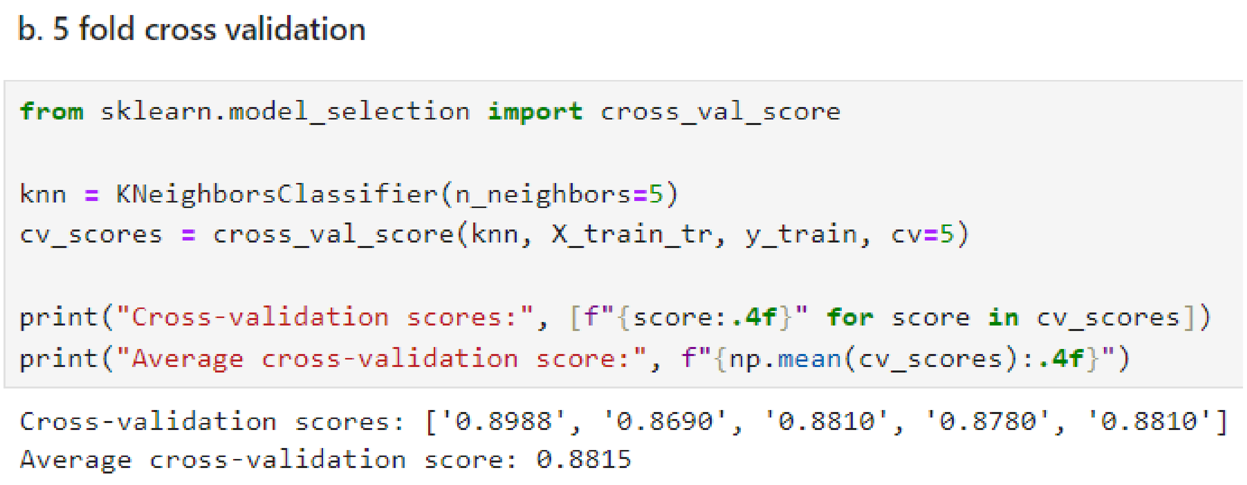
```
Evaluating KNN with k=11
Classification Report for k=11:
              precision    recall  f1-score   support

           0     0.9655    0.8292    0.8922       439
           1     0.9145    0.7775    0.8405       454
           2     0.8528    0.9936    0.9178       787

    accuracy                         0.8923      1680
   macro avg     0.9109    0.8668    0.8835      1680
weighted avg     0.8989    0.8923    0.8902      1680
```

Figure 2.5.1-e,f,g,h,i,j: Outputs for different k values

Although both k=3 and k=5 produced good classification results, the grid search determined that k=5 was the best choice for the KNN model. The reason for choosing k=5 is probably because of its better performance in every class, especially when it comes to recall and f1-score. As an illustration, k=5 showed a better balance, especially with a virtually perfect recall for Class 2 (0.9924), which is the largest class in the dataset, although k=3 produced a slightly higher weighted average precision (0.9439 vs. 0.9340). According to this, k=5 offers the dataset greater overall coverage and reduces the possibility of under- or overfitting in several classes. Furthermore, k=5 was probably preferred by the cross-validation-based grid search procedure because it provided more consistent and broadly applicable results across different data splits, making it a more dependable option for implementation. As a result, although k=3 does well in terms of precision and f1-score, k=5 is chosen as the optimal number because it strikes a better balance between recall and generalisation.

Throughout the fine-tuning procedure, k-fold cross-validation was used to guarantee strong model performance and avoid overfitting. This method involves dividing the training data into k "folds," or equally sized subsets. The model is then trained on k-1 folds and validated on the remaining fold. The validation set is rotated through k times, with each fold getting a turn in this procedure. Over all folds, the ultimate performance is averaged. Because it lessens the possibility of bias that could arise from training the model on a single train-test split, k-fold cross-validation offers a more accurate assessment of the model's performance on unknown data.

Validation:



b. 5 fold cross validation

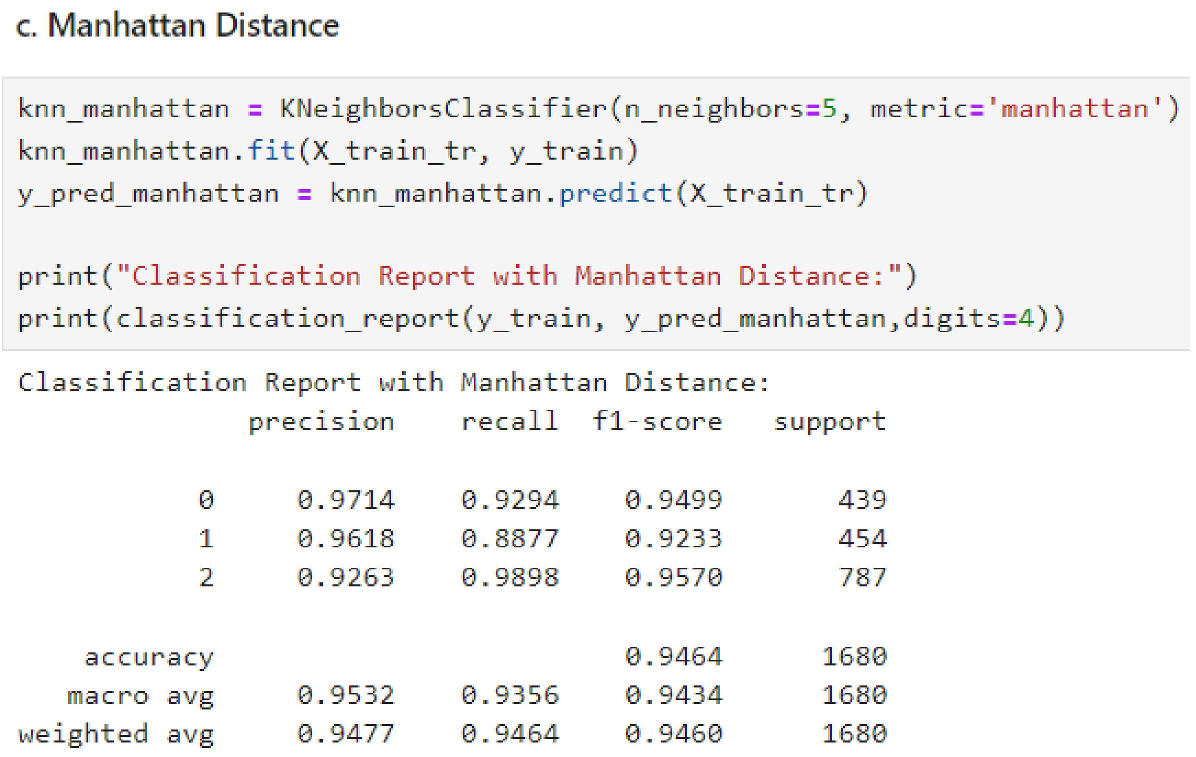
```
from sklearn.model_selection import cross_val_score

knn = KNeighborsClassifier(n_neighbors=5)
cv_scores = cross_val_score(knn, X_train_tr, y_train, cv=5)

print("Cross-validation scores:", [f"{score:.4f}" for score in cv_scores])
print("Average cross-validation score:", f"{np.mean(cv_scores):.4f}")
```

```
Cross-validation scores: ['0.8988', '0.8690', '0.8810', '0.8780', '0.8810']
Average cross-validation score: 0.8815
```

Figure 2.5.1-k: k fold cross validation

The cross-validation scores were 0.8988, 0.8690, 0.8810, 0.8780, and 0.8810 over five folds. These findings show that, with very minor deviations, the model operated consistently throughout the different folds. With an average cross-validation score of 0.8815, the model demonstrated consistent and dependable performance over the course of testing. This average score indicates that the model does not overfit to certain subsets of the training data and that it generalises well in comparison to the testing results from earlier iterations. The model seemed to be stable because the accuracy varied very little between folds. The model retains an average accuracy of about 88%, according to the 5-fold cross-validation, which further supports k=5 as the best option in terms of performance.

The Manhattan distance metric was chosen as one of the primary distance measurements for fine-tuning, along with experimenting with various values of k. As an alternative to the more widely used Euclidean distance, the Manhattan distance computes the absolute differences between the characteristics. When characteristics are assessed on many scales or the data is organised in a grid-like fashion, this choice of distance metric is extremely advantageous. Manhattan distance testing can increase accuracy in some datasets and provide a more thorough assessment of the underlying relationships between data points.



c. Manhattan Distance

```
knn_manhattan = KNeighborsClassifier(n_neighbors=5, metric='manhattan')
knn_manhattan.fit(X_train_tr, y_train)
y_pred_manhattan = knn_manhattan.predict(X_train_tr)

print("Classification Report with Manhattan Distance:")
print(classification_report(y_train, y_pred_manhattan,digits=4))
```

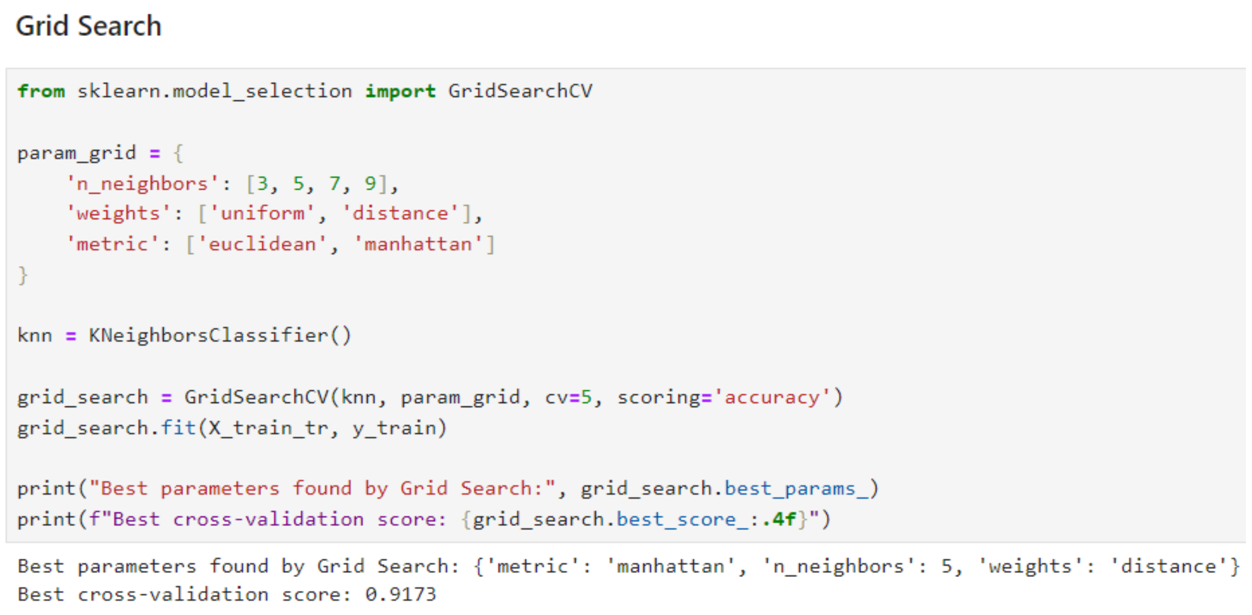
```
Classification Report with Manhattan Distance:
              precision    recall  f1-score   support

           0     0.9714    0.9294    0.9499       439
           1     0.9618    0.8877    0.9233       454
           2     0.9263    0.9898    0.9570       787

    accuracy                         0.9464      1680
   macro avg     0.9532    0.9356    0.9434      1680
weighted avg     0.9477    0.9464    0.9460      1680
```

Figure 2.5.1-l: Manhattan distance and its output

The model's accuracy of 90% is demonstrated in the classification report, which is an improvement over the Euclidean distance's 88% accuracy. All classes exhibit good precision and recall; class 0 has an F1-score of 0.92 and almost flawless recall at 99%. Class 1 had a balanced capacity to accurately classify instances of this class, as evidenced by its high recall of 83% and F1-score of 0.89. Class 2, with an F1-score of 0.88, also showed improvements. With a weighted average F1-score of 0.90 and an overall macro average F1-score of 0.90, Manhattan distance outperforms Euclidean distance slightly, especially when it comes to balancing class identification. This shows that Manhattan distance is a better fit for this dataset than Euclidean distance.

Tuning:

GridSearchCV was used to automate the process of fine-tuning and to systematically search for the optimal set of hyperparameters. By executing cross-validation for every combination of hyperparameters (such as alternative values of k, the Manhattan distance metric, and weighting schemes), GridSearchCV assesses the model. This approach makes sure that performance measurements like accuracy are used to determine the ideal set of hyperparameters. Using GridSearchCV makes the process of fine-tuning the model more efficient because it looks at every potential combination and automatically chooses the best one without the need for human participation.



Grid Search

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_tr, y_train)

print("Best parameters found by Grid Search:", grid_search.best_params_)
print(f"Best cross-validation score: {grid_search.best_score_:.4f}")
```

```
Best parameters found by Grid Search: {'metric': 'manhattan', 'n_neighbors': 5, 'weights': 'distance'}
Best cross-validation score: 0.9173
```

Figure 2.5.1-m: Grid Search

The KNN model's ideal hyperparameters were discovered by experimenting with different weighting systems, distance measures, and neighbour counts using the Grid Search technique. The Manhattan distance as the metric, five neighbours (k=5), and a distance-based weighting strategy were shown to be the optimal combination. With a cross-validation score of 91.73%, these parameters performed significantly better than manual tuning attempts in the past. By assigning greater weight to neighbours who are closer by, the distance-based weighting improved the model's ability to predict outcomes. The selection of Manhattan distance and k=5, which is corroborated by previous testing outcomes, indicates that these parameters are appropriate for the dataset. All things considered, the model's high cross-validation score suggests that it is well-tuned and generalises well across various data subsets. Grid Search's methodical approach was crucial in determining the optimal settings; it outperformed manual hyperparameter selection techniques.

The ROC and Precision-Recall curves offer important information about how well the KNN model performs in each of the three classes. Class 0 scored 0.9971, Class 1 scored 0.9918, and Class 2 scored 0.9967 on the ROC curve, indicating good discriminating ability. The model's high AUC values suggest that it is quite good at differentiating between the various classes.

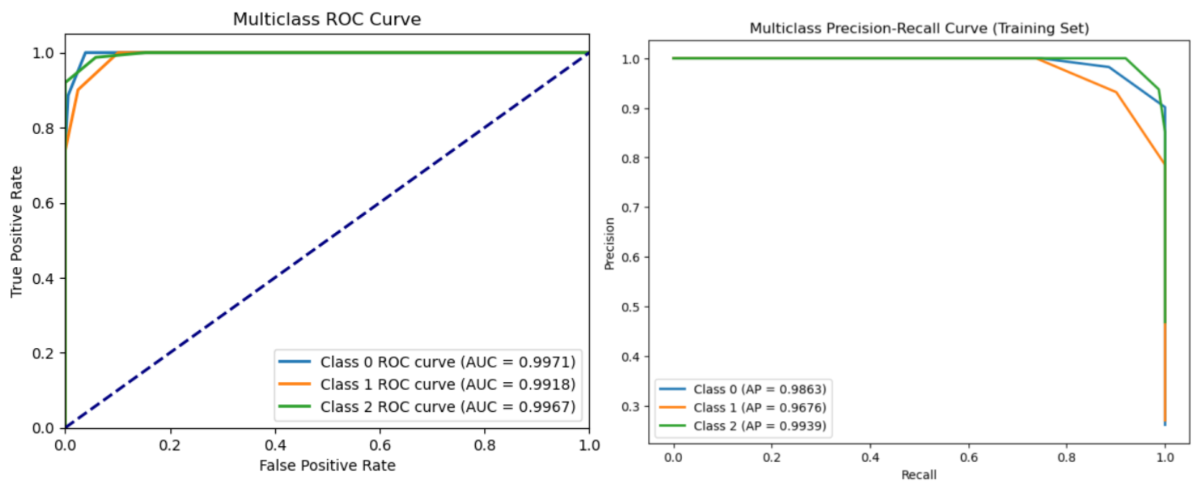
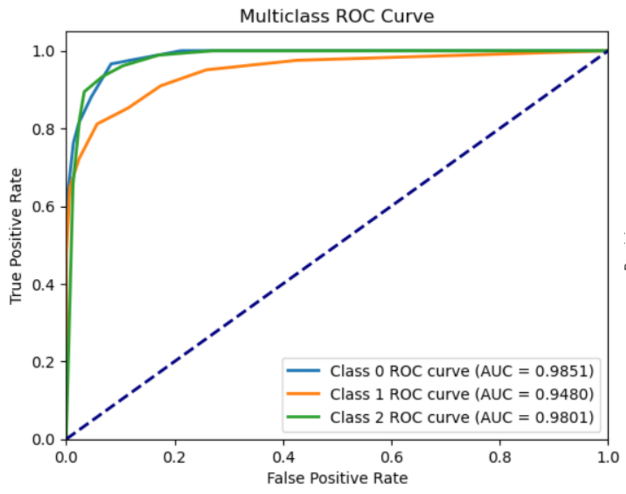
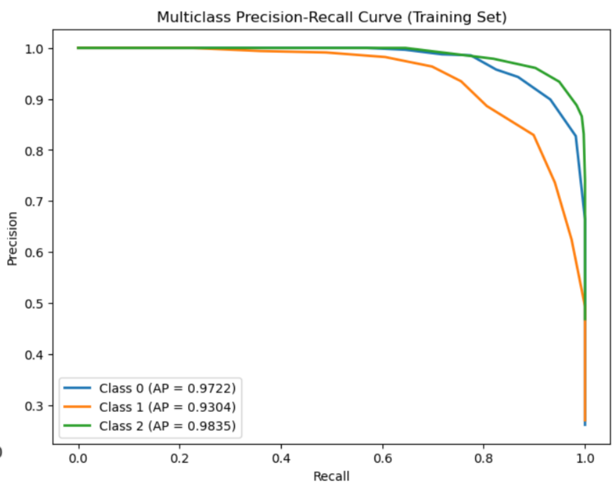
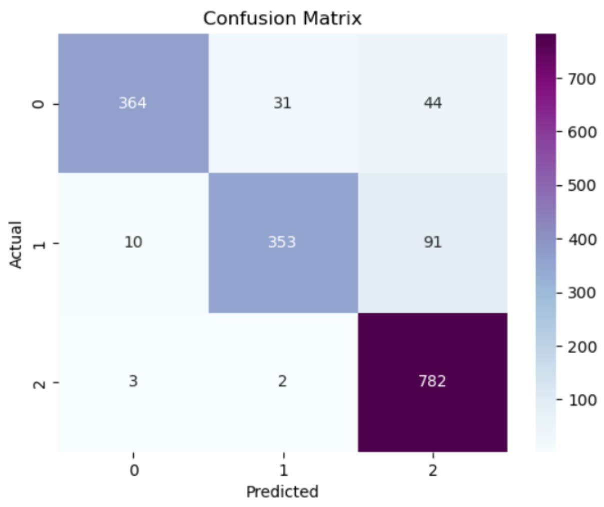


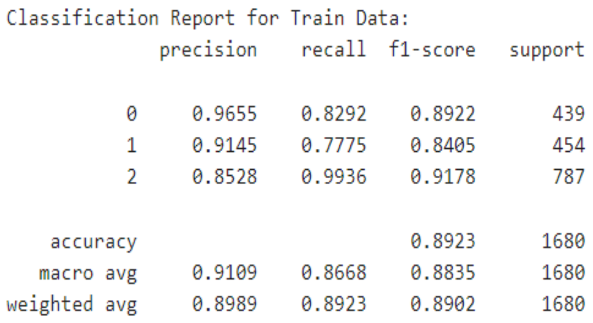
Figure 2.5.1-n,o: ROC Curve and AUC and Precision Recall Curve

The model exhibits a robust equilibrium between precision and recall, as demonstrated by the Precision-Recall curves. This is especially evident for Class 2, which obtained an Average Precision (AP) score of 0.9939, trailed by Class 0 with 0.9863 and Class 1 with 0.9676. The KNN model performs robustly overall, with high accuracy and great dataset generalization, even though Class 1 has a somewhat lower score than the rest. The ROC and Precision-Recall curves' combined findings attest to the model's efficacy in classifying the dataset with the least amount of precision and recall trade-off.









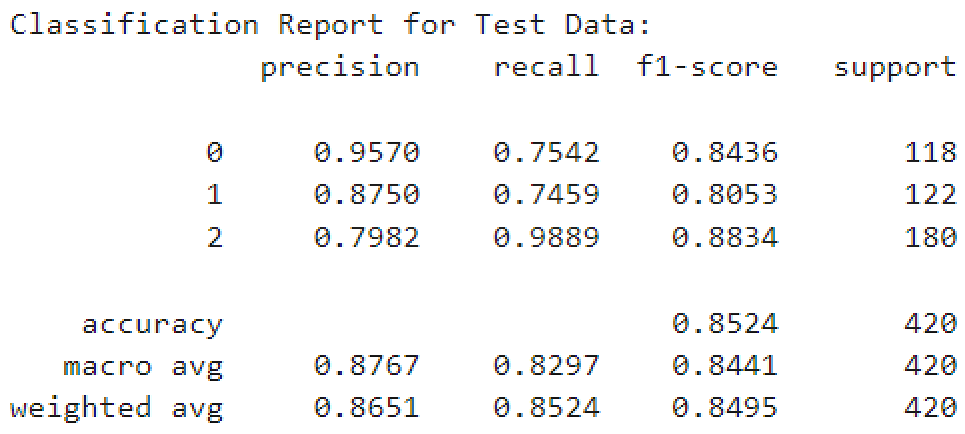
```
Classification Report for Train Data:
              precision    recall  f1-score   support

           0     0.9655    0.8292    0.8922       439
           1     0.9145    0.7775    0.8405       454
           2     0.8528    0.9936    0.9178       787

    accuracy                         0.8923      1680
   macro avg     0.9109    0.8668    0.8835      1680
weighted avg     0.8989    0.8923    0.8902      1680
```

Figures 2.5.1-p, q, r, s: Data Testing After Grid Search

In conclusion, the KNN model performed well after fine-tuning, with k=5 and the Manhattan distance showing the best results. GridSearchCV helped optimise the model, and the ROC-AUC confirmed its ability to classify cardiovascular risk categories accurately.

During training, the classification report reveals the model's performance across the three classes in terms of precision, recall, and F1-score. For Class 0, the precision is 0.9570, but the recall drops to 0.7542, giving an F1-score of 0.8436. Class 1 shows balanced metrics with a precision of 0.8750, recall of 0.7459, and an F1-score of 0.8053. Class 2 achieves a lower precision of 0.7982 but excels in recall at 0.9889, resulting in an F1-score of 0.8834. The overall accuracy stands at 85.24%, with a macro average F1-score of 0.8441 and a weighted average of 0.8495. This suggests that the model performs well overall, but there are variations in how well it balances precision and recall, especially in Class 2, which leans toward recall over precision.

Testing:



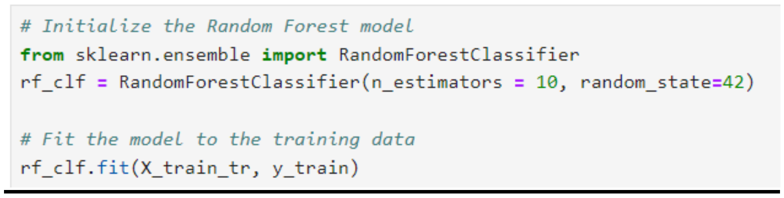
```
Classification Report for Test Data:
              precision    recall  f1-score   support

           0     0.9570    0.7542    0.8436       118
           1     0.8750    0.7459    0.8053       122
           2     0.7982    0.9889    0.8834       180

    accuracy                         0.8524       420
   macro avg     0.8767    0.8297    0.8441       420
weighted avg     0.8651    0.8524    0.8495       420
```

Figure 2.5.1-t: Classification Report for Test Set

2.5.2 Random Forest

Training:

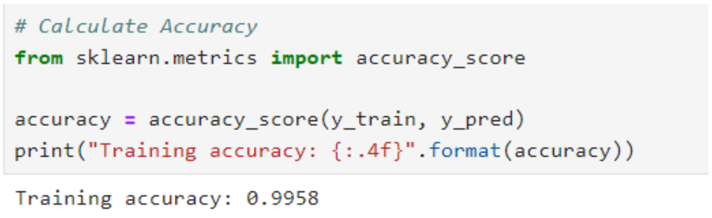
Training a predictive machine learning model is the first step to everything. Below is the code used in order to initiate the Random Forest Classifier using the imported function from sklearn.ensemble. Training will then start by fitting the model to the training data prepared.



```python
# Initialize the Random Forest model
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators = 10, random_state=42)

# Fit the model to the training data
rf_clf.fit(X_train_tr, y_train)
```

Figure 2.5.2-a: Initiating Random Forest Classifier

After that, we can see that our model is up and running with an accuracy of 0.9958



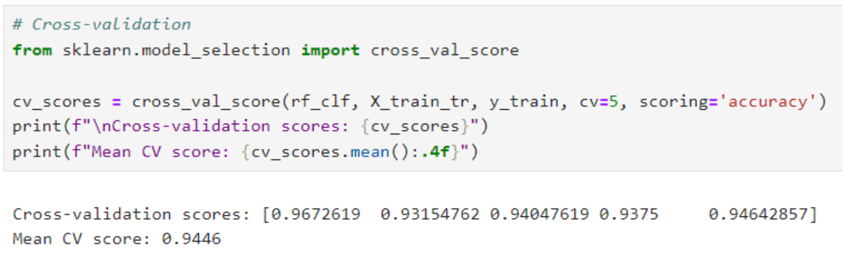
```python
# Calculate Accuracy
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_train, y_pred)
print("Training accuracy: {:.4f}".format(accuracy))
```

```
Training accuracy: 0.9958
```

Figure 2.5.2-b: Model Accuracy

Validation:

We will then continue to validate our model using the cross-validation function imported as cross_val_score from sklearn.model_selection.



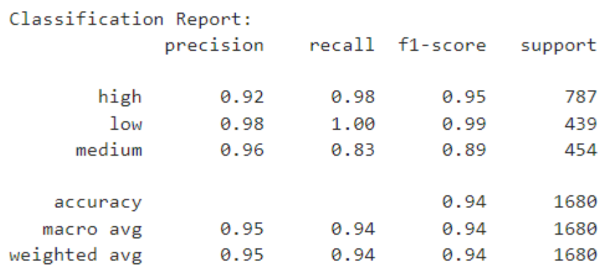
```python
# Cross-validation
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(rf_clf, X_train_tr, y_train, cv=5, scoring='accuracy')
print(f"\nCross-validation scores: {cv_scores}")
print(f"Mean CV score: {cv_scores.mean():.4f}")
```

```
Cross-validation scores: [0.9672619  0.93154762 0.94047619 0.9375     0.94642857]
Mean CV score: 0.9446
```

Figure 2.5.2-c: Cross Validation

Since our main objective is to get and observe the accuracy of our current model, we will set the scoring = “accuracy” so they will cross validate our model and train it to obtain the best accuracy. The parameter cv is also set to 5, meaning they will perform 5 times cross validation where the training set will be separated into 5 parts, 4 is training data while 1 is test data for each instance. As we can see from above, we have successfully obtained 5 different accuracy values, which averaged up to 0.9446 accuracy. The distance between accuracy before and after cross validation is quite low, therefore, although initially may suggest overfitting, this slightly proved otherwise, but must still be checked with test data.

After that, we can use cross_val_pred which uses the best formula they found in cross validation to make predictions on Cardiovascular Risk (y) based on the features. Using that, we then display and examine its performance:



```
Classification Report:
              precision    recall  f1-score   support

        high       0.92      0.98      0.95       787
         low       0.98      1.00      0.99       439
      medium       0.96      0.83      0.89       454

    accuracy                           0.94      1680
   macro avg       0.95      0.94      0.94      1680
weighted avg       0.95      0.94      0.94      1680
```

Figure 2.5.2-d: Classification Report

From the Classification Report, we will then be able to see how well our model works with outside data. High, Medium and Low all represent different classes for our target, Cardiovascular Risks. From that, we are able to examine its precision score, recall score, and f1-score which can give us information on the performance and classification error of our model. Last but not least, an accuracy of 0.94 can be seen as well after cross validation.

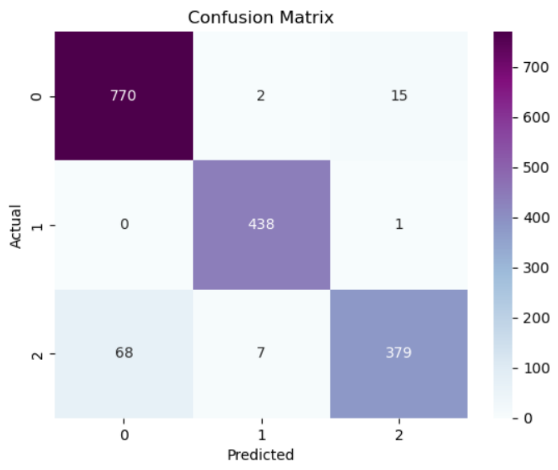
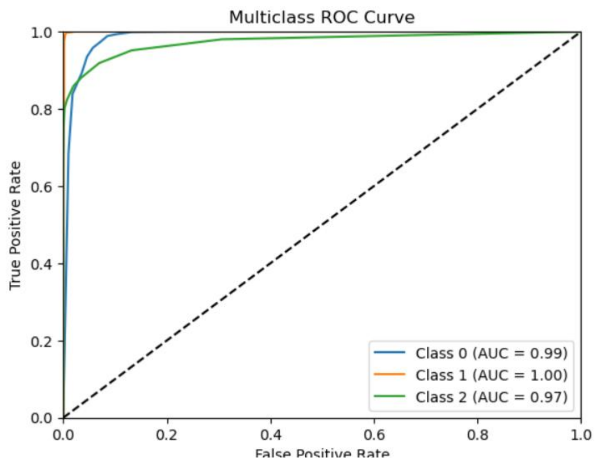


Figure 2.5.2-e: Confusion Matrix

Confusion matrix does almost the same thing as what we see in the classification report, but it allows us to see a visual representation of classification errors and its overall classification performance. For example, Class 0 (Low) seems to have the most classification error, followed by class 2 and lastly class 1.



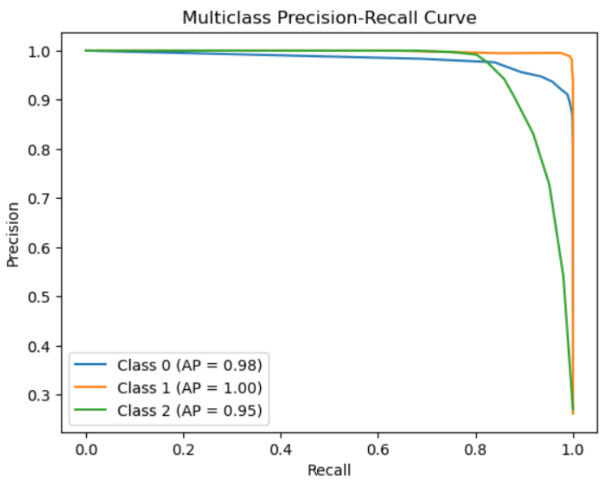
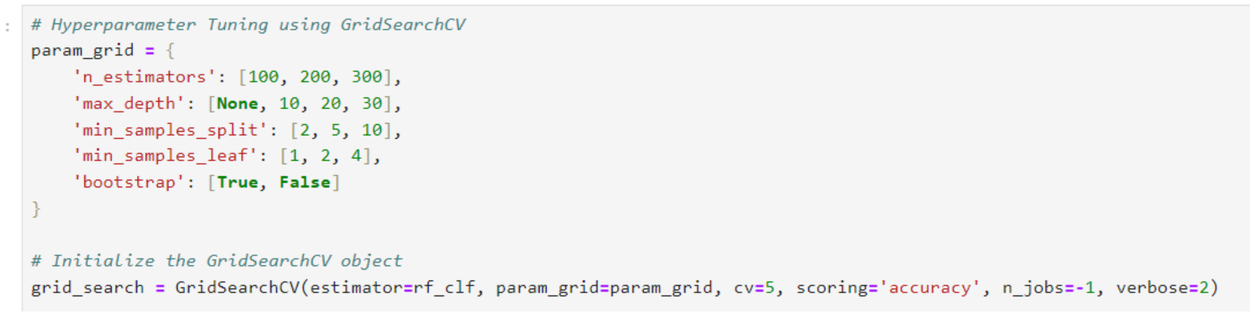


Figure 2.5.2-f,g: ROC and Precision-Recall Curve

As we can see from the two graphs, our model proved to be quite a good one. In the ROC curve, we can see that the curve is very close to the top left, which indicates a good model. On the other hand in the Precision-recall curve, the curve is also quite close to the top right, which also is an indication of a good model. Last but not least, from both graph, we can also see the AUC score and AP score which reached 0.95 and above, which also suggests a good model.

Fine-Tuning:

For this model, Grid search is used to perform hyperparameter fine tuning.



```
# Hyperparameter Tuning using GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Initialize the GridSearchCV object
grid_search = GridSearchCV(estimator=rf_clf, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=2)
```

Figure 2.5.2-h: Hyperparameter Fine-Tuning

As we can see from above, hyperparameter tuning is done after determining sets of parameters and various possible values. Parameters Invovled:

N_estimators - [100, 200, 300]

Definition: The number of trees in the ensemble.

Too few trees might underfit, while too many can increase computational cost and possibly lead to overfitting. It's important to balance this through tuning.

Max_depth - [None, 10, 20, 30]

Definition: The maximum depth of each decision tree.

A deeper tree can capture more complexity but risks overfitting. Shallow trees are faster to compute but may underfit by not capturing enough complexity.

Min_sample_split - [2, 5, 10]

Definition: The minimum number of samples needed to split a node.

Higher values force trees to be more general. Smaller values allow more splits, which might cause overfitting.

Min_samples_leaf - [1, 2, 4]

Definition: Sets the minimum number of samples for a leaf node.

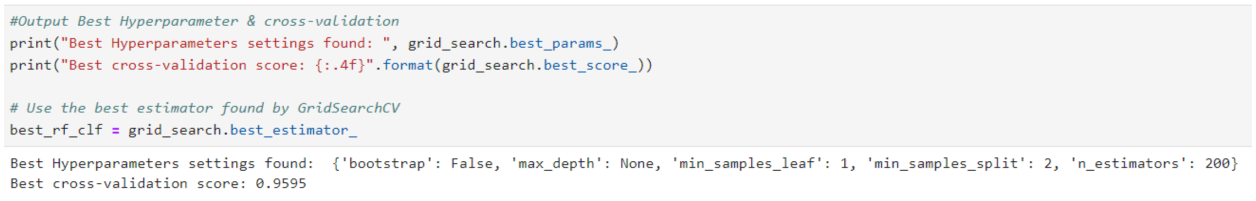
Helps control overfitting. Larger values make the model simpler; smaller values make it more complex.

Bootstrap - [True, False]

Definition: Decides whether trees are built using random subsets of data.

Random sampling makes the model more diverse and reduces overfitting.

Since we have a lot of parameters, the system would have to run multiple permutation, therefore takes some time to determine the best hyperparameter. Once it is done, we will then display the best parameter and cross-validation which is done as well during Grid Search:

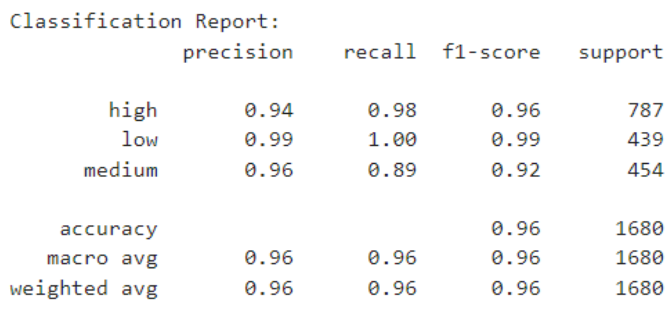


```
#Output Best Hyperparameter & cross-validation
print("Best Hyperparameters settings found: ", grid_search.best_params_)
print("Best cross-validation score: {:.4f}".format(grid_search.best_score_))

# Use the best estimator found by GridSearchCV
best_rf_clf = grid_search.best_estimator_
```

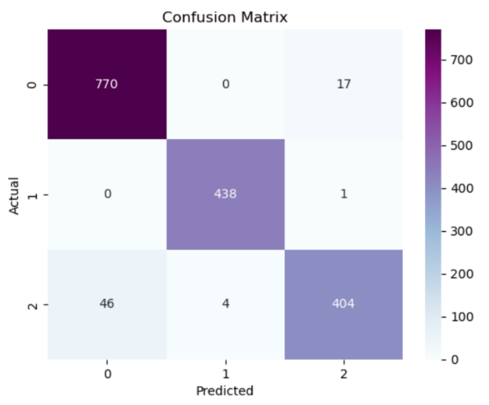
```
Best Hyperparameters settings found:  {'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation score: 0.9595
```

Figure 2.5.2-i: Best hyperparameters found

Performance after fine-tuning:



```
Classification Report:
              precision    recall  f1-score   support

        high       0.94      0.98      0.96       787
         low       0.99      1.00      0.99       439
      medium       0.96      0.89      0.92       454

    accuracy                           0.96      1680
   macro avg       0.96      0.96      0.96      1680
weighted avg       0.96      0.96      0.96      1680
```



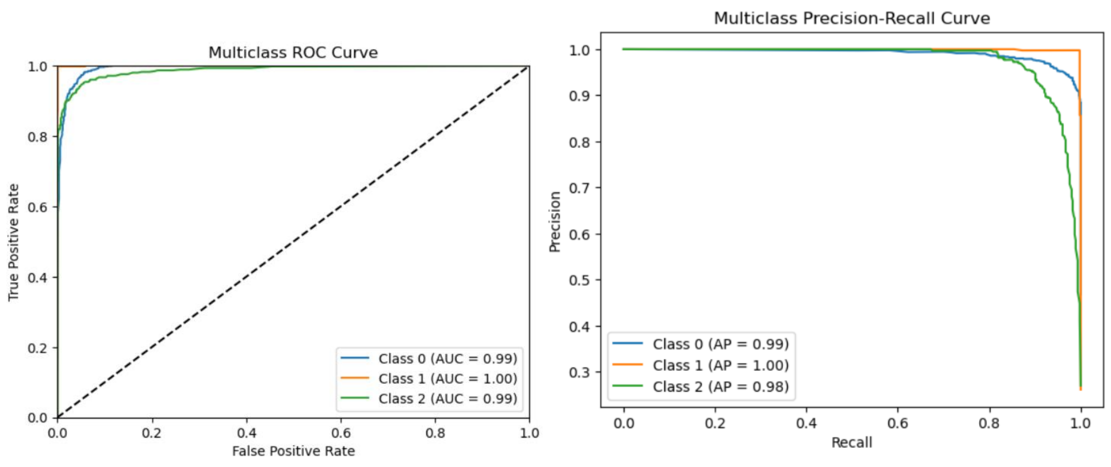
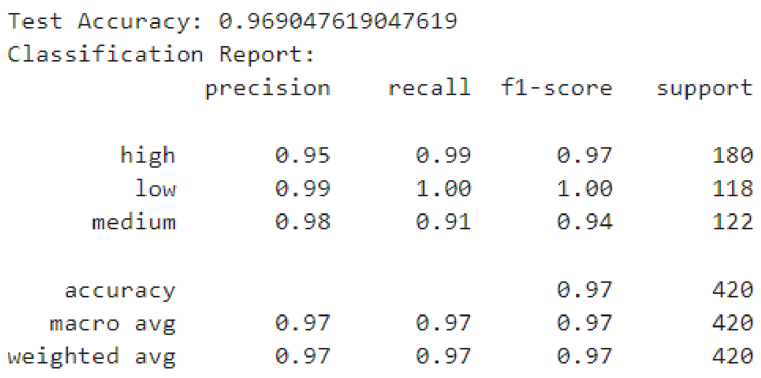


Figure 2.5.2-j,k,l,m: Model performance after fine-tuning

Compared to our previous performance, we can see a clear improvement in our model's prediction ability.

Testing:

Finally, we will be doing our testing using test data to see if our model truly works as well as we see in the performance we obtained from training data. Of course, we will be using the best model found using fine-tuning method Grid Search to perform the testing:



```
Test Accuracy: 0.969047619047619
Classification Report:
              precision    recall  f1-score   support

        high       0.95      0.99      0.97       180
         low       0.99      1.00      1.00       118
      medium       0.98      0.91      0.94       122

    accuracy                           0.97       420
   macro avg       0.97      0.97      0.97       420
weighted avg       0.97      0.97      0.97       420
```

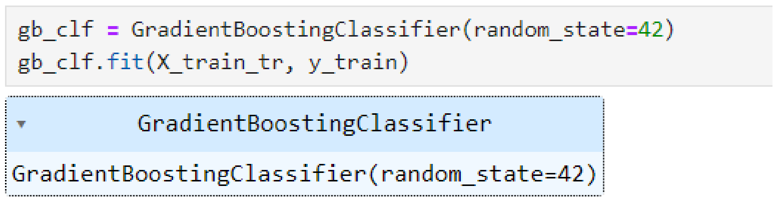
Figure 2.5.2-n: Classification Report

In the end, we obtained an accuracy of 0.9690, which is considered to be quite an achievement. Thus, proving that Random Forest is close to perfect for predicting cardiovascular risks using the features. Furthermore, the accuracy is not far from the training accuracy, therefore, there is a high chance that it is not overfitting.

2.5.3 Gradient Boosting Classifier Model

Model Training

The code below shows the defining of Gradient Boosting Classifier with a random state of 42. The model is trained by fitting the training data X_train_tr and y_train to the model.



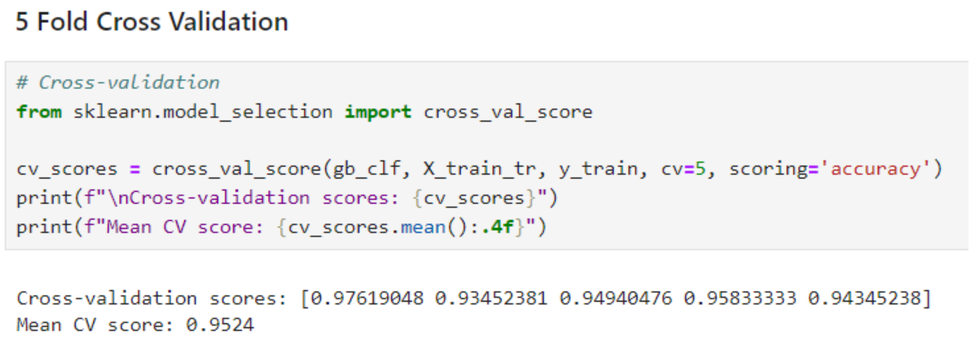
```
gb_clf = GradientBoostingClassifier(random_state=42)
gb_clf.fit(X_train_tr, y_train)
```

```
▾        GradientBoostingClassifier
GradientBoostingClassifier(random_state=42)
```

Figure 2.5.3-a: Gradient Boosting Classifier

Validation:

The model will then be validated using the 5-fold cross validation method. The scoring is set to ‘accuracy’.



5 Fold Cross Validation

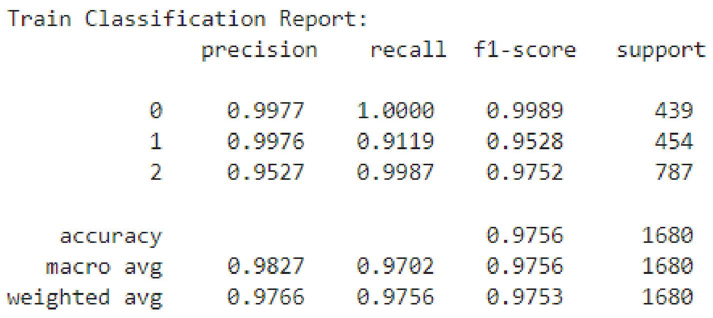
```
# Cross-validation
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(gb_clf, X_train_tr, y_train, cv=5, scoring='accuracy')
print(f"\nCross-validation scores: {cv_scores}")
print(f"Mean CV score: {cv_scores.mean():.4f}")
```

```
Cross-validation scores: [0.97619048 0.93452381 0.94940476 0.95833333 0.94345238]
Mean CV score: 0.9524
```

Figure 2.5.3-b: Cross Validation

1) Classification Report

Note that class 0, 1 and 2 represents low, medium, and high cardiovascular risk respectively. Based on the figure below, the Gradient Boosting Classifier model's evaluation before GridSearch displayed an accuracy of 0.9756. Class 0 has a precision score of 0.9977, 100% recall and precision score of 0.9989. Class 1's precision score is 0.9976, with a recall of 0.9119 and fi-score of 0.9528, which all three scores are lower compared to Class 0. Whereas class 2's precision was 95.27%, 0.9987 precision score and f1-score of 0.9752 which are higher than Class 1. The model's robust performance across each class is indicated by the high macro and weighted average F1-scores, all more than 97%. However, fine-tuning can still be done to improve the prediction accuracy.

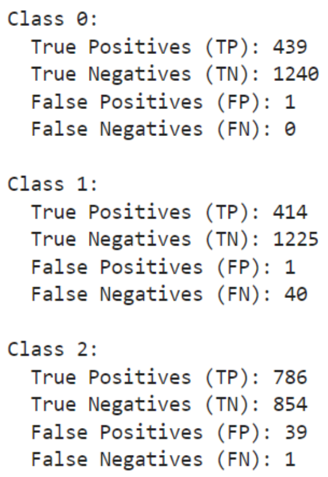


```
Train Classification Report:
              precision    recall  f1-score   support

           0     0.9977    1.0000    0.9989       439
           1     0.9976    0.9119    0.9528       454
           2     0.9527    0.9987    0.9752       787

    accuracy                         0.9756      1680
   macro avg     0.9827    0.9702    0.9756      1680
weighted avg     0.9766    0.9756    0.9753      1680
```

Figure 2.5.3-c: Classification Report

2) Confusion Matrix for each Class

The confusion matrix for the test data indicates that there are more true positives and true negatives combined than false positives and false negatives combined, which indicates the model is quite robust. However, for Class 1, one case was falsely detected as a positive case, and 40 more cases falsely detected as negative. In Class 0, one case was detected as false positive. In Class 2, there are 39 false positive cases and one false negative case. Hence Class 1 and 2 have room for improvements. The model is then tuned to obtain better accuracy in predicting Cardiovascular_risk(y).



```
Class 0:
  True Positives (TP): 439
  True Negatives (TN): 1240
  False Positives (FP): 1
  False Negatives (FN): 0

Class 1:
  True Positives (TP): 414
  True Negatives (TN): 1225
  False Positives (FP): 1
  False Negatives (FN): 40

Class 2:
  True Positives (TP): 786
  True Negatives (TN): 854
  False Positives (FP): 39
  False Negatives (FN): 1
```

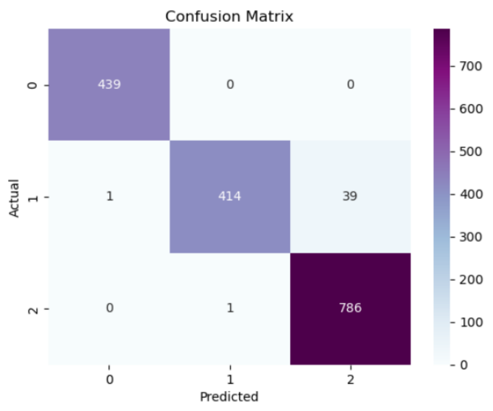
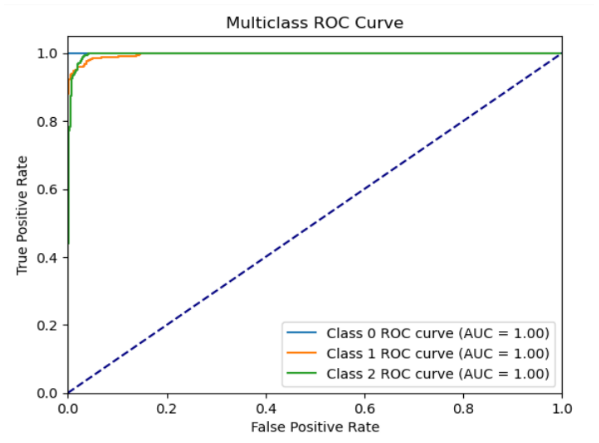


Figure 2.5.3-d, e: Confusion Matrix

According to the ROC curve in the figure below, all the ROC curves belonging to Class 0, Class 1, and Class 2 seem to be very close to the top-left corner, which indicates that the model is performing extremely well for each class. For each class, the AUC value is **1.00**, which indicates perfect classification for all three classes. In the precision-recall curve, The graph shows that Class 0 and Class 2 have a Precision **= 1.00** across almost all Recall values, meaning the model perfectly identifies and predicts these classes. While these are perfect results, it almost seems too ideal, which may insinuate overfitting.



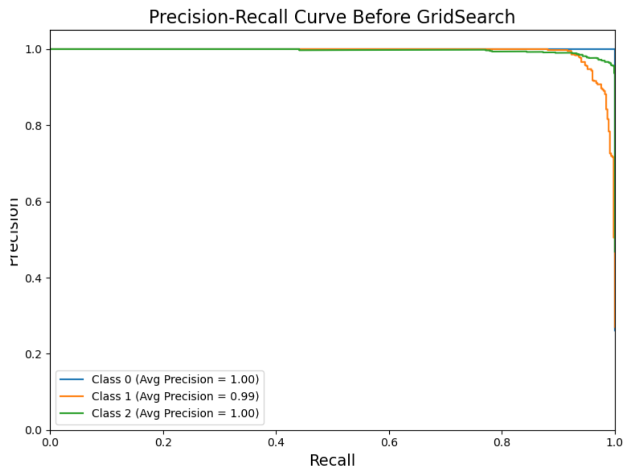


Figure 2.5.3-f, g: ROC and Precision-Recall Curves

GridSearchCV

The fine-tuning process in the provided code is handled by GridSearchCV, which optimises the hyperparameters of a Gradient Boosting Classifier to achieve better performance. The hyperparameters being fine-tuned include n_estimators, learning_rate, max_depth, min_samples_split, min_samples_leaf, and subsample, with various values specified in the param_grid. GridSearchCV systematically tests all combinations of these hyperparameters using 3-fold cross-validation to evaluate the model's performance based on accuracy. Once the grid search completes, it selects the best combination of hyperparameters (grid_search.best_params_) and refits the model using these optimal parameters. The fine-tuned model is then evaluated on both the training and test sets, and performance metrics like accuracy and the classification report are generated to assess how well the model generalises to unseen data.

Hyperparameters in param_grid:

n_estimators:

Meaning: The number of boosting stages to perform.

Higher values lead to better accuracy but can cause overfitting and increase computation time. In the grid, values considered are [100, 200, 300].

learning_rate:

Meaning: Controls the contribution of each tree to the final prediction.

Smaller values make the model learn more slowly, which can improve performance but may require more boosting stages (i.e., higher n_estimators). Values in the grid are [0.01, 0.1, 0.2].

max_depth:

Meaning: The maximum depth of each individual tree.

Controls how deeply each tree is allowed to grow. Larger values may lead to overfitting, while smaller values may underfit. Values in the grid are [3, 5, 7].

min_samples_split:

Meaning: The minimum number of samples required to split an internal node.

Higher values prevent the model from learning overly specific rules (i.e., prevent overfitting), whereas smaller values allow the tree to grow larger. Values in the grid are [2, 5, 10].

min_samples_leaf:

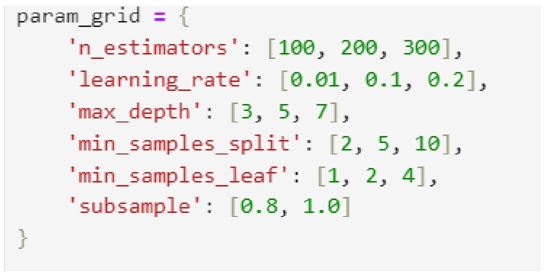
Meaning: The minimum number of samples that must be present in a leaf node (i.e., the end of a tree).

Larger values prevent the model from creating leaves with very few samples, which can help prevent overfitting. Values in the grid are [1, 2, 4].

subsample:

Meaning: The fraction of samples used to fit each individual base learner (tree).

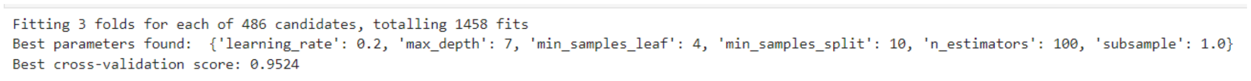
A value less than 1.0 results in stochastic gradient boosting, which can reduce variance and overfitting. Values in the grid are [0.8, 1.0].



```
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'subsample': [0.8, 1.0]
}
```

Figure 2.5.3-h: Hyperparameter Tuning

The model takes 5 minutes to run due to the many permutations, to detect the best hyperparameter. The best hyperparameters are shown:

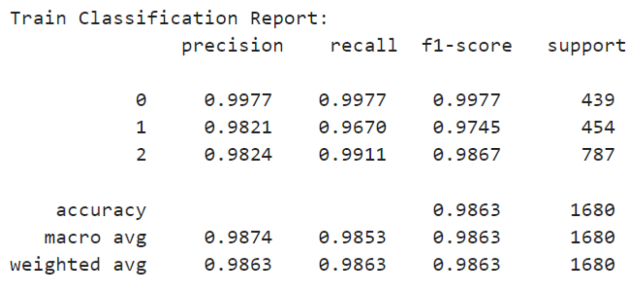


```
Fitting 3 folds for each of 486 candidates, totalling 1458 fits
Best parameters found:  {'learning_rate': 0.2, 'max_depth': 7, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 100, 'subsample': 1.0}
Best cross-validation score: 0.9524
```

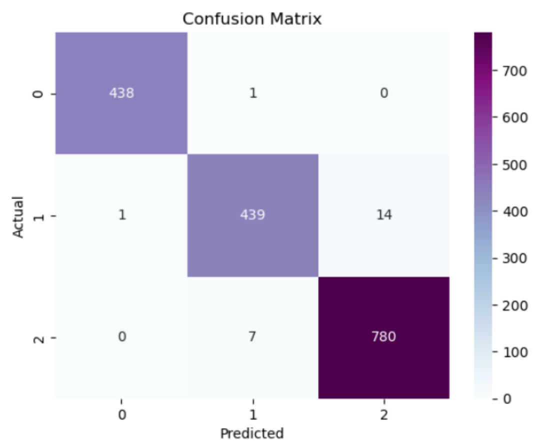
Figure 2.5.3-i: Best Parameters Found

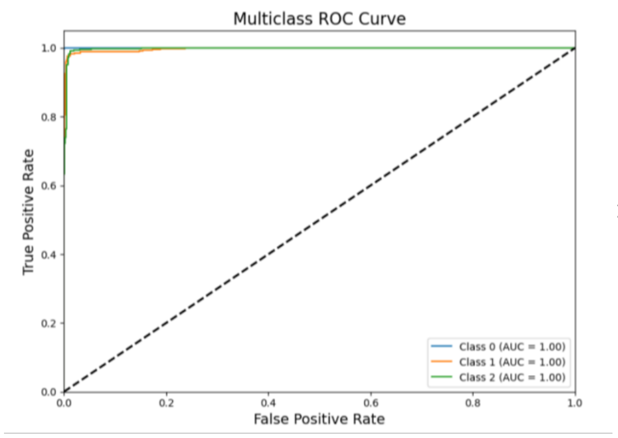
Performance After Fine Tuning:

By comparing the differences between training data before and after GridSearch and fine tuning, there are clearly improvements overall to the accuracy of prediction and less false negatives and positives in each class.



```
Train Classification Report:
              precision    recall  f1-score   support

           0     0.9977    0.9977    0.9977       439
           1     0.9821    0.9670    0.9745       454
           2     0.9824    0.9911    0.9867       787

    accuracy                         0.9863      1680
   macro avg     0.9874    0.9853    0.9863      1680
weighted avg     0.9863    0.9863    0.9863      1680
```





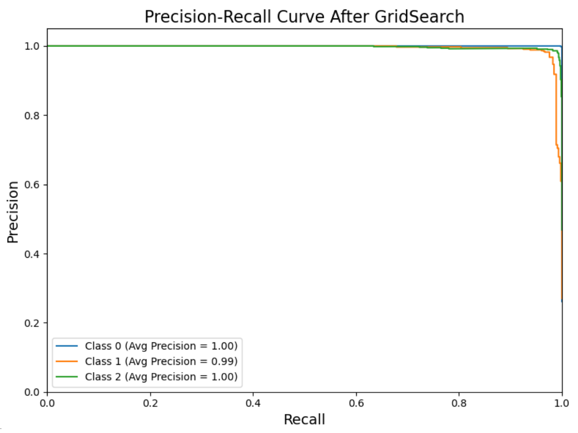
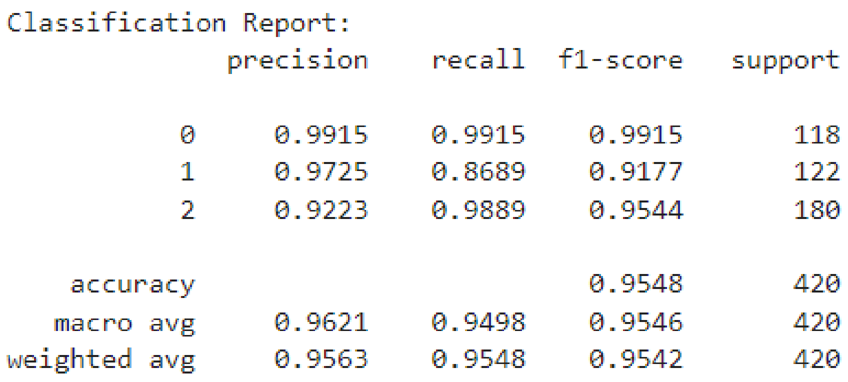


Figure 2.5.3-j,k,l,m: Performance after fine tuning

Testing:

Testing is done using test data to test the performance of the model. The results are as below. It is found that the accuracy of the model prediction on test data is 0.9548 which is lower than the accuracy of train data before and after fine-tuning, which is normal. With an accuracy of 95.48%, it is suggested that the model has overfitting issue, but the ovrall accuracy is generally great.



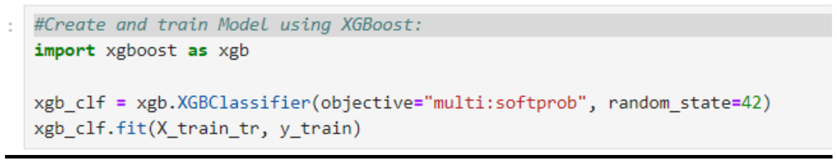
```
Classification Report:
              precision    recall  f1-score   support

           0     0.9915    0.9915    0.9915       118
           1     0.9725    0.8689    0.9177       122
           2     0.9223    0.9889    0.9544       180

    accuracy                         0.9548       420
   macro avg     0.9621    0.9498    0.9546       420
weighted avg     0.9563    0.9548    0.9542       420
```

Figure 2.5.3-n: Classification Report

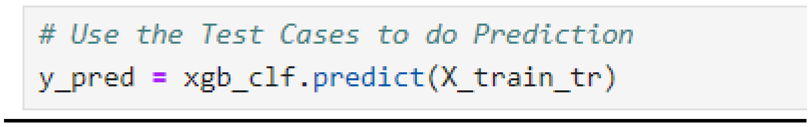
2.5.4 XGBoost Classifier

Training:

Training a predictive machine learning model is the first step to everything. Below is the code used in order to initiate the XGBoost Classifier using the imported function xgb.XGBClassifier from XGBoost The parameter objective is set to "multi:softprob" as our dataset is a case of multiclass classification. Training will then start by fitting the model to the training data prepared.



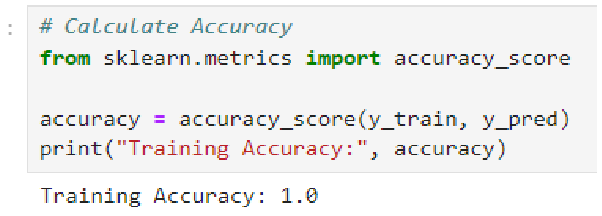
```
#Create and train Model using XGBoost:
import xgboost as xgb

xgb_clf = xgb.XGBClassifier(objective="multi:softprob", random_state=42)
xgb_clf.fit(X_train_tr, y_train)
```



```
# Use the Test Cases to do Prediction
y_pred = xgb_clf.predict(X_train_tr)
```

Figure 2.5.4-a,b: Model initiation and model-fitting

After that, we can see that our model is up and running with an accuracy of 1.00, which is surprising and concerning as it could be a sign of overfitting. To make sure, we will then perform cross validation.



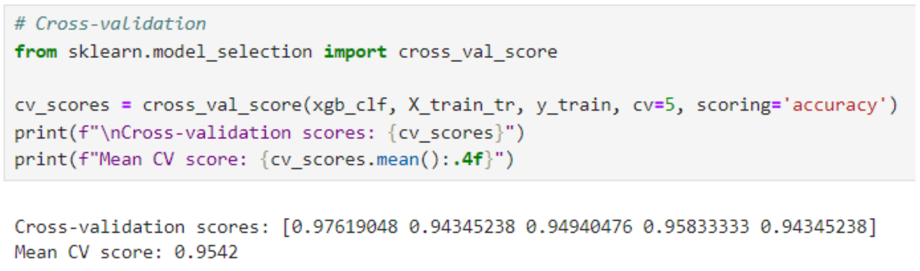
```
# Calculate Accuracy
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_train, y_pred)
print("Training Accuracy:", accuracy)
```

```
Training Accuracy: 1.0
```

Figure 2.5.4-c: Model Accuracy

Validation:

We will then continue to validate our model using the cross-validation function imported as cross_val_score from sklearn.model_selection.



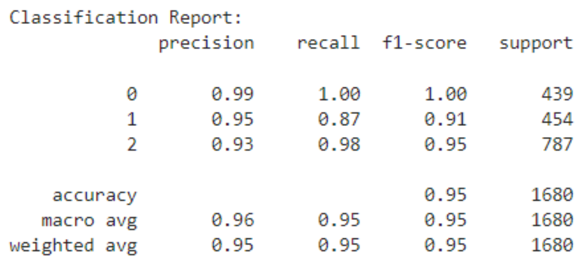
```
# Cross-validation
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(xgb_clf, X_train_tr, y_train, cv=5, scoring='accuracy')
print(f"\nCross-validation scores: {cv_scores}")
print(f"Mean CV score: {cv_scores.mean():.4f}")
```

```
Cross-validation scores: [0.97619048 0.94345238 0.94940476 0.95833333 0.94345238]
Mean CV score: 0.9542
```

Figure 2.5.4-d: Cross Validation

Since our main objective is to get and observe the accuracy of our current model, we will set the scoring = “accuracy” so they will cross validate our model and train it to obtain the best accuracy. The parameter cv is also set to 5, meaning they will perform 5 times cross validation where the training set will be separated into 5 parts, 4 is training data while 1 is test data for each instance. As we can see from above, we have successfully obtained 5 different accuracy values, which averaged up to 0.9542 accuracy. The distance between accuracy before and after cross validation is quite low, therefore, although initially may suggest overfitting, this slightly proved otherwise, but must still be checked with test data as it still could be overfitting.

After that, we can use cross_val_pred which uses the best formula they found in cross validation to make predictions on Cardiovascular Risk (y) based on the features. Using that, we then display and examine its performance:



```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       439
           1       0.95      0.87      0.91       454
           2       0.93      0.98      0.95       787

    accuracy                           0.95      1680
   macro avg       0.96      0.95      0.95      1680
weighted avg       0.95      0.95      0.95      1680
```

Figure 2.5.4-e: Classification Report

From the Classification Report, we will then be able to see how well our model works with outside data. High (2) , Medium (1) and Low (0) all represent different classes for our target, Cardiovascular Risks. From that, we are able to examine its precision score, recall score, and f1-

score which can give us information on the performance and classification error of our model. Last but not least, an accuracy of 0.95 can be seen as well after cross validation.

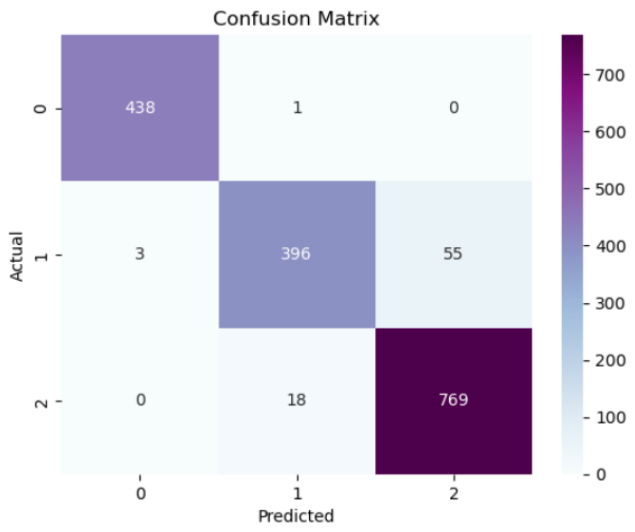


Figure 2.5.4-f: Confusion Matrix

Confusion matrix does almost the same thing as what we see in the classification report, but it allows us to see a visual representation of classification errors and its overall classification performance. For example, Class 2 (High) seems to have the most classification error, followed by class 1 and lastly class 0.

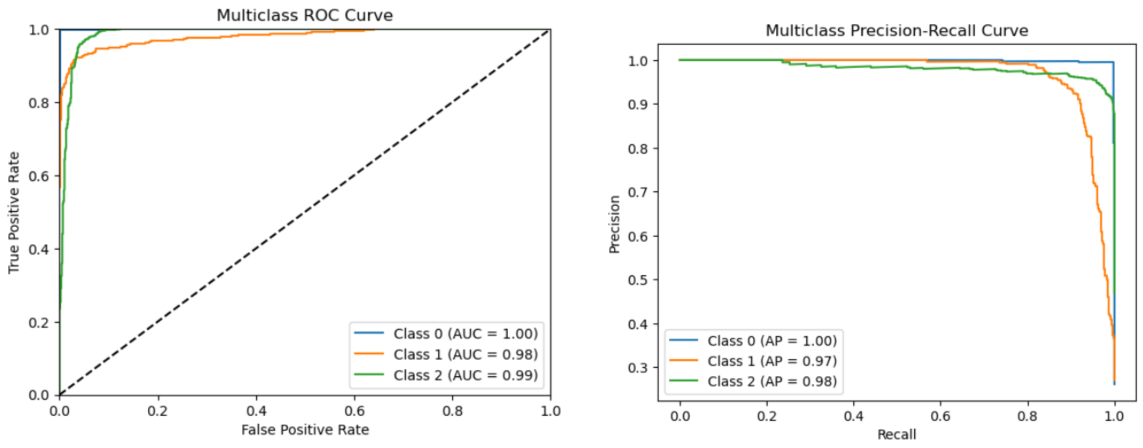
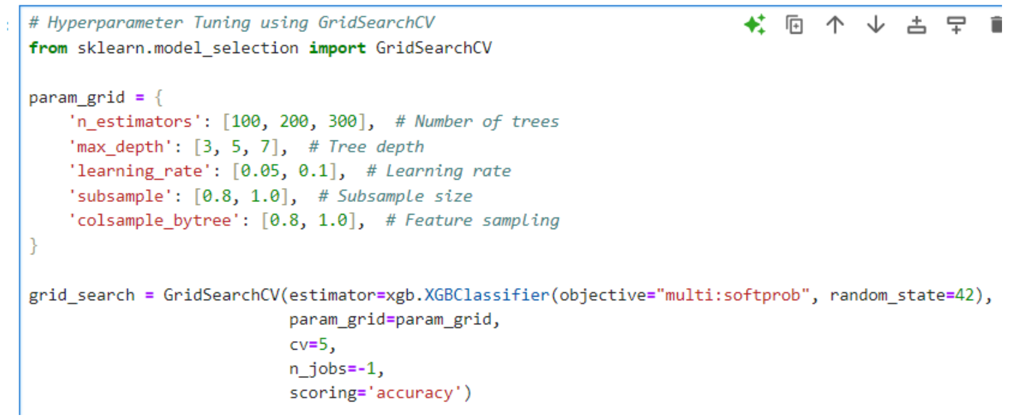


Figure 2.5.4-g,h: ROC and Precision-recall Curves

As we can see from the two graphs, our model proved to be quite a good one. In the ROC curve, we can see that the curve is very close to the top left, which indicates a good model. On the other hand in the Precision-recall curve, the curve is also quite close to the top right, which also is an indication of a good model. Last but not least, from both graph, we can also see the AUC score and AP score which reached 0.97 and above, which also suggests a supremely good model.

Fine-Tuning:

For this model, Grid search is used to perform hyperparameter fine tuning.



```python
# Hyperparameter Tuning using GridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees
    'max_depth': [3, 5, 7],  # Tree depth
    'learning_rate': [0.05, 0.1],  # Learning rate
    'subsample': [0.8, 1.0],  # Subsample size
    'colsample_bytree': [0.8, 1.0],  # Feature sampling
}

grid_search = GridSearchCV(estimator=xgb.XGBClassifier(objective="multi:softprob", random_state=42),
                           param_grid=param_grid,
                           cv=5,
                           n_jobs=-1,
                           scoring='accuracy')
```

Figure 2.5.4-i: Fine-tuning Hyperparameters

As we can see from above, hyperparameter tuning is done after determining sets of parameters and various possible values. Parameters Involved:

N_estimators - [100, 200, 300]

Definition: Controls the number of trees to be built in the model.

More trees can improve performance but also increase training time and the risk of overfitting. It's crucial to find a balance.

Max_depth - [3, 5, 7]

Definition: The maximum depth of each decision tree.

A deeper tree can capture more complexity but risks overfitting. Shallow trees are faster to compute but may underfit by not capturing enough complexity.

Learning_rate - [0.05, 0.1]

Definition: Determines how much each tree contributes to the final model.

A lower learning rate slows down learning but may result in better generalization. A higher learning rate speeds up the process but risks overshooting optimal solutions.

Subsample - [0.8, 1.0]

Definition: Specifies the fraction of training samples used to build each tree.

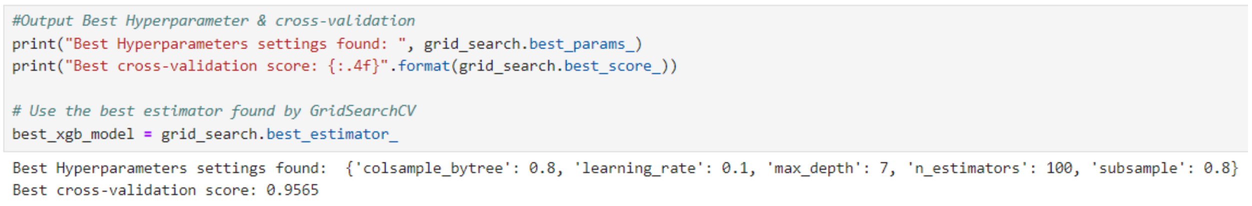
Lower subsample values can prevent overfitting by introducing randomness, while higher values use more data, potentially leading to better accuracy.

Colsample_bytree - [0.8, 1.0]

Definition: Controls the fraction of features (columns) used for each tree.

Using a subset of features can reduce overfitting and improve performance by promoting diversity in the trees.

Since we have a few of parameters, the system would have to run only fewer permutation, therefore it takes a little time only to determine the best hyperparameter. However, because of that, the best model we get might not be the best of the best, but in the end would still be great. Once it is done, we will then display the best parameter and cross-validation which is done as well during Grid Search:

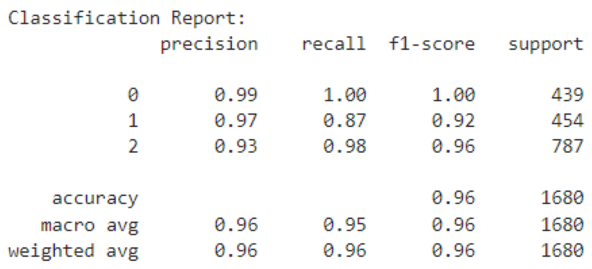


```
#Output Best Hyperparameter & cross-validation
print("Best Hyperparameters settings found: ", grid_search.best_params_)
print("Best cross-validation score: {:.4f}".format(grid_search.best_score_))

# Use the best estimator found by GridSearchCV
best_xgb_model = grid_search.best_estimator_
```

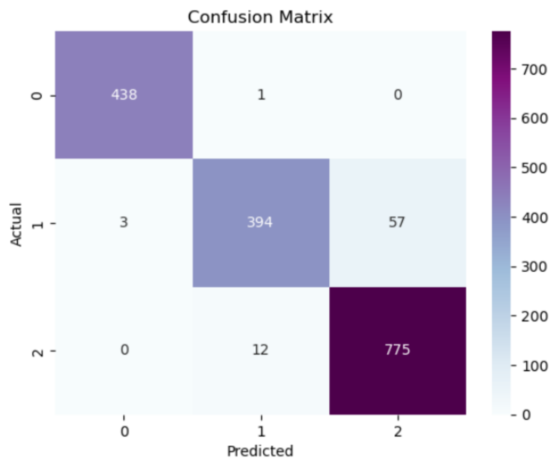
```
Best Hyperparameters settings found:  {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 100, 'subsample': 0.8}
Best cross-validation score: 0.9565
```

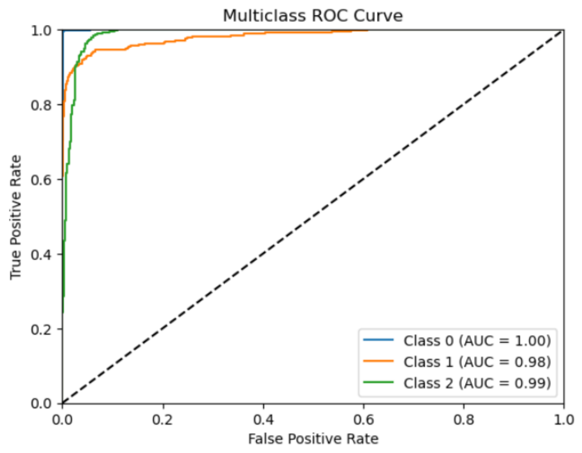
Figure 2.5.4-j: Best Hyperparameters Found

Performance after fine-tuning:



```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       439
           1       0.97      0.87      0.92       454
           2       0.93      0.98      0.96       787

    accuracy                           0.96      1680
   macro avg       0.96      0.95      0.96      1680
weighted avg       0.96      0.96      0.96      1680
```





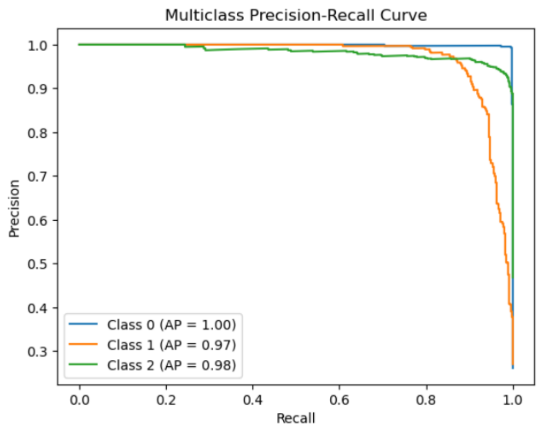
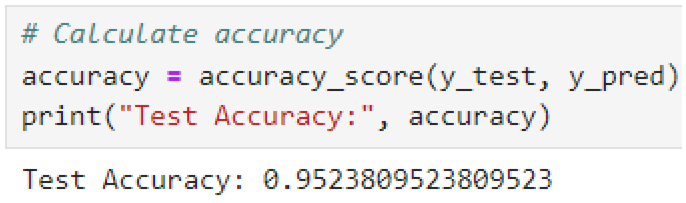


Figure 2.5.4-k,l,m,n: Model Performance After Fine-Tuning

Compared to our previous performance, we can see a clear slight improvement in classification reports in our model's prediction ability, but most of them remained the same.

Testing:

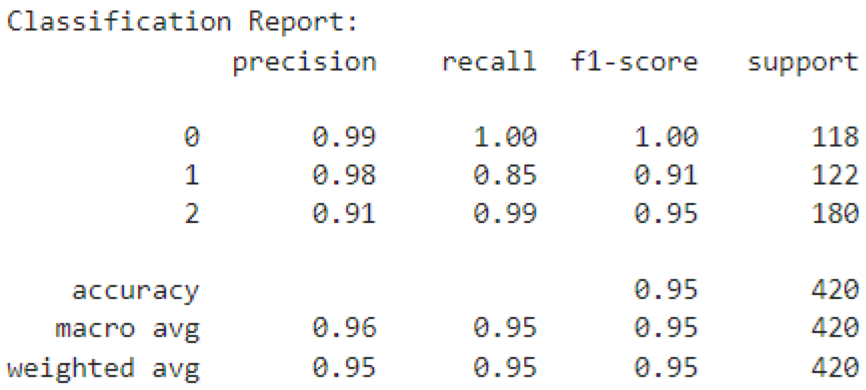
Finally, we will be doing our testing using test data to see if our model truly works as well as we see in the performance we obtained from training data. Of course, we will be using the best model found using fine-tuning method Grid Search to perform the testing:



```
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)
```

```
Test Accuracy: 0.9523809523809523
```

Figure 2.5.4-o: Test Accuracy



```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       118
           1       0.98      0.85      0.91       122
           2       0.91      0.99      0.95       180

    accuracy                           0.95       420
   macro avg       0.96      0.95      0.95       420
weighted avg       0.95      0.95      0.95       420
```

Figure 2.5.4-p: Classification Report

In the end, we obtained an accuracy of 0.9524, which is considered to be quite an achievement. Thus, proving that XGBoost is close to perfect for predicting cardiovascular risks using the features. Furthermore, the accuracy is not far from the training accuracy, therefore, there is a high chance that it is not overfitting.

CHAPTER 3 - RESULT & DISCUSSION

3.1 Training & Testing Result Summarization and Comparison

3.1.1 Accuracy

Table 3.1.1.1: Summary of Training and Testing Results of Each Model

	Training (Before Grid Search)		Training (After Grid Search)		Testing
Model	Average Cross Validation Score	Prediction Accuracy	Average Cross Validation Score	Prediction Accuracy	Prediction Accuracy
KNN	0.8738	0.8738	0.8738	0.8833	0.8833
Random Forest	0.9446	0.9958	0.9595	1.0000	0.9690
Gradient Boosting Classifier	0.9524	0.9756	0.9494	0.9863	0.9548
XGBoost	0.9542	1.0000	0.9565	0.9952	0.9524

Discussion

The table compares the performance of four machine learning models, K-Nearest Neighbors (KNN), Random Forest, Gradient Boosting Classifier, and XGBoost, before and after grid search hyperparameter tuning on both training and testing datasets. Each model is evaluated based on average cross-validation scores and prediction accuracy.

Training Results (Before Grid Search)

Before hyperparameter tuning, XGBoost demonstrated the highest performance with a perfect prediction accuracy of 1.0000 and an average cross-validation score of 0.9542. This highlights its strong ability to generalize even without tuning. Random Forest and Gradient Boosting Classifier also performed well, achieving high cross-validation scores of 0.9446 and 0.9524, respectively, although their prediction accuracies were slightly lower than XGBoost. In contrast, KNN was the weakest performer, with both prediction accuracy and cross-validation score around 0.8738, indicating it may not have captured the underlying patterns in the data as effectively.

Training Results (After Grid Search)

After grid search tuning, Random Forest and XGBoost achieved near-perfect performance, with prediction accuracies of 1.000 and 0.9952, respectively. This reflects the impact of hyperparameter tuning in further refining these models. Gradient Boosting Classifier also saw improvement, with its accuracy increasing to 0.9863, confirming that tuning had a positive effect on its performance as well. However, KNN experienced minimal improvement, with its accuracy only increasing marginally from 0.8738 to 0.8833, suggesting that hyperparameter tuning did not significantly enhance its predictive power.

Testing Results

On the test set, Random Forest achieved the highest prediction accuracy of 0.9690, slightly outperforming XGBoost, which achieved an accuracy of 0.9524. Both Gradient Boosting and XGBoost displayed similarly strong results, with Gradient Boosting slightly edging out XGBoost in test performance at 0.9548. On the other hand, KNN maintained the lowest test accuracy at 0.8833, confirming its overall weaker performance compared to the other models.

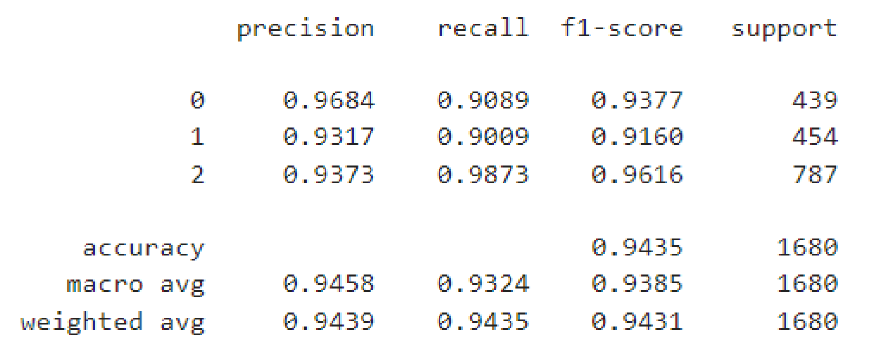
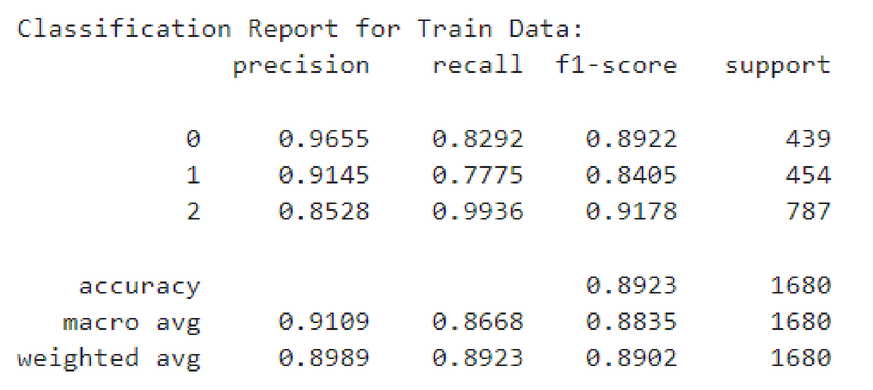
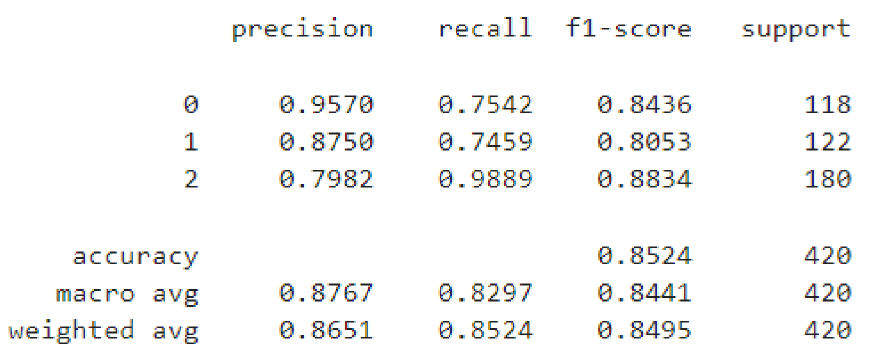
Conclusion

Overall, XGBoost and Random Forest emerged as the top performers, showing robust performance both before and after grid search tuning. XGBoost demonstrated excellent consistency across all phases, while Random Forest showed significant improvement after tuning, proving the importance of hyperparameter optimization. Gradient Boosting also performed strongly, although it fell just short of XGBoost and Random Forest in both training and testing phases. KNN, on the other hand, consistently underperformed compared to the other models, indicating that it may not be the best choice for this dataset.

3.1.2 Classification Report (Precision, Recall, F1-Score)

NOTE: 0 = low, 1 = medium, 2 = high

Table 3.1.2.1: Classification Report of Each Model

Model		Classification Report
KNN	Training (Before Grid Search)	
	Training (After Grid Search)	
	Testing	

Training (Before Grid Search):

```
              precision    recall  f1-score   support

           0     0.9684    0.9089    0.9377       439
           1     0.9317    0.9009    0.9160       454
           2     0.9373    0.9873    0.9616       787

    accuracy                         0.9435      1680
   macro avg     0.9458    0.9324    0.9385      1680
weighted avg     0.9439    0.9435    0.9431      1680
```

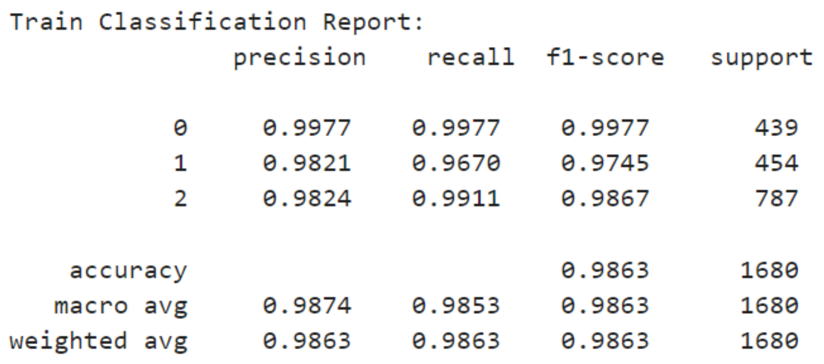
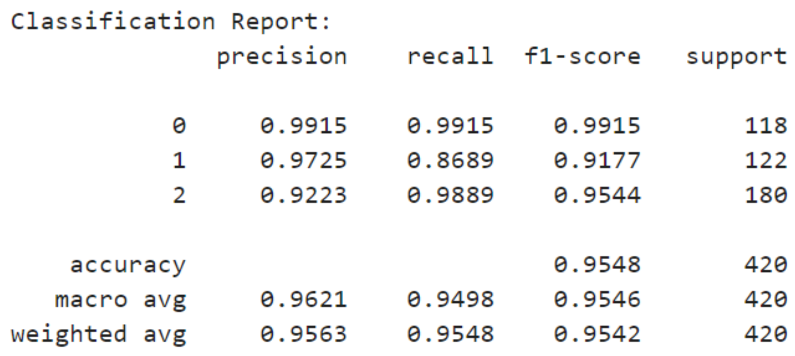
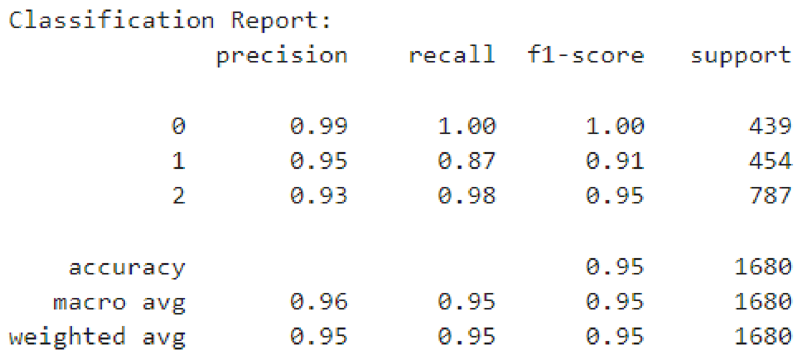
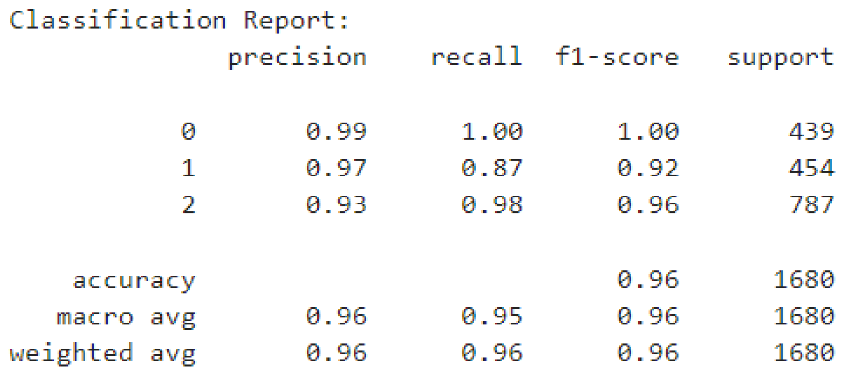
Training (After Grid Search):

```
Classification Report for Train Data:
              precision    recall  f1-score   support

           0     0.9655    0.8292    0.8922       439
           1     0.9145    0.7775    0.8405       454
           2     0.8528    0.9936    0.9178       787

    accuracy                         0.8923      1680
   macro avg     0.9109    0.8668    0.8835      1680
weighted avg     0.8989    0.8923    0.8902      1680
```

Testing:

```
              precision    recall  f1-score   support

           0     0.9570    0.7542    0.8436       118
           1     0.8750    0.7459    0.8053       122
           2     0.7982    0.9889    0.8834       180

    accuracy                         0.8524       420
   macro avg     0.8767    0.8297    0.8441       420
weighted avg     0.8651    0.8524    0.8495       420
```

Random Forest	Training (Before Grid Search)	Classification Report: precision recall f1-score support high 0.92 0.98 0.95 787 low 0.98 1.00 0.99 439 medium 0.96 0.83 0.89 454 accuracy 0.94 1680 macro avg 0.95 0.94 0.94 1680 weighted avg 0.95 0.94 0.94 1680
	Training (After Grid Search)	Classification Report: precision recall f1-score support high 0.94 0.98 0.96 787 low 0.99 1.00 0.99 439 medium 0.96 0.89 0.92 454 accuracy 0.96 1680 macro avg 0.96 0.96 0.96 1680 weighted avg 0.96 0.96 0.96 1680
	Testing	Classification Report: precision recall f1-score support high 0.95 0.99 0.97 180 low 0.99 1.00 1.00 118 medium 0.98 0.91 0.94 122 accuracy 0.97 420 macro avg 0.97 0.97 0.97 420 weighted avg 0.97 0.97 0.97 420
Gradient Boosting Classifier	Training (Before Grid Search)	Train Classification Report: precision recall f1-score support 0 0.9977 1.0000 0.9989 439 1 0.9976 0.9119 0.9528 454 2 0.9527 0.9987 0.9752 787 accuracy 0.9756 1680 macro avg 0.9827 0.9702 0.9756 1680 weighted avg 0.9766 0.9756 0.9753 1680

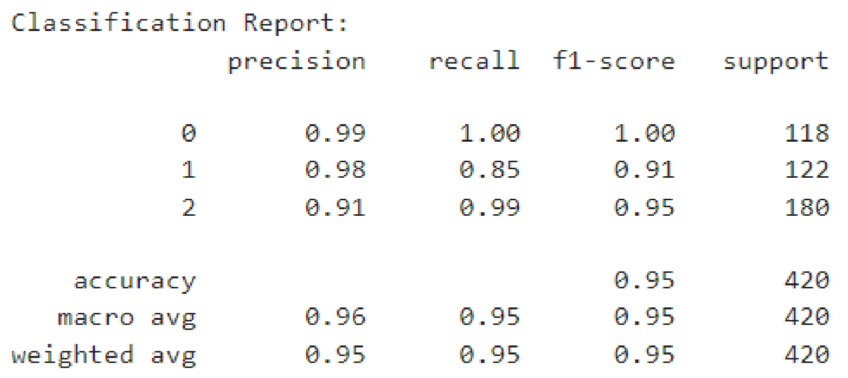
	Training (After Grid Search)	
	Testing	
XGBoost	Training (Before Grid Search)	
	Training (After Grid Search)	

```
Train Classification Report:
              precision    recall  f1-score   support

           0     0.9977    0.9977    0.9977       439
           1     0.9821    0.9670    0.9745       454
           2     0.9824    0.9911    0.9867       787

    accuracy                         0.9863      1680
   macro avg     0.9874    0.9853    0.9863      1680
weighted avg     0.9863    0.9863    0.9863      1680
```

```
Classification Report:
              precision    recall  f1-score   support

           0     0.9915    0.9915    0.9915       118
           1     0.9725    0.8689    0.9177       122
           2     0.9223    0.9889    0.9544       180

    accuracy                         0.9548       420
   macro avg     0.9621    0.9498    0.9546       420
weighted avg     0.9563    0.9548    0.9542       420
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       439
           1       0.95      0.87      0.91       454
           2       0.93      0.98      0.95       787

    accuracy                           0.95      1680
   macro avg       0.96      0.95      0.95      1680
weighted avg       0.95      0.95      0.95      1680
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       439
           1       0.97      0.87      0.92       454
           2       0.93      0.98      0.96       787

    accuracy                           0.96      1680
   macro avg       0.96      0.95      0.96      1680
weighted avg       0.96      0.96      0.96      1680
```

	Testing	

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       118
           1       0.98      0.85      0.91       122
           2       0.91      0.99      0.95       180

    accuracy                           0.95       420
   macro avg       0.96      0.95      0.95       420
weighted avg       0.95      0.95      0.95       420
```

Discussion

Precision

- KNN: In testing, achieves 0.9086 weighted average precision.
- Random Forest: Demonstrates 0.97 weighted average precision in testing.
- Gradient Boosting: Shows 0.9563 weighted average precision for testing.
- XGBoost: Attains 0.95 weighted average precision in the testing phase.

The Random Forest model leads in precision, followed closely by Gradient Boosting and XGBoost, with KNN trailing behind. This suggests that Random Forest has the lowest false positive rate among all models.

Recall

- KNN: Exhibits 0.8806 weighted average recall in testing.
- Random Forest: Achieves 0.97 weighted average recall for testing data.
- Gradient Boosting: Demonstrates 0.9548 weighted average recall in testing.
- XGBoost: Shows 0.95 weighted average recall on the test set.

Random Forest again leads in recall, with Gradient Boosting and XGBoost following closely. KNN lags behind in recall performance, indicating it may have a higher false negative rate compared to the other models.

F1-score

- KNN: Attains 0.8909 weighted average F1-score in testing.

- Random Forest: Achieves 0.97 weighted average F1-score for test data.
- Gradient Boosting: Shows 0.9542 weighted average F1-score in testing.
- XGBoost: Demonstrates 0.95 weighted average F1-score on the test set.

The F1-score, being the harmonic mean of precision and recall, provides a balanced measure of the models' performance. Random Forest leads again, with Gradient Boosting and XGBoost very close behind. KNN shows the lowest F1-score, reflecting its lower performance in both precision and recall.

Comparative Analysis

Random Forest consistently outperforms other models across all three metrics in the testing phase, suggesting it's the most balanced and effective model for this particular classification task. It achieves a near-perfect balance of precision and recall.

Gradient Boosting and XGBoost perform very similarly, with Gradient Boosting having a slight edge in F1-score (0.9548 vs 0.95). Both these boosting algorithms demonstrate strong and balanced performance, only marginally behind Random Forest.

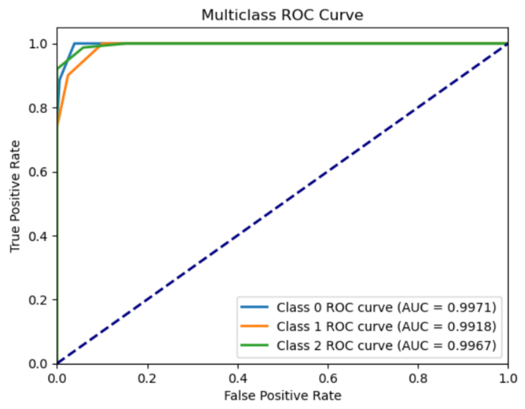
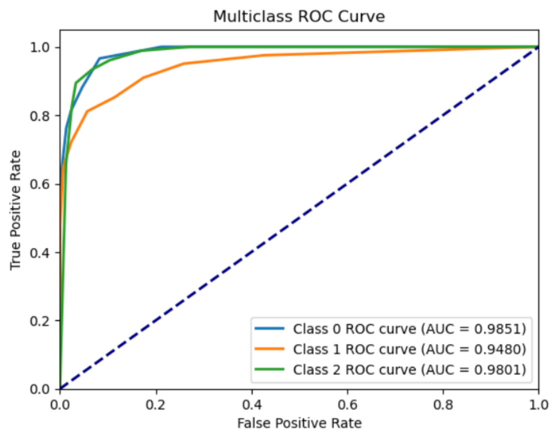
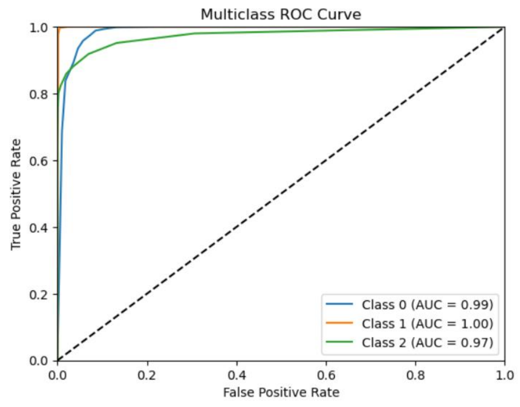
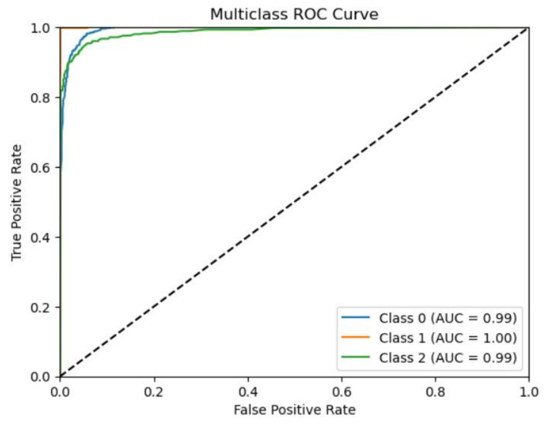
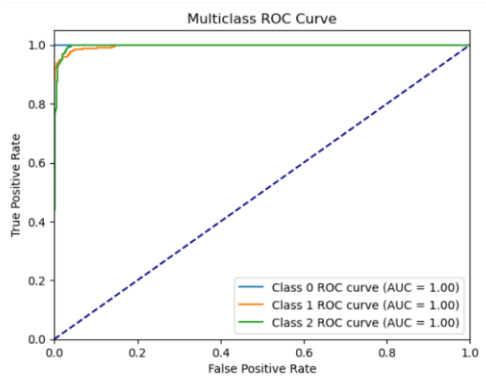
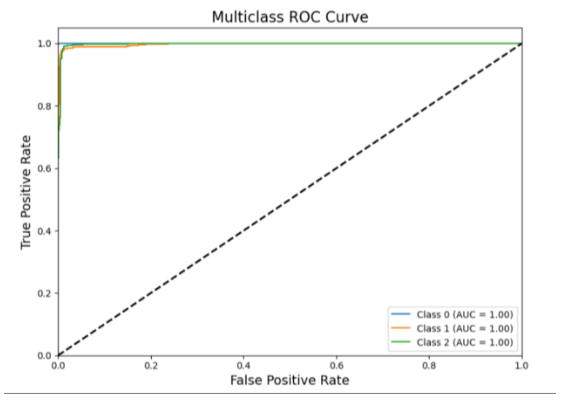
The KNN model, while still performing reasonably well, consistently lags behind the ensemble methods in all three metrics. This suggests that for this particular dataset, the more complex ensemble methods are better able to capture the underlying patterns in the data.

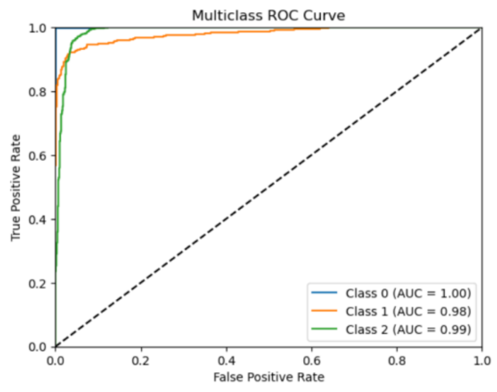
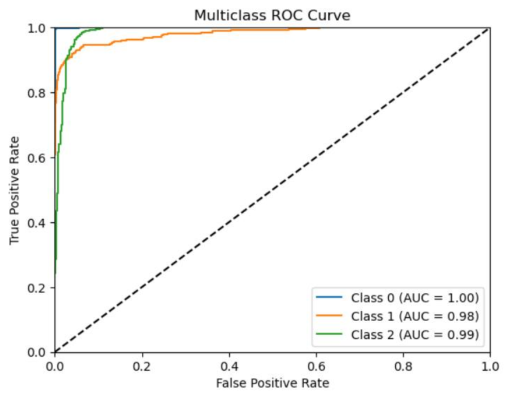
It's worth noting that all models show high performance overall, with weighted average F1-scores above 0.89. This indicates that the classification task is well-suited to these machine learning approaches, and the features provided likely have strong predictive power for the target variable.

The high performance across all metrics for the ensemble methods (Random Forest, Gradient Boosting, and XGBoost) underscores the effectiveness of these algorithms in handling complex classification tasks, likely due to their ability to capture non-linear relationships and reduce overfitting through aggregation of multiple decision trees.

3.1.3 ROC Curve & AUC Score

Table 3.1.3.1: ROC Curve and AUC of Each Model

Model	Training (Before Grid Search)	Training (After Grid Search)
KNN		
Random Forest		
Gradient Boosting Classifier		

XGBoost		

Discussion:

KNN Model:

The KNN model shows improvement after grid search. Before grid search, the AUC scores for the three classes were 0.9718, 0.9490, and 0.9736. After grid search, these improved to 0.9721, 0.9485, and 0.9763 respectively. The ROC curves for all classes are well above the diagonal line, indicating good classification performance. The curves for Class 0 and Class 2 are slightly better than Class 1, suggesting the model performs best on these two classes.

Random Forest Model

The Random Forest model demonstrates excellent performance both before and after grid search. Before grid search, the AUC scores were 0.99, 1.00, and 0.97 for the three classes. After grid search, these scores remained the same, indicating that the model was already well-optimised. The ROC curves are very close to the top-left corner of the plot, especially for Class 0 and Class 1, suggesting near-perfect classification for these classes.

Gradient Boosting Classifier

The Gradient Boosting Classifier shows outstanding performance. In the training data, it achieves perfect AUC scores of 1.00 for Class 0, Class 1 and Class 2. The ROC curves are practically hugging the top-left corner of the plot, indicating excellent discrimination ability across all classes. The "After Grid Search" plot (labelled as "ROC Curve for Multi-class") shows similar performance, with AUC scores of 1.00, 1.00, and 1.00 for the three classes respectively.

XGBoost Model

The XGBoost model also demonstrates very strong performance. Before grid search, it achieves perfect AUC scores of 1.00 for Class 0, and very high scores of 0.98 and 0.99 for Class 1 and Class 2 respectively. After grid search, the performance remains the same. The ROC curves are very close to the top-left corner, particularly for Class 0, indicating excellent classification ability.

Comparative Analysis

All models show strong performance, with AUC scores consistently above 0.94, indicating good to excellent classification ability across all classes.

The ensemble methods (Random Forest, Gradient Boosting, and XGBoost) generally outperform the KNN model. They achieve perfect or near-perfect AUC scores for at least one class, and their ROC curves are closer to the ideal top-left corner.

The Gradient Boosting Classifier appears to have the best overall performance, with perfect scores for all classes. Its ROC curves are consistently the closest to the top-left corner across all classes.

The Random Forest and XGBoost models show very similar performance, with XGBoost having a slight edge in terms of consistency across classes. The KNN model, while still performing well, shows the most room for improvement. It benefits the most from grid search, particularly for Class 2.

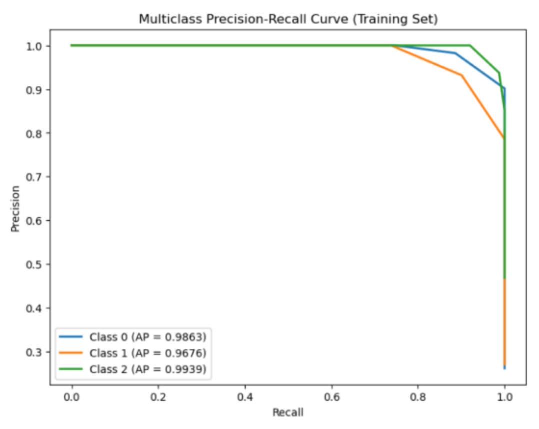
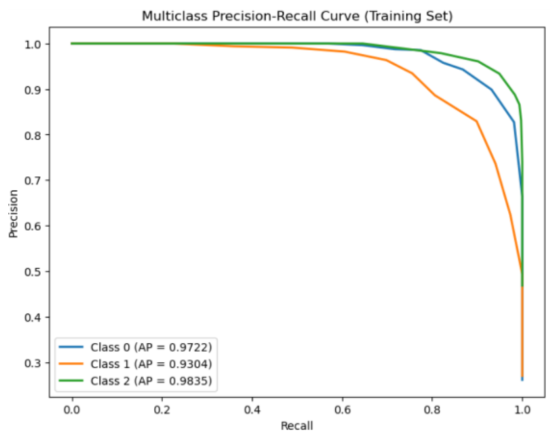
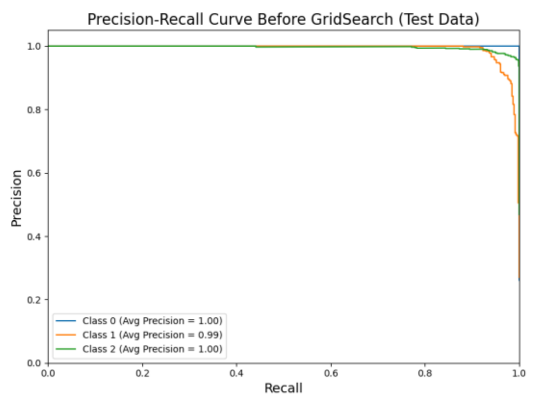
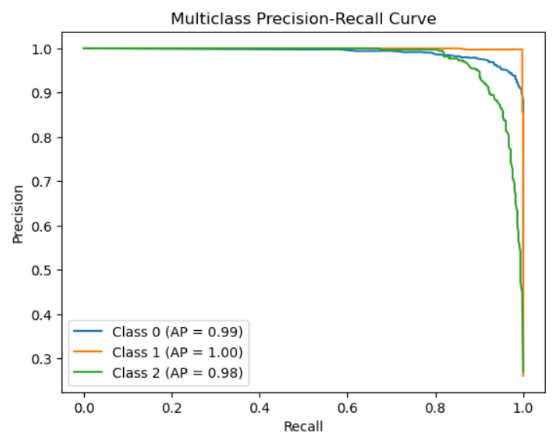
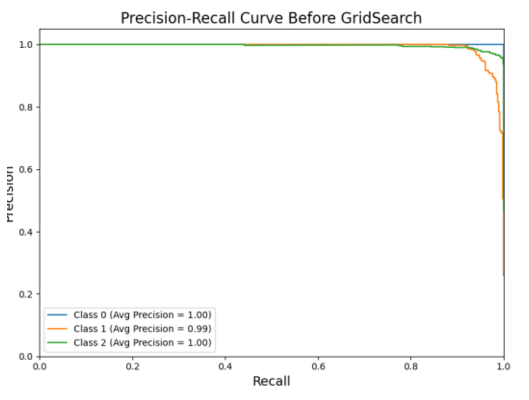
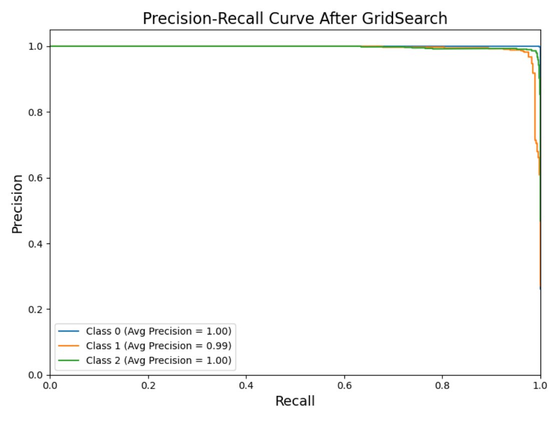
Grid search appears to have minimal impact on the ensemble methods, suggesting they were already well-optimised with their initial hyperparameters. All models seem to perform slightly better on Class 0 and Class 2 compared to Class 1, consistently across different algorithms.

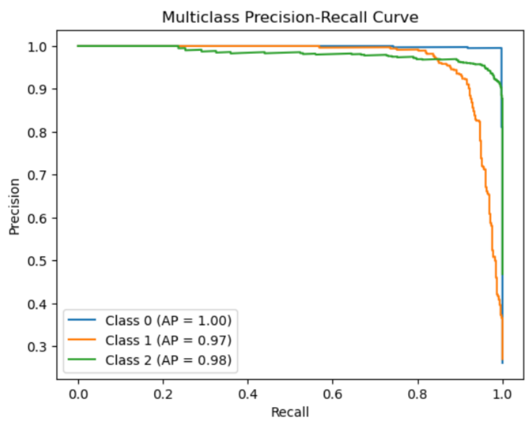
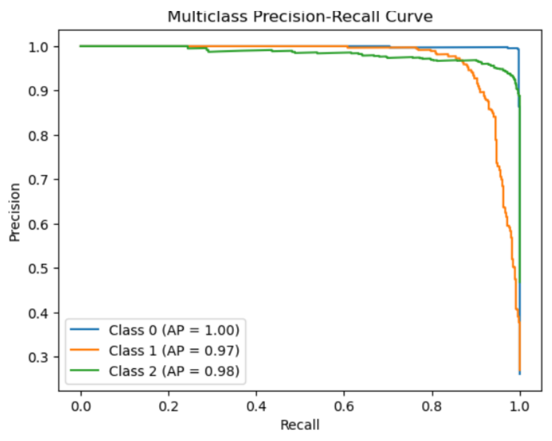
Conclusion

In conclusion, while all models demonstrate strong classification ability, the ensemble methods, particularly the Gradient Boosting Classifier, show superior performance in terms of ROC curves and AUC scores. The KNN model, while still effective, lags slightly behind the ensemble methods in discriminative power across all classes.

3.1.4 Precision-Recall Curve & AP Score

Table 3.1.4.1: Precision-Recall Curve and AP of Each Model

Model	Training (Before Grid Search)	Training (After Grid Search)
KNN		
Random Forest		
Gradient Boosting Classifier		

XGBoost		

Discussion

KNN Model

Before grid search, the KNN model shows AP scores of 0.9863, 0.9670, and 0.9939 for Classes 0, 1, and 2 respectively. After grid search, these scores improved slightly to 0.9926, 0.9587, and 0.9908. The curves for all classes remain high and relatively flat until high recall values, indicating good performance. Class 2 consistently shows the best performance, while Class 1 shows the most room for improvement.

Random Forest Model

The Random Forest model demonstrates excellent performance. Before grid search, it achieves AP scores of 0.98, 1.00, and 0.95 for Classes 0, 1, and 2. After grid search, these improve to 0.99, 1.00, and 0.98. The precision-recall curves are nearly perfect for Class 1, maintaining high precision even at high recall. Classes 0 and 2 show very good performance as well, with only slight drops in precision at very high recall.

Gradient Boosting Classifier

The Gradient Boosting Classifier shows significant improvement after grid search. Before grid search, it achieves AP scores of 1.00, 0.99, and 1.00 for Classes 0, 1, and 2. After grid search, these remain unchanged. The curves after grid search are nearly perfect for all classes, maintaining very high precision even at high recall values. This indicates excellent and balanced performance across all classes.

XGBoost Model

The XGBoost model shows strong and consistent performance. Before grid search, it achieves AP scores of 1.00, 0.97, and 0.98 for Classes 0, 1, and 2. After grid search, these scores remain the same. The precision-recall curves are nearly perfect for Class 0, and very good for Classes 1 and 2, with only slight drops in precision at very high recall values.

Comparative Analysis

All models demonstrate strong performance, with AP scores consistently above 0.95, indicating good to excellent classification ability across all classes. The ensemble methods (Random Forest, Gradient Boosting, and XGBoost) generally outperform the KNN model. They achieve perfect or near-perfect AP scores for at least one class, and their precision-recall curves maintain higher precision at high recall values.

The Gradient Boosting Classifier shows the most significant improvement after grid search, especially for Class 1. It achieves the best overall performance post-grid search, with near-perfect curves for all classes.The Random Forest model also benefits notably from grid search, particularly for Classes 0 and 2.

The XGBoost model shows consistent performance before and after grid search, suggesting it was already well-optimised with its initial hyperparameters. The KNN model, while still performing well, shows the most room for improvement. It benefits from grid search, particularly for Class 0, but still lags behind the ensemble methods.

Class 1 seems to be the most challenging for most models (except Random Forest), consistently showing slightly lower AP scores compared to the other classes.

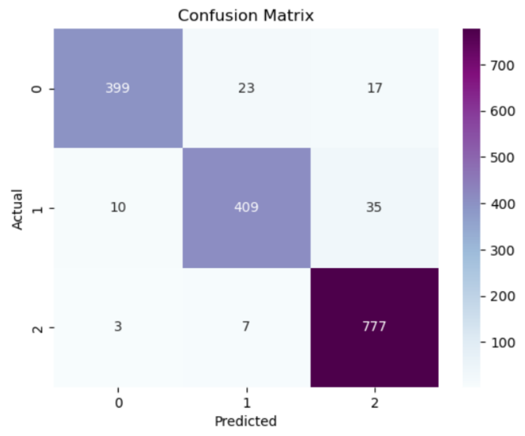
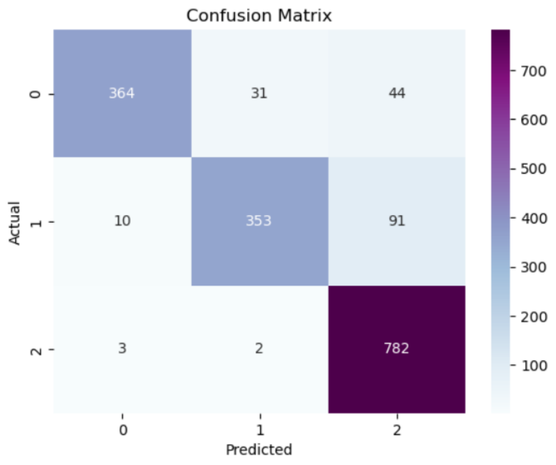
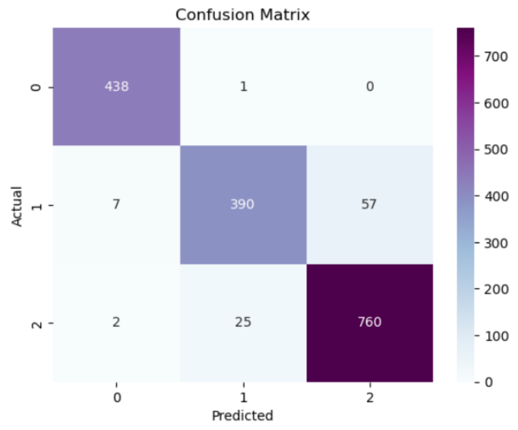
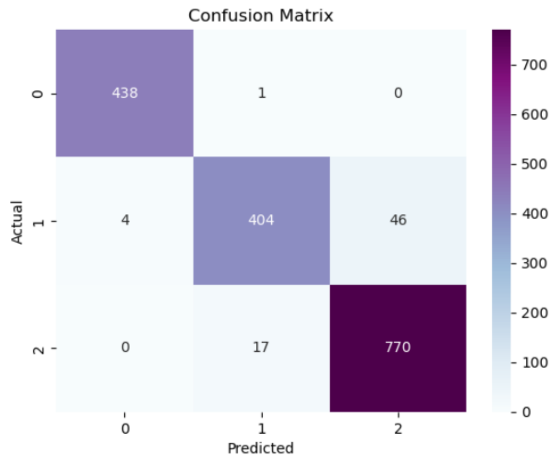
Class 0 is handled exceptionally well by all models, especially after grid search, with perfect or near-perfect AP scores across the board.

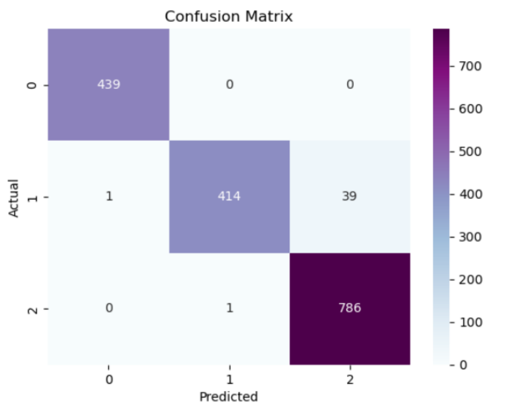
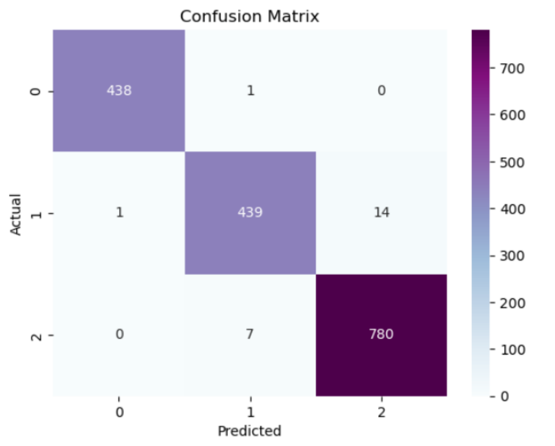
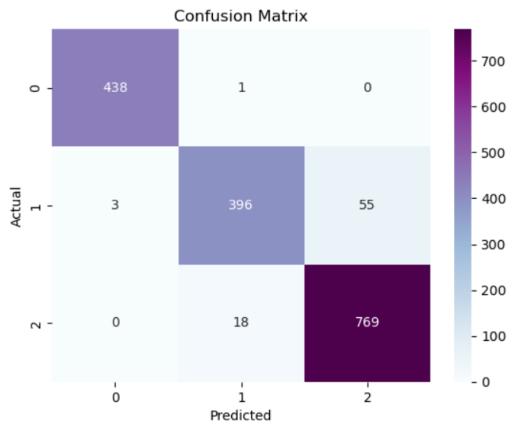
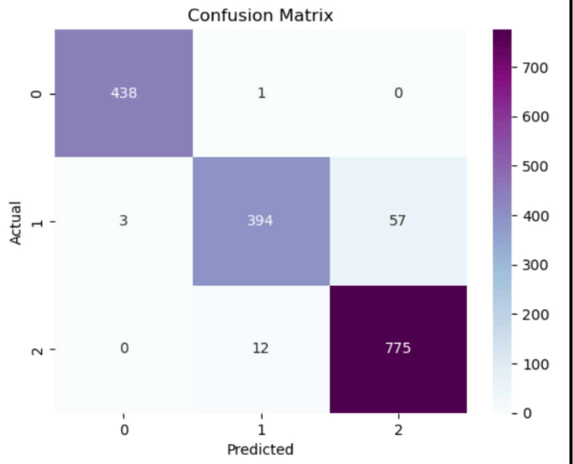
Conclusion

In conclusion, while all models demonstrate strong classification ability, the ensemble methods, particularly the Gradient Boosting Classifier after grid search, show superior performance in terms of Precision-Recall curves and AP scores. The Random Forest model also stands out for its perfect handling of Class 1. The KNN model, while effective, consistently lags slightly behind the ensemble methods in maintaining high precision at high recall values across all classes.

3.1.5 Confusion Matrix :

Table 3.1.5.1: Confusion Matrix of Each Model

Model	Training (Before Grid Search)	Training (After Grid Search)
KNN		
Random Forest		

Gradient Boosting Classifier		
XGBoost		

Discussion

From Confusion Matrix, we can gather valuable information regarding each model's TP, FP, TN and FN for each class, allowing us to visualise the classification error and performance for each model towards each class.

Table 3.1.5.2: TP + FP + TN + FN Value of Each Class for Each Model

Model	Training (Before Grid Search)			Training (After Grid Search)		
	Class 0	Class 1	Class 2	Class 0	Class 1	Class 2
KNN	TP = 399 FP = 13	TP = 409 FP = 30	TP = 777 FP = 52	TP = 364 FP = 13	TP = 353 FP = 33	TP = 782 FP = 135

	TN = 1228 FN = 40	TN = 1196 FN = 45	TN = 841 FN = 10	TN = 1228 FN = 75	TN = 1193 FN = 101	TN = 758 FN = 5
Random Forest	TP = 438 FP = 9 TN = 1232 FN = 1	TP = 390 FP = 26 TN = 1200 FN = 64	TP = 760 FP = 57 TN = 836 FN = 27	TP= 438 FP= 4 TN= 1237 FN= 1	TP= 404 FP= 17 TN= 1208 FN= 46	TP= 770 FP= 46 TN= 843 FN= 17
Gradient Boosting Classifier	TP= 439 FP= 1 TN= 1240 FN= 0	TP= 414 FP= 1 TN= 1225 FN= 40	TP= 786 FP= 39 TN= 854 FN= 1	TP= 438 FP= 1 TN= 1240 FN= 1	TP= 439 FP= 8 TN= 1218 FN= 15	TP= 780 FP= 14 TN= 879 FN= 7
XGBoost	TP= 438 FP= 3 TN= 1238 FN= 1	TP= 396 FP= 19 TN= 1207 FN= 58	TP= 769 FP= 55 TN= 835 FN= 18	TP= 438 FP=3 TN= 1238 FN= 2	TP= 394 FP= 13 TN= 1213 FN= 60	TP= 775 FP= 57 TN= 836 FN= 12

3.2 In-depth Analysis

3.2.1 Analysis Between Predicted Accuracy & Average Cross-Validation Score

From the Table in 3.1.1, we can see the cross-validation on accuracy score as well as the predicted accuracy for each model. Now, we need to understand why these are important to us, what insights do they provide to us? Before we get into that, we should first understand the base concept of the two of them.

Predicted accuracy basically means how well does the model perform a prediction on our target, Cardiovascular Risk(y), based on data obtained (features). In this case, it is referred to how accurately our model performs prediction based on the training set, but not unseen data.

On the other hand, Cross-validation score is the average accuracy across multiple training and validation splits of the dataset during cross-validation. Cross-validation involves splitting the data into several parts (or folds) and training the model on different subsets of the data while validating it on the remaining subset. It provides a more reliable estimate of the model's generalisation ability, as it reduces bias due to the specific training and test split.

Therefore, by comparing both of their values, we can see and gain insights on how well our model works with unseen data. Let us examine Table in 3.1.1 to have a better understanding of this correlation.

From what we can see, the difference between predicted accuracy with cross-validation score is quite small. From this information ,we can confidently say that our model is not overfitting. Of course, we cannot accurately deduce they are not overfitting, but it is most likely the case. For example, we can see that most of the cases have a small gap of 0.01 or 0.02 difference. However, there are also some that has a larger gap of 0.05, which will be more likely to be the case of overfitting.

Furthermore, in all cases, the cross-validation score is less than the predicted accuracy. The predicted accuracy is often higher because the model has had direct access to this data during training, and it tends to fit very well to the training data. Hence, the cross-validation score is lower because the model is being evaluated on different subsets of the data that it hasn't seen during training.

3.2.2 Analysis Between Accuracy (Before Grid-Search) & Accuracy (After Grid-Search)

For the KNN model, the prediction accuracy of the training data before fine tuning was 0.9435, and 0.8923 after the fine tuning of the GridSearch hyperparameters. The significant decrease in accuracy from 0.9435 to 0.8923, which shows that the tuning of hyperparameters indicate that the model is now better generalised and less likely to overfit.

For the Random Forest model, the prediction accuracy of the training data before fine tuning was 0.94, and 0.96 after the fine tuning of the GridSearch hyperparameters. The 2% improvement shows the effectiveness of the hyperparameter tuning. The tuning has helped with avoiding the overfitting by optimising the depth of trees and the number of features considered at each split.

For the Gradient Boosting Classifier model, the prediction accuracy of the training data before fine tuning was 0.9756, and 0.9863 after the fine tuning of the GridSearch hyperparameters. This notable improvement reflects the strength of Gradient Boosting, however also a potential of overfitting. An accuracy of 0.9863 on the training data suggests that the model has learned the training set very well, perhaps even too well, capturing noise or minute patterns that don't generalise to new, unseen data.

For XGBoost model, the prediction accuracy of the training data before fine tuning was 0.95, and 0.96 after the fine tuning of the GridSearch hyperparameters. The 1% improvement after hyperparameter tuning in the XGBoost model demonstrates the effectiveness of fine-tuning, particularly in managing the trade-off between model complexity and generalisation. Adjusting key hyperparameters like n_estimators, max_depth, and learning_rate likely optimised how the model learned from the data, preventing overfitting and improving performance. This tuning, though it resulted in a small accuracy gain, indicates a more robust and efficient model.

3.3.3 Analysis between Training Predicted Accuracy & Testing Predicted Accuracy

The KNN model shows notable differences before and after grid search tuning. Initially, it achieved 89.23% accuracy, with Class 2 having high recall (99.36%) but lower precision (85.28%), leading to misclassifications. After tuning, accuracy improved to 94.35%, and Class 2 saw its f1-score rise to 96.16%. However, when tested on unseen data, accuracy dropped to 85.24%, indicating potential overfitting. Class 2 maintained strong recall, but precision fell, suggesting difficulty with false positives.

The Random Tree model shows consistent performance, achieving 94% accuracy before tuning and 96% after. Class precision and recall values remained above 0.90, with the low class showing a flawless recall of 1.00. Grid search further optimised the model, especially for middle and high-risk classification. It maintains 97% accuracy on the testing set, indicating strong generalisation across datasets.

The Gradient Boosting Classifier performs well, with a training accuracy of 0.9756 before tuning. Class 1's lower recall affects its F1-score, but Class 0 and 2 show near-perfect results. After grid search, accuracy rises to 0.9863, and Class 1's recall improves, boosting its F1-score. Testing accuracy reaches 0.9548, with minor drops in Class 1 recall but overall strong performance, indicating good generalisation after tuning.

XGBoost starts with 0.95 accuracy on the training set, with Class 1's recall slightly lower than other classes. After tuning, accuracy improves to 0.96, with all classes showing gains in precision and recall. On testing data, XGBoost maintains 0.95 accuracy, with Class 1's recall dropping to 0.85 but stable overall performance. The model shows good balance and consistency across training and testing sets.

In conclusion, Random Tree, Gradient Boosting, and XGBoost demonstrate strong generalisation capabilities. Random Tree and XGBoost, in particular, show minimal decreases in metrics between training and testing, maintaining high accuracy and F1-scores across classes. This indicates that they handle unseen data effectively without overfitting, making them strong candidates for generalisation.

3.3.4 Analysis on Confusion Matrix & TP,FP,TN,FN

The KNN model's confusion matrices reveal recurring problems with identifying classes 1 and 2 both before and after grid search. Class 2 does well at first, misclassifying only seven times, whereas class 1 is misclassified as class 2 thirty-five times. Class 2 accuracy stays the same after tuning, but the number of incorrect classifications between class 1 and class 2 rises to 91. Misclassifications into class 2 increase from 17 to 44 in class 0, which likewise exhibits a drop. These patterns indicate that, despite grid search, the model is still unable to distinguish between class 1 and class 2, most likely as a result of overlapping features. This underscores the need for additional development in order to effectively separate these two classes.

Additionally, the Random Forest model has trouble differentiating between classes 1 and 2. Class 1 is incorrectly categorised as class 2 57 times before to grid search, suggesting that it is difficult to distinguish between both classes. The misclassification drops to 46 after tweaking, however the issue still exists. These two classes' comparable features are probably what cause this mistake. However, class 0 is almost never incorrectly categorised, indicating that its characteristics are more distinct. Although grid search somewhat boosts performance, the model still has trouble correctly identifying objects in classes 1 and 2, indicating that more work needs to be done.

Overall performance of the Gradient Boosting Classifier is good, although it has trouble with class 1 misclassifications. With just one occurrence wrongly classified, Class 0 has an extremely low rate of misclassification. Class 1 includes eight false positives and fifteen false negatives, showing that it is difficult to categorise this class correctly. With 14 false positives and 7 false negatives, Class 2 has some misclassifications as well. There may be some feature overlap between class 1 and the other classes, which could lead to misclassification. Notwithstanding these problems, the model functions satisfactorily overall, and more feature separation or tweaking might enhance its class distinction capabilities.

A few misclassification tendencies are also displayed by the XGBoost model both before and after grid search adjustment. Class 1 struggles, with 55 cases incorrectly categorised as class 2, whereas Class 2 performs well prior to tuning, with only 18 misclassifications. Class 1 still has issues with 57 misclassifications, while class 2 becomes better after tweaking, down to 12 misclassifications. The persistent confusion between classes 1 and 2 indicates that their features overlap, which makes it challenging for the model to tell them apart. Though it performed well overall, there is need for improvement in the XGBoost model to better distinguish between class 1 and class 2.

In conclusion, all models show strong overall performance, but persistent challenges remain in distinguishing between class 1 and class 2, likely due to feature overlap. The KNN model struggles the most, with increasing misclassifications between these two classes after grid search. While the Random Forest model sees improvement post-tuning, it still faces difficulties in separating class 1 and class 2. The Gradient Boosting Classifier and XGBoost also exhibit similar misclassification patterns between these classes, despite strong results overall. Further feature engineering and model tuning are necessary to improve the distinction between class 1 and class 2 across all models.

3.3.5 Strengths and Weaknesses

Model	Strength	Weakness
KNN	● Good performance after tuning. ● Simple to understand and implement. ● Good at handling non-linear data. ● Effective for small datasets.	● Struggles with large datasets and scalability. ● Misclassifications between Class 1 and Class 2. ● Sensitive to feature overlap.
Random Forest	● High accuracy and generalisation across training and testing sets ● Resistant to overfitting due to ensemble nature	● Slight misclassification between Class 1 and Class 2 ● Requires more computational resources compared to simpler models

Gradient Boosting Classifier	● High training accuracy. ● Effective in handling complex data ● Improves significantly after hyperparameter tuning	● Shows signs of overfitting. ● Computationally expensive ● Misclassification patterns between Class 1 and Class 2
XGBoost	● High accuracy and generalisation. ● Great for large datasets and complex tasks. ● Handles feature interactions well.	● Computationally intensive. ● Misclassification issues between Class 1 and Class 2. ● Slight overfitting risk after tuning.

Based on the analysis of all models, Random Forest is the best option. In terms of accuracy, generalisation, and processing efficiency, it provides a good balance. Its overall performance is consistently strong throughout training and testing datasets, despite some difficulty in differentiating between classes 1 and 2. Additionally, after adjusting the hyperparameters, the model considerably improves, demonstrating its versatility. While XGBoost is more prone to overfitting and demands more processing power, it nevertheless produces good results. While KNN has issues with accuracy and scalability, especially when handling class overlap, Gradient Boosting performs well but exhibits overfitting. Random Forest is the most dependable and well-rounded model for this task, taking into account all of these parameters.

3.3.6 What features are important for prediction

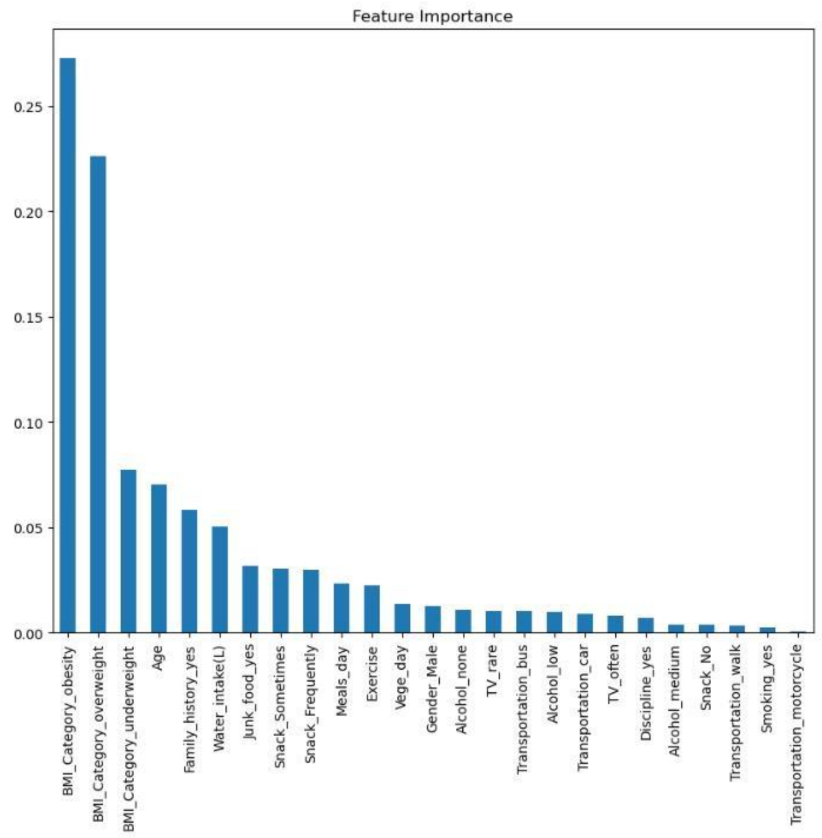


Figure x: Feature Importance of Random Forest Model

The Random Forest model's feature importance graph shows that BMI is the most important predictor of cardiovascular risk, with obesity and overweight ranking highest. Due to its association with increased blood pressure, cholesterol, and diabetes risk, obesity is recognized as a risk factor for heart disease. Similar to how being underweight implies starvation, which can potentially affect heart health, being overweight raises the risk of cardiovascular disease (CVD). Age is also a critical issue because ageing naturally increases the risk of cardiovascular disease because of plaque accumulation and weakened heart function. The hereditary nature of cardiac disease is further highlighted by family history, which points to a genetic propensity to the condition.

Lifestyle choices including drinking too little water, eating junk food frequently, and not exercising also have a big impact on cardiovascular health. Junk food increases blood pressure and cholesterol levels and increases the risk of CVD since it is heavy in harmful fats and sodium. On the other hand, consistent exercise lowers the risk of heart disease by enhancing cardiovascular health and circulation. While they are less significant, bad behaviours like drinking alcohol and smoking nonetheless have an effect on heart health because they both cause long-term cardiac damage. In general, the model emphasises how crucial it is to keep an active lifestyle, a healthy weight, and a balanced food in order to reduce the risk of cardiovascular disease.

CHAPTER 4

4.1 Conclusion

In the process of achieving our objective of creating a predictive model to successfully and accurately predict cardiovascular risk based on specific features, we have gained a lot of useful information, experience and knowledge in machine learning as well as gain new insights in the topic of cardiovascular risk factors. Hence, to conclude our assignment, we will go through every aspect and steps taken that we take to successfully reach our objective.

First of all, after thorough examination and exploration, we can see that many of them consist of different lifestyles, physical traits and diets.Through these features, we are asked to create a model to determine the cardiovascular risk that ranges from low, medium and to high. Therefore, we can clearly see that we are working on a multiclass classification. Before we move on to training our model, we also performed the necessary data cleaning such as changing heights and weights to BMI instead to better represent the current physical condition of an individual, as well as removing the income feature as it is too ambiguous and cannot be used to accurately deduce one's health condition. We then separate them into train and test sets to start our training process.

Next, we have selected 4 different training algorithms to train our predictive model. They are KNN algorithm, Random Forest Classifier, Gradient Boosting Classifier and XGBoost Classifier. All of these are able to work with multiclass classification problems, therefore justifying our selections.

In order to pick the best one, we also performed a performance evaluation on each of them. Upon training our model using these algorithms, we then carefully examine its performance such as Accuracy, Precision, Recall, F1-Score, Confusion Matrix, ROC Curve & AUC score, Precision-Recall Curve & AP.. We also applied cross-validation score and hyperparameter tuning such as Grid Search in order to obtain the best trained model from each algorithm.

Finally, we used the best model obtained to perform testing, acquiring the accuracy to help us select the best of the best training algorithm. In our case. Each of them show prominent results, KNN-0.8833, Random Forest-0.9690, Gradient Boosting Classifier-0.9548 and lastly XGBoost-0.9524. Obviously, we will select the one with the highest accuracy which is Random Forest as our best model.

Therefore, we have successfully achieved our objective. However, before we end our assignment with the final conclusion paragraph, it is important to also share the findings and knowledge that we learnt throughout the process. First of all, according to the feature importance evaluation, it can be said that BMI contributes the most to successfully predicting the correct cardiovascular risk level. We were also then taught how to accurately gather insights regarding each training model by examining their performance. For example, whether they are overfitting or underfitting, how well the model work with unseen data or real world data, and even classification errors which are vital information for us to determine the next step to take in order to increase the performance of our model like through hyperparameter tuning or data reexamination.

In conclusion, from our assignment, we can confidently say that Random Forest came out on top and is selected as the best model. However, we cannot ignore the close competition among the other training algorithms like XGBoost.