

Iteration 4

GITHUB LINK : <https://github.com/agneslee01/Infosys-722---Iteration-4/tree/main>

Step 1: Business understanding

1.1 - Identify the objectives of the business/situation

Good health has consistently remained a paramount concern on a global scale, garnering widespread attention across all demographics. While historically more prevalent among adults, the advent of the Covid-19 pandemic has notably heightened good health awareness across all age groups. In light of this, our focus will be specifically directed towards addressing the significant issue of heart attacks.

Heart attack is one of the life-threatening diseases that frequently afflict adults, particularly those aged 40 and older. Conversely, younger adults typically experience a lower risk of heart attack. However, Henderson (2022) reported that “heart attack death rates took a sharp turn and increased for all age groups during the pandemic and the increase was most significant among individuals ages 25-44”. From this information, it does support our assumption that the covid-19 pandemic has contributed to heightened vulnerability of heart attack among various demographics. Henderson (2022) has furthered elaborated the reason heart attack has increased is due to “psychological and social challenges associated with the pandemic, including job loss and other financial pressures that can cause acute or chronic stress.” This may seem relevant since many individuals in that mentioned age group need to earn a

living to support the family and the absence of reliable income sources inevitably leads to heightened stress and significant challenges.

We have evidence supporting the assumption that the risk of heart attacks affects younger people more significantly. Besides the impact of the COVID-19 pandemic, daily habits also play a crucial role. One prevalent habit among this age group is smoking. American Heart Association News (2021) reported that “young men who smoked had the highest long-term risk for heart attacks – 24%”. From this, it is noted that it is long-term risk, thus the occurrence of heart attack may be at a later age, and also depends on other underlying factors and habits each individual has. But at least we know that smoking will contribute to the risk of having heart attack in the long run. According to Fogoros (2022), some other lifestyle habits that may cause risk of heart attack is “poor diet, inactivity, and being overweight or obese”. While these habits alone may not directly cause a heart attack, when combined with other underlying health issues that individuals may have, they can contribute to an increased risk of heart attack.

Thus, study will be carried out with the following objectives:

- Assess the likelihood of an individual developing heart disease in different age group.
- Find out the main contributor(s) to heart attack disease.

1.2 Assess the situation

Resources:

I have a dataset which encompass individual health information, lifestyle-choices and some socioeconomic features etc. This dataset includes information sourced from Kaggle, an open- source data platform.

I will be using Big Data Analytics Software Stack – PySpark in Jupyter to interpret and analyse the dataset. Then, the code will be uploaded to GitHub via the AWS server. PySpark in Jupyter software enables to conduct statistical analysis such as summary statistics and regression analysis, data visualization and much more.

Hardware requirement would be a high and fast performance laptop with sufficient processing power and memory to run large dataset and produce output more quickly while using the software tool. I am using my personal laptop with the above specification.

Requirement:

Data to be analysed would be health information such as heart rate, cholesterol level etc, lifestyle-choices such as if the individual smokes and some socioeconomic attributes such as income. The expected outcome is to identify high-risk age group, statistical model that can analyse various risk factors contributing to heart attack etc. The criteria for evaluating success would be the accuracy of the statistical model.

Assumption:

Given that the data is obtained from Kaggle, it qualifies as secondary data. Thus, we can only make certain assumptions in relation to the data.

We can assume that the data is thorough and includes relevant variables essential for subsequent analysis, which minimize the potential of bias. Additionally, we assume that the data is accurate and free from errors, which ensures that the subsequent analysis on is based on reliable and true information.

Constraint:

Perhaps some important variables may be removed from the dataset before it was published to Kaggle due to data privacy issue and the way to obtained it is to request permission from the primary collector of the data. This can be challenging as there may be no further information of the author and many underlying issues may arise such as the process of getting consent could be complicated and/or time-consuming.

Risks:

Given that Kaggle is an open-source platform, it is accessible to anyone, potentially leading to data alteration that could impact subsequent analysis results. The risk of underfitting is a concern issue as well if the model is too simple and does not include all relevant features.

Contingencies:

To deal with risk of data alteration, we may perform checks such as anomalies and outliers and missing values in subsequent analysis, and decide what to do with it if any is present. On the other hand, to minimize underfitting risk, we can refine the model by including more features so that we are able to capture more relationship in the data.

1.3 Determine data mining objective/goals

The goal for the initial study to address the issue of heart attack is:

- To generate a predictive model to assess the likelihood of individuals towards developing heart disease by analysing comprehensive data encompassing their medical history, lifestyle factors, and demographic attributes.
- The strategic focus is on leveraging advanced analytics to provide actionable insights that empower informed decision-making and promote proactive measures by exploring relationship between factors and heart attack in order to identify patterns and promote proactive measures for heart disease prevention.

May use feature selection to determine which is the main contributor to heart attack disease.

1.4 Produce a project plan

<i>Phase</i>	<i>Time (in %)</i>	<i>How it is carried out</i>
1. Business understanding	5	Extra research to dig up real world data to back up my business objective
2. Data understanding	5	Assess the situation based on my project
3. Data Preparation	20	Using the Jupyter - PySpark software
4. Data transformation	10	Using the Jupyter - PySpark software

5. Data-mining method(s) selection	10	Based on own data mining objective + extra research
6. Data-mining algorithm(s) selection	10	Based on own data mining objective + extra research
7. Data Mining	30	Using the Jupyter - PySpark software
8. Interpretation	10	According to output from models.



Figure 1 : Day to day timeline

The day-to-day timeline depicted for this iteration project is shown in Figure 1 above. Please note that the timeline should be read from bottom to top, even though the Gantt chart may export with a descending order. This formatting does not impact the clarity of the timeline presentation.

Step 2: Data understanding

2.1 Collect Initial Data

For this project, I will only use one dataset, which is the heart attack risk prediction dataset, obtained from Kaggle, an open-source data hub that is assessable and free for everyone. The link to the dataset is <https://www.kaggle.com/datasets/iamsouravbanerjee/heart-attack-prediction-dataset>. This is secondary data; thus, I lack information regarding any potential complications encountered while collecting data. This dataset encompasses a wide array of attributes, such as age, cholesterol levels, blood pressure, smoking habits, exercise patterns, dietary preferences, and additional factors. Its purpose is to illuminate the intricate interactions among these variables in assessing the probability of a heart attack.

2.2 Describe the data

The data is in CSV file format. I have used PySpark in Jupyter to derive further initial insights about the dataset.

```
In [1]: import findspark
findspark.init('/home/ubuntu/spark-3.2.1-bin-hadoop2.7')
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('basics').getOrCreate()

WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/home/ubuntu/spark-3.2.1-bin-hadoop2.7/jars/spark-unsafe_2.12-3.2.1.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/05/22 03:37:18 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

In [2]: # Read in dataset
data = spark.read.csv('Dataset/heart_attack_dataset.csv', header=True, inferSchema=True)
```

Figure 2

Figure 2 above shows that I have uploaded my dataset into Jupyter with the lines of code. I have assigned the name *data* to the heart attack dataset that will be used for this project.

In [3]:

```
data.printSchema()
```

```
root
|-- Patient ID: string (nullable = true)
|-- Age: integer (nullable = true)
|-- Sex: string (nullable = true)
|-- Cholesterol: integer (nullable = true)
|-- Systolic: integer (nullable = true)
|-- Diastolic : integer (nullable = true)
|-- Heart Rate: integer (nullable = true)
|-- Diabetes: integer (nullable = true)
|-- Family History (1: Yes): integer (nullable = true)
|-- Smoking: integer (nullable = true)
|-- Obesity: integer (nullable = true)
|-- Alcohol Consumption: integer (nullable = true)
|-- Exercise Hours Per Week: double (nullable = true)
|-- Diet: string (nullable = true)
|-- Previous Heart Problems (1 : Yes): integer (nullable = true)
|-- Medication Use: integer (nullable = true)
|-- Stress Level: integer (nullable = true)
|-- Sedentary Hours Per Day: double (nullable = true)
|-- Income: integer (nullable = true)
|-- BMI: double (nullable = true)
|-- Triglycerides: integer (nullable = true)
|-- Physical Activity Days Per Week: integer (nullable = true)
|-- Sleep Hours Per Day: integer (nullable = true)
|-- Country: string (nullable = true)
|-- Continent: string (nullable = true)
|-- Hemisphere: string (nullable = true)
|-- Heart Attack Risk (1: Yes): integer (nullable = true)
```

Figure 3

In Figure 3, it depicts the code and format of the attributes. The output provides information about the types of variables present. Attribute such as Country etc. are identified as string because it takes string value. Attributes such as Age etc. are identified as integer since it only takes whole number. Attributes such as exercise per week, etc. are identified as double since it can take decimal values.

```
In [4]: # Count the number of rows
num_rows = data.count()

# Count the number of columns
num_columns = len(data.columns)

# Print the results
print(f"Number of rows: {num_rows}")
print(f"Number of columns: {num_columns}")
```

```
Number of rows: 2499
Number of columns: 27
```

Figure 4

In Figure 4 , it is evident there are 2499 rows and 27 columns (the output is circled in red).

In my dataset, there are 8 attributes that are binary variables. Attributes such as heart rate, diabetes, previous heart problem etc are represented with values [0,1], indicating binary variables. In Figure 3, these variables are being identified as integer data type. In the context of our dataset, 0 signifies "No" and 1 signifies "Yes."

Figure 5 below showcases the code and output for the summary statistics for the attributes. However, summary statistics is only meaningful for numerical attributes. Therefore, I have written another line of code, which shows the summary statistics for some numerical variables, which is to illustrate the idea of how to compute summary statistics.

```
In [5]: from pyspark.sql.functions import round

selected_columns = ['Age', 'Exercise Hours Per Week', 'Sleep Hours Per Day', 'Income']

summary = data.select(selected_columns).describe()

summary_rounded = summary.select(summary["summary"],
                                  *[round(summary[col], 2).alias(col) for col in selected_columns])

# Show the summary statistics
summary_rounded.show()
```

	Age	Exercise Hours Per Week	Sleep Hours Per Day	Income
count	2499.0	2496.0	2497.0	2499.0
mean	53.55	9.98	7.0	157514.21
stddev	21.29	5.78	2.03	80349.55
min	18.0	0.01	2.0	5000.0
max	103.0	40.55	20.0	299769.0

Figure 5

```
In [6]: # Perform value counts for the variable "Gender"
value_counts = data.groupBy("Sex").count()

# Show the value counts
value_counts.show()
```

Sex	count
Female	776
Male	1723

Figure 6

In Figure 6 depicted above, the code tallies the counts for each category within a variable. The corresponding output reveals that there are 1723 individuals classified as male and 776 individuals classified as female. The function `value_counts()` can be used for other variables such as diet, country etc.. the aforementioned examples provide a layout of the output by using the function.

```

from pyspark.sql.functions import mean

# Compute the mean values of Income and Age for each gender
mean_values = data.groupBy("Sex").agg(round(mean("Age"), 2).alias("Mean_Age"), round(mean("Income"), 2).alias("Mean_Income"))

# Show the mean values
mean_values.show()

```

Sex	Mean_Age	Mean_Income
Female	52.25	159931.71
Male	54.14	156425.42

Figure 7

In Figure 7 displayed above, the code demonstrates how to utilize the `group_by()` function to generate insightful summaries, as depicted in Figure 7. Specifically, within Figure 7, we have selected the variables Income, Age, and Sex, and applied a group-by operation on Sex to compute the mean values of Income and Age for each gender. From the resulting output shown in Figure 7, it appears that females tend to have a higher average income compared to males. Conversely, males exhibit a higher average age compared to females.

Below is the list of parameters and definition:

- **Patient ID** - Unique identifier of the patient
- **Age** - Age
- **Sex** - Gender (Male/Female)
- **Cholesterol** - Cholesterol levels
- **Systolic** - Blood pressure
- **Diastolic** - Blood pressure
- **Heart Rate** - Heart rate
- **Diabetes** - Whether the patient has diabetes (1: Yes, 0: No)
- **Family History** - Family history of heart-related problems (1: Yes, 0: No)
- **Smoking** - Smoking status (1: Smoker, 0: Non-smoker)
- **Obesity** - Obesity status (1: Obese, 0: Not obese)
- **Alcohol Consumption** - Alcohol consumption (1: Yes, 0: No)
- **Exercise Hours Per Week** - Number of exercise hours per week
- **Diet** - Dietary habits (Healthy/Average/Unhealthy)
- **Previous Heart Problems** - Previous heart problems (1: Yes, 0: No)

- **Medication Use** - Medication usage (1: Yes, 0: No)
- **Stress Level** - Stress level (1-10)
- **Sedentary Hours Per Day** - Hours of sedentary activity per day
- **Income** - Income level
- **BMI** - Body Mass Index (BMI)
- **Triglycerides** - Triglyceride levels
- **Physical Activity Days Per Week** - Days of physical activity per week
- **Sleep Hours Per Day** - Hours of sleep per day
- **Country** - Country
- **Continent** - Continent
- **Hemisphere** - Hemisphere
- **Heart Attack Risk** - Presence of heart attack risk (1: Yes, 0: No)

2.3 Explore the data

```
import matplotlib.pyplot as plt

# Compute the distribution of the "Country" variable
country_distribution = data.groupby("Country").count()

country_distribution_local = country_distribution.toLocalIterator()

# Prepare data for plotting
countries = []
counts = []
for row in country_distribution_local:
    countries.append(row["Country"])
    counts.append(row["count"])

# Plot the bar graph
plt.figure(figsize=(10, 6))
plt.bar(countries, counts, color='skyblue')
plt.xlabel('Country')
plt.ylabel('Count')
plt.title('Distribution of Country')
plt.xticks(rotation=45, ha='right', fontsize=12) # Rotate x-axis labels and align to the right
plt.tight_layout() # Adjust subplots to fit into the figure area
plt.show()
```

Figure 8

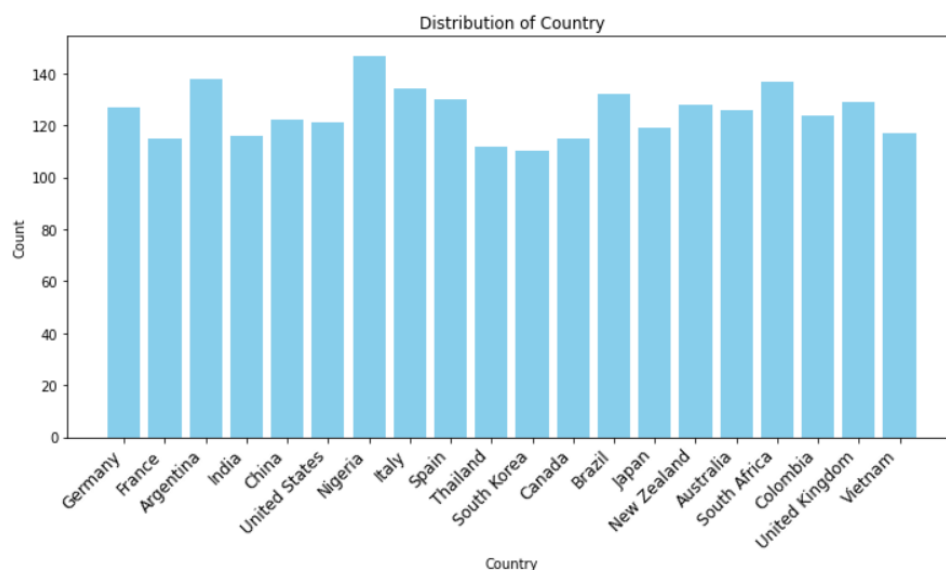


Figure 8.1

I would like to explore the distribution of specific attributes, such as the Country attribute, as demonstrated in the code in Figure 8. I opt to use bar graph to visualise the unique count because country is a categorical variable, and there are 20 different countries thus it is sensible to use a bar graph to illustrate count. The output, presented in Figure 8.1, displays countries on the x-axis and their corresponding counts on the y-axis. From the graph, it is evident that Nigeria has the highest proportion of patients, and South Korea has the lowest proportion of patients.

```
from pyspark.sql.functions import col

# Compute the distribution of the "Gender" variable
gender_distribution = data.groupBy("Sex").count()

# Compute the total count
total_count = data.count()

# Compute the percentage distribution
gender_percentage = gender_distribution.withColumn("Percentage", (col("count") / total_count) * 100)

gender_percentage_local = gender_percentage.collect()

# Prepare data for plotting
genders = [row["Sex"] for row in gender_percentage_local]
percentages = [row["Percentage"] for row in gender_percentage_local]

# Plot the percentage distribution
plt.bar(genders, percentages, color='orange')
plt.xlabel('Sex')
plt.ylabel('Percentage')
plt.title('Percentage Distribution of Gender')
plt.ylim(0, 80) # Set y-axis limits to 0-100%
plt.tight_layout() # Adjust subplots to fit into the figure area
plt.show()
```

Figure 9

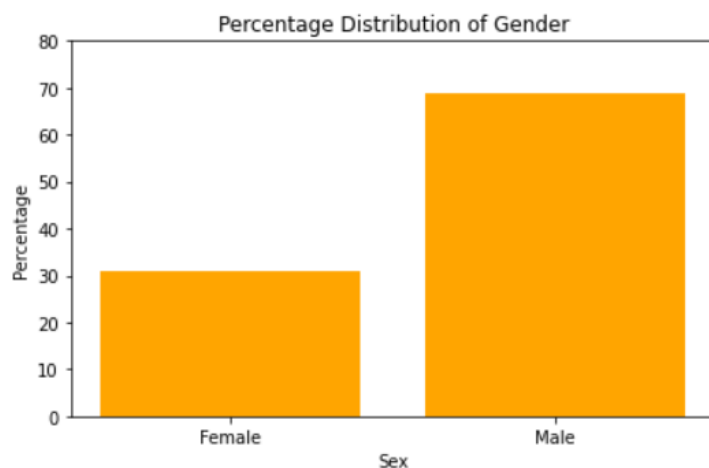


Figure 9.1

As an example, the percentage distribution of gender is illustrated in Fig 9.1 in a bar graph. The findings reveal a higher proportion of males compared to females within the dataset. The percentage for male is roughly 68% and female is roughly 32%. The code to produce this graph is shown in Figure 9 above.

```
# Compute the count of stress levels for each gender
count_stress_by_gender = data.groupBy("Sex", "Stress Level").count().orderBy("Sex", "Stress Level")

# Collect data locally for plotting
count_stress_by_gender_local = count_stress_by_gender.collect()

# Prepare data for plotting
genders = list(set(row["Sex"] for row in count_stress_by_gender_local))
stress_levels = sorted(set(row["Stress Level"] for row in count_stress_by_gender_local if row["Stress Level"] is not None))
count_matrix = [[row["count"] for row in count_stress_by_gender_local if row["Sex"] == gender for gender in genders]

# Get the maximum count of stress levels across genders
max_count = max(len(counts) for counts in count_matrix)

# Pad count_matrix with zeros if the count of stress levels is not consistent across genders
count_matrix_padded = [counts + [0] * (max_count - len(counts)) for counts in count_matrix]

# Plot the count of stress levels by gender
plt.figure(figsize=(10, 6))
for i, gender in enumerate(genders):
    plt.bar([x + i * 0.2 for x in range(max_count)], count_matrix_padded[i], width=0.2, label=gender)
plt.xlabel('Stress Level')
plt.ylabel('Count')
plt.title('Count of Stress Levels by Gender')
# plt.xticks([x + bar_width / 2 for x in range(len(stress_levels))], stress_levels)
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 10

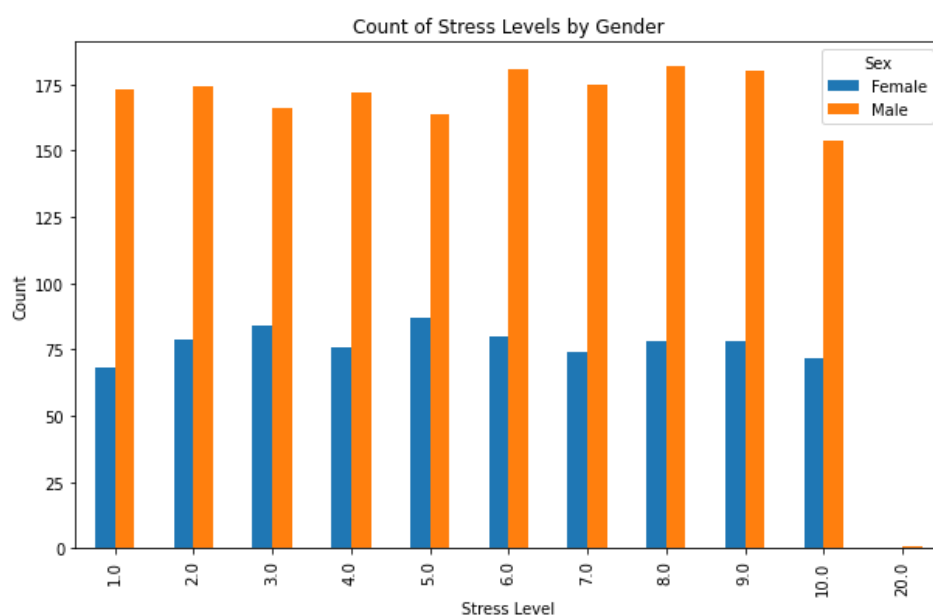


Figure 10.1

I have chosen to depict stress level using a bar graph because I would want to summarize the count of discrete variable, and stress level, ranging from 1-10, fits this criterion. Fig 10.1 illustrates the output bar graph categorized by gender. By comparison, the mode for stress level for female is 5, contrasting with the mode for stress level of 8 for male. Figure 10 above is the code to produce the bar graph.

```
# Group the DataFrame by "Diet" and count the occurrences
diet_counts = data.groupby("Diet").count().orderBy("Diet")

diet_counts_local = diet_counts.collect()

# Prepare data for plotting
diets = [row["Diet"] for row in diet_counts_local]
counts = [row["count"] for row in diet_counts_local]

# Plot the pie chart
plt.figure(figsize=(8, 8))
plt.pie(counts, labels=diets, autopct='%1.1f%%', startangle=140)
plt.title('Diet Distribution')
plt.show()
```

Figure 11

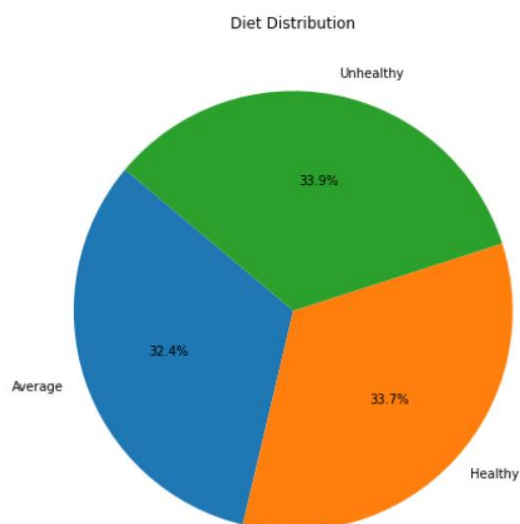


Figure 11.1

A pie chart is suitable to summarize nominal data, hence I have selected the *diet* variable for illustration. From the pie chart in fig 11.1, although the proportions are relatively similar, it is noteworthy that unhealthy diet has the highest percentage, with 33.9%.

The code to produce the pie chart is in Figure 11. `'autopct='%1.1f%%'` displays the percentage value on each slice, and `'startangle=140'` specifies the starting angle for the first slice.

2.4 Verify Data Quality

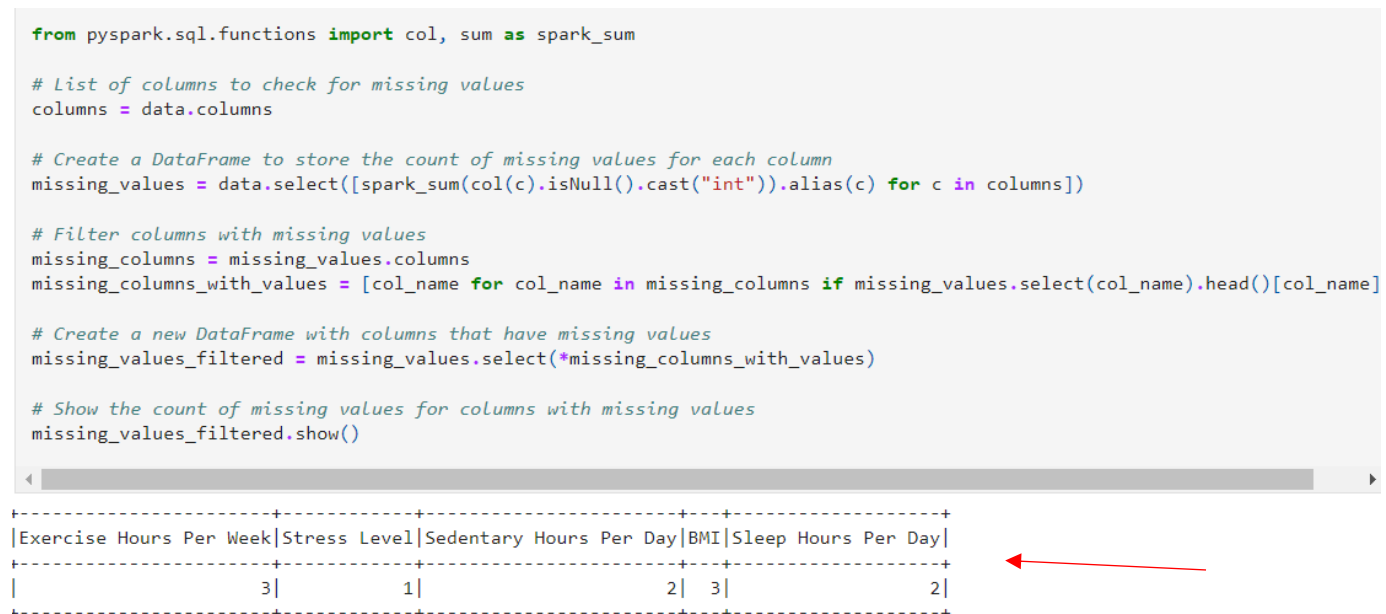


Figure 12

In Fig 12, it's evident that there are some missing values for 5 variables, which are *exercise hours per week*, *stress level*, *sedentary hours per day*, *sleep hours per day* and *BMI* (the arrow points at). The count of missing values are produced at the bottom to the respective variables.

It is also important to check if any outlier is present in the dataset. Upon further examination, it is noticeable that there are outlier in few variables.

```

from pyspark.sql.types import IntegerType, DoubleType, StructType, StructField, StringType

# Function to calculate outliers for a column
def calculate_outliers(data, column):
    # Calculate Q1 and Q3
    quantiles = data.approxQuantile(column, [0.25, 0.75], 0.01)
    Q1, Q3 = quantiles[0], quantiles[1]
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter outliers
    outliers = data.filter((col(column) < lower_bound) | (col(column) > upper_bound))
    outlier_count = outliers.count()

    return outlier_count

# List to store outlier counts
outlier_counts = []

# Iterate through each column
for column in data.columns:
    # Check if the column is of numeric type (integer or double)
    if isinstance(data.schema[column].dataType, (IntegerType, DoubleType)):
        outlier_count = calculate_outliers(data, column)
        if outlier_count > 0:
            outlier_counts.append((column, outlier_count))

# Define schema for the results DataFrame
schema = StructType([
    StructField("Column", StringType(), True),
    StructField("Outlier Count", IntegerType(), True)
])

# Create DataFrame from the results
outlier_df = spark.createDataFrame(outlier_counts, schema)

# Show the results in tabular format
outlier_df.show(truncate=False)

```

Figure 13

In Figure 13 above shows the codes to identify outliers. I have identify outliers using the Interquartile Range (IQR) method with a specified threshold ($1.5 * \text{IQR}$). Since outlier is usually associated for numerical variables, thus it is sensible that we only look for presence of outliers in numerical variables. I have written a line of code to include only numerical variables (circled in red) in Figure 13. The code (circled in purple) is to calculate the outliers. The codes (circled in blue) is to calculate the sum of outliers for each variable and only produce the variables that have outlier instead of the full list of variables.

Column	Outlier Count
Heart Rate	1
Smoking	276
Exercise Hours Per Week	1
Stress Level	1
Sleep Hours Per Day	1

Figure 13.1

The analysis results are presented and illustrated in Figure 13.1 above. PySpark has identified outliers in 5 variables. However, it's important to note that the variable "Smoking" is binary (taking values 0 or 1) and should not have any outliers due to its nature. Further investigation is needed to understand the context of the "Smoking" variable. Excluding "Smoking," the remaining four variables each contain one outlier.

Additional examination is provided in Figure 13.2 below. The noticeable disparity in the proportion of smokers and non-smokers likely led PySpark to flag non-smokers (0) as outliers.

```
# Perform value counts for the variable "Gender"
value_counts1 = data.groupBy("Smoking").count()

# Show the value counts
value_counts1.show()
```

Smoking	count
1	2223
0	276

Figure 13.2

After I have identified outliers, I would like to identify if there are any extreme values. Figure 14 below provides the code to calculate extreme values. I have once again added the codes (circled in blue) to only produce the variables that have extreme values instead of the full list of variables.

```

# Function to find extreme values for a column
def find_extreme_values(data, column):
    # Calculate Q1 and Q3
    quantiles = data.approxQuantile(column, [0.25, 0.75], 0.01)
    Q1, Q3 = quantiles[0], quantiles[1]
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter extreme values
    extreme_values_df = data.filter((col(column) < lower_bound) | (col(column) > upper_bound))
    return extreme_values_df

exclude_columns = ["Smoking"]

# Iterate through each column and find extreme values if the column is numeric
for column in data.columns:
    if column not in exclude_columns and isinstance(data.schema[column].dataType, (IntegerType, DoubleType)):
        extreme_values_df = find_extreme_values(data, column)
        if extreme_values_df.count() > 0:
            extreme_value = extreme_values_df.select(column).head()[0]
            print(f"{column} has an extreme value: {extreme_value}")

```

Figure 14

```

Heart Rate has an extreme value: 200
Exercise Hours Per Week has an extreme value: 40.546388
Stress Level has an extreme value: 20
Sleep Hours Per Day has an extreme value: 20

```

Figure 14.1

The output in Figure 14.1 above displays the presence of extreme values in the dataset. Specifically, there are four variables identified as containing extreme values, which aligns with the variables previously identified as containing outliers by PySpark. Notably, the variable *smoking* is not flagged for extreme values, which is expected given its binary nature. And this supports our statement above regarding the significant difference between smokers and non-smokers thus being mistakenly identified as outliers. Considering that each of these four variables contains only one extreme value, and these are the same variables identified to have outliers, we can conclude that, in this context, the terms "outliers" and "extreme values" are synonymous.

```
git config --global --edit

After doing this, you may fix the identity used for this commit with:

git commit --amend --reset-author

1 file changed, 611 insertions(+)
 create mode 100644 Part 2 - Data Understanding.ipynb
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git push
Username for 'https://github.com': agneslee01
Password for 'https://agneslee01@github.com':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 60.87 KiB | 10.15 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/agneslee01/Infosys-722---Iteration-4
 * [new branch]      main -> main
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$
```

Screenshot of AWS terminal to proof that Part 2 – Data Understanding file is pushed to github.

Step 3: Data Preparation

3.1 Data Selection

After reviewing our dataset, I have decided to retain all the variables and only remove the Patient ID variable column. The rationale behind this is because the patient ID seems irrelevant and unrelated to our analysis, as it is merely a unique identifier for each patient. Additionally, no sensitive information is included in the dataset that would identify the patients. I have decided to keep all other variables because we would not want to remove variables that potentially be a factor for heart attack. We want to consider all possible factors and in subsequent analysis only we conduct feature selection to ascertain the main contributor to heart attack more precisely.

The decision to include all features also align well with our data mining goals, where we consider their medical history, lifestyle choices etc, to provide insight and comes up with proactive measures.

```
# Drop a column
column_to_drop = "Patient ID"
data = data.drop(column_to_drop)

# Show the resulting DataFrame
data.printSchema()

root
|-- Age: integer (nullable = true)
|-- Sex: string (nullable = true)
|-- Cholesterol: integer (nullable = true)
|-- Systolic: integer (nullable = true)
|-- Diastolic : integer (nullable = true)
|-- Heart Rate: integer (nullable = true)
|-- Diabetes: integer (nullable = true)
|-- Family History (1: Yes): integer (nullable = true)
|-- Smoking: integer (nullable = true)
|-- Obesity: integer (nullable = true)
|-- Alcohol Consumption: integer (nullable = true)
|-- Exercise Hours Per Week: double (nullable = true)
|-- Diet: string (nullable = true)
|-- Previous Heart Problems (1 : Yes): integer (nullable = true)
|-- Medication Use: integer (nullable = true)
|-- Stress Level: integer (nullable = true)
|-- Sedentary Hours Per Day: double (nullable = true)
|-- Income: integer (nullable = true)
|-- BMI: double (nullable = true)
|-- Triglycerides: integer (nullable = true)
|-- Physical Activity Days Per Week: integer (nullable = true)
|-- Sleep Hours Per Day: integer (nullable = true)
|-- Country: string (nullable = true)
|-- Continent: string (nullable = true)
|-- Hemisphere: string (nullable = true)
|-- Heart Attack Risk (1: Yes): integer (nullable = true)
```

Figure 15

```
# Count the number of columns
num_columns = len(data.columns)

# Print the results
print(f"Number of columns: {num_columns}")

Number of columns: 26
```

Figure 15.1

Figure 15 above illustrates the code to drop the Patient ID variable and the corresponding output. Figure 15 shows that Age variable is now the first column. Figure 15.1 above shows now there is a total of 26 columns compare to the initial 27 columns, after dropping the Patient ID variable.

In prior steps, we observed that four variables – *heart rate*, *exercise hours per week*, *stress level* and *sleep hours per day* have 1 outlier each. Despite this, we

do not want to exclude these variables because lifestyle choice can also potentially be a factor for heart attack. The outliers will handle in subsequent analysis.

```
: # Perform value counts for the variable "Heart Attack Risk (1: Yes)"
value_counts = data.groupBy("Heart Attack Risk (1: Yes)").count()

# Show the value counts
value_counts.show()
```

Heart Attack Risk (1: Yes)	count
1	925
0	1574

Figure 16

I have picked *heart attack risk* variable as the target. In Figure 16, we can gain insight that 0 indicates no heart attack risk, and 1 indicates presence of heart attack risk.

3.2 Data Cleaning

We have previously identified missing values for 5 variables in Figure 12 above. In this subsection, we will need to perform data cleaning and decides how to deal with the missing values. The missing values are *Exercise Hours Per Week*, *Sedentary Hours Per Day*, *BMI*, *Sleep Hours Per Day* and *Stress Level*.

```
from pyspark.sql.functions import when

# Calculate mode of the "Stress Level" column
mode_value = data.groupBy("Stress Level").count().orderBy("count", ascending=False).first()[0]

# Impute missing values with the mode
data = data.withColumn("Stress Level", when(data["Stress Level"].isNull(), mode_value).otherwise(data["Stress Level"]))
```

Figure 17

The code in Figure 17 above, I demonstrate how I handle missing values in a categorical variable. The *Stress Level* variable, despite ranging from 1 to 10 where 10 represents the highest level of stress, is treated as a categorical variable. I have decided to impute missing values for the *stress level* variable using the mode because it is sensible to use mode imputation for categorical variable. Thus, the missing value will be replaced with the most frequent stress level. The code (circled in red) imputes missing value for the variable with the mode.

```
from pyspark.sql.functions import mean

exercise_mean = data.select(mean("Exercise Hours Per Week")).collect()[0][0]
sedentary_mean = data.select(mean("Sedentary Hours Per Day")).collect()[0][0]
bmi_mean = data.select(mean("BMI")).collect()[0][0]
sleep_mean = data.select(mean("Sleep Hours Per Day")).collect()[0][0]

# Impute missing values with the mean
data = data.withColumn("Exercise Hours Per Week", when(data["Exercise Hours Per Week"].isNull(), exercise_mean).otherwise(data["Exercise Hours Per Week"]))
data = data.withColumn("Sedentary Hours Per Day", when(data["Sedentary Hours Per Day"].isNull(), sedentary_mean).otherwise(data["Sedentary Hours Per Day"]))
data = data.withColumn("BMI", when(data["BMI"].isNull(), bmi_mean).otherwise(data["BMI"]))
data = data.withColumn("Sleep Hours Per Day", when(data["Sleep Hours Per Day"].isNull(), sleep_mean).otherwise(data["Sleep Hours Per Day"]))
```

Figure 18

The code in Figure 18 shows how I deal with continuous variables that have missing values. I have decided to impute missing value for continuous variables using the mean. The code (circled in red) calculates the respective mean for the four variables. The code (circled in blue) imputes missing value for the variables with the mean.


```

from pyspark.sql.functions import col, sum as spark_sum

# List of columns to check for missing values
columns = data.columns

# Create a DataFrame to store the count of missing values for each column
missing_values = data.select([spark_sum(col(c).isNull().cast("int")).alias(c) for c in columns])

# Filter columns with missing values
missing_columns = missing_values.columns
missing_columns_with_values = [col_name for col_name in missing_columns if missing_values.select(col_name).head()[col_name]

# Create a new DataFrame with columns that have missing values
missing_values_filtered = missing_values.select(*missing_columns_with_values)

# Show the count of missing values for columns with missing values
missing_values_filtered.show()

```

```

++
||
++
||
++

```




Figure 19

```

: # Count the number of rows
num_rows1 = data.count()

# Count the number of columns
num_columns1 = len(data.columns)

# Print the results
print(f"Number of rows: {num_rows1}")
print(f"Number of columns: {num_columns1}")

```

```

Number of rows: 2499
Number of columns: 26

```

Figure 19.1

In Figure 19 is the code to check for missing values once again after I have filled the missing values with mean and mode. Figure 19 shows the output too. An empty series (shown by the red arrow) is returned which indicates no more missing values in the dataset. Figure 19.1 shows no fields (rows) are removed. We still have 2499 entries (rows).

```
Heart Rate has an extreme value: 200
Exercise Hours Per Week has an extreme value: 40.546388
Stress Level has an extreme value: 20
Sleep Hours Per Day has an extreme value: 20
```

Figure 20 (taken from figure 14.1)

In Figure 20 (taken from figure 14.1), shows the extreme value for the four variables. For instance, it is impossible for an individual to sleep 20 hours per day. Since these values are too extreme, it is best if we discard these extreme values. By removing these extreme values, it may improve our model performance in the subsequent analysis.

```
data = data.filter(data["Heart Rate"] != 200)
data = data.filter(data["Exercise Hours Per Week"] != 40.546388)
data = data.filter(data["Stress Level"] != 20)
data = data.filter(data["Sleep Hours Per Day"] != 20)
```

Figure 21

```
# Print the results
print(f"Number of rows: {data.count()}")
print(f"Number of columns: {len(data.columns)}")
```

```
Number of rows: 2495
Number of columns: 26
```

Figure 21.1

Figure 21 above shows the code to remove the extreme values from the four variables. Figure 21.1 above shows the number of entries has decreased from 2499 entries to 2495 entries once we have dealt with the extreme values by removing them. Figure 21.3 below also proof that there is no presence of extreme values. Figure 21.2 below is the code to produce Figure 21.3.

```

from pyspark.sql.types import IntegerType, DoubleType, StructType, StructField, StringType

# Function to find extreme values for a column

def find_extreme_values(data, column):
    # Calculate Q1 and Q3
    quantiles = data.approxQuantile(column, [0.25, 0.75], 0.01)
    Q1, Q3 = quantiles[0], quantiles[1]
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter extreme values
    extreme_values_df = data.filter((col(column) < lower_bound) | (col(column) > upper_bound))
    return extreme_values_df

exclude_columns = ["Smoking"]

# Iterate through each column and find extreme values if the column is numeric
for column in data.columns:
    if column not in exclude_columns and isinstance(data.schema[column].dataType, (IntegerType, DoubleType)):
        extreme_values_df = find_extreme_values(data, column)
        if extreme_values_df.count() > 0:
            extreme_values = [row[column] for row in extreme_values_df.collect()]
            for value in extreme_values:
                print(f"{column} has an extreme value: {value}")
        else:
            print(f"No extreme values found for {column}")

```

Figure 21.2

```

No extreme values found for Age
No extreme values found for Cholesterol
No extreme values found for Systolic
No extreme values found for Diastolic
No extreme values found for Heart Rate
No extreme values found for Diabetes
No extreme values found for Family History (1: Yes)
No extreme values found for Obesity
No extreme values found for Alcohol Consumption
No extreme values found for Exercise Hours Per Week
No extreme values found for Previous Heart Problems (1 : Yes)
No extreme values found for Medication Use
No extreme values found for Stress Level
No extreme values found for Sedentary Hours Per Day
No extreme values found for Income
No extreme values found for BMI
No extreme values found for Triglycerides
No extreme values found for Physical Activity Days Per Week
No extreme values found for Sleep Hours Per Day
No extreme values found for Heart Attack Risk (1: Yes)

```

Figure 21.3

3.3 Construct of new variables / features

In our heart attack dataset, there is the *Age* variable. However, for a more insightful analysis, I think it is beneficial to categorize the patients into age groups rather than analysing individual ages. As a result, I created a new variable *Age Group*, from existing *Age* variable. The new variable consists of three different age groups, which are *Young Adults* (age 18-29), *Middle Age* (30 – 59) and *Old* (60-103). The code to create the new variable *Age Group* is depicted in the below Figure 22. Figure 22.1 below shows that there is an additional column now (from 26 columns initially to 27 columns now).

```
from pyspark.sql.functions import when

# Create Age Group column based on age bins
data = data.withColumn(
    "Age Group",
    when((col("Age") >= 18) & (col("Age") < 30), "young adults")
    .when((col("Age") >= 30) & (col("Age") < 60), "middle age")
    .when((col("Age") >= 60) & (col("Age") <= 103), "old")
)
```

Figure 22

```
print(f"Number of columns: {len(data.columns)}")
```

Number of columns: 27

Figure 22.1

Figure 23 below shows the count for the *Age Group* variable. It is evident that majority of the patients belong to the “*Old*” age group of age 60-90.

```
# Perform value counts for the variable "Heart Attack Risk (1: Yes)"
value_counts = data.groupBy("Age Group").count()

# Show the value counts
value_counts.show()
```

```
+-----+-----+
| Age Group|count|
+-----+-----+
|      old| 1045|
| middle age| 1001|
| young adults| 449|
+-----+-----+
```

Figure 23

```

import matplotlib.pyplot as plt

# Compute the distribution of the "Country" variable
agegroup_distribution = data.groupBy("Age Group").count()

agegroup_distribution_local = agegroup_distribution.toLocalIterator()

# Prepare data for plotting
agegroup = []
counts = []
for row in agegroup_distribution_local:
    agegroup.append(row["Age Group"])
    counts.append(row["count"])

# Plot the bar graph
plt.bar(agegroup, counts, color='blue')
plt.xlabel('Age Group')
plt.ylabel('Count')
plt.title('Distribution of Age Group')
plt.tight_layout() # Adjust subplots to fit into the figure area
plt.show()

```

Figure 24

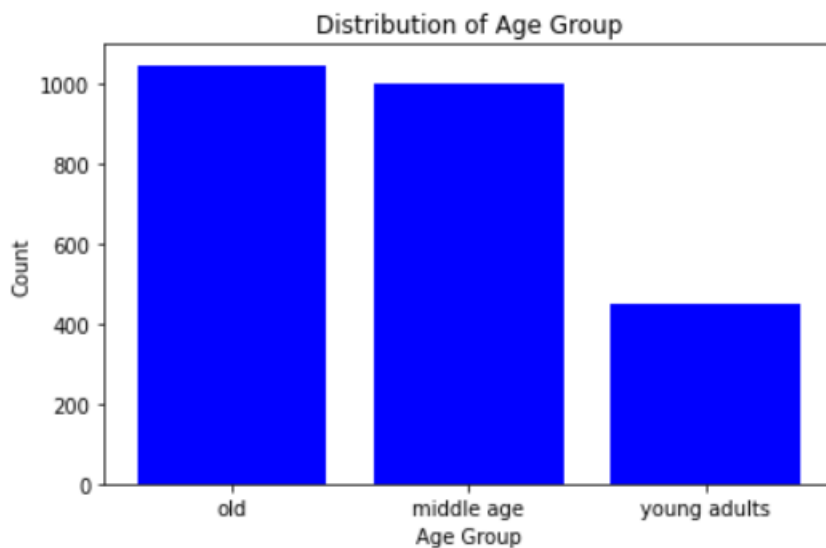


Figure 24.1

An alternative way to get an estimate of count for each age group is to illustrate it in a graph. Figure 24 is the code to produce Figure 24.1. Fig 24.1 illustrates the bar graph for the distribution of *Age Group* variable.

```

from pyspark.sql.functions import count, lit

# Perform value counts for the variable "Heart Attack Risk (1: Yes)"
total_count = data.count()

value_counts = data.groupBy("Heart Attack Risk (1: Yes)").count()
value_counts_percentage = value_counts.withColumn("percentage", (col("count") / total_count) * 100)

# Show the value counts
value_counts_percentage.show()

```

Heart Attack Risk (1: Yes)	count	percentage
1	922	36.95390781563127
0	1573	63.04609218436874

Figure 25

In Figure 25, it shows the distribution % for the target variable *Heart Attack Risk*. We observe that the distribution is relatively skewed (63% for no risk (0) and 37% for presence of risk (1)). This might have a negative influence on the overall predictive power of our model in subsequent analysis. Thus, we have to take some steps to address this issue.

```

import random

# Count the number of samples in each class
class_counts = data.groupBy("Heart Attack Risk (1: Yes)").count().collect()
count_0 = next(row['count'] for row in class_counts if row['Heart Attack Risk (1: Yes)'] == 0)
count_1 = next(row['count'] for row in class_counts if row['Heart Attack Risk (1: Yes)'] == 1)

# Determine the minority and majority classes
minority_class = 1 if count_1 < count_0 else 0
majority_class = 0 if minority_class == 1 else 1
minority_count = min(count_0, count_1)

# Sample from the majority class
majority_class_df = data.filter(col("Heart Attack Risk (1: Yes)") == majority_class)
sampled_majority_class_df = majority_class_df.sample(False, minority_count / count_0)

# Get all minority class samples
minority_class_df = data.filter(col("Heart Attack Risk (1: Yes)") == minority_class)

# Combine both classes to get a balanced DataFrame
balanced_data = sampled_majority_class_df.union(minority_class_df)

# Count the number of samples in each class in the balanced dataset
balanced_class_counts = balanced_data.groupBy("Heart Attack Risk (1: Yes)").count()
balanced_class_counts.show()

```

Figure 26

Figure 26 shows the code to balance out the count for the target variable. The code basically compares the counts of class 0 and class 1 to determine which one is the minority class and which one is the majority class. Then, it balance the dataset by randomly selecting a subset of the majority class.

Heart Attack Risk (1: Yes)		count
0		936
1		922

Figure 26.1

Figure 26.1 above shows the final count for the target variable. It is evident that now there is a close count for presence and absence of heart attack risk (1 and 0).

```
# Save the balanced DataFrame to CSV
#balanced_data.write.csv("balanced_heartdata.csv", header=True)
```

```
# Read in dataset
balanced_data = spark.read.csv('Dataset/balanced_heartdata.csv', header=True, inferSchema=True)
```

Figure 27

After we have cleaned the dataset, we will now save it into a new CSV file.

Figure 27 shows the code to do so. The new csv file has the name 'balanced_heartdata.csv'. Figure 27 also shows the code where we read in the new csv file. This new csv file will be used for subsequent analysis.

```
: print(f"Number of rows: {balanced_data.count()}")  
  print(f"Number of columns: {len(balanced_data.columns)}")
```

Number of rows: 1858

Number of columns: 27

Figure 28

In Figure 28, it is evident that after the balancing for target variable, reduction of data has taken place. The number of entries has reduced from 2495 entries to 1858 entries. The reduction in dataset occurs because we are intentionally removing a subset of instances (from the majority class) to balance the dataset. Specifically, we are reducing the number of instances in the majority class to match the count of the minority class. It is worthy to note that the number of columns still remain the same.

3.4 Data Integration

In my project, I have worked with a single dataset. The challenge arises from the unavailability of another dataset with similar variables to the original one. Therefore, in order to demonstrate my integration skill, I have taken 20% of the cleaned dataset that I export from Jupyter in the previous step. I further split the 20% dataset into two dataset, which is 10% each. I intend to merge the two datasets together so it becomes one again. The reason I chosen to use only 20% of the cleaned dataset is because my dataset is large with 1858 records after the reduction of data I did in previous step. Due to a large dataset, it might not capture important factors so I decided to work on a subset of data instead.

```
# Read in dataset  
file1 = spark.read.csv('Dataset/heart_data_10%(1).csv', header=True, inferSchema=True)  
file2 = spark.read.csv('Dataset/heart_data_10%(2).csv', header=True, inferSchema=True)
```

Figure 29

Figure 29 shows the code to read in the two dataset.

```
: print(f"Number of rows: {file1.count()}")
  print(f"Number of columns: {len(file1.columns)}")

  print(f"Number of rows: {file2.count()}")
  print(f"Number of columns: {len(file2.columns)}")

Number of rows: 500
Number of columns: 27
Number of rows: 500
Number of columns: 27
```

Figure 29.1

Figure 29.1 shows the shape of the two datasets before merging. The two dataset each has 500 rows and 27 columns.

```
: # Merge vertically
  merge_data = file1.union(file2)

: print(f"Number of rows: {merge_data.count()}")
  print(f"Number of columns: {len(merge_data.columns)}")

Number of rows: 1000
Number of columns: 27
```

Figure 30

Figure 30 above shows the code that merge the two datasets together. I have used *union()* function to do the merging job. The shape of the newly merged data is shown in Figure 30. Also, the name of the merged dataset is *merge_data*, and this is the dataset that we will used for subsequent analysis.

3.5 Reformatting data

The current format of the dataset does meet the requirement for analysis. It is considered well-structed and do not have any significant quality issues at this point. If we approach some weird issue in the subsequent analysis, we may have to return to this section to reformat the data then.

Figure 31 and figure 31.1 below depicts the information for the dataset. There are no missing values (empty series returned, shown by blue arrow) and the data types are correct for each variable. Thus, no reformatting is needed.

```
merge_data.printSchema()

root
|-- Age: integer (nullable = true)
|-- Sex: string (nullable = true)
|-- Cholesterol: integer (nullable = true)
|-- Systolic: integer (nullable = true)
|-- Diastolic : integer (nullable = true)
|-- Heart Rate: integer (nullable = true)
|-- Diabetes: integer (nullable = true)
|-- Family History (1: Yes): integer (nullable = true)
|-- Smoking: integer (nullable = true)
|-- Obesity: integer (nullable = true)
|-- Alcohol Consumption: integer (nullable = true)
|-- Exercise Hours Per Week: double (nullable = true)
|-- Diet: string (nullable = true)
|-- Previous Heart Problems (1 : Yes): integer (nullable = true)
|-- Medication Use: integer (nullable = true)
|-- Stress Level: integer (nullable = true)
|-- Sedentary Hours Per Day: double (nullable = true)
|-- Income: integer (nullable = true)
|-- BMI: double (nullable = true)
|-- Triglycerides: integer (nullable = true)
|-- Physical Activity Days Per Week: integer (nullable = true)
|-- Sleep Hours Per Day: integer (nullable = true)
|-- Country: string (nullable = true)
|-- Continent: string (nullable = true)
|-- Hemisphere: string (nullable = true)
|-- Heart Attack Risk (1: Yes): integer (nullable = true)
|-- Age group: string (nullable = true)
```

Figure 31

```
from pyspark.sql.functions import col, sum as spark_sum

# List of columns to check for missing values
columns = merge_data.columns

# Create a DataFrame to store the count of missing values for each column
missing_values = merge_data.select([spark_sum(col(c).isNull().cast("int")).alias(c) for c in columns])

# Filter columns with missing values
missing_columns = missing_values.columns
missing_columns_with_values = [col_name for col_name in missing_columns if missing_values.select(col_name).head()[col_name]]

# Create a new DataFrame with columns that have missing values
missing_values_filtered = missing_values.select(*missing_columns_with_values)

# Show the count of missing values for columns with missing values
missing_values_filtered.show()
```

```
++
||
++
||
++
```




Figure 31.1

```
ubuntu@ip-172-31-60-13:~$ cd Infosys-722---Iteration-4
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git add 'Part 3 - Data Preparation.ipynb'
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git commit -m 'Part 3'
[main 7b1a35c] Part 3
  Committer: Ubuntu <ubuntu@ip-172-31-60-13.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 1 insertion(+), 1 deletion(-)
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git push
Username for 'https://github.com': agneslee01
Password for 'https://agneslee01@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
```

Screenshot of AWS terminal to proof that Part 3 – Data Preparation file is pushed to github.

Step 4: Data Transformation

4.1 Data reduction

In my dataset, there are originally 27 features. However, some features might not be the main contributor to heart attack. Therefore, in order to find out the variables that are potentially contributors, I have used feature selection technique.

However, in order for feature selection chunk of code to run successfully, I will have to convert variables that are of string data type to numerical format. Variables that are of object data type in my dataset are *Sex*, *Diet*, *Country*, *Continent*, *Hemisphere* and *Age group*. To convert these variables, I perform encoding.

```
from pyspark.sql.functions import when

# Encode 'Sex' column
data = data.withColumn("Sex", when(data["Sex"] == "Male", 0).when(data["Sex"] == "Female", 1))
```

Figure 32

In Figure 32 above, it is the code used to perform encoding for the Sex column. Since this column only takes 2 values (male / female), thus I have manually replaced Male with the value 0 and female with the value 1.

```
from pyspark.sql.types import StringType

# Identify columns with string data type and store them in categorical_columns list
categorical_columns = [field.name for field in data.schema.fields if isinstance(field.dataType, StringType)]

# Show the list of categorical columns
print("Categorical columns:", categorical_columns)
```

Categorical columns: ['Diet', 'Country', 'Continent', 'Hemisphere', 'Age group']

Figure 33

In figure 33 above, it is the code that groups the other 5 variables that are still of string data type, into this new string *categorical_columns*. This code is needed because the other 5 variables have more than two categories, such as the Country variable has 19 different countries. We know there are 5 variables of string data type from the above output.

```

from pyspark.sql.types import IntegerType
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline
from pyspark.sql.functions import col

# Dictionary to store mappings
mappings = {}

# Apply StringIndexer to each categorical column and collect the mappings
for column in categorical_columns:
    indexer = StringIndexer(inputCol=column, outputCol=f"{column}_indexed")
    indexer_model = indexer.fit(data)
    data = indexer_model.transform(data)

# Extract and print the mapping of original values to numerical values
labels = indexer_model.labels
mapping = {label: index for index, label in enumerate(labels)}
mappings[column] = mapping
print(f"Mapping for column '{column}': {mapping}")

# Cast the indexed column to integer type
data = data.withColumn(f"{column}_indexed", col(f"{column}_indexed").cast(IntegerType()))

```

```

Mapping for column 'Diet': {'Healthy': 0, 'Unhealthy': 1, 'Average': 2}
Mapping for column 'Country': {'Nigeria': 0, 'Germany': 1, 'United States': 2, 'Australia': 3, 'Italy': 4, 'United Kingdom': 5, 'South Africa': 6, 'Argentina': 7, 'Brazil': 8, 'France': 9, 'Colombia': 10, 'Spain': 11, 'Thailand': 12, 'Vietnam': 13, 'China': 14, 'India': 15, 'Japan': 16, 'Canada': 17, 'New Zealand': 18, 'South Korea': 19}
Mapping for column 'Continent': {'Asia': 0, 'Europe': 1, 'South America': 2, 'Africa': 3, 'North America': 4, 'Australia': 5}
Mapping for column 'Hemisphere': {'Northern Hemisphere': 0, 'Southern Hemisphere': 1}
Mapping for column 'Age group': {'Old': 0, 'Middle Age': 1, 'Young Adults': 2}

```

Figure 33.1

In Figure 33.1 above, the chunk of code is we apply label encoding to each categorical columns that are stored in the new string *categorical_columns* (in Figure 33). The code (circled in red) is to inspect the mapping of the original categorical values to their encoded numerical representations. This helps in understanding which number represents which category.

The output shown in Figure 33.1 corresponds to the code output presented at the bottom of Figure 33.1. This output clarifies that, for example, in the variable "Diet," the value 0 represents "Healthy," while 1 represents "unhealthy," and so on.

```

# Drop the original categorical columns
data = data.drop(*categorical_columns)

```

Figure 34

```

print(f"Number of columns: {len(data.columns)}")

```

Number of columns: 27

Figure 34.1

In Figure 34, the code is to drop the original categorical columns after label encoding. Figure 34.1 shows we have 27 columns after dropping the original categorical columns. This basically means we do not have duplicate columns.

Now that we have completed the encoding for the categorical variables, we can move on to feature selection.

```
# Define the target column
target_column = "Heart Attack Risk (1: Yes)"

# Define the feature columns (excluding the target column)
feature_columns = [col for col in data.columns if col != target_column ]

# Split the DataFrame into X (features) and Y (target)
X = data.select(*feature_columns)
Y = data.select(target_column)

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.linalg import Vectors

# Combine feature columns into a single vector column
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data_assembled = assembler.transform(data)
```

Figure 35

```
# Train a RandomForest model
rf = RandomForestClassifier(labelCol=target_column, featuresCol="features", numTrees=10)
model = rf.fit(data_assembled)

# Extract feature importance
feature_importance = model.featureImportances

# Combine feature names and importance scores
feature_names = feature_columns
feature_importance_dict = dict(zip(feature_names, feature_importance))

# Sort features by importance
sorted_features = sorted(feature_importance_dict.items(), key=lambda x: x[1], reverse=True)

# Print selected feature names with importance scores
#top_n = 12 # Specify the number of top features you want to print
#for feature, importance in sorted_features[:top_n]:
#    print(f"Feature: {feature}, Importance: {importance}")

# Print selected feature names with importance scores
#top_n = 12 # Specify the number of top features you want to print
top_features = sorted_features[:top_n]

# Print the table header
print("Feature | Importance")
print("-----")

# Print selected feature names with importance scores
for feature, importance in top_features:
    print(f"Feature: {feature} | Importance: {importance}")
```

Figure 35.1

Figure 35 and Figure 35.1 above is the code to produce the important features. Firstly, I have set *top_n=12* . This will output the top 12 features. We use the `RandomForestClassifier()` function to generate feature importance (code circled in red). Then, we sort the features based on their importance value in descending order.

Feature	Importance
Income	0.10
Cholesterol	0.10
Country_indexed	0.09
Diastolic	0.09
Exercise Hours Per Week	0.08
Heart Rate	0.07
Triglycerides	0.07
BMI	0.06
Sedentary Hours Per Day	0.06
Age	0.05
Stress Level	0.05
Systolic	0.03

Figure 35.2

Figure 35.2 above can provide insight into the importance of each selected feature. In Figure 35.2 it depicts the importance of the variable with their corresponding scores. The highest score is 0.10 (round to two decimal place). However, I have decided to take into consideration variables that have a score of 0.05 and higher, which are *Income, Cholesterol, Country_indexed, Diastolic, Exercise Hours Per Week, Heart Rate, Triglycerides, BMI, Sedentary Hours Per day, Age and Stress Level*.

```
# Specify the column names you want to keep
columns_to_keep = ["Heart Attack Risk (1: Yes)", "Income", "Cholesterol", "Country_indexed", "Diastolic ",
                  "Exercise Hours Per Week", "Heart Rate", "Triglycerides", "BMI", "Sedentary Hours Per Day", "Age",
                  "Stress Level"]

# Select only the columns to keep
new_data = data_assembled.select(*columns_to_keep)
```

Figure 36

```

: new_data.printSchema()

root
 |-- Heart Attack Risk (1: Yes): integer (nullable = true)
 |-- Income: integer (nullable = true)
 |-- Cholesterol: integer (nullable = true)
 |-- Country_indexed: integer (nullable = true)
 |-- Diastolic : integer (nullable = true)
 |-- Exercise Hours Per Week: double (nullable = true)
 |-- Heart Rate: integer (nullable = true)
 |-- Triglycerides: integer (nullable = true)
 |-- BMI: double (nullable = true)
 |-- Sedentary Hours Per Day: double (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Stress Level: integer (nullable = true)

```

Figure 36.1

Figure 36.1 above shows the features we have selected from the dataset. Previously, we have decided to keep only those features with score of 0.05 and greater. The 12 columns (shown in Figure 36.1), are the target and features with value of 0.05 and greater. The code is shown in Figure 36.

```

# Coalesce the DataFrame to a single partition
new_data_single_partition = new_data.coalesce(1)

# Save the DataFrame to a single CSV file
new_data_single_partition.write.csv("new_data.csv", header=True)

# Read in dataset
new_data = spark.read.csv('Dataset/new_data.csv', header=True, inferSchema=True)

```

Figure 37

```

: print(f"Number of rows: {new_data.count()}")
  print(f"Number of columns: {len(new_data.columns)}")

```

```

Number of rows: 1000
Number of columns: 12

```

Figure 37.1

Figure 37 above shows that I have saved and load the new dataset (with only features of interest) into jupyter, with the name *new_data*. As we can see from the shape of the new dataset, there are 1000 rows and 12 columns.

```
# Perform value counts for the variable "Heart Attack Risk (1: Yes)"
value_counts = new_data.groupBy("Heart Attack Risk (1: Yes)").count()

# Show the value counts
value_counts.show()
```

Heart Attack Risk (1: Yes)	count
1	509
0	491

Figure 38

Figure 38 is the value count for the target variable. The count seems reasonable as it's not too big difference between them.

4.2 Data Projection

```
: from pyspark.sql.functions import stddev

# Print header
print("{:<30} {:<20}".format("Column", "Standard Deviation"))
print("-----")

# Calculate and print the standard deviation for each column
for column in new_data.columns:
    std_dev = new_data.agg(stddev(column)).collect()[0][0]
    print("{:<30} | {:<20}".format(column, std_dev))
```

Figure 39

Column	Standard Deviation
Heart Attack Risk (1: Yes)	0.5001691405606395
Income	80973.01257467821
Cholesterol	79.82551652252607
Country_indexed	5.749640803508454
Diastolic	14.649868588135842
Exercise Hours Per Week	5.735343348205532
Heart Rate	20.165317156345836
Triglycerides	221.82688765586357
BMI	6.298476971781967
Sedentary Hours Per Day	3.5290078215488867
Age	21.32262719344007
Stress Level	2.8425418280201344

Figure 39.1

Figure 39.1 is the standard deviation for each variable and the code is in Figure 39. Looking at the standard deviation values, it is noticeable that *Income* variable has a very high standard deviation.

In order to lower the standard deviation, we can do statistical transformation, such as the log of (x), where x is the variable. In this section, we will take the log of income variable. Upon doing the log transformation, the standard deviation has indeed become smaller than before. This is because the log transformation has effectively stabilized the variance, thus results in a much smaller standard deviation. This is depicted in Figure 40 below.

```

: from pyspark.sql.functions import log

# Log transformation of 'income' variable
log_income_values = new_data.select(log(new_data["Income"] + 1).alias("log_income"))

# Calculate the standard deviation after log transformation
std_dev_log_income = log_income_values.agg(stddev("log_income")).collect()[0][0]

# Print the standard deviation after log transformation
print("Standard deviation of log transformed income:", std_dev_log_income)

```

Standard deviation of log transformed income: 0.6780097873473351

Figure 40

This is to demonstrate how to undergo a transformation, but I do not want to transform the income variable because a significant decrease in standard deviation means a reduction in income variability. In subsequent analysis, I wish to take into account the variability of income in order to produce a more powerful predictive model.

```
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ cd Infosys-722---Iteration-4
-bash: cd: Infosys-722---Iteration-4: No such file or directory
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git add 'Part 4 - Data Transformation.ipynb'
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git commit -m 'Part 4'
[main c969dc0] Part 4
  Committer: Ubuntu <ubuntu@ip-172-31-60-13.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 637 insertions(+)
create mode 100644 Part 4 - Data Transformation.ipynb
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git push
Username for 'https://github.com': agneslee01
Password for 'https://agneslee01@github.com':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
```

Screenshot of AWS terminal to proof that Part 4 – Data Transformation file is pushed to github.

Step 5: Data- mining method(s) selection

5.1 – Discussion of data mining methods within the context of data mining objectives

In this section, we aim to discuss some data mining methods which align with our data mining objectives. But firstly, we will have a quick look into the data types for our subsequent data mining process. The data types are a mixture of continues variable and binary variable. There are various modelling techniques that can be used, such as classification etc. We will delve deeper to have a better understanding of some data mining methods. Ali (2023) talks about one of the data mining techniques is **association**. “Association technique involves looking for certain occurrences with connected attributes” (Ali, 2023). Like the term itself, it refers to the relationship between different variables. This technique could be useful especially when we wish to draw certain assumption to determine if one variable has an effect on another variable. The next common data mining method is **classification**. “Classification in data mining is a technique used to assign labels or classify each instance, record, or data object in a dataset based on their features or attributes” (Utkarsh, 2023). Again, from the term itself, this suggest grouping in common term. (Utkarsh, 2023) also mention that classification techniques can be divided into categories - binary classification and multi-class classification. Binary means it only has 2 distinct values, while multi-class means it has more than 2 values, as the word multi suggest. Another common data mining method is **clustering**. (Sehgal, 2022) describes clustering as putting items with similarity together but could look for values that are out of the norm range, such as outliers. Classification and clustering on simple term may sound alike, but in fact, both techniques are different. Firstly, classification is supervised learning, and clustering is

unsupervised learning. “Supervised Learning works with the help of a well-labelled dataset, in which the target output is well known” (Baheti, 2021). On the other hand, unsupervised learning is the opposite. “Algorithm is trained using data that is unlabelled. The machine tries to identify the hidden patterns and give the response” (Baheti, 2021). Additionally, **feature selection** is also an option. Feature selection by its own term means select features that are consider important in relation to the target variable as often dataset could consist of many features but some may be redundant or irrelevant. In the next subsection, we will select the appropriate data mining method, which align with our data mining objectives / goal.

5.2 - Select the appropriate data mining methods

The first objective is to measure the likelihood of individuals towards developing heart disease. The target variable is *heart attack risk*, which is a binary variable that takes the value 0 or 1. In simpler term, we are interested to predict the chances of individuals developing heart disease, thus we will use classification method for this first objective. In the above subsection, it is mentioned that classification can handle binary variables, thus it is the appropriate method to tackle this particular objective.

The second objective is to provide actionable insights that empower informed decision-making by exploring relationship between factors and heart attack in order to identify patterns and promote proactive measures for heart disease prevention based on findings. We will use association method to handle this objective. This is because we intend to find the significance of association between variables, which contributes to more effective strategies for heart disease prevention. We may also implement clustering method to identify

distinct risk profile with the aim to make proactive measures more personalized.

The third objective is to determine the main contributor to heart attack disease. In order to achieve this objective, we will utilise the feature selection method. As mentioned in the subsection above, feature selection will only keep some features and filter out the rest that are being determined to be irrelevant or play an unimportant role to heart attack disease. Feature selection will only select a few variables that are potentially main contributors to heart attack disease.

These are the method that we will implement in subsequent analysis. The algorithm selection will be discussed in the next step.

Step 6: Algorithm Selection

6.1 – Exploratory analysis of data mining objectives

Association technique is used to find relationship between features and produce result that can be used for the purpose of prediction and / or decision. Among association technique, there are a few algorithms. FPGrowth algorithm is one of them. “FPGrowth algorithm uses a bottom-up approach, starting with individual items and gradually building up to more complex itemsets” (All, 2023). FPGrowth algorithm search for common patterns and association rules within extensive datasets.

Classification techniques consists of many different algorithms. For instance, decision tree is one of them. “Decision tree works best for simple cases with few variables, thus often used as first line classification method” (Keserer, 2023). Decision tree algorithm is able to handle categorical variable without

the need to transform it to a factor. This makes the process easier and straightforward since dataset may contain more than one type of data types. However, decision tree has some disadvantages. “It is a high variance algorithm, may easily overfit because it has no inherent mechanism to stop, thereby creating complex decision rule” (Kapil, 2022). From this piece of statement, this problem might arise if there are too many variables in the dataset. So, it does support the reference above quoted by (Keserer, 2023) that decision tree is a powerful algorithm for dataset with less variables.

On the other hand, according to (AIML.com, 2024), logistic regression which is another classification algorithm, is able to handle large number of features. (AIML.com, 2024) has also mentioned that different data types are acceptable for logistic regression, for instance continuous and categorical variables.

For clustering technique, one of the common algorithms is the K-means clustering algorithm. “K-means minimize the variance of data points within a cluster” (McGregor, 2020). (McGregor, 2020) has also noted that K-means cluster is suitable for small dataset because the algorithm is designed to go through all data point. From this, we may determine that the disadvantage of this algorithm is it can be time consuming for large dataset since it needs to iterate over all the data.

6.2 – Select algorithm(s) based on discussion

The first objective is to predict the chances of individuals developing heart disease. The target variable is *heart attack risk*, which is a binary variable that takes the value 0 or 1. It is mentioned in section 5.2 that we will using the

classification method. In particular, we will use **logistic regression** since the target is a binary variable. Also, we will implement the **decision tree algorithm** because this algorithm, as mentioned above, is a powerful algorithm for dataset with less variables. There are only 8 features and 1 target in our dataset, so it is sensible to use this algorithm for our analysis and hopefully derive interesting patterns from. The second objective is to provide actionable insights that empower informed decision-making by exploring relationship between factors and heart attack in order to identify patterns and promote proactive measures for heart disease prevention based on findings. As mentioned previously, this objective will be solved using the association method. In particular, we will try using the **FPGrowth algorithm**. As for clustering method, we will use the **two-steps clustering algorithm**.

The third objective is to find the main contributor. We will use **feature selection method**, based on the RandomForestClassifier scoring function, as this is a straightforward and easy method to find out the variables quickly.

6.3 – Build / Select model

Feature selection:

```
: print(f"Number of rows: {new_data.count()}")  
  print(f"Number of columns: {len(new_data.columns)}")
```

```
Number of rows: 1000  
Number of columns: 12
```

Figure 41


```
root
|-- Heart Attack Risk (1: Yes): integer (nullable = true)
|-- Income: integer (nullable = true)
|-- Cholesterol: integer (nullable = true)
|-- Country_indexed: integer (nullable = true)
|-- Diastolic: integer (nullable = true)
|-- Exercise Hours Per Week: double (nullable = true)
|-- Heart Rate: integer (nullable = true)
|-- Triglycerides: integer (nullable = true)
|-- BMI: double (nullable = true)
|-- Sedentary Hours Per Day: double (nullable = true)
|-- Age: integer (nullable = true)
|-- Stress Level: integer (nullable = true)
```

Figure 41.1

There are originally 27 fields, but after the feature selection, only 12 fields left. These variables are potentially contributor to heart attack, and we will do further analysis with these 12 fields, to get a better understanding.

Logistic regression:

In the below Figure 42, it shows the chunk of code where we combine feature columns into a single vector column.

In the below figure 42.1, it shows the chunk of code for logistic regression. The output will be shown in step 7. We proceed to fit the logistic regression. Subsequently, we also make prediction on the testing and training set and produce accuracy score for both sets.

In Figure 42.2 below, the code sections extract the features coefficients, which indicate the significance of each feature. In Figure 42.3, the code section create a cross-tabulation of predicted vs. actual class for the target variable.

```

from pyspark.sql.functions import col
from pyspark.ml.feature import VectorAssembler

# Assuming 'data' is your original DataFrame
target_column = "Heart Attack Risk (1: Yes)"
feature_columns = [col for col in data.columns if col != target_column]

# Combine feature columns into a single vector column
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data1 = assembler.transform(data).select("features", target_column)

# Split the assembled data into training and testing sets
train_data, test_data = data1.randomSplit([0.8, 0.2], seed=42)

```

Figure 42

```

from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.functions import expr

# Fit the Logistic Regression model
lr = LogisticRegression(labelCol=target_column, featuresCol="features")
lr_model = lr.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = lr_model.transform(train_data)
test_predictions = lr_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.2f}")
print(f"Testing Accuracy: {test_accuracy:.2f}")

```

Figure 42.1

```

# Extract feature coefficients
coefficients = lr_model.coefficients
intercept = lr_model.intercept

# Convert feature coefficients to a DataFrame for better readability
coefficients_df = spark.createDataFrame(
    [(feature, round(float(coeff), 6)) for feature, coeff in zip(feature_columns, coefficients)],
    ["Feature", "Coefficient"]
)

# Show feature coefficients
coefficients_df.show(truncate=False)

```

Figure 42.2

```
# Create a cross-tabulation of predicted vs. actual classes for the test set  
cross_tab = test_predictions.groupby("prediction").pivot(target_column).count()  
  
# Display the cross-tabulation  
cross_tab.show()
```

Figure 42.3

Decision tree algorithm:

Figure 43 below shows the code for the decision tree algorithm. The output will be shown in step 7. We proceed to fit the decision tree. Subsequently, we also make prediction on the testing and training set and produce accuracy score for both sets.

In Figure 43.1 below, the decision tree algorithm allow us to see the feature importance values based on how much each feature contributes to the model's decision-making process .However, decision trees do not provide coefficient magnitudes like logistic regression. In Figure 43.2, the chunk of code section produce tree rules in a more readable manner. In Figure 43.3, the code section produce the cross-tab result for target variable. Output will be shown in next step.

```

from pyspark.ml.classification import DecisionTreeClassifier

# Fit the Decision Tree model
dt = DecisionTreeClassifier(labelCol=target_column, featuresCol="features")
dt_model = dt.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = dt_model.transform(train_data)
test_predictions = dt_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Testing Accuracy: {test_accuracy:.4f}")

```

Figure 43

```

# Produce feature importance values
importances = dt_model.featureImportances
feature_importances = [(feature, round(float(importance), 4)) for feature, importance in zip(feature_columns, importances)]

# Convert feature importances to a DataFrame
importances_df = spark.createDataFrame(feature_importances, ["Feature", "Importance"])
importances_df = importances_df.orderBy(col("Importance").desc())

# Show feature importances
importances_df.show(truncate=False)

```

Figure 43.1

```

from pyspark.ml.classification import DecisionTreeClassificationModel

def extract_rules(tree_model):
    # Get the debug string
    tree_debug_string = tree_model._java_obj.toDebugString()

    # Print the debug string for debugging
    print(tree_debug_string)

    # Split the debug string by line and remove empty lines
    lines = filter(None, tree_debug_string.split("\n"))

    # Initialize an empty list to store the rules
    rules = []

    # Iterate through each line in the debug string
    for line in lines:
        # Skip lines that don't represent decision nodes
        if not line.startswith("If") and not line.startswith("Else"):
            continue

        # Extract the rule from the line
        rule = line.strip().replace("(", "").replace(")", "").replace("feature ", "").replace("<=", "<=").replace(">", ">")
        rules.append(rule)

    return rules

# Assuming dt_model is your trained DecisionTreeClassificationModel
rules = extract_rules(dt_model)

# Print the rules
for rule in rules:
    print(rule)

```

Figure 43.2

```

# Create a cross-tabulation of predicted vs. actual classes for the test set
cross_tab = test_predictions.groupBy("prediction").pivot(target_column).count()

# Display the cross-tabulation
cross_tab.show()

```

Figure 43.3

Association algorithm:

```

from pyspark.sql.functions import array, col, concat_ws

# Combine all columns into a single column containing the items for each transaction
data2 = data.withColumn("items", concat_ws(",", *data.columns))

# Select only the items column
transaction_data = data2.select("items")

```

```

from pyspark.ml.fpm import FPGrowth
from pyspark.sql.functions import split, array_distinct
from pyspark.sql.functions import col, create_map, lit
from itertools import chain

# Split the concatenated string into an array of items
transaction_data = data2.withColumn("items", split("items", ","))

# Deduplicate the items within each transaction
transaction_data = transaction_data.withColumn("items", array_distinct("items"))

# Create an FPGrowth object
fp_growth = FPGrowth(itemsCol="items", minSupport=0.1, minConfidence=0.1)

# Fit the FPGrowth model
model = fp_growth.fit(transaction_data)

# Get the frequent itemsets
frequent_itemsets = model.freqItemsets

# Get the association rules
association_rules = model.associationRules

# Get the association rules
association_rules = model.associationRules

# Define a mapping dictionary for feature names
feature_map = {
    0: "Heart Attack Risk (1: Yes)",
    1: "Income",
    2: "Cholesterol",
    3: "Country_indexed",
    4: "Diastolic",
    5: "Exercise Hours Per Week",
    6: "Heart Rate",
    7: "Triglycerides",
    8: "BMI",
    9: "Sedentary Hours Per Day",
    10: "Age",
    11: "Stress Level",
}

# Convert the mapping dictionary to a Spark DataFrame column
mapping_expr = create_map([lit(x) for x in chain(*feature_map.items())])

# Apply the mapping to the antecedent and consequent columns
association_rules_mapped = association_rules.withColumn(
    "antecedent_feature", mapping_expr.getItem(col("antecedent")[0])
).withColumn(
    "consequent_feature", mapping_expr.getItem(col("consequent")[0])
).drop("antecedent", "consequent")

# Show the association rules with feature names
association_rules_mapped.show(truncate=False)

```

Figure 44

In Figure 44 above shows the code and parameter setting for FPGrowth algorithm.

The parameter *min_support = 0.1* means only itemsets appearing in at least 10% of the dataset will be considered frequent. The parameter *minConfidence=0.1* means that it sets the minimum confidence threshold for generating association rules. *truncate=False* parameter ensures that the column values are not truncated when displayed.

Two-step clustering:

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline

# Define your clustering model
kmeans = KMeans().setK(3).setSeed(42) # Example: 3 clusters, seed for reproducibility

# Pipeline to assemble features and fit KMeans model
pipeline1 = Pipeline(stages=[assembler, kmeans])

train_data1, test_data1 = data.randomSplit([0.8, 0.2], seed=42)

# Fit the pipeline to the train data
model1 = pipeline1.fit(train_data1)

# Make predictions on both train and test data
train_predictions1 = model1.transform(train_data1)
test_predictions1 = model1.transform(test_data1)

# Evaluate clustering performance using silhouette score
evaluator = ClusteringEvaluator()

# Compute silhouette score for train data
train_silhouette_score = evaluator.evaluate(train_predictions1)
print("Silhouette Score for Train Data:", train_silhouette_score)

# Compute silhouette score for test data
test_silhouette_score = evaluator.evaluate(test_predictions1)
print("Silhouette Score for Test Data:", test_silhouette_score)
```

```

import matplotlib.pyplot as plt
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.feature import VectorAssembler

# Fit KMeans model
kmeans = KMeans().setK(3).setSeed(42) # 3 clusters, seed for reproducibility
pipeline = Pipeline(stages=[assembler, kmeans])
model = pipeline.fit(data)

# Make predictions
predictions = model.transform(data)

# Evaluate clustering performance using silhouette score
evaluator = ClusteringEvaluator()
silhouette_score = evaluator.evaluate(predictions)
print("Silhouette Score:", silhouette_score)

# Get count of samples in each cluster
cluster_counts = predictions.groupBy('prediction').count().orderBy('prediction').collect()

# Plot count of samples in each cluster
labels = [f"Cluster {row['prediction']}" for row in cluster_counts]
sizes = [row['count'] for row in cluster_counts]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)
plt.axis('equal')
plt.title('Count of Samples in Each Cluster')
plt.show()

```

```

# Draw boxplot for each variable in each cluster (showing only the first 4 features)
num_clusters = 3 # Assuming 3 clusters
num_features = 4 # Show boxplots for the first 4 features
fig, axes = plt.subplots(num_features, num_clusters, figsize=(15, 10))

# Select only the first 4 features
feature_columns_subset = feature_columns[:num_features]

for i, feature in enumerate(feature_columns_subset):
    for j in range(num_clusters):
        cluster_data = predictions.filter(predictions["prediction"] == j).select(feature)
        cluster_values = [row[feature] for row in cluster_data.collect()]
        axes[i, j].boxplot(cluster_values)
        axes[i, j].set_title(f'Cluster {j}')
        axes[i, j].set_xlabel(feature)
        axes[i, j].set_ylabel('Values')

plt.tight_layout()
plt.show()

```

Figure 45

Figure 45 shows the code and parameter setting for K-Means clustering. I have decided to have 3 clusters, and that is what the parameter *num_clusters* = 3 does (circled in red). The *set.seed* parameter (circled in red too) controls the randomness of the algorithm's. If we do not specify a value for this parameter,

the algorithm will use a different random seed each time it is run. This can lead to variations in the results on every runs, even with the same dataset and parameters. The code chunk circled in blue, is a for loop that iterates over each feature and create separate boxplot for each cluster.

```
ubuntu@ip-172-31-60-13:~$ cd Infosys-722---Iteration-4
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git add 'Part 6 - Algorithm Selection.ipynb'
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git commit -m 'Part 6'
[main 3466a84] Part 6
Committer: Ubuntu <ubuntu@ip-172-31-60-13.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 772 insertions(+)
create mode 100644 Part 6 - Algorithm Selection.ipynb
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git push
Username for 'https://github.com': agneslee01
Password for 'https://agneslee01@github.com':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 38.94 KiB | 9.73 MiB/s, done.
```

Screenshot of AWS terminal to proof that Part 6 – Algorithm Selection file is pushed to github.

Step 7: Data Mining

7.1 – Create logical test

```
# Split the assembled data into training and testing sets
train_data, test_data = data1.randomSplit([0.8, 0.2], seed=42)
```

Figure 46

In the above Figure 46 shows the code to split the dataset into train and test set. I have decided to go with the typical split ratio, 80:20. 80% for training and 20% for testing. The reason behind the split is due to no other data to test our

model. I have also added the parameter *seed= 42*. This parameter ensures reproducibility of the results when the same code is run multiple times. This also ensure we get consistent results every time we run it.

The model performance is being evaluated on the 20% test set to see how the model perform on new data. Train set has more data, this means that model may have a better chance of capturing the underlying pattern in the data with lower bias and lead to better model performance. Train test split is used to estimate the performance of algorithms that are used to make predictions on data not used to train the model. Train test split is important because often in real world, the algorithm is needed to make prediction on data that is completely new or unseen data. Also, it is important that we get a result such as accuracy score, that is as accurate as possible. This is when train test split comes in useful for us to test.

7.2 – Conduct data mining

In this section, we will be using the 80% of our data since we have decided to split the data in the 80:20 ratio. All the models are run successfully and the output are included in this section. However, interpretation of the output will be in step 8.

Logistic regression output:

Figure 47 : code

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.functions import expr

# Fit the Logistic Regression model
lr = LogisticRegression(labelCol=target_column, featuresCol="features")
lr_model = lr.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = lr_model.transform(train_data)
test_predictions = lr_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.2f}")
print(f"Testing Accuracy: {test_accuracy:.2f}")
```

```
# Extract feature coefficients
coefficients = lr_model.coefficients
intercept = lr_model.intercept

# Convert feature coefficients to a DataFrame for better readability
coefficients_df = spark.createDataFrame(
    [(feature, round(float(coef), 6)) for feature, coef in zip(feature_columns, coefficients)],
    ["Feature", "Coefficient"]
)

# Show feature coefficients
coefficients_df.show(truncate=False)
```

```
# Create a cross-tabulation of predicted vs. actual classes for the test set
cross_tab = test_predictions.groupBy("prediction").pivot(target_column).count()

# Display the cross-tabulation
cross_tab.show()
```

Training Accuracy: 0.55

Testing Accuracy: 0.49

Figure 47.1 : Accuracy score on test and train set

Feature	Coefficient
Income	-1.0E-6
Cholesterol	0.001431
Country_indexed	2.9E-5
Diastolic	-0.004038
Exercise Hours Per Week	-0.007842
Heart Rate	0.001061
Triglycerides	-2.17E-4
BMI	-0.008633
Sedentary Hours Per Day	0.007461
Age	0.006027
Stress Level	0.03614

Figure 47.2 : coefficient result

prediction	0	1
0.0	31	33
1.0	50	48

Figure 47.3 : cross-tab output

Decision tree output:

```
from pyspark.ml.classification import DecisionTreeClassifier

# Fit the Decision Tree model
dt = DecisionTreeClassifier(labelCol=target_column, featuresCol="features")
dt_model = dt.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = dt_model.transform(train_data)
test_predictions = dt_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Testing Accuracy: {test_accuracy:.4f}")

# Produce feature importance values
importances = dt_model.featureImportances
feature_importances = [(feature, round(float(importance), 4)) for feature, importance in zip(feature_columns, importances)]

# Convert feature importances to a DataFrame
importances_df = spark.createDataFrame(feature_importances, ["Feature", "Importance"])
importances_df = importances_df.orderBy(col("Importance").desc())

# Show feature importances
importances_df.show(truncate=False)
```

```

from pyspark.ml.classification import DecisionTreeClassificationModel

def extract_rules(tree_model):
    # Get the debug string
    tree_debug_string = tree_model._java_obj.toDebugString()

    # Print the debug string for debugging
    print(tree_debug_string)

    # Split the debug string by line and remove empty lines
    lines = filter(None, tree_debug_string.split("\n"))

    # Initialize an empty list to store the rules
    rules = []

    # Iterate through each line in the debug string
    for line in lines:
        # Skip lines that don't represent decision nodes
        if not line.startswith("If") and not line.startswith("Else"):
            continue

        # Extract the rule from the line
        rule = line.strip().replace("(", "").replace(")", "").replace("feature ", "").replace("<=", "<=").replace(">", ">")
        rules.append(rule)

    return rules

# Assuming dt_model is your trained DecisionTreeClassificationModel
rules = extract_rules(dt_model)

# Print the rules
for rule in rules:
    print(rule)

```

Figure 48: Code

Training Accuracy: 0.6444
 Testing Accuracy: 0.5123

Figure 48.1 : Accuracy score on test & train set

Feature	Importance
Sedentary Hours Per Day	0.1724
BMI	0.1652
Exercise Hours Per Week	0.1141
Triglycerides	0.1118
Cholesterol	0.0947
Age	0.0927
Country_indexed	0.0643
Heart Rate	0.0591
Diastolic	0.0508
Stress Level	0.048
Income	0.0269

Figure 48.2 : Importance value for each features

```

If (feature 5 <= 69.5)
  Predict: 1.0
Else (feature 5 > 69.5)
  Predict: 0.0

If (feature 6 <= 766.0)
  If (feature 8 <= 4.1592155245)
    If (feature 10 <= 2.5)
      If (feature 2 <= 1.5)
        Predict: 0.0
      Else (feature 2 > 1.5)
        Predict: 1.0
    
```

Figure 48.3 : Rule set example

```

+-----+-----+
|prediction| 0| 1|
+-----+-----+
|      0.0| 40| 38|
|      1.0| 41| 43|
+-----+-----+

```

Figure 48.3 : Cross-tab output for decision tree

FPGrowth Output:

confidence	lift	support	antecedent_feature	consequent_feature
0.20342205323193915	0.3501240158897404	0.107	Heart Attack Risk (1: Yes)	Income
0.6024096385542169	1.036849636065778	0.1	Heart Rate	Income
0.1721170395869191	1.0368496360657777	0.1	Income	Heart Rate
0.18416523235800344	0.35012401588974035	0.107	Income	Heart Attack Risk (1: Yes)

Figure 49 : Result is displayed in this table format

Two-step clustering output:

```

Silhouette Score for Train Data: 0.7501455549984403
Silhouette Score for Test Data: 0.763603385563619

```

Figure 50: Silhouette score

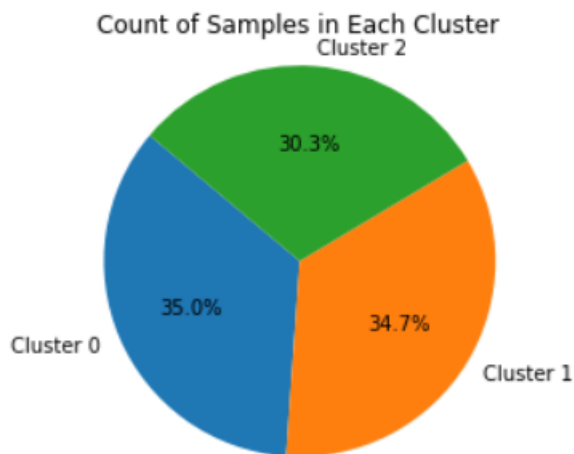


Figure 50.1: Cluster Distribution chart

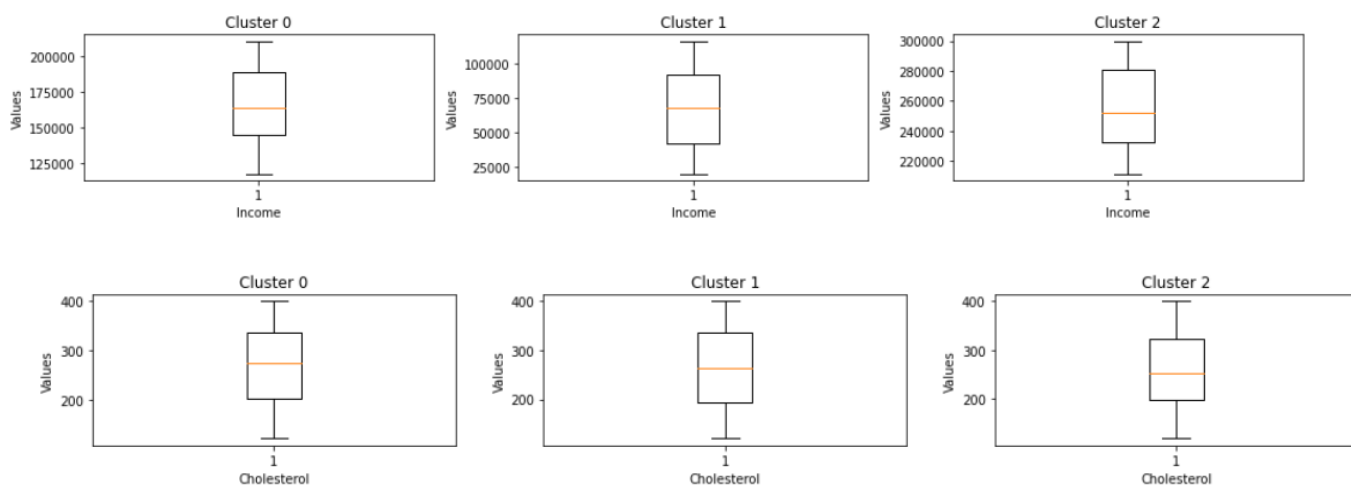


Figure 50.2: Boxplot for each feature (example)

7.3 – Search for patterns

Logistic regression

Training Accuracy: 0.55
Testing Accuracy: 0.49

+-----+-----+			
prediction	0	1	
+-----+-----+			
	0.0	31	33
	1.0	50	48
+-----+-----+			

Feature	Coefficient
Income	-1.0E-6
Cholesterol	0.001431
Country_indexed	2.9E-5
Diastolic	-0.004038
Exercise Hours Per Week	-0.007842
Heart Rate	0.001061
Triglycerides	-2.17E-4
BMI	-0.008633
Sedentary Hours Per Day	0.007461
Age	0.006027
Stress Level	0.03614

Figure 51 : Logistic regression model accuracy (top left), cross-tab output (top right) & coefficient summary (bottom)

From the coefficient summary output, we can see there are multiple features that are considered significant to predict the target. From the cross-tab output, we can get an insight about the true negative and false positive values. Result shall be interpreted in the step 8.

Decision tree algorithm

Training Accuracy: 0.6444
Testing Accuracy: 0.5123

Figure 52 : Decision tree accuracy score

Feature	Importance
Sedentary Hours Per Day	0.1724
BMI	0.1652
Exercise Hours Per Week	0.1141
Triglycerides	0.1118
Cholesterol	0.0947
Age	0.0927
Country_indexed	0.0643
Heart Rate	0.0591
Diastolic	0.0508
Stress Level	0.048
Income	0.0269

Figure 52.1 : Feature importance (descending)


```

If (feature 5 <= 69.5)
  Predict: 1.0
Else (feature 5 > 69.5)
  Predict: 0.0

If (feature 6 <= 766.0)
  If (feature 8 <= 4.1592155245)
    If (feature 10 <= 2.5)
      If (feature 2 <= 1.5)
        Predict: 0.0
      Else (feature 2 > 1.5)
        Predict: 1.0
    
```

Figure 52.2 : Rule set output

```

+-----+-----+
|prediction| 0| 1|
+-----+-----+
|         0.0| 40| 38|
|         1.0| 41| 43|
+-----+-----+

```

Figure 52.3 : Cross-tab output for decision tree

From fig 52.2, we can see the pattern where the algorithm list out possible set of rules, such as rules for heart attack labelled as class 1 or 0 and the feature importance in descending order in Figure 52.1. From the cross-tab output, we can get an insight about the true negative and false positive values. Result shall be interpreted in the step 8.

FPGrowth algorithm

```

+-----+-----+-----+-----+-----+
|confidence|lift|support|antecedent_feature|consequent_feature|
+-----+-----+-----+-----+-----+
|0.20342205323193915|0.3501240158897404|0.107|Heart Attack Risk (1: Yes)|Income|
|0.6024096385542169|1.036849636065778|0.1|Heart Rate|Income|
|0.1721170395869191|1.0368496360657777|0.1|Income|Heart Rate|
|0.18416523235800344|0.35012401588974035|0.107|Income|Heart Attack Risk (1: Yes)|
+-----+-----+-----+-----+-----+

```

Figure 53: FPGrowth algorithm output

We found the pattern that the algorithm takes the variable as the target and compare it to other variables, to find the relationship between them.

Two-step clustering

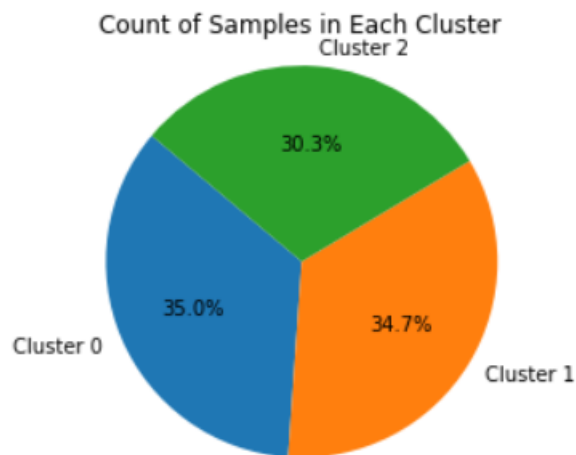


Figure 54: Cluster Distribution chart

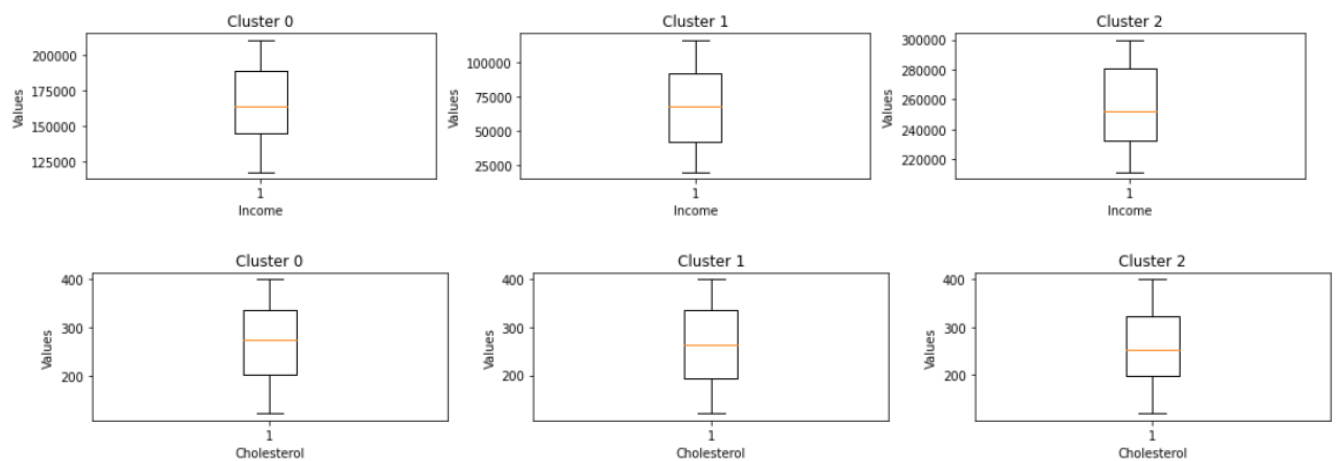


Figure 54.1: Cluster comparison

According to figure 54, the algorithm separates the data into 3 distinct cluster, and each cluster characteristics are depicted above.

Step 8: Interpretation

8.1 – Study and discuss mined patterns

Logistic regression:

Feature	Coefficient
Income	-1.0E-6
Cholesterol	0.001431
Country_indexed	2.9E-5
Diastolic	-0.004038
Exercise Hours Per Week	-0.007842
Heart Rate	0.001061
Triglycerides	-2.17E-4
BMI	-0.008633
Sedentary Hours Per Day	0.007461
Age	0.006027
Stress Level	0.03614

Logistic regression belongs to predictive pattern, because it trains the dataset to predict on the test set. In the figure above, the coefficient value for each variable is given, and we will be using the coefficient to determine if a feature is significantly important.

Decision tree:

```
If (feature 5 <= 69.5)
  Predict: 1.0
Else (feature 5 > 69.5)
  Predict: 0.0
```

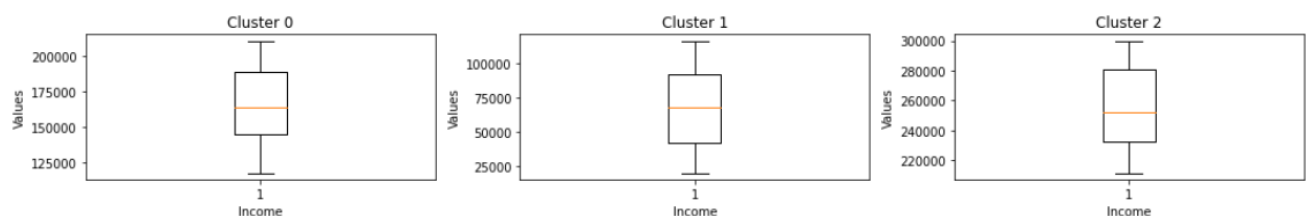
Decision tree algorithm belongs to classification; thus, it also depicts a predictive pattern. This output is different because it is the rule set output. This algorithm helps predicts potential combination of factors in relation to target variable.

FPGrowth algorithm:

confidence	lift	support	antecedent_feature	consequent_feature
0.20342205323193915	0.3501240158897404	0.107	Heart Attack Risk (1: Yes)	Income
0.6024096385542169	1.036849636065778	0.1	Heart Rate	Income
0.1721170395869191	1.0368496360657777	0.1	Income	Heart Rate
0.18416523235800344	0.35012401588974035	0.107	Income	Heart Attack Risk (1: Yes)

FPGrowth algorithm falls under association method. Thus, it depicts a descriptive pattern. We are able to derive information by looking at these pattern as there are many information we can get from the different columns. Antecedent represents the condition or the input of the association rule. Consequent represents the outcome or the output of the association rule. It depicts a “A likely to cause C” relationship pattern.

Two-step clustering



Clustering also belongs to descriptive pattern. We are able to find interesting pattern just by looking at the above image. Clustering also helps to reveal underlying relationship that may be interesting and surprising. For instance, in the boxplot, we can tell which cluster has a lower median value and derive further insights from there.

8.2 – Visualise the data, result, model, pattern

In this subsection, we will visualise the entire output we have produced from our models. The interpretation of these output will be in the next subsection (8.3).

```
Training Accuracy: 0.55
Testing Accuracy: 0.49
```

prediction	0	1
0.0	31	33
1.0	50	48

Figure 55: Logistic Regression accuracy (left) & cross-tab output (right)

Feature	Coefficient
Income	-1.0E-6
Cholesterol	0.001431
Country_indexed	2.9E-5
Diastolic	-0.004038
Exercise Hours Per Week	-0.007842
Heart Rate	0.001061
Triglycerides	-2.17E-4
BMI	-0.008633
Sedentary Hours Per Day	0.007461
Age	0.006027
Stress Level	0.03614

Figure 55.1: Logistic Regression output

```
Training Accuracy: 0.6444
Testing Accuracy: 0.5123
```

Figure 56: Decision tree output

prediction	0	1
0.0	40	38
1.0	41	43

Figure 56.1: Decision tree cross-tab output

Feature	Importance
Sedentary Hours Per Day	0.1724
BMI	0.1652
Exercise Hours Per Week	0.1141
Triglycerides	0.1118
Cholesterol	0.0947
Age	0.0927
Country_indexed	0.0643
Heart Rate	0.0591
Diastolic	0.0508
Stress Level	0.048
Income	0.0269

Figure 56.2: Decision tree output

```
If (feature 5 <= 69.5)
  Predict: 1.0
Else (feature 5 > 69.5)
  Predict: 0.0
```

```
If (feature 6 <= 766.0)
  If (feature 8 <= 4.1592155245)
    If (feature 10 <= 2.5)
      If (feature 2 <= 1.5)
        Predict: 0.0
      Else (feature 2 > 1.5)
        Predict: 1.0
```

Figure 56.3: Decision tree output

confidence	lift	support	antecedent_feature	consequent_feature
0.20342205323193915	0.3501240158897404	0.107	Heart Attack Risk (1: Yes)	Income
0.6024096385542169	1.036849636065778	0.1	Heart Rate	Income
0.1721170395869191	1.0368496360657777	0.1	Income	Heart Rate
0.18416523235800344	0.35012401588974035	0.107	Income	Heart Attack Risk (1: Yes)

Figure 57: FPGrowth output

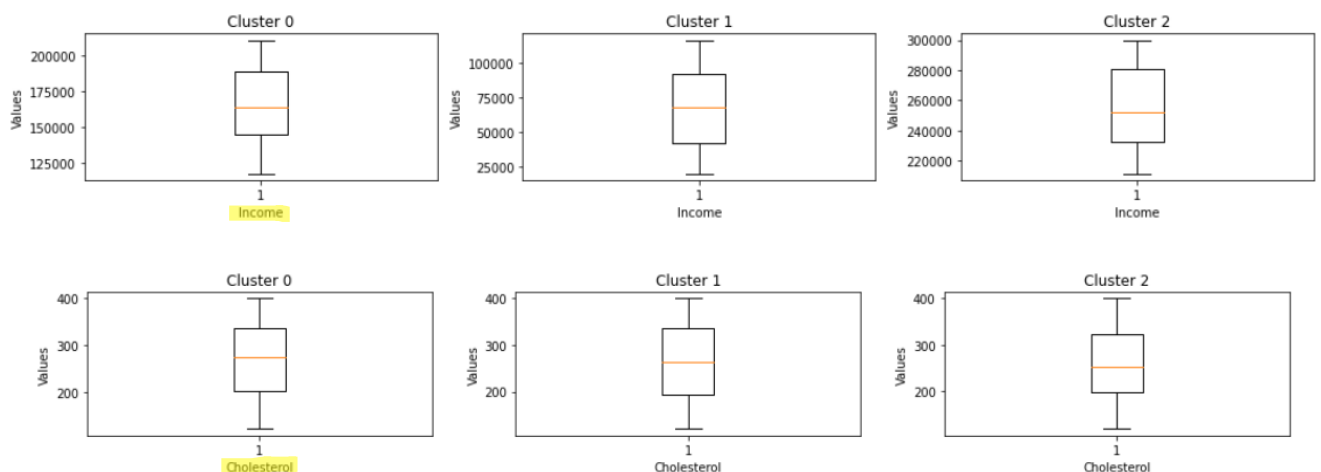


Figure 58: Two-step cluster output

Silhouette Score for Train Data: 0.7501455549984403
Silhouette Score for Test Data: 0.763603385563619

Figure 58.1: Silhouette score output

8.3 – Interpret the result, model, pattern

Note: *This subsection will only be interpretation of subsection 8.2. Please refer to the output image above.*

According to **figure 55** – the logistic regression output, we can infer from the accuracy score that logistic regression correctly predicts the heart attack risk 49% with test set and 55% with train set. In **figure 55.1**, we can infer from the coefficient values that all features are significantly important in predicting the heart attack risk. From the cross-tab output in **Figure 55**, we can interpret the result that was tested on the test set as the cell (Actual = 0, Predicted = 0) indicates the model correctly predicted 31 instances of class 0 (no heart attack risk). On the other hand, the cell (Actual = 1, Predicted = 0) indicates the model incorrectly predicted 50 instances of class 0 when the actual class was 1 (heart attack risk presence) using the logistic regression algorithm.

According to **figure 56.2** – the decision tree output, we found that sedentary hours per day is ranked as the most important features. This piece of information may imply that if an individual spend more hour seated, then the chances of heart attack risk is high. However, note that sedentary hours per day values vary, so an individual with less hour of sedentary hours per day may have low heart attack risk. **Figure 56.3** will provide a more detailed insight. For example, as depicted, a diastolic (feature 5) reading of less than or equal to 69.5, indicates class 1 (presence of risk), but a reading greater than 69.5 indicates class 0 (no risk). **Figure 56.1** provides the cross-tab output. The cell

(Actual = 0, Predicted = 0) indicates the model correctly predicted 40 instances of class 0 (no heart attack risk). On the other hand, the cell (Actual = 1, Predicted = 0) indicates the model incorrectly predicted 41 instances of class 0 when the actual class was 1 (heart attack risk presence) using the decision tree algorithm.

According to **figure 57**– FPGrowth output, I have chosen to interpret the second and third row because these two rows have the highest lift value. We can interpret the output in a way such as, income earning is likely to be associated with an individual heart rate (though does not sound sensible). It is important to note that income seems to have relationship / association with another variable - heart attack risk. Therefore, we may infer that income may be a more prominent factor since it appears in all the rows.

According to **figure 58** – the cluster comparison output depicts cluster 2 tends to have a higher median value for income, compare to the other 2 cluster. This could indicate that majority of the data points are larger for cluster 2, skewing the distribution towards higher values. Whereas, for example, cluster 1 has the lowest median value for income, which indicates that larger portion of income values are lower for cluster 1, compare to the other cluster.

8.4 – Assess and evaluate result, models, pattern

According to Figure 55 – logistic regression output, the model is not overfitting neither underfitting. The accuracy score for both test and train set are close. An accuracy of 49% on the test set suggests that the model's performance in making correct predictions on unseen data is relatively low. The model is correct in its predictions for almost half of the instances in the test set. As

mentioned above, we can determine that all variables are significantly important variables in this logistic regression based on their coefficient. If the coefficient is greater than p-value (usually 0.05), then we can conclude the variable is not significantly important. Although the accuracy score is acceptable, but it can be improved to produce a higher accuracy score.

According to Figure 56 – decision tree algorithm, the accuracy for train and test set varies more compare to logistic regression accuracy. Train set has an accuracy score of 64% while test set has an accuracy score of 51%. This indicates the model is overfitting because train set has a much higher accuracy. The accuracy score is not good compare to logistic regression. There are several reasons why it has big difference compare to logistic regression and this might be because the algorithm works differently. The model can be improved to perform better.

For FPGrowth algorithm, the values such as confidence % is not high but yet able to capture some association / relationship between variables. However, we can try to improve our model in order to achieve a higher value to show stronger relationship.

According to Figure 58.1 - Silhouette score for two-step clustering, the score for both train and test data is 0.75 and 0.76 respectively. The cluster quality is considered fairly good since is it greater than 0.5. This score indicates that the features have captured some relevant and important patterns of the data.

8.5 – Iteration

1st iteration:

Since the result we obtained is not as ideal as expected, we will have to improve our model performance. In order to do so, I have decided to further delve into the logistic regression algorithm. In this first iteration, I have decided to remove the features that are not as significantly important compare to other variables in predicting heart attack risk. The decision to remove which variable is based on their coefficient in the earlier output above. Figure 59 below shows the code to do so.

```
data = data.drop("Stress Level", "Sedentary Hours Per Day", "Cholesterol", "Heart Rate" )
```

```
from pyspark.sql.functions import col
from pyspark.ml.feature import VectorAssembler

# Assuming 'data' is your original DataFrame
target_column = "Heart Attack Risk (1: Yes)"
feature_columns = [col for col in data.columns if col != target_column]

# Combine feature columns into a single vector column
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data1 = assembler.transform(data).select("features", target_column)

# Split the assembled data into training and testing sets
train_data, test_data = data1.randomSplit([0.8, 0.2], seed=42)
```

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.functions import expr

# Fit the Logistic Regression model
lr = LogisticRegression(labelCol=target_column, featuresCol="features")
lr_model = lr.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = lr_model.transform(train_data)
test_predictions = lr_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.2f}")
print(f"Testing Accuracy: {test_accuracy:.2f}")
```

Figure 59 : the logistic regression code

	+-----+-----+-----+
	prediction 0 1
	+-----+-----+-----+
Training Accuracy: 0.53	0.0 29 34
Testing Accuracy: 0.47	1.0 52 47
	+-----+-----+-----+

Figure 59.1: the accuracy score (left) and cross-tab value (right)

Figure 59.1 above shows the accuracy score and cross-tab value after removing the four less significantly important features. The accuracy on test set has decreased a little, from 49% to 47%. The accuracy on train set has decreased a little, from 55% to 53%. The result from cross-tab value has not changed too much as well. Although this approach has led to a slight decrease in test set accuracy, but still it is not good yet. Another approach is needed.

2nd Iteration:

Since the accuracy for test set is still not as good as expected, it might be due to insufficient data. Therefore, an approach is to boost the dataset, so that we get more data. Figure 60 below is the code to boost the model. I have decided to double the row. Figure 60 also shows the boosted shape.

```
boost_data = data.sample(withReplacement=True, fraction=2.0, seed = 42)
```

```
print(f"Number of rows: {boost_data.count()}")
print(f"Number of columns: {len(boost_data.columns)}")
```

```
Number of rows: 2022
Number of columns: 12
```

Figure 60: Boost model (code) and shape of boosted model

Figure 60.1 below shows the logistic regression algorithm once again. Figure 60.2 shows the accuracy score and cross-tab value. It is evident that the test set

accuracy has now increased from 49% to 51%. This means the model is performance on test set has improved compare to train set, because train set accuracy has decreased from 55% to 53%. The largest proportion from the cross-tab output is the cell (Actual = 1, Predicted = 1). This cell indicates that the model correctly predicted 117 instances of class 1.

```
# Fit the Logistic Regression model
lr = LogisticRegression(labelCol=target_column, featuresCol="features")
lr_model = lr.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = lr_model.transform(train_data)
test_predictions = lr_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.2f}")
print(f"Testing Accuracy: {test_accuracy:.2f}")

# Create a cross-tabulation of predicted vs. actual classes for the test set
cross_tab = test_predictions.groupBy("prediction").pivot(target_column).count()

# Display the cross-tabulation
cross_tab.show()
```

Figure 60.1 : Logistic regression code

		+-----+---+---+			
		prediction 0 1			
		+-----+---+---+			
			0.0	65	77
Training Accuracy: 0.53			1.0	101	117
Testing Accuracy: 0.51		+-----+---+---+			

Figure 60.2 Accuracy score (left) & cross-tab value (right)

3rd Iteration:

I had taken another approach, where I used the boosted dataset but this round I use the original features (did not remove the features). The accuracy on test set has increased a little, from 49% to 51%. But, the accuracy on train set has not changed. Overall, the second and third iteration performs better on the test set compare to the first iteration.

```
# Fit the Logistic Regression model
lr = LogisticRegression(labelCol=target_column, featuresCol="features")
lr_model = lr.fit(train_data)

# Make predictions on the training and testing sets
train_predictions = lr_model.transform(train_data)
test_predictions = lr_model.transform(test_data)

# Define the evaluator for accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol=target_column, predictionCol="prediction", metricName="accuracy"
)

# Calculate accuracy scores for training and testing sets
train_accuracy = evaluator.evaluate(train_predictions)
test_accuracy = evaluator.evaluate(test_predictions)

# Print accuracy scores
print(f"Training Accuracy: {train_accuracy:.2f}")
print(f"Testing Accuracy: {test_accuracy:.2f}")

# Create a cross-tabulation of predicted vs. actual classes for the test set
cross_tab = test_predictions.groupBy("prediction").pivot(target_column).count()

# Display the cross-tabulation
cross_tab.show()
```

Figure 61: code

```
Training Accuracy: 0.55
Testing Accuracy: 0.51
+-----+-----+
|prediction|  0|  1|
+-----+-----+
|          0.0| 73| 84|
|          1.0| 93|110|
+-----+-----+
```

Figure 61.1: 3 Accuracy score (top) & cross-tab value (right)

```
ubuntu@ip-172-31-60-13:~$ cd Infosys-722---Iteration-4
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git add 'Part 8 - Iterations.ipynb'
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git commit -m 'Part 8'
[main c78f310] Part 8
Committer: Ubuntu <ubuntu@ip-172-31-60-13.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 501 insertions(+)
 create mode 100644 Part 8 - Iterations.ipynb
ubuntu@ip-172-31-60-13:~/Infosys-722---Iteration-4$ git push
Username for 'https://github.com': agneslee01
Password for 'https://agneslee01@github.com':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
```

Screenshot of AWS terminal to proof that Part 8 – Iterations file is pushed to github.

Reference list

AIML. (2024). *What are the advantages and disadvantages of logistic regression?*. AIML.com. <https://aiml.com/what-are-advantages-and-disadvantages-of-logistic-regression/>

Ali, A. (2023). *Top 10 Data Mining Techniques*. Astera. <https://www.astera.com/type/blog/top-10-data-mining-techniques/>

All, M. (2023). *Association Rule Mining in Python Tutorial*. Datacamp. <https://www.datacamp.com/tutorial/association-rule-mining-python>

American Heart Association News (2021). *For smokers, fatal heart attack or stroke may be first sign of cardiovascular disease*. American Heart Association News. <https://www.heart.org/en/news/2021/11/17/for-smokers-fatal-heart-attack-or-stroke-may-be-first-sign-of-cardiovascular-disease#:~:text=Young%20men%20who%20smoked%20had,such%20strokes%20or%20heart%20failure>.

Baheti, P. (2021). *Supervised and Unsupervised Learning [Differences & Examples]*. V7. <https://www.v7labs.com/blog/supervised-vs-unsupervised-learning>

Fogoros, R.N. (2022). *Heart Attack Risks in Young People*. Verywellhealth. <https://www.verywellhealth.com/how-common-are-heart-attacks-in-young-people-3866059>

Henderson, E. (2022). *Heart attack death rates took a sharp turn and increased during the pandemic, study shows*. News Medical Life Sciences. <https://www.news-medical.net/news/20221024/Heart-attack-death-rates-took-a-sharp-turn-and-increased-during-the-pandemic-study-shows.aspx>

Kapil, A.R. (2022). *Decision Tree Algorithm in Machine Learning: Advantages, Disadvantages, and Limitations*. AnalytixLabs. <https://www.analytixlabs.co.in/blog/decision-tree-algorithm/>

Keserer, E. (2023). *8 Types of Machine Learning Classification Algorithms*. Akkio <https://www.akkio.com/post/5-types-of-machine-learning-classification-algorithms#:~:text=With%20an%20input%20training%20dataset,similar%20patterns%20in%20future%20data>.

McGregor, M. (2020). *8 Clustering Algorithms in Machine Learning that All Data Scientists Should Know*. FreeCodeCamp.

<https://www.freecodecamp.org/news/8-clustering-algorithms-in-machine-learning-that-all-data-scientists-should-know/>

Sehgal, A. (2022). *Clustering Data Mining Techniques: 5 Critical Algorithms* 2024. Hevo. <https://hevodata.com/learn/clustering-data-mining-techniques/>

Utkarsh. (2023). *Classification in Data Mining*. Scaler. <https://www.scaler.com/topics/data-mining-tutorial/classification-in-data-mining/>

Disclaimer

"I acknowledge that the submitted work is my own original work in accordance with the University of Auckland guidelines and policies on academic integrity and copyright. (See: <https://www.auckland.ac.nz/en/students/forms-policies-and-guidelines/student-policies-and-guidelines/academic-integrity-copyright.html>).

I also acknowledge that I have appropriate permission to use the data that I have utilised in this project. (For example, if the data belongs to an organisation and the data has not been published in the public domain, then the data must be approved by the rights holder.) This includes permission to upload the data file to Canvas. The University of Auckland bears no responsibility for the student's misuse of data."