2023

Cardiovascular Disease

INVESTIGATION OF THE RISK FACTORS FOR CARDIOVASCULAR DISEASE

PHONE KO (PKO996)

GitHub: https://github.com/Pokekaya/722-BDAS.git

Table of Contents

1. Business Understanding

1.1. Business/Situation Objectives

Studying the risk factors for cardiovascular disease (CVD) is essential because of CVD-related deaths. Which is taking around 17.9 million lives in 2019, and it is about 32% of all deaths worldwide (World Health Organization, 11 June 2021). Heart attacks and strokes are the main reasons for these deaths, responsible for 85% of CVD-related fatalities. Surprisingly, more than three-quarters of these deaths happened in low-income and middle-income countries.

CVD also cause many premature deaths, making up 38% of the estimated 17 million early deaths related to noncommunicable diseases in 2019 worldwide (World Health Organization, 11 June 2021). The good news is that we can prevent most cardiovascular diseases by addressing certain behaviours. By reducing tobacco use, promoting healthy eating habits, fighting obesity, encouraging physical activity, and tackling harmful alcohol consumption, we might decrease the number of CVD cases.

Detecting cardiovascular disease is important for managing it effectively. When we find CVD in its early stages, we can start counselling and medical treatments promptly, which might prevent more problems and help patients get better. So, studying the risk factors linked to cardiovascular disease is crucial. It can help create specific plans to avoid CVD, develop ways to detect it early and develop treatments based on evidence. In the end, this will lead to a healthier world population and lessen the impact of CVD on societies and healthcare systems.

Data analysis plays a significant role in identifying individuals at risk of heart problems, improving preventive measures, and enhancing healthcare. By closely examining data, we can gain insights into the causes of heart issues, make more accurate predictions, and take necessary actions. It benefits public health, saves money, and promotes overall well-being. It is essential for both research and business purposes.

1.2. Situation Assessment

We will source our project data from Kaggle website, which offers free CSV datasets. We will analyse the data using PySpark which combines Python's API and the power of Apache Spark to analyse any data size. Fortunately, PySpark is open source and free, making this suitable for our academic project in INFOSYS 722 without incurring any costs.

I will conduct this study independently, with support and guidance from INFOSYS 722 tutors and the lecturer. The tutors and lecturer will help with the project's technical and theoretical aspects.

This study had limitations because it used only a small amount of data. People are interested in seeing the study's results and the model used, as they could use it to confirm suspected patients in the future. The research does not pose any immediate risks. However, some potential risks are listed in the following table (Table-1.1), and the backup plan to address them.

S/N	Risk	Contingency Plan
1.	Data Quality	Clean and validate the data to make sure it's accurate. Use appropriate methods to handle any missing or incomplete data.
2.	Model Accuracy	Fine-tuning the model parameters and examining the data to identify any missing values.

3.	Domain Knowledge of Data Mining	Seeking help from tutors and the lecturer when it needs assistance with your study's technical and theoretical aspects.
4.	Timeline Planning	Make a schedule for each stage of the project and stick to it. Adjust the schedule accordingly at each Milestone Point if there are any delays.

Table-1.1: Project's Risk and Contingency Plan

1.3. Data Mining Goals

In 2013, the World Health Organization (WHO) created a plan what is called "Global action plan for the prevention and control of NCDs 2013-2020." The goal is to reduce premature deaths from NCDs by 25% before 2025. This plan has nine global targets, two focusing on preventing and controlling cardiovascular diseases (World Health Organization, 14 November 2013).

In line with this global plan, our main objective is to find and deal with cardiovascular disease (CVD) as early as possible by looking into its risk factors. We will focus on a few points as follows:

- We will study how lifestyle choices and environmental factors affect the risk of CVD. We will consider things like age, gender, height, weight, and smoking habits of different people.
- And check a person's physical activity level, blood pressure, cholesterol levels, and chances of getting cardiovascular disease.

By doing so, we aim to identify patterns and connections within this information to help us remember and prevent CVD early on. We intend to utilise data mining techniques to uncover hidden insights that will aid in the early detection and prevention of cardiovascular disease, aligning with the World Health Organization's global target.

1.4. Project Planning

To ensure the success of our study, we will utilise the free and user-friendly Smartsheet[1] online tool (free version) to create a well-organized project timeline in different views, such as Grid View (Figure-1.1) and Gantt View (Figure-1.2). This timeline will outline each phase with a specific date range, indicating the number of days allocated for each stage.

[1] https://www.smartsheet.com

ITERATION-4

	Tasks	Start Date	End Date	Days
1	**Iteration 1 - Proposal (Steps 1 - 2)**	**07/17/23**	**07/28/23**	**11**
2	Step 1. Business and/or Situation understanding	07/17/23	07/22/23	**5**
3	Step 2. Data understanding	07/23/23	07/28/23	**5**
4	**Iteration 2 - ISAS (Steps 1 - 8)**	**07/17/23**	**08/18/23**	**32**
5	Step 1. Business and/or Situation understanding	07/17/23	07/22/23	**5**
6	Step 2. Data understanding	07/23/23	07/28/23	**5**
7	Step 3. Data preparation	07/29/23	08/04/23	**6**
8	Step 4. Data transformation	07/29/23	08/04/23	**6**
9	Step 5. Data-mining method(s) selection	08/05/23	08/11/23	**6**
10	Step 6. Data-mining algorithm(s) selection	08/12/23	08/18/23	**6**
11	Step 7. Data Mining	08/12/23	08/18/23	**6**
12	Step 8. Interpretation	08/12/23	08/18/23	**6**
13	**Iteration 3 - OSAS (Steps 1 - 8)**	**07/17/23**	**09/22/23**	**67**
14	Step 1. Business and/or Situation understanding	07/17/23	07/22/23	**5**
15	Step 2. Data understanding	07/23/23	07/28/23	**5**
16	Step 3. Data preparation	08/19/23	08/25/23	**6**
17	Step 4. Data transformation	08/19/23	08/25/23	**6**
18	Step 5. Data-mining method(s) selection	08/26/23	09/01/23	**6**
19	Step 6. Data-mining algorithm(s) selection	09/02/23	09/08/23	**6**
20	Step 7. Data Mining	09/09/23	09/15/23	**6**
21	Step 8. Interpretation	09/16/23	09/22/23	**6**
22	**Iteration 4 - BDAS (Steps 1 - 8)**	**07/17/23**	**10/13/23**	**88**
23	Step 1. Business and/or Situation understanding	07/17/23	07/22/23	**5**
24	Step 2. Data understanding	07/23/23	07/28/23	**5**
25	Step 3. Data preparation	09/23/23	09/29/23	**6**
26	Step 4. Data transformation	09/23/23	09/29/23	**6**
27	Step 5. Data-mining method(s) selection	09/30/23	10/06/23	**6**
28	Step 6. Data-mining algorithm(s) selection	09/30/23	10/06/23	**6**
29	Step 7. Data Mining	10/07/23	10/13/23	**6**
30	Step 8. Interpretation	10/07/23	10/13/23	**6**
31	**Research Paper - BDAS (Steps 1 - 9)**	**08/19/23**	**10/20/23**	**62**
32	Step 1. Business and/or Situation understanding	**08/19/23**	08/25/23	**6**
33	Step 2. Data understanding	08/26/23	09/01/23	**6**
34	Step 3. Data preparation	09/02/23	09/08/23	**6**
35	Step 4. Data transformation	09/09/23	09/15/23	**6**
36	Step 5. Data-mining method(s) selection	09/16/23	09/22/23	**6**
37	Step 6. Data-mining algorithm(s) selection	09/23/23	09/29/23	**6**
38	Step 7. Data Mining	09/30/23	10/06/23	**6**
39	Step 8. Interpretation	10/07/23	10/13/23	**6**
40	Step 9. Action	10/14/23	10/20/23	**6**

Figure-1.1: Project Plan in Grid View

ITERATION-4

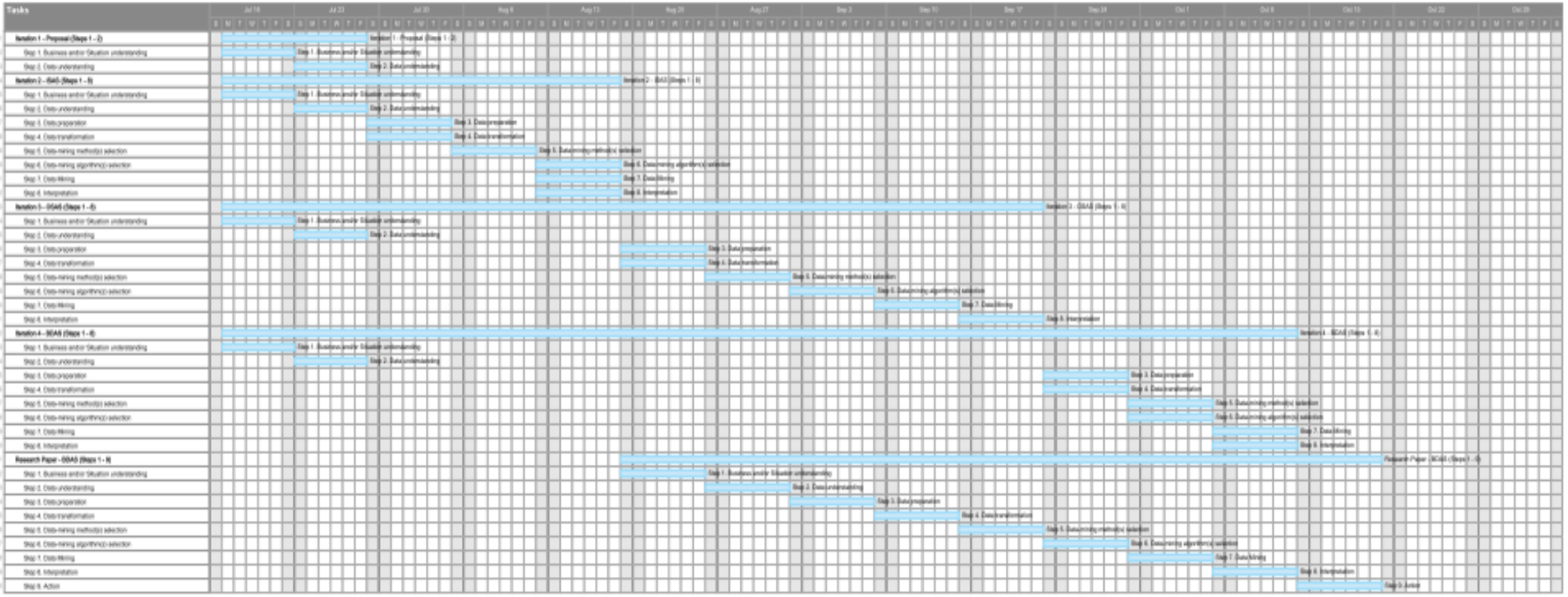


Figure-1.2: Project Plan in Gantt View

2. Data Understanding

2.1. Collecting Initial Data

We initially searched various online sources for datasets covering lifestyle choices, demographics, and health indicators for our data mining project. A recurring challenge was that many datasets needed specific critical attributes we deemed necessary for our project. After navigating through several data-providing platforms, our persistence paid off when we stumbled upon the perfect dataset on Kaggle. With that in hand, we proceeded with the data collection and a thorough review to ensure its relevance and quality as following:

1. As shown in Figure-2.1, we loaded the downloaded CSV dataset using the "csv" method from the "PySpark.sql.DataFrameReader" library and confirmed the total loaded data with the "count" method.

```
In [12]: # load the csv file
         df = spark.read.csv('Datasets/heart_dataset_1.csv', inferSchema=True, header=True)

         df.count()

Out[12]: 69000
```

Figure-2.1: Load the csv dataset.

In Figure-2.2, we used the "show "method to verify the loaded data.

```
In [27]: # Verify the loaded data
         df.show()
```

```
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index| id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    0|  0|18393|     2|   168|  62.0|  110|   80|          1|   1|    0|   0|     1|     0|
|    1|  1|20228|     1|   156|  85.0|  140|   90|          3|   1|    0|   0|     1|     1|
|    2|  2|18857|     1|   165|  64.0|  130|   70|          3|   1|    0|   0|     0|     1|
|    3|  3|17623|     2|   169|  82.0|  150|  100|          1|   1|    0|   0|     1|     1|
|    4|  4|48Yrs|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|    5|  8|21914|     1|   151|  67.0|  120|   80|          2|   2|    0|   0|     0|     0|
|    6|  9|22113|     1|   157|  93.0|  130|   80|          3|   1|    0|   0|     1|     0|
|    7| 12|22584|     2|   178|  95.0|  130|   90|          3|   3|    0|   0|     1|     1|
|    8| 13|17668|     1|   158|  71.0|  110|   70|          1|   1|    0|   0|     1|     0|
|    9| 14|19834|     1|   164|  68.0|  110|   60|          1|   1|    0|   0|     0|     0|
|   10| 15|22530|     1|   169|  80.0|  120|   80|          1|   1|    0|   0|     1|     0|
|   11| 16|18815|     2|   173|  60.0|  120|   80|          1|   1|    0|   0|     1|     0|
|   12| 18|14791|     2|   165|  60.0|  120|   80|          1|   1|    0|   0|     0|     0|
|   13| 21|19809|     1|   158|  78.0|  110|   70|          1|   1|    0|   0|     1|     0|
|   14| 23|14532|     2|   181|  95.0|  130|   90|          1|   1|    1|   1|     1|     0|
|   15| 24|16782|     2|   172| 112.0|  120|   80|          1|   1|    0|   0|     0|     1|
|   16| 25|21296|     1|   170|  75.0|  130|   70|          1|   1|    0|   0|     0|     0|
|   17| 27|16747|     1|   158|  52.0|  110|   70|          1|   3|    0|   0|     1|     0|
|   18| 28|17482|     1|   154|  68.0|  100|   70|          1|   1|    0|   0|     0|     0|
|   19| 29|21755|     2|   162|  56.0|  120|   70|          1|   1|    1|   0|     1|     0|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
only showing top 20 rows
```

Figure-2.2: Verify the loaded data.

2.2. Describing Data

The Kaggle dataset[2/3] we downloaded contain 70,000 records for the combined two datasets. It includes information about people with and without the disease and other relevant data attributes for this study. The details information on each attribute can be found in the table below (Table-2.1):

S/N	Attribute Name	Description
1.	index	Serialise running number
2.	id	Unique identifier of the participant
3.	age	Age of participants in days

[2] https://www.kaggle.com/datasets/thedevastator/exploring-risk-factors-for-cardiovascular-diseas
[3] https://www.kaggle.com/code/alenreuel/cardiovascular-heart-disease/notebook

4.	gender	Gender of participant (1 = Male / 2 = Female)
5.	height	Height measured in centimetres
6.	weight	Weight measured in kilograms
7.	ap_hi	The systolic blood pressure reading taken from patient
8.	ap_lo	The diastolic blood pressure reading taken from the patient
9.	cholesterol	Zero means less than 100 mg/dL; every increment is a 20 units addition.
10.	gluc	Zero references <3.3 mmol/L, and every increment is a 1-unit addition.
11.	smoke	Whether a person smokes or not (0= No, 1=Yes)
12.	alco	Whether person drinks alcohol or not (0 =No ,1 =Yes)
13.	active	Whether person physically active or not (0 =No,1 = Yes)
14.	cardio	Whether a person suffers from cardiovascular diseases (0 = no, 1 = yes).

Table-2.1: Details information on the data

Attributes such as “age”, “gender”, “height”, and “weight” are foundational demographics that allow for categorisation and ensure a broad population representation. These factors are often considered when evaluating an individual’s susceptibility to various diseases.

Attributes like “ap_hi”, “ap_lo”, “cholesterol”, and “gluc” are crucial health indicators. Variations in these metrics can often signal or predict cardiovascular anomalies. Their inclusion helps create a robust model that can predict the onset or risk of the disease based on historical and current data.

Lastly, lifestyle factors such as smoking (smoke), alcohol consumption (alco), and physical activity (active) have proven to influence cardiovascular health significantly. Their inclusion ensures a comprehensive consideration of external factors that might impact an individual’s health.

Furthermore, we will use the “printSchema” methods to analyse the data type for each attribute, and this will help us understand the nature of the data and its characteristics, as shown in the following figure (Figure-2.3).

```
In [13]: # Verify the data schema
         df.printSchema()

         root
          |-- index: integer (nullable = true)
          |-- id: integer (nullable = true)
          |-- age: string (nullable = true)
          |-- gender: integer (nullable = true)
          |-- height: integer (nullable = true)
          |-- weight: double (nullable = true)
          |-- ap_hi: integer (nullable = true)
          |-- ap_lo: integer (nullable = true)
          |-- cholesterol: integer (nullable = true)
          |-- gluc: integer (nullable = true)
          |-- smoke: integer (nullable = true)
          |-- alco: integer (nullable = true)
          |-- active: integer (nullable = true)
          |-- cardio: integer (nullable = true)
```

Figure-2.3: Data Type for each attribute

2.3. Exploring Data

We have already identified the data type for each attribute in the previous section, as shown in Figure-2.3. In this section, we will apply some methods of PySpark’s data frame to visualise each data attribute and evaluate the overall data quality, such as missing data, data errors, inconsistencies, etc.

We will begin our examination of the dataset using various PySpark's methods as follows:

- **columns**: This method retrieves the column labels of the Dataframe, allowing us to confirm the dataset's labelling, as shown in Figure-2.4.

```
In [21]: df.columns

Out[21]: ['index',
          'id',
          'age',
          'gender',
          'height',
          'weight',
          'ap_hi',
          'ap_lo',
          'cholesterol',
          'gluc',
          'smoke',
          'alco',
          'active',
          'cardio']
```

Figure-2.4: Columns labels for each attribute

- **head**: This method shows the first "n" rows of the DataFrame, as seen in Figure-2.5. Since it's not in table form, we'll use the "OrderBy" method to sort by the "Index" column and display the first ten rows as shown in Figure-2.6.

```
In [22]: df.head(10)

Out[22]: [Row(index=0, id=0, age='18393', gender=2, height=168, weight=62.0, ap_hi=110, ap_lo=80, cholesterol=1, gluc=1, smoke=0, alc
         o=0, active=1, cardio=0),
          Row(index=1, id=1, age='20228', gender=1, height=156, weight=85.0, ap_hi=140, ap_lo=90, cholesterol=3, gluc=1, smoke=0, alc
         o=0, active=1, cardio=1),
          Row(index=2, id=2, age='18857', gender=1, height=165, weight=64.0, ap_hi=130, ap_lo=70, cholesterol=3, gluc=1, smoke=0, alc
         o=0, active=0, cardio=1),
          Row(index=3, id=3, age='17623', gender=2, height=169, weight=82.0, ap_hi=150, ap_lo=100, cholesterol=1, gluc=1, smoke=0, al
         co=0, active=1, cardio=1),
          Row(index=4, id=4, age='48Yrs', gender=1, height=156, weight=56.0, ap_hi=100, ap_lo=60, cholesterol=1, gluc=1, smoke=0, alc
         o=0, active=0, cardio=0),
          Row(index=5, id=8, age='21914', gender=1, height=151, weight=67.0, ap_hi=120, ap_lo=80, cholesterol=2, gluc=2, smoke=0, alc
         o=0, active=0, cardio=0),
          Row(index=6, id=9, age='22113', gender=1, height=157, weight=93.0, ap_hi=130, ap_lo=80, cholesterol=3, gluc=1, smoke=0, alc
         o=0, active=1, cardio=0),
          Row(index=7, id=12, age='22584', gender=2, height=178, weight=95.0, ap_hi=130, ap_lo=90, cholesterol=3, gluc=3, smoke=0, al
         co=0, active=1, cardio=1),
          Row(index=8, id=13, age='17668', gender=1, height=158, weight=71.0, ap_hi=110, ap_lo=70, cholesterol=1, gluc=1, smoke=0, al
         co=0, active=1, cardio=0),
          Row(index=9, id=14, age='19834', gender=1, height=164, weight=68.0, ap_hi=110, ap_lo=60, cholesterol=1, gluc=1, smoke=0, al
         co=0, active=0, cardio=0)]
```

Figure-2.5: First "10" rows of the Dataframe

```
In [26]: df.orderBy(df['Index'].asc()).show(10)

+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index| id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    0|  0|18393|     2|   168|  62.0|  110|   80|          1|   1|    0|   0|     1|     0|
|    1|  1|20228|     1|   156|  85.0|  140|   90|          3|   1|    0|   0|     1|     1|
|    2|  2|18857|     1|   165|  64.0|  130|   70|          3|   1|    0|   0|     0|     1|
|    3|  3|17623|     2|   169|  82.0|  150|  100|          1|   1|    0|   0|     1|     1|
|    4|  4|48Yrs|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|    5|  8|21914|     1|   151|  67.0|  120|   80|          2|   2|    0|   0|     0|     0|
|    6|  9|22113|     1|   157|  93.0|  130|   80|          3|   1|    0|   0|     1|     0|
|    7| 12|22584|     2|   178|  95.0|  130|   90|          3|   3|    0|   0|     1|     1|
|    8| 13|17668|     1|   158|  71.0|  110|   70|          1|   1|    0|   0|     1|     0|
|    9| 14|19834|     1|   164|  68.0|  110|   60|          1|   1|    0|   0|     0|     0|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
only showing top 10 rows
```

Figure-2.6: First "10" rows of data which are ascending order on "Index" column.

- **tail**: It displays the last "n" rows of the DataFrame as illustrated in Figure-2.7. Since it is not displayed in a table format, we will apply the "OrderBy" method to arrange the data in descending order based on the "Index" column to get the top ten rows, as displayed in Figure-2.8.

```
In [23]: df.tail(10)

Out[23]: [Row(index=68990, id=98559, age='20508', gender=2, height=173, weight=91.0, ap_hi=140, ap_lo=90, cholesterol=1, gluc=1, smoke=0, alco=1, active=1, cardio=0),
 Row(index=68991, id=98561, age='21787', gender=1, height=168, weight=70.0, ap_hi=140, ap_lo=90, cholesterol=3, gluc=1, smoke=0, alco=0, active=1, cardio=1),
 Row(index=68992, id=98562, age='17574', gender=1, height=157, weight=63.0, ap_hi=150, ap_lo=100, cholesterol=1, gluc=1, smoke=0, alco=0, active=1, cardio=0),
 Row(index=68993, id=98563, age='14643', gender=2, height=175, weight=75.0, ap_hi=100, ap_lo=70, cholesterol=1, gluc=1, smoke=0, alco=0, active=1, cardio=0),
 Row(index=68994, id=98564, age='18949', gender=2, height=172, weight=62.0, ap_hi=110, ap_lo=70, cholesterol=1, gluc=1, smoke=1, alco=0, active=1, cardio=0),
 Row(index=68995, id=98565, age='17548', gender=2, height=173, weight=72.0, ap_hi=120, ap_lo=80, cholesterol=1, gluc=1, smoke=0, alco=0, active=0, cardio=0),
 Row(index=68996, id=98566, age='20430', gender=1, height=159, weight=105.0, ap_hi=120, ap_lo=80, cholesterol=1, gluc=1, smoke=0, alco=0, active=1, cardio=1),
 Row(index=68997, id=98567, age='21682', gender=1, height=158, weight=78.0, ap_hi=130, ap_lo=90, cholesterol=1, gluc=1, smoke=0, alco=0, active=1, cardio=1),
 Row(index=68998, id=98568, age='19107', gender=1, height=154, weight=77.0, ap_hi=14, ap_lo=90, cholesterol=1, gluc=1, smoke=0, alco=0, active=1, cardio=0),
 Row(index=68999, id=98569, age='19591', gender=2, height=175, weight=88.0, ap_hi=120, ap_lo=80, cholesterol=1, gluc=1, smoke=0, alco=0, active=0, cardio=1)]
```

Figure-2.7: Last “10” rows of the Dataframe

```
In [25]: df.orderBy(df['Index'].desc()).show(10)

+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index|   id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|68999|98569|19591|     2|   175|  88.0|  120|   80|          1|   1|    0|   0|     0|     1|
|68998|98568|19107|     1|   154|  77.0|   14|   90|          1|   1|    0|   0|     1|     0|
|68997|98567|21682|     1|   158|  78.0|  130|   90|          1|   1|    0|   0|     1|     1|
|68996|98566|20430|     1|   159| 105.0|  120|   80|          1|   1|    0|   0|     1|     1|
|68995|98565|17548|     2|   173|  72.0|  120|   80|          1|   1|    0|   0|     0|     0|
|68994|98564|18949|     2|   172|  62.0|  110|   70|          1|   1|    1|   0|     1|     0|
|68993|98563|14643|     2|   175|  75.0|  100|   70|          1|   1|    0|   0|     1|     0|
|68992|98562|17574|     1|   157|  63.0|  150|  100|          1|   1|    0|   0|     1|     0|
|68991|98561|21787|     1|   168|  70.0|  140|   90|          3|   1|    0|   0|     1|     1|
|68990|98559|20508|     2|   173|  91.0|  140|   90|          1|   1|    0|   1|     1|     0|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
only showing top 10 rows
```

Figure-2.8: First “10” rows of data which are descending order on “Index” column.

- **describe**: It presents the descriptive statistics of the Dataframe, as shown in Figure-2.9. We split the attributes and trigger the “describe” method to avoid a compact display.

```
In [24]: df.select('index','id').describe().show()

         df.select('age','gender','height','weight').describe().show()

         df.select('ap_hi','ap_lo','cholesterol','gluc').describe().show()

         df.select('smoke','alco','active','cardio').describe().show()

+-------+-----------------+-----------------+
|summary|            index|               id|
+-------+-----------------+-----------------+
|  count|            69000|            69000|
|   mean|          34499.5| 49257.99002898551|
| stddev|19918.728624086427|28438.175895091368|
|    min|                0|                0|
|    max|            68999|            98569|
+-------+-----------------+-----------------+

+-------+------------------+------------------+------------------+------------------+
|summary|               age|            gender|            height|            weight|
+-------+------------------+------------------+------------------+------------------+
|  count|             69000|             69000|             69000|             69000|
|   mean|19469.362175167036| 1.3495217391304348|164.35059420289855|74.21126086956522|
| stddev|2467.2731299236907|0.47682238629511076| 8.211304686499416|14.404311068284651|
|    min|             10798|                 1|                55|              10.0|
|    max|             48Yrs|                 2|               250|             200.0|
+-------+------------------+------------------+------------------+------------------+

+-------+------------------+------------------+------------------+------------------+
|summary|             ap_hi|             ap_lo|       cholesterol|              gluc|
+-------+------------------+------------------+------------------+------------------+
|  count|             69000|             69000|             68997|             69000|
|   mean|128.80997101449276| 96.65495652173912|1.3673637984260185|1.2263623188405797|
| stddev|154.94479317115955|189.36793185426143|0.6805505273319785|0.5722543067125166|
|    min|              -150|               -70|                 1|                 1|
|    max|             16020|             11000|                 3|                 3|
+-------+------------------+------------------+------------------+------------------+

+-------+--------------------+--------------------+--------------------+--------------------+
|summary|               smoke|                alco|              active|              cardio|
+-------+--------------------+--------------------+--------------------+--------------------+
|  count|               69000|               69000|               69000|               69000|
|   mean|0.08828985507246377|0.05376811594202899| 0.8040434782608695|0.4997246376811594|
| stddev|0.28371803462474876|0.22556117354304892|0.39693808938305636| 0.5000035474028258|
|    min|                   0|                   0|                   0|                   0|
|    max|                   1|                   1|                   1|                   1|
+-------+--------------------+--------------------+--------------------+--------------------+
```

Figure-2.9: Descriptive statistics of the Dataframe

After exploring the data using panda library, we have some insights about the overall data quality, as below points. However, a more detailed evaluation is still required.

1. The “index” and “id” can be excluded from data mining, as they are separate from our study.
2. The “age” data appears to have varying data value formats, as shown in Figure-2.6 and Figure-2.9.
3. Given the differing row count compared to other data, there’s a potential for missing or null values in the “cholesterol” data, as highlighted in Figure-2.9.
4. Some data seem strange, such as the minimum “height” of 55 cm and the minimum “weight” of 10 kg, as indicated in Figure-2.9.
5. Irregularities were found in both the minimum and maximum values of “ap_hi” and “ap_lo” data, as these values seem wrong and irrelevant, as shown in Figure-2.9.

Upon further examining each data, we observed the following points:

1. We used the “agg” method in mixture with the “isNull” method to check each attribute for missing or null values, as shown in Figure-2.10. This confirmed that the “cholesterol” column has missing values, which we initially suspected during our data exploration.

```
In [7]: # Check for missing values in each column
missing_values = df.agg(*[(sum_(col(c).isNull().cast("int"))).alias(c) for c in df.columns])

missing_values.show()
```

```
+-----+---+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index| id|age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+---+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    0|  0|  0|     0|     0|     0|    0|    0|          3|   0|    0|   0|     0|     0|
+-----+---+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
```

Figure-2.10: Check missing/null value of the Dataframe.

2. To analyse the “age” column, we utilised PySpark’s library’s “filter” method and “rlike” method to filter out the non-numeric data value of “age” column, as shown in Figure-2.11. It is confirmed that the “age” data attribute has inconsistence data format.

```
In [13]: df.filter(~col("age").rlike("^[0-9]+$")).show()
```

```
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index|   id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    4|    4|48Yrs|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|   28|   39|38Yrs|     2|   166|  66.0|  120|   80|          1|   1|    0|   0|     1|     0|
|23867|34120|44Yrs|     1|   161|  92.0|  906|    0|          2|   1|    0|   0|     1|     1|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
```

Figure-2.11: To analyse the “age” column.

3. To analyse the “height”, “weight”, “ap_hi” and “ap_lo” columns, we will apply PySpark’s library’s “filter” with SQL’s aggregation methods such as ‘max’ and ‘min’, as illustrated in Figure-2.12.

```
In [26]:
# Further analyse the "ap_hi", "ap_lo", "heigh" and "weight"
df.select(min('ap_hi').alias('Min-ap_hi'), max('ap_hi').alias('Max-ap_hi')).show()
df.select(min('ap_lo').alias('Min-ap_lo'), max('ap_lo').alias('Max-ap_lo')).show()
df.select(min('height').alias('Min-Height'), max('height').alias('Max-Height')).show()
df.select(min('weight').alias('Min-Weight'), max('weight').alias('Max-weight')).show()

+---------+---------+
|Min-ap_hi|Max-ap_hi|
+---------+---------+
|     -150|    16020|
+---------+---------+

+---------+---------+
|Min-ap_lo|Max-ap_lo|
+---------+---------+
|      -70|    11000|
+---------+---------+

+----------+----------+
|Min-Height|Max-Height|
+----------+----------+
|        55|       250|
+----------+----------+

+----------+----------+
|Min-Weight|Max-weight|
+----------+----------+
|      10.0|     200.0|
+----------+----------+
```

Figure-2.12: To analyse the columns “ap_hi”, “ap_lo”, “height” and “weight.

As illustrated in Figure-2.12, the following observations raise some questions about the dataset which we find out at earlier data explore:

- The minimum and maximum heights appear incorrect. Even though the shortest and tallest humans ever recorded were 55 cm and 250 cm respectively, the values in this dataset seem erroneous.
- Similar doubts arise regarding the minimum weight 10kg.
- Systolic (ap_hi) and Diastolic (ap_lo) blood pressures shouldn’t be negative.
- While Systolic (ap_hi) blood pressures above 180mmHg and Diastolic (ap_lo) above 120mmHg indicate emergency situations, our data shows extreme values like 16020mmHg and 11000mmHg, which are questionable.

4. For further analysis, we will examine the remaining data attributes: “gender”, “cholesterol”, “gluc”, “smoke”, “alco”, “active”, and “cardio”. To verify the data, we will utilize PySpark’s “filter” function combined with the SQL aggregation method “countDistinct”, as illustrated in Figure-2.13

```
In [39]:
# Further analyse the remaining attributes
df.select(countDistinct('gender'),countDistinct('cholesterol'),countDistinct('gluc'),countDistinct('smoke')).show()
df.select(countDistinct('alco'),countDistinct('active'),countDistinct('cardio')).show()

+---------------------+--------------------------+--------------------+--------------------+
|count(DISTINCT gender)|count(DISTINCT cholesterol)|count(DISTINCT gluc)|count(DISTINCT smoke)|
+---------------------+--------------------------+--------------------+--------------------+
|                    2|                         3|                   3|                   2|
+---------------------+--------------------------+--------------------+--------------------+

+--------------------+----------------------+----------------------+
|count(DISTINCT alco)|count(DISTINCT active)|count(DISTINCT cardio)|
+--------------------+----------------------+----------------------+
|                   2|                     2|                     2|
+--------------------+----------------------+----------------------+
```

Figure-2.13: To analysis the remaining data attributes.

As illustrated in Figure-2.13, the data values serve the flag type described in Table-2.1. To analyse the unique values of each attribute, we will use the “distinct” method, as demonstrated in Figure-2.14.

```
In [38]:
df.select('gender').distinct().show()
df.select('cholesterol').distinct().show()
df.select('gluc').distinct().show()
df.select('smoke').distinct().show()
df.select('alco').distinct().show()
df.select('active').distinct().show()
df.select('cardio').distinct().show()
```

```
+------+
|gender|
+------+
|     1|
|     2|
+------+

+-----------+
|cholesterol|
+-----------+
|       null|
|          1|
|          3|
|          2|
+-----------+

+----+
|gluc|
+----+
|   1|
|   3|
|   2|
+----+

+-----+
|smoke|
+-----+
|    1|
|    0|
+-----+

+----+
|alco|
+----+
|   1|
|   0|
+----+

+------+
|active|
+------+
|     1|
|     0|
+------+

+------+
|cardio|
+------+
|     1|
|     0|
+------+
```

Figure-2.14: To analyse distinct values of each attribute.

As shown in Figure-2.14, it is confirmed that the "cholesterol" column contains missing values.

5. Furthermore, we will investigate the connection between the target and feature data using the "groupBy" method combined with the SQL aggregation method "count". First, we will study the data relationship between "gender" and "cardio"; we found that the occurrence of cardiovascular disease (CVD) is equally distributed among both genders as shown in Figure-2.15. Additionally, we will visualise and confirm this distribution between "gender" and "cardio", as displayed in Figure-2.16. In order to plot the bar chart, we need to convert the PySpark's dataframe to pandas' dataframe type first and then using "seaborn" and "matplotlib" libraries to plot the diagram.

```
In [38]:
# 'Cardio vs Gender'
df.groupBy('cardio', 'gender').agg(count('*').alias('Count')).show()
```

```
+------+------+-----+
|cardio|gender|Count|
+------+------+-----+
|     1|     2|12195|
|     1|     1|22286|
|     0|     1|22597|
|     0|     2|11922|
+------+------+-----+
```

Figure-2.15: "cardio" vs "gender"

```
In [55]:
# Plotting 'Cardio vs Gender'
df_pd = df.toPandas()
sns.countplot(x='gender', hue='cardio', data=df_pd)
plt.title("Gender vs Cardio", size=14)
plt.show()
```

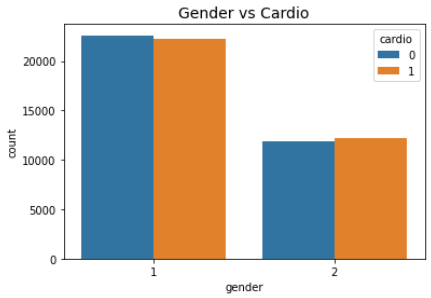


Figure-2.16: Plot “cardio vs gender”

Examining “cardio” relationships with other attributes, there’s an even distribution of CVD occurrences, suggesting balanced data as illustrated in Figure-2.17. we will plot the bar charts and visualise these relationships using “seaborn” and “matplotlib” libraries as shown in Figure-2.18.

```
In [37]:
df.groupBy('cardio','active').agg(count('*').alias('Count')).show()

df.groupBy('cardio', 'smoke').agg(count('*').alias('Count')).show()

df.groupBy('cardio', 'alco').agg(count('*').alias('Count')).show()
```

```
+------+------+-----+
|cardio|active|Count|
+------+------+-----+
|     1|     0| 7239|
|     1|     1|27242|
|     0|     0| 6282|
|     0|     1|28237|
+------+------+-----+

+------+-----+-----+
|cardio|smoke|Count|
+------+-----+-----+
|     1|    0|31590|
|     1|    1| 2891|
|     0|    0|31318|
|     0|    1| 3201|
+------+-----+-----+

+------+----+-----+
|cardio|alco|Count|
+------+----+-----+
|     1|   0|32688|
|     1|   1| 1793|
|     0|   0|32602|
|     0|   1| 1917|
+------+----+-----+
```

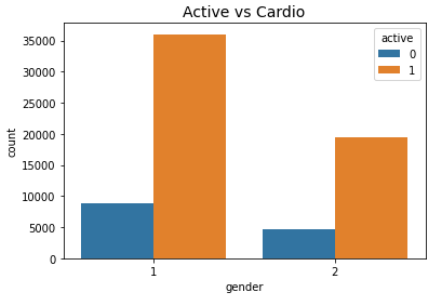
Figure-2.17: “cardio” vs other attributes

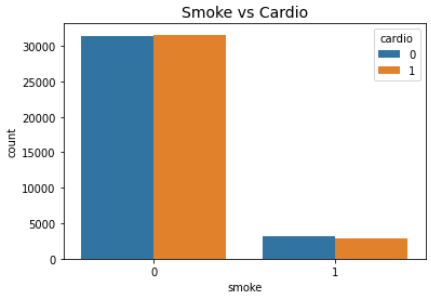
In [36]:

```
# Plotting 'Cardio vs active'
sns.countplot(x='gender', hue='active', data=df_pd)
plt.title("Active vs Cardio", size=14)
plt.show()

# Plotting 'Cardio vs active'
df_pd = df.toPandas()
sns.countplot(x='smoke', hue='cardio', data=df_pd)
plt.title("Smoke vs Cardio", size=14)
plt.show()

# Plotting 'Cardio vs active'
df_pd = df.toPandas()
sns.countplot(x='alco', hue='cardio', data=df_pd)
plt.title("Alco vs Cardio", size=14)
plt.show()
```





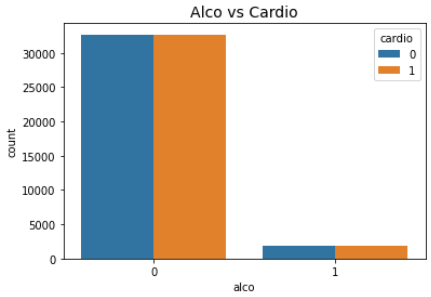


Figure-2.18: Plot “cardio” vs other attributes

Understanding our data’s quality is essential. If the data has errors or we overlook key details, our predictions could be wrong. With the insights from our visualisations:

- We know which columns need format standardisation.
- We are aware of missing values that need imputation.
- We have identified outliers that might require treatment or removal.
- We have discovered initial insights into attribute importance.

As we transition to data preprocessing, each point will be our checklist. A cleansed dataset sets the foundation for an effective predictive model.

2.4. Verifying Data Quality

After looking at the data quality in the previous Section 2.3, "Exploring Data", we found some issues with the data. Here are the problems we detected:

- **Missing Data**: There were some cases where the "cholesterol" attribute had no information recorded (shown in Figure-2.10 / Figure-2.14).
- **Data Errors and Extreme Outlier**: The "ap_hi" and "ap_lo" are measured in mm Hg (millimetres of mercury), but We noticed that the minimum and maximum values appear to be wrong. And the minimum value of "height" and "weight" too (shown in Figure-2.9 / Figure-2.12).
- **Inconsistent Data Value**: The "age" attribute had data recorded in years format instead of the total number of days, causing inconsistency (shown in Figure-2.9 / Figure-2.11).

To improve the data quality, we must address these issues by removing the missing cholesterol data records, setting a boundary to remove any impossible measurements of blood pressure, weight, and height data, and converting the inconsistent age data into the correct format (total number of days).

3. Data Preparation

3.1. Selection Data

For our study to predict CVD outcomes, we have thoroughly explored our dataset in the "Data Understanding", Section 2. From this exploration, we have established a strategy for data selection. We will process the following steps for Data Selection:

1. ***Correlation Method***: We utilised the "corr" method from the "pyspark.ml.stat" library, as illustrated in Figure-3.1, and the printed results display the correlation between the feature and the target and also visualise in heat-map diagram as shown in Figure-3.2

```
In [62]:
# prepare the columns and "age" column cannot correlation yet due to inconsistent data format
tmpDf = df.drop("age", "cardio")
columns = tmpDf.columns
target = 'cardio'

# apply correlation method
correlations = {}
for col in columns:
    corr_value = df.stat.corr(col, target)
    correlations[col] = corr_value

# print the result in descending order
print("\nCorrelation Results")
print("-----------------------------------")
sorted_correlations = sorted(correlations.items(), key=lambda x: x[1], reverse=True)
for col, corr_value in sorted_correlations:
    print(f"Correlation between \"{target}\" and \"{col}\": {corr_value:.4f}")
```

```
Correlation Results
-----------------------------------
Correlation between "cardio" and "cholesterol": 0.2215
Correlation between "cardio" and "weight": 0.1814
Correlation between "cardio" and "gluc": 0.0882
Correlation between "cardio" and "ap_lo": 0.0656
Correlation between "cardio" and "ap_hi": 0.0541
Correlation between "cardio" and "gender": 0.0087
Correlation between "cardio" and "index": 0.0040
Correlation between "cardio" and "id": 0.0040
Correlation between "cardio" and "alco": -0.0078
Correlation between "cardio" and "height": -0.0109
Correlation between "cardio" and "smoke": -0.0157
Correlation between "cardio" and "active": -0.0352
```

Figure-3.1: To print the correlation between features and target.

```
In [60]:
# preparing data to plot the heatmap diagram
attributes = list(correlations.keys())
correlation_values = list(correlations.values())
corr_matrix = [correlation_values]

# Generate the heatmap plot
plt.figure(figsize=(10, 1))  # Adjust size as per your preference
sns.heatmap(corr_matrix, annot=True, yticklabels=['cardio'], xticklabels=attributes, cmap="coolwarm", cbar=True, linewidths=0
plt.title("Correlation with 'cardio'")
plt.show()
```

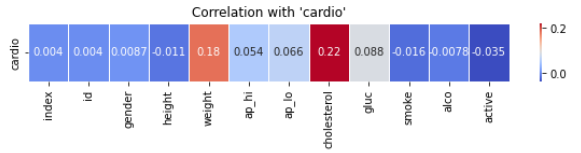


Figure-3.1: Correlation heat-map diagram between features and target “cardio”

2. ***Findings from Correlation Method***: From Figure-3.1, we identified the printed result correlations between the features and the target, as summarized in Table-3.1

Features	Correlation with Target “cardio”
cholesterol	0.2215
weight	0.1814
gluc	0.0882
ap_lo	0.0656
ap_hi	0.0541
gender	0.0087
Index	0.0040
Id	0.0040
alco	-0.0078
height	-0.0109
smoke	-0.0157
active	-0.0352

Table-3.1: Correlation value between features and target.

3. ***Attribute Relevance***:
   - High Correlation: Attributes “cholesterol” and “weight” displayed the highest correlation with the target.
   - Low Correlation: Attributes such as “gender”, “alco”, “smoke”, “height”, and “active” showed lower correlation values.

   Note: It is worth mentioning that these insights were obtained from unclean data. We will re-evaluate after the data-cleaning stage, because the “age” column is excluded from the feature list due to inconsistent data format.

4. ***Dropping Irrelevant Features***: Attributes like “index” and “id” appear not to offer meaningful information for predicting CVD outcomes. Therefore, we’ve decided to drop them and verified with “columns” method, as illustrated in Figure-3.3.

```
In [21]: # drop the "Index" and "Id" columns
         df= df.drop("Index", "Id")

         df.columns

Out[21]: ['age',
          'gender',
          'height',
          'weight',
          'ap_hi',
          'ap_lo',
          'cholesterol',
          'gluc',
          'smoke',
          'alco',
          'active',
          'cardio']
```

Figure-3.3: Remove the “id” and “index” columns.

5. ***Constructing New Relevant Features***: Some attributes may not significantly influence the outcome, but when combined with others, they become potent indicators. A couple of key examples include:
   - “Height”, despite its lower correlation, can be paired with “Weight” to determine the “BMI” (Body Mass Index).
   - The interplay between Systolic and Diastolic blood pressure can have a bearing on pulse rate.

   We will address upon these considerations in the “Section 3.3. Data Construction“ section.

With our data selection groundwork, we can transition to the next section: “Data Cleaning”. This stage will involve refining our dataset, removing anomalies, and ensuring our features are optimally prepared for analysis.

3.2. Cleaning Data

In this section, we will perform the data cleaning process to improve the data quality; we identified the following data quality issues missing data, data errors and the inconsistent value in the previous section, “section 2.4. Verifying Data Quality”.

3.2.1. Missing Data Issues

We know that values are missing in the “cholesterol” feature. To handle this, we will use the “na.drop” method to remove the entire row of missed and verified that the rows of missing value are dropped as shown in Figure-3.4

```
In [22]: # drop the missing value rows
         df= df.na.drop()

         df.select('cholesterol').distinct().show()

         +-----------+
         |cholesterol|
         +-----------+
         |          1|
         |          3|
         |          2|
         +-----------+
```

Figure-3.4: Dropped the rows of missing value and verified.

3.2.2. Data Errors and Extreme Outlier Issues

In this section, we’ll address “Data Errors and Extreme Outlier” concerns for the features “ap-hi”, “ap_lo”, “height”, and “weight”. We will eliminate rows with values in these features that fall outside the following conditions:

- Blood pressure measurements must range between 40 mm Hg and 370 mm Hg.

- Height must be at least 145 cm.
- Weight must be at least 45 kg.

As shown in Figure-3.5, we will use the “filter” method with the conditions, to confirm that the issues have been resolved, we will run the “describe” method again and examine the results.

```
In [72]: # handle data errors and extreme outlier
df = df.filter(
    ~(
        (df.ap_hi < 40) |
        (df.ap_hi > 370) |
        (df.ap_lo < 40) |
        (df.ap_lo > 370) |
        (df.height < 145) |
        (df.weight < 45)
    )
)

df.select('ap_hi','ap_lo','height','weight').describe().show()
```

```
+-------+------------------+-----------------+------------------+------------------+
|summary|             ap_hi|            ap_lo|            height|            weight|
+-------+------------------+-----------------+------------------+------------------+
|  count|             67181|            67181|             67181|             67181|
|   mean|126.64908232982539| 81.412795284381|164.56456438576384| 74.28256426668254|
| stddev|16.728044719625508|9.671091681265171| 7.702318486155489|14.150908077532737|
|    min|                60|               40|               145|              45.0|
|    max|               240|              190|               250|             200.0|
+-------+------------------+-----------------+------------------+------------------+
```

Figure-3.5: Handling the “Data Errors and Extreme Outlier” issues and verified

3.2.3. Inconsistent Data Value Issues

To address the problem of inconsistent data values, we will take the following steps:

1. We will filter the year formatted “age” rows from Dataframe as shown in Figure-3.6.

```
In [188]: # handle inconsistence data format

## First filter out the year format data
df_filtered = df.filter(df["age"].like("%Yrs%"))

df_filtered.show()
```

```
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|48Yrs|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|38Yrs|     2|   166|  66.0|  120|   80|          1|   1|    0|   0|     1|     0|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
```

Figure-3.6: Filter the year formatted “age” rows.

2. We will employ the “withColumn” method to transform the year format into an age format represented in days. This conversion will be accomplished using the “regexp_replace” function described in Figure-3.7.

```
In [189]: df_Correction = df_filtered.withColumn("age", regexp_replace(df["age"], "Yrs", "") * 365)
df_Correction.show()
```

```
+-------+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-------+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|17520.0|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|13870.0|     2|   166|  66.0|  120|   80|          1|   1|    0|   0|     1|     0|
+-------+------+------+------+-----+-----+-----------+----+-----+----+------+------+
```

Figure-3.7: Correction of data value

3. And then we will filter out the year format and update back corrected Dataframe to orginal Dataframe and confirmed the issue is solved as shown in Figure-3.8

```
In [249]:
## adding back to orginal dataframe and drop the inconsistant format

df = df.filter(~(df["age"].like("%Yrs%")))

df = df.union(df_Correction)

df.filter(df["age"].like("%Yrs%")).show()
```

```
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+
```

Figure-3.8: Update back to Dataframe and confirmed the issue is solved.

3.2.4. Verify and Remove the Duplicate Rows

After solving the above three main issues, we make sure on the current Dataframe should not have duplicate data rows. we will perform the following steps to handle this:

1. We will utilise the methods “groupBy” and “count” to verify the Dataframe have duplicate rows as described in Figure-3.9.

```
In [326]:
# Count occurrences of each row
df_grouped = df.groupBy(df.columns).agg(count("*").alias("count"))

# Check for duplicates
if df_grouped.filter("`count` > 1").count() > 0:
    print(f"DataFrame has duplicate rows! - {df_grouped.filter('count > 1').count()} rows")
else:
    print("DataFrame does not have duplicate rows!")
```

```
DataFrame has duplicate rows! - 24 rows
```

Figure-3.9: Verify the duplicate rows.

2. We will use “dropDuplicates” method to remove the duplicate rows and confirmed the duplicated rows are removed as shown in Figure-3.10.

```
In [300]:
# drop the duplicates row
df = df.dropDuplicates()

# Count occurrences of each row
df_grouped = df.groupBy(df.columns).agg(count("*").alias("count"))

# Check for duplicates
if df_grouped.filter("`count` > 1").count() > 0:
    print(f"DataFrame has duplicate rows! - {df_grouped.filter('count > 1').count()} rows")
else:
    print("DataFrame does not have duplicate rows!")
```

```
DataFrame does not have duplicate rows!
```

Figure-3.10: Remove the duplicate rows.

3.3. Data Constructing

This section will introduce two new features: “BMI” and “Pulse_rate”. We found that the “Height” and “Weight” variables had limited impact during our data selection process. Yet, we recognised an opportunity to derive a more relevant “BMI” feature from these variables. Similarly, the “Pulse_rate” feature can be deduced from the relationship between Systolic and Diastolic blood pressures.

$$\text{BMI} = \frac{\text{Weight (in kilograms)}}{\text{Height}^2 \text{ (in meters)}}$$

Figure-3.11: BMI formula in metric system

As illustrated in Figure-3.12, we will utilise the “withColumn” method to add the new feature “BMI” with the formula (shown in Figure-3.11).

```
In [334]:
# adding new feature "BMI"

df = df.withColumn("BMI", round(df["weight"] / ((df["height"]/100) ** 2),2))

df.show()
```

```
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+
|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|  BMI|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+
|18393|     2|   168|  62.0|  110|   80|          1|   1|    0|   0|     1|     0|21.97|
|20228|     1|   156|  85.0|  140|   90|          3|   1|    0|   0|     1|     1|34.93|
|18857|     1|   165|  64.0|  130|   70|          3|   1|    0|   0|     0|     1|23.51|
|17623|     2|   169|  82.0|  150|  100|          1|   1|    0|   0|     1|     1|28.71|
|21914|     1|   151|  67.0|  120|   80|          2|   2|    0|   0|     0|     0|29.38|
|22113|     1|   157|  93.0|  130|   80|          3|   1|    0|   0|     1|     0|37.73|
|22584|     2|   178|  95.0|  130|   90|          3|   3|    0|   0|     1|     1|29.98|
|17668|     1|   158|  71.0|  110|   70|          1|   1|    0|   0|     1|     0|28.44|
|19834|     1|   164|  68.0|  110|   60|          1|   1|    0|   0|     0|     0|25.28|
|22530|     1|   169|  80.0|  120|   80|          1|   1|    0|   0|     1|     0|28.01|
|18815|     2|   173|  60.0|  120|   80|          1|   1|    0|   0|     1|     0|20.05|
|14791|     2|   165|  60.0|  120|   80|          1|   1|    0|   0|     0|     0|22.04|
|19809|     1|   158|  78.0|  110|   70|          1|   1|    0|   0|     1|     0|31.24|
|14532|     2|   181|  95.0|  130|   90|          1|   1|    1|   1|     1|     0| 29.0|
|16782|     2|   172| 112.0|  120|   80|          1|   1|    0|   0|     0|     1|37.86|
|21296|     1|   170|  75.0|  130|   70|          1|   1|    0|   0|     0|     0|25.95|
|16747|     1|   158|  52.0|  110|   70|          1|   3|    0|   0|     1|     0|20.83|
|17482|     1|   154|  68.0|  100|   70|          1|   1|    0|   0|     0|     0|28.67|
|21755|     2|   162|  56.0|  120|   70|          1|   1|    1|   0|     1|     0|21.34|
|19778|     2|   163|  83.0|  120|   80|          1|   1|    0|   0|     1|     0|31.24|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+
only showing top 20 rows
```

Figure-3.12: Add the new feature “BMI”.

As shown in Figure-3.13, we will use the “withColumn” method to construct the new feature “Pulse_Rate”.

```
In [336]:
# Adding new feature Pulse_Rate
df = df.withColumn("Pulse_Rate", df["ap_hi"] - df["ap_lo"])

df.show()
```

```
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+----------+
|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|  BMI|Pulse_Rate|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+----------+
|18393|     2|   168|  62.0|  110|   80|          1|   1|    0|   0|     1|     0|21.97|        30|
|20228|     1|   156|  85.0|  140|   90|          3|   1|    0|   0|     1|     1|34.93|        50|
|18857|     1|   165|  64.0|  130|   70|          3|   1|    0|   0|     0|     1|23.51|        60|
|17623|     2|   169|  82.0|  150|  100|          1|   1|    0|   0|     1|     1|28.71|        50|
|21914|     1|   151|  67.0|  120|   80|          2|   2|    0|   0|     0|     0|29.38|        40|
|22113|     1|   157|  93.0|  130|   80|          3|   1|    0|   0|     1|     0|37.73|        50|
|22584|     2|   178|  95.0|  130|   90|          3|   3|    0|   0|     1|     1|29.98|        40|
|17668|     1|   158|  71.0|  110|   70|          1|   1|    0|   0|     1|     0|28.44|        40|
|19834|     1|   164|  68.0|  110|   60|          1|   1|    0|   0|     0|     0|25.28|        50|
|22530|     1|   169|  80.0|  120|   80|          1|   1|    0|   0|     1|     0|28.01|        40|
|18815|     2|   173|  60.0|  120|   80|          1|   1|    0|   0|     1|     0|20.05|        40|
|14791|     2|   165|  60.0|  120|   80|          1|   1|    0|   0|     0|     0|22.04|        40|
|19809|     1|   158|  78.0|  110|   70|          1|   1|    0|   0|     1|     0|31.24|        40|
|14532|     2|   181|  95.0|  130|   90|          1|   1|    1|   1|     1|     0| 29.0|        40|
|16782|     2|   172| 112.0|  120|   80|          1|   1|    0|   0|     0|     1|37.86|        40|
|21296|     1|   170|  75.0|  130|   70|          1|   1|    0|   0|     0|     0|25.95|        60|
|16747|     1|   158|  52.0|  110|   70|          1|   3|    0|   0|     1|     0|20.83|        40|
|17482|     1|   154|  68.0|  100|   70|          1|   1|    0|   0|     0|     0|28.67|        30|
|21755|     2|   162|  56.0|  120|   70|          1|   1|    1|   0|     1|     0|21.34|        50|
|19778|     2|   163|  83.0|  120|   80|          1|   1|    0|   0|     1|     0|31.24|        40|
+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+-----+----------+
only showing top 20 rows
```

Figure-3.13: Add the new feature “Pulse_Rate”.

3.4. Data Integration

In this study, we will work with two sets of data. The first dataset consists of 69,000 participants (Figure-3.14), and the second has 1,000 participants (Figure-3.15) applying the “info” method form pandas’ library.

```
In [340]:
# load the csv file - 1
df1 = spark.read.csv('Datasets/heart_dataset_1.csv', inferSchema=True, header=True)

print(f"Total records: {df1.count()}")
```

```
Total records: 69000
```

Figure-3.14: First Dataset total records

```
In [341]:
# load the csv file - 2
df2 = spark.read.csv('Datasets/heart_dataset_2.csv', inferSchema=True, header=True)

print(f"Total records: {df2.count()}")

Total records: 1000
```

Figure-3.15: Second Dataset total records

We will use the union method to merge the two datasets, as illustrated in Figure-3.16. After merging, we will validate by checking the total number of records in the combined dataframe.

```
In [343]:
# Merge two datasets
merge_df = df1.union(df2)

print(f"Total records: {merge_df.count()}")

Total records: 70000
```

Figure-3.16: Merging two datasets.

Upon completing the merging process, we will inspect the merged data with PySpark's DataFrame methods, as illustrated in Figures 3.17, 3.18, 3.19 and 3.20.

```
In [121]:
# verify the schema
merge_df.printSchema()

root
 |-- index: integer (nullable = true)
 |-- id: integer (nullable = true)
 |-- age: string (nullable = true)
 |-- gender: integer (nullable = true)
 |-- height: integer (nullable = true)
 |-- weight: double (nullable = true)
 |-- ap_hi: integer (nullable = true)
 |-- ap_lo: integer (nullable = true)
 |-- cholesterol: integer (nullable = true)
 |-- gluc: integer (nullable = true)
 |-- smoke: integer (nullable = true)
 |-- alco: integer (nullable = true)
 |-- active: integer (nullable = true)
 |-- cardio: integer (nullable = true)
```

Figure-3.17: Using "printSchema" method to verify the merge dataset schema.

```
In [122]:
# verify the first 10 rows in table
merge_df.orderBy(merge_df['Index'].asc()).show(10)

+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index| id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|    0|  0|18393|     2|   168|  62.0|  110|   80|          1|   1|    0|   0|     1|     0|
|    1|  1|20228|     1|   156|  85.0|  140|   90|          3|   1|    0|   0|     1|     1|
|    2|  2|18857|     1|   165|  64.0|  130|   70|          3|   1|    0|   0|     0|     1|
|    3|  3|17623|     2|   169|  82.0|  150|  100|          1|   1|    0|   0|     1|     1|
|    4|  4|48Yrs|     1|   156|  56.0|  100|   60|          1|   1|    0|   0|     0|     0|
|    5|  8|21914|     1|   151|  67.0|  120|   80|          2|   2|    0|   0|     0|     0|
|    6|  9|22113|     1|   157|  93.0|  130|   80|          3|   1|    0|   0|     1|     0|
|    7| 12|22584|     2|   178|  95.0|  130|   90|          3|   3|    0|   0|     1|     1|
|    8| 13|17668|     1|   158|  71.0|  110|   70|          1|   1|    0|   0|     1|     0|
|    9| 14|19834|     1|   164|  68.0|  110|   60|          1|   1|    0|   0|     0|     0|
+-----+---+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
only showing top 10 rows
```

Figure-3.18: Using "orderBy" method on "Index" column to inspect the first ten rows of data.

```
In [123]:
# verify the last 10 rows in table
merge_df.orderBy(merge_df['Index'].desc()).show(10)
```

```
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|index|   id|  age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
|69999|99999|20540|     1|   170|  72.0|  120|   80|          2|   1|    0|   0|     1|     0|
|69998|99998|22431|     1|   163|  72.0|  135|   80|          1|   2|    0|   0|     0|     1|
|69997|99996|19066|     2|   183| 105.0|  180|   90|          3|   1|    0|   1|     0|     1|
|69996|99995|22601|     1|   158| 126.0|  140|   90|          2|   2|    0|   0|     1|     1|
|69995|99993|19240|     2|   168|  76.0|  120|   80|          1|   1|    1|   0|     1|     0|
|69994|99992|21074|     1|   165|  80.0|  150|   80|          1|   1|    0|   0|     1|     1|
|69993|99991|19699|     1|   172|  70.0|  130|   90|          1|   1|    0|   0|     1|     1|
|69992|99990|18792|     1|   161|  56.0|  170|   90|          1|   1|    0|   0|     1|     1|
|69991|99988|20609|     1|   159|  72.0|  130|   90|          2|   2|    0|   0|     1|     0|
|69990|99986|15094|     1|   168|  72.0|  110|   70|          1|   1|    0|   0|     1|     1|
+-----+-----+-----+------+------+------+-----+-----+-----------+----+-----+----+------+------+
only showing top 10 rows
```

Figure-3.19: Using “orderBy” method on “Index” column to inspect the last ten rows of data.

```
In [124]:
merge_df.select('index','id').describe().show()

merge_df.select('age','gender','height','weight').describe().show()

merge_df.select('ap_hi','ap_lo','cholesterol','gluc').describe().show()

merge_df.select('smoke','alco','active','cardio').describe().show()
```

```
+-------+------------------+------------------+
|summary|             index|                id|
+-------+------------------+------------------+
|  count|             70000|             70000|
|   mean|           34999.5|        49972.4199|
| stddev|20207.403758688713|28851.302323172746|
|    min|                 0|                 0|
|    max|             69999|             99999|
+-------+------------------+------------------+

+-------+------------------+-------------------+------------------+------------------+
|summary|               age|             gender|            height|            weight|
+-------+------------------+-------------------+------------------+------------------+
|  count|             70000|              70000|             70000|             70000|
|   mean|19469.010557595324| 1.3495714285714286|164.35922857142856|          74.20569|
| stddev|2467.1940676214967|0.47683801558286276| 8.210126364538075|14.395756678511344|
|    min|             10798|                  1|                55|              10.0|
|    max|             48Yrs|                  2|               250|             200.0|
+-------+------------------+-------------------+------------------+------------------+

+-------+------------------+------------------+------------------+------------------+
|summary|             ap_hi|             ap_lo|       cholesterol|              gluc|
+-------+------------------+------------------+------------------+------------------+
|  count|             70000|             70000|             69997|             70000|
|   mean| 128.8172857142857|96.63041428571428|1.3668871523065274| 1.226457142857143|
| stddev|154.01141945609197|188.4725302963891|0.6802606859560671|0.5722702766138457|
|    min|              -150|              -70|                 1|                 1|
|    max|             16020|             11000|                 3|                 3|
+-------+------------------+------------------+------------------+------------------+

+-------+--------------------+--------------------+-------------------+--------------------+
|summary|               smoke|                alco|             active|              cardio|
+-------+--------------------+--------------------+-------------------+--------------------+
|  count|               70000|               70000|              70000|               70000|
|   mean|0.08812857142857143|0.053771428571428574|0.8037285714285715|              0.4997|
| stddev| 0.2834838167699366| 0.22556770360410552|0.3971790635049264|0.5000034814661851|
|    min|                   0|                   0|                  0|                   0|
|    max|                   1|                   1|                  1|                   1|
+-------+--------------------+--------------------+-------------------+--------------------+
```

Figure-3.20: Using “describe” method to check the descriptive statistics for each attribute.

After successfully merging the two datasets and examining the combined data, we observed that certain portions require cleaning. It’s essential to revisit the data processing steps to ensure the quality of the data is optimal for analysis. We will focus on this in the upcoming “Data Formatting” section.

3.5. Data Formatting

In this section, we will work with the newly merged CSV file and repeat all the steps we took in the previous section. This includes selecting the data, cleaning it up, and constructing it as illustrated in Figure-3.21. The end goal is to create the final dataset that we are going to use for further analysis.

```
In [125]:
# reprocessing the data-processing on merge dataframe
## drop the "Index" and "Id" columns
merge_df= merge_df.drop("Index", "Id")

## drop the missing value rows
merge_df= merge_df.na.drop()

## handle data errors and extreme outlier
merge_df = merge_df.filter(
    ~(
        (merge_df.ap_hi < 40) |
        (merge_df.ap_hi > 370) |
        (merge_df.ap_lo < 40) |
        (merge_df.ap_lo > 370) |
        (merge_df.height < 145) |
        (merge_df.weight < 45)
    )
)

## handle inconsistence data format
### filter out the year format data
df_filtered = merge_df.filter(merge_df["age"].like("%Yrs%"))

### correction of age format
df_Correction = df_filtered.withColumn("age", regexp_replace(merge_df["age"], "Yrs", "") * 365)

### adding back to orginal dataframe and drop the inconsistant format
merge_df = merge_df.filter(~(merge_df["age"].like("%Yrs%")))
merge_df = merge_df.union(df_Correction)

### drop the duplicates row
merge_df = merge_df.dropDuplicates()

### adding new feature "BMI"
merge_df = merge_df.withColumn("BMI", round(merge_df["weight"] / ((merge_df["height"]/100) ** 2),2))

### Adding new feature Pulse_Rate
merge_df = merge_df.withColumn("Pulse_Rate", merge_df["ap_hi"] - merge_df["ap_lo"])
```

Figure-3.21: Revisit the earlier process.

After executing this process, we will verify the data's quality on the new merged dataset, as illustrated in Figures 3.22, 3.23, 3.24, and 3.25. Following these validations, we can confirm the success of the reprocessing.

1. As illustrated in Figure-3.22, we used to "describe" method to verify the dataset of descriptive statistics.

```
In [342]:
# verify the descriptive statistics
merge_df.select('age','gender','height','weight').describe().show()

merge_df.select('ap_hi','ap_lo','cholesterol','gluc').describe().show()

merge_df.select('smoke','alco','active','cardio').describe().show()
```

```
+-------+-----------------+-----------------+-----------------+-----------------+
|summary|              age|           gender|           height|           weight|
+-------+-----------------+-----------------+-----------------+-----------------+
|  count|            68126|            68126|            68126|            68126|
|   mean|19463.850776502364|1.3509673252502716|164.57329066729295| 74.28221604086548|
| stddev| 2465.976972345727| 0.477276236107067| 7.704646575797962|14.143546525455188|
|    min|            10798|                1|              145|             45.0|
|    max|            23713|                2|              250|            200.0|
+-------+-----------------+-----------------+-----------------+-----------------+

+-------+-----------------+-----------------+-----------------+-----------------+
|summary|            ap_hi|            ap_lo|      cholesterol|             gluc|
+-------+-----------------+-----------------+-----------------+-----------------+
|  count|            68126|            68126|            68126|            68126|
|   mean|126.66249302762529|81.41396823532865| 1.365176290990224|1.2264333734550685|
| stddev| 16.73257009698175| 9.6633849885015|0.6794400741975859|0.5725595361834471|
|    min|               60|               40|                1|                1|
|    max|              240|              190|                3|                3|
+-------+-----------------+-----------------+-----------------+-----------------+

+-------+-------------------+-------------------+-------------------+-------------------+
|summary|              smoke|               alco|             active|             cardio|
+-------+-------------------+-------------------+-------------------+-------------------+
|  count|              68126|              68126|              68126|              68126|
|   mean| 0.088541819569621|0.05391480492029475|0.8032322461321668| 0.495596394915304|
| stddev|0.2840833511064471|0.22585116221079535|0.3975582031711206|0.4999842774551799|
|    min|                  0|                  0|                  0|                  0|
|    max|                  1|                  1|                  1|                  1|
+-------+-------------------+-------------------+-------------------+-------------------+
```

Figure-3.22: Descriptive statistics of new dataset.

2. Used the “filter” method to verify the inconsistent date format are handled as shown in Figure-3.23.

```
In [340]: # Verify the inconsistent data format on "age" is handled
          merge_df.filter(merge_df["age"].like("%Yrs%")).show()

+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
|age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|BMI|Pulse_Rate|
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
```

Figure-3.23: Verify the inconsistent date format.

3. Applied “select” method with conditions to check missing value, as displayed in Figure-3.24

```
In [341]: # Verify the missing values are handled
          missing_counts = merge_df.select([count(when(isnan(c), c)).alias(c) for c in merge_df.columns])
          missing_counts.show()

+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
|age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke|alco|active|cardio|BMI|Pulse_Rate|
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
|  0|     0|     0|     0|    0|    0|          0|   0|    0|   0|     0|     0|  0|         0|
+---+------+------+------+-----+-----+-----------+----+-----+----+------+------+---+----------+
```

Figure-3.24: Check the missing value.

4. As illustrated in Figure-3.25, we will use the “printSchema” to check the data type for each attribute.

```
In [343]: # verify the schema
          merge_df.printSchema()

root
 |-- age: string (nullable = true)
 |-- gender: integer (nullable = true)
 |-- height: integer (nullable = true)
 |-- weight: double (nullable = true)
 |-- ap_hi: integer (nullable = true)
 |-- ap_lo: integer (nullable = true)
 |-- cholesterol: integer (nullable = true)
 |-- gluc: integer (nullable = true)
 |-- smoke: integer (nullable = true)
 |-- alco: integer (nullable = true)
 |-- active: integer (nullable = true)
 |-- cardio: integer (nullable = true)
 |-- BMI: double (nullable = true)
 |-- Pulse_Rate: integer (nullable = true)
```

Figure-3.25: Verify the data type.

Upon examining Figure-3.25, we observed that “age” attribute is not appropriately typed based on their content which must be “numeric” data type instead of “string”. We will use the “withColumn” method combine with “cast” method to rectify the data type mismatches and confirm that the data type mismatches are rectified as shown in Figure-3.26.

```
In [87]:
# Data type correction
merge_df = merge_df.withColumn("age", merge_df["age"].cast('int'))

merge_df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- gender: integer (nullable = true)
 |-- height: integer (nullable = true)
 |-- weight: double (nullable = true)
 |-- ap_hi: integer (nullable = true)
 |-- ap_lo: integer (nullable = true)
 |-- cholesterol: integer (nullable = true)
 |-- gluc: integer (nullable = true)
 |-- smoke: integer (nullable = true)
 |-- alco: integer (nullable = true)
 |-- active: integer (nullable = true)
 |-- cardio: integer (nullable = true)
 |-- BMI: double (nullable = true)
 |-- Pulse_Rate: integer (nullable = true)
```

Figure-3.27: Rectify the mismatch datatype.

Based on our previous "Section 2. Data Understanding" section, we noted that the 'age' attribute is represented in days. We will use "withColumn" method to convert this to years to enhance clarity, as shown in Figure-3.28, and confirm that the converting is successful.

```
In [401]:
# Convert age in days to years
merge_df = merge_df.withColumn("age", round(merge_df["age"] / 365, 0))

merge_df.show()
```

```
+----+------+------+------+-----+-----+-----------+----+-----+-----+------+------+-----+----------+
| age|gender|height|weight|ap_hi|ap_lo|cholesterol|gluc|smoke| alco|active|cardio|  BMI|Pulse_Rate|
+----+------+------+------+-----+-----+-----------+----+-----+-----+------+------+-----+----------+
|57.0|     1|   162|  85.0|  120|   70|          3|   3|false|false|  true|  true|32.39|        50|
|60.0|     1|   159| 101.0|  170|   90|          1|   1|false|false|  true|  true|39.95|        80|
|65.0|     1|   165|  60.0|  120|   80|          1|   1|false|false| false|  true|22.04|        40|
|65.0|     2|   165|  67.0|  120|   80|          1|   1|false|false|  true| false|24.61|        40|
|48.0|     1|   156|  74.0|  150|   90|          1|   1|false|false| false|  true|30.41|        60|
|57.0|     1|   163|  63.0|  120|   80|          3|   3|false|false|  true| false|23.71|        40|
|44.0|     1|   169|  73.0|  100|   70|          1|   1| true|false|  true| false|25.56|        30|
|54.0|     1|   158|  50.0|  120|   80|          1|   1|false|false|  true| false|20.03|        40|
|44.0|     1|   166|  68.0|  100|   60|          1|   1|false|false|  true|  true|24.68|        40|
|47.0|     2|   176|  71.0|  100|   70|          1|   2|false|false|  true| false|22.92|        30|
|46.0|     1|   160|  55.0|  110|   80|          1|   1|false|false|  true|  true|21.48|        30|
|54.0|     1|   165|  86.0|  140|   90|          1|   1|false|false|  true|  true|31.59|        50|
|55.0|     1|   164|  64.0|  110|   70|          1|   1|false|false|  true| false| 23.8|        40|
|54.0|     2|   170|  84.0|  150|  100|          1|   1| true| true|  true|  true|29.07|        50|
|46.0|     1|   157|  68.0|  180|  100|          2|   1|false|false|  true| false|27.59|        80|
|57.0|     1|   169|  70.0|  120|   80|          1|   1|false|false|  true| false|24.51|        40|
|52.0|     1|   154| 120.0|  120|   89|          3|   3|false|false|  true| false| 50.6|        31|
|50.0|     2|   170|  72.0|  140|   90|          2|   1|false|false|  true|  true|24.91|        50|
|58.0|     2|   166|  75.0|  120|   80|          1|   3|false|false|  true| false|27.22|        40|
|62.0|     1|   165|  65.0|  120|   90|          1|   1|false|false|  true| false|23.88|        30|
+----+------+------+------+-----+-----+-----------+----+-----+-----+------+------+-----+----------+
only showing top 20 rows
```

Figure-3.28: Convert the age in days to years and verified.

Having addressed the data understanding and processing, we are now transitioning into the next stage: Data Transformation. This section will address various techniques to refine and optimise our dataset for subsequent analyses.

4. Data Transformation

4.1. Data Reduction

In this section, we will employ two methods to determine features based on their significance:

1. ***Correlation Method***: A traditional approach used to identify the correlation between features and the target. We previously applied this method in the 'Data Selection' phase.
2. ***Random Forest Classifier:*** It is a tree-based method and offers feature importance scores, giving insights into which features most influence the target outcome.

4.1.1. Correlation Method

We will apply the correlation method to generate the result and visualise in heat-map diagram, as shown in Figure-4.1 and Figure4.2. To facilitate easier visualisation, we will compile the correlation values between the features and the target into a structured table format, as presented in Table-4.1.

```
In [47]:
# prepare the columns
tmpDf = merge_df.drop("cardio")
columns = tmpDf.columns
target = 'cardio'

# apply correlation method
correlations = {}
for col in columns:
    corr_value = merge_df.stat.corr(col, target)
    correlations[col] = corr_value

# print the result in descending order
print("\nCorrelation Results")
print("-----------------------------------")
sorted_correlations = sorted(correlations.items(), key=lambda x: x[1], reverse=True)
for col, corr_value in sorted_correlations:
    print(f"Correlation between \"{target}\" and \"{col}\": {corr_value:.4f}")
```

```
Correlation Results
-----------------------------------
Correlation between "cardio" and "ap_hi": 0.4243
Correlation between "cardio" and "ap_lo": 0.3340
Correlation between "cardio" and "Pulse_Rate": 0.3182
Correlation between "cardio" and "age": 0.2386
Correlation between "cardio" and "cholesterol": 0.2212
Correlation between "cardio" and "BMI": 0.1900
Correlation between "cardio" and "weight": 0.1787
Correlation between "cardio" and "gluc": 0.0893
Correlation between "cardio" and "gender": 0.0066
Correlation between "cardio" and "alco": -0.0081
Correlation between "cardio" and "height": -0.0133
Correlation between "cardio" and "smoke": -0.0166
Correlation between "cardio" and "active": -0.0376
```

Figure-4.1: Correlation Results

```
In [91]:
# preparing data to plot the heatmap diagram
attributes = list(correlations.keys())
correlation_values = list(correlations.values())
corr_matrix = [correlation_values]

# Generate the heatmap plot
plt.figure(figsize=(10, 1))  # Adjust size as per your preference
sns.heatmap(corr_matrix, annot=True, yticklabels=['cardio'], xticklabels=attributes, cmap="coolwarm", cbar=True, linewidths=0
plt.title("Correlation with 'cardio'")
plt.show()
```

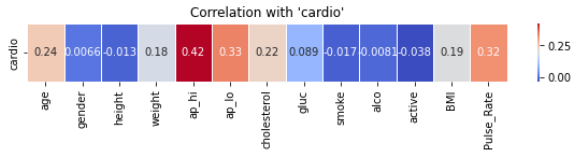


Figure-4.2: Visualise the correlation results in heat-map.

Features	Correlation with Target value
ap_hi	0.4243
ap_lo	0.3340
Pulse_Rate	0.3182

age	0.2390
cholesterol	0.2212
BMI	0.1900
weight	0.1787
gluc	0.0893
gender	0.0066
alco	-0.0081
height	-0.0133
smoke	-0.0166
active	-0.0376

Table-4.1: Correlation value between features and target

Studying the correlation result (Figure-4.1) and the detailed values in Table-4.1, we can draw the following conclusions:

- ***High Correlation:*** "ap_hi" prominently displays the strongest positive correlation of 0.4243 with the target value. it points to its crucial role as a predictor.
- ***Substantial Positive Influences:*** Several other features, including "ap_lo" (0.3340), "Pulse_Rate" (0.3182), "age" (0.2390), and "cholesterol" (0.2212) also register noticeable positive correlations. This implies they have a significant bearing on the target value.
- ***Derived Metrics:*** The feature "BMI", which is derived, possesses a notable correlation of 0.1900, emphasizing its relevance within this dataset.
- ***Moderate Influences:*** Features such as "weight" (0.1787) and "gluc" (0.0893) have modest positive correlations, suggesting they have some moderate influence on the target but are not as strong as the top influencers.
- ***Minor & Negative Impact:*** Some features, namely "gender" (0.0066), "alco" (-0.0081), "height" (-0.0133), "smoke" (-0.0166), and "active" (-0.0376), manifest weak or negative correlations. Their impact on the target is minimal or, in some cases, inversely related.

Overall, this data clarifies which features might be key influencers for our target, guiding us for the next steps in our analysis.

4.1.2. Random Forest Classifier

As shown in Figure-4.3, we will begin by separating the dataset into features and the target, and then assemble the features into a single vector column. Next, we will construct a "Random Forest Classifier", fit the data, and then rank the features based on their importance to the target.

```
In [ ]:
# separate the features and target
tmpDf = merge_df.drop("cardio")
feature_cols = tmpDf.columns

# Assemble features into a single vector column
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
df_assembled = assembler.transform(merge_df)

# Create and train a RandomForest model
rfCls = RandomForestClassifier(labelCol="cardio", featuresCol="features", numTrees=100)
model = rfCls.fit(df_assembled)

# Extract feature importances and sort them in descending order
importances = model.featureImportances.toArray()
sorted_indices = importances.argsort()[::-1]

# Print the result
print("\nImportance Features")
print("-----------------------------------")
for index in sorted_indices:
    print(f"{feature_cols[index]}: {importances[index]:.4f}")
```

Figure-4.3: Build Random Forest Classifier

```
Importance Features
-----------------------------------
ap_hi: 0.4653
ap_lo: 0.1923
Pulse_Rate: 0.1652
cholesterol: 0.0796
age: 0.0751
BMI: 0.0091
weight: 0.0083
gluc: 0.0029
active: 0.0010
height: 0.0005
smoke: 0.0004
alco: 0.0002
gender: 0.0000
```

Figure-4.4: Ranking Importance Features Result

From the ranking results in Figure-4.4, it shows that “ap_hi” is the most influential feature, followed closely by “ap_lo” and “Pulse_Rate”, in determining the target variable ‘cardio’. In contrast, features like “alco” and “gender” have less influence.

Utilising both the Correlation and Random Forest Classifier methods, we have derived clear insights into the importance of each feature for predicting our target variable,” cardio.” A summarised recommendation is as follows:

Primary Features for Selection:

- ap_hi: This is a primary predictor with the highest correlation of 0.4243 and dominates the Random Forest importance score at 0.4653.
- ap_lo and Pulse_Rate: These features rank highly in correlation and the Random Forest importance scores, cementing their status as influential predictors.
- age: With a correlation value of 0.2390 and an importance score of 0.0751, it continues to be a feature of significance.

Secondary Considerations:

- cholesterol, BMI, and weight: While they may not top the list, their consistent presence in both methods underlines their potential importance in predicting the target.
- gluc: Although it has a modest correlation value of 0.0893, its Random Forest importance score is relatively low at 0.0029, making it a secondary consideration.

Tertiary Features (To be Considered Cautiously):

- active, height, smoke, alco, and gender: All these features either have less importance, negative correlations, or insignificant importance scores in the Random Forest ranking. These features might be necessary in some situations, but they don't seem very influential in our current data study.

Based on the recommendations, our study will exclude the ‘active’, ‘height’, ‘smoke’, ‘alco’, and ‘gender’ features as illustrated in Figure-4.5.

```
In [105]:
# Drop the columns which are less correlation/importance
merge_df = merge_df.drop("active", "height", "smoke", "alco", "gender")

merge_df.columns

Out[105]:
['age',
 'weight',
 'ap_hi',
 'ap_lo',
 'cholesterol',
 'gluc',
 'cardio',
 'BMI',
 'Pulse_Rate']
```

Figure-4.5: Dropped the ‘active’, ‘height’, ‘smoke’, ‘alco’, and ‘gender’ features.

With these features identified and prioritised, we will proceed to the next section, Data Projection, where we will transform the selected data in preparation for modelling.

4.2. Data Projection

In the previous section, we implemented various data transformation techniques. For instance, we introduced a new feature named “Pulse_Rate,” derived from the relationship between Systolic and Diastolic blood pressures. Additionally, we calculated the “BMI” using the “height” and “weight” parameters. These transformations were discussed in detail in Section 3.3 titled ‘Data Construction’.

Furthermore, we will employ the “skewness” method to compute the skewness values and print out the value for each feature and also visualise in distribution chart using “seaborn” and “matplotlib” libraries, as displayed in Figure-4.6 and Figure-4.7.

```
In [111]:
tmpDf = merge_df.drop("cardio")
feature_cols = tmpDf.columns

# Compute and categorize skewness for each column
print("\nSkewness Values")
print("-------------------------------------")
for column in feature_cols:
    skew_value = merge_df.select(skewness(column)).collect()[0][0]
    print(f"Skewness value for {column}: {skew_value:.4f}")


Skewness Values
-------------------------------------
Skewness value for age: -0.3042
Skewness value for weight: 1.0454
Skewness value for ap_hi: 0.9155
Skewness value for ap_lo: 0.6647
Skewness value for cholesterol: 1.5947
Skewness value for gluc: 2.3981
Skewness value for BMI: 1.2111
Skewness value for Pulse_Rate: 0.5382
```

Figure-4.6: Compute the skewness value and print out the value for each feature.

```
In [52]:
# Covert to Pandas
tmpDf_pd = tmpDf.toPandas()

# Setting up the subplot grid
num_features = len(feature_cols)
num_rows = (num_features + 2) // 3  # Ceiling division to ensure enough rows

plt.figure(figsize=(15, 5 * num_rows))
for idx, column in enumerate(feature_cols):
    # Setting the position for the subplot
    plt.subplot(num_rows, 3, idx+1)

    # Plotting distribution
    sns.histplot(tmpDf_pd[column], kde=True)

    # Computing and displaying skewness
    skew_value = merge_df.select(skewness(column)).collect()[0][0]
    plt.title(f"Distribution of {column} - Skewness: {skew_value:.4f}")

plt.tight_layout()  # Adjusting layout for better visualization
plt.show()
```

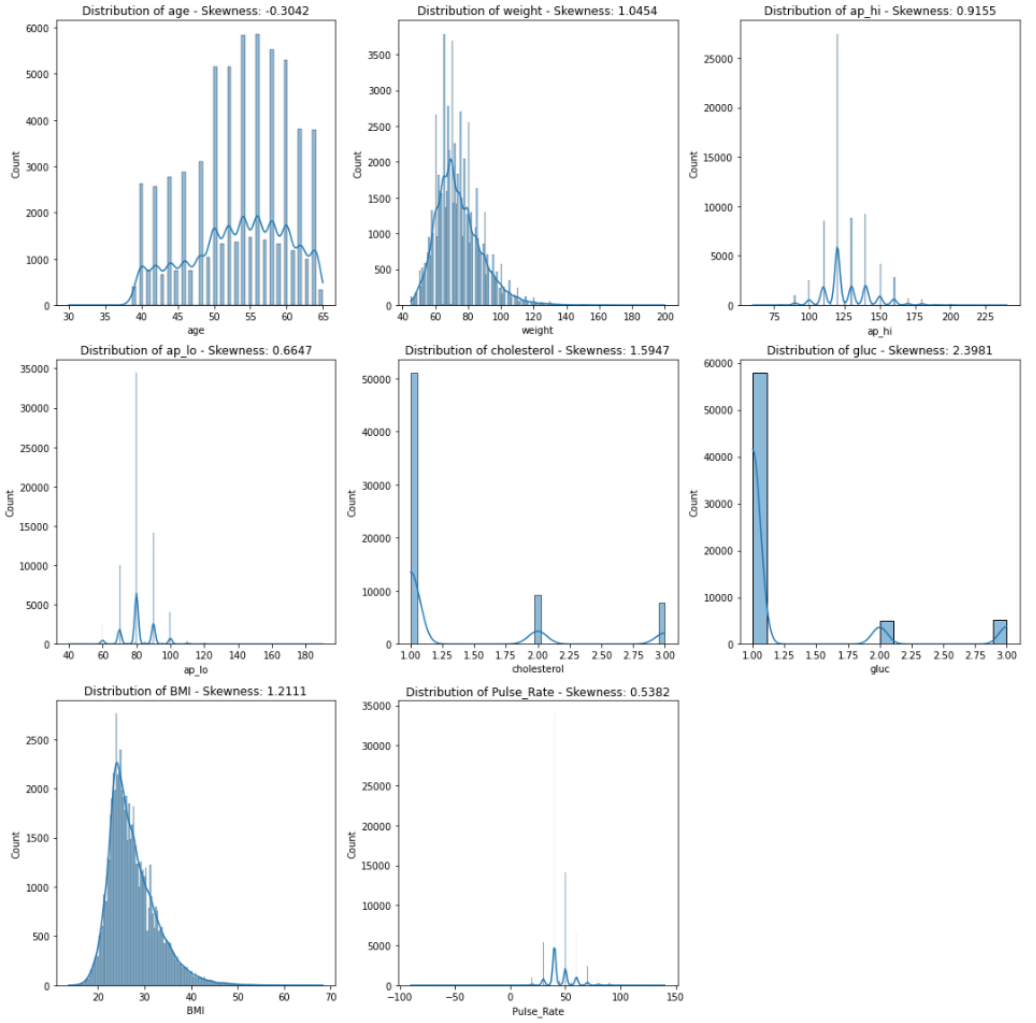


Figure-4.7: To visualise the distribution of each features

As observed in Figure-4.6 and Figure-4.7, the “gluc” feature displays a skewness value of 2.3981, which slightly falls outside the acceptable range that suggested by IBM SPSS Documentation[4]. While the other features remain within the acceptable range, the noted right-skewness of “gluc” requires rectification. We will apply an inverse transformation method using the “withColumn” method to “gluc” and assess the resulting skewness value and visualise the distribution chart, as shown in Figure-4.8 and Figure-4.9.

[4] https://www.ibm.com/docs/en/spss-statistics/29.0.0?topic=summarize-statistics

```
In [112]:
# Applying Inverse Transformation to 'gluc' column
merge_df = merge_df.withColumn("gluc", 1/merge_df["gluc"])

# Verify the transformed data
skew_value = merge_df.select(skewness("gluc")).collect()[0][0]
print(f"Skewness value for gluc: {skew_value:.4f}")

Skewness value for gluc: -2.0554
```

Figure-4.8: Applying the Inverse Transformation on "gluc" column and verify the skewness value.

```
In [113]:
## Plotting the distribution for "BMI" feature
plt.figure(figsize=(8, 6))
sns.histplot(tmpDf_pd['gluc'], kde=True)
skew_value = merge_df.select(skewness('gluc')).collect()[0][0]
plt.title(f"Distribution of {'gluc'} - Skewness: {skew_value:.4f}")
plt.show()
```

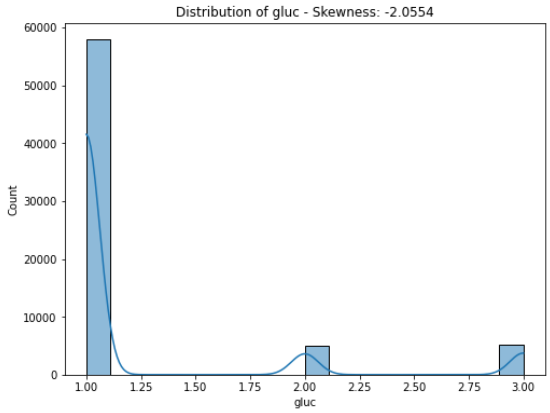


Figure-4.9: Visualise the distribution chart of "gluc"

As shown in Figure-4.8 and Figure-4.9, the skewness of the "gluc" distribution appears to reduce effectively "-2.06" which is near to the range.

5. Data-Mining Method(s) Selection

5.1. Discussion of Data Mining Methods in Context of Data Mining Objectives

We consider several factors when selecting the best model, such as the types of data we can use, our data analysis goals, the specific modelling requirements, and the criteria we use to pick models. According to PySpark's Machine Learning Library (MLlib) Guide[5], there are different categories of modelling methods:

- ***Classification:*** It is a supervised approach used to categorize or label data points into distinct classes or groups
- ***Regression:*** A statistical approach is used to predict a continuous or quantitative output based on one or more input features.
- ***Clustering:*** An unsupervised approach is used to group data points with similar characteristics or features into clusters or segments.
- ***Collaborative Filtering:*** It is a method used primarily in recommendation systems.

For our study, we are aiming to predict outcomes related to CVD. It means we use a supervised method, leading us to focus on classification modelling methods. Within this category, tree-based algorithms are particularly interesting to us, namely the Decision Tree Classifier, Random Forest

[5] https://spark.apache.org/docs/latest/ml-guide.html

Classifier, and the Gradient-Boosted Tree Classifier. In the subsequent section, we will provide a detailed assessment of these algorithms.

5.1.1. Data Type Accessible for Data Mining

From our earlier sections on "Data Understanding" and "Data Processing," we have developed a comprehensive grasp of our dataset. Our primary focus is on the target field, distinguishing individuals into two categories: those with cardiovascular disease ("cardio" = 1) and those without ("cardio" = 0). Regression methods are unsuitable because the categories and labels are predefined, and our aim is not numeric prediction. With our dataset containing over 6,000 entries and featuring a mix of numeric and categorical data types, our direction for data mining is clear.

5.1.2. Data Mining Goals/Objectives

Its data mining goals shape the methods we employ in this project:

- We aim to investigate how lifestyle decisions and environmental elements influence cardiovascular disease (CVD) likelihood. We will analyse factors such as age, gender, and smoking habits across various individuals.
- We will also assess an individual's physical activity, blood pressure, cholesterol levels, and the risk of developing cardiovascular disease.

By focusing on these aspects, we aim to unravel the connections between lifestyle, environment, and the potential for CVD. This analysis will contribute to a comprehensive understanding of risk factors and aid in developing prevention and early intervention strategies.

5.1.3. Data Mining Modelling Success Criteria

Tree-based algorithms are efficient and intuitive, making them highly desirable for many data scenarios. They can be trained quickly, are computationally efficient, and their outcomes are often accessible for businesses to understand. PySpark's ML Algorithm offers three key tree-based classifier models: the Decision Tree Classifier, the Random Forest Classifier, and the Gradient-Boosted Tree Classifier. While these algorithms share a foundational approach, it's essential to recognise their distinctions:

- ***Decision Tree Classifier*** [6]***:*** Represented by a singular tree, this model is straightforward. However, its simplicity can sometimes lead to overfitting, especially when the tree is exceedingly detailed.
- ***Random Forest Classifier*** [7]***:*** This ensemble technique employs multiple trees for predictions. By averaging outcomes across various trees, it trims overfitting. But it sacrifices some of the transparency inherent to a singular decision tree.
- ***Gradient-Boosted Tree Classifier*** [8]***:*** Unique in its approach, this classifier builds trees in succession, each addressing the errors of its predecessor. This method can achieve remarkable accuracy with tuning, though it demands careful configuration to avoid overfitting.

These tree-based models are especially adept at handling large datasets and are relatively unfazed by occasional missing data. Their straightforward rules resonate well in business settings, facilitating easy comprehension and often enhancing classification accuracy.

[6] https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-classifier
[7] https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier
[8] https://spark.apache.org/docs/latest/ml-classification-regression.html#gradient-boosted-tree-classifier

5.2. Select the Appropriate Data Mining Methods Based on Discussion.

In the previous section, we discovered that we are dealing with a classification problem. We found a set of data that works well for what we need and meets the rules for making models. Now, we will split this data into two parts: one for training and one for testing. It will help us see which modelling method works the best. We will try different ways of splitting the data to avoid overfitting or underfitting. We will also test different ratios to find what works best. We have picked three tree-based methods for a reason:

- ***Decision Tree Classifier:*** It is straightforward but can sometimes get too specific.
- ***Random Forest Classifier:*** It combines multiple trees, giving a more balanced view.
- ***Gradient-Boosted Tree Classifier:*** It builds trees step-by-step, aiming to reduce mistakes each time.

These methods are known to handle our data effectively. We will dive deeper into them in the next section.

6. Data Mining Algorithm(s) Selection

6.1. Conduct Exploratory Analysis and Discuss

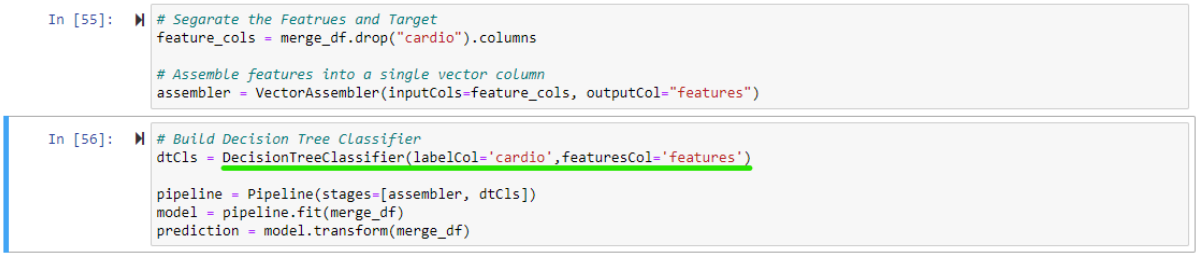
In this section, we will conduct exploratory analysis and model selection as part of our data mining process. We will focus on identifying appropriate data-mining algorithms and techniques to reveal significant trends within the dataset with the following step:

- **Finding Important Features**: First, we will explore the data to determine which features are most important for our target, "cardio". We will utilise the "Feature Selection" techniques to determine the importance of various features to the target variable, as detailed in Section 4.
- **Using Selected Three Algorithms**: Once we know the essential features, we will use the selected three algorithms (Decision Tree Classifier, Random Forest Classifier, and Gradient-Boosted Tree Classifier) for data mining. These algorithms will help us understand how the input features relate to the target field.

Our study is to figure out what's essential in the data input and ensure the choice of method. To achieve this, we will explore it in the following sections.

6.1.1. 1st Data Mining Objective: Decision Tree Classifier Algorithm

As illustrated in Figure-6.1, we use the "Decision Tree Classifier" model with default parameters. After we built the model, we will evaluate the model with a few methods such as printing classification report, confusion matrix, and feature importance. Before this, we prepared our data into features and targets. We have also made some custom functions to print the classification report, confusion matrix, and feature importance, which we will use for other models that the customise functions can be found after importing libraries section at Jupyter Notebook.



```
In [55]:
# Segarate the Featrues and Target
feature_cols = merge_df.drop("cardio").columns

# Assemble features into a single vector column
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

```
In [56]:
# Build Decision Tree Classifier
dtCls = DecisionTreeClassifier(labelCol='cardio',featuresCol='features')

pipeline = Pipeline(stages=[assembler, dtCls])
model = pipeline.fit(merge_df)
prediction = model.transform(merge_df)
```

Figure-6.1: Build "Decision Tree Classifier" model.

In Figure-6.2, the model evaluation process is described. We employ the Multiclass Classification Evaluator to assess the accuracy rate, setting “metricName” to “accuracy”, which we have 73% accuracy rate. For evaluating the Area Under the Receiver Operating Characteristic curve (ROC), we use the Binary Classification Evaluator method with “metricName” set to “areaUnderROC”, which got 59%. Similarly, to measure the Area Under the Precision-Recall curve (PR), the “metricName” is set to “areaUnderPR” within the binary classification evaluator, which has 67%.

```
In [145]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nDecision Tree Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nDecision Tree Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nDecision Tree Classifier - Area Under PR = {pr_auc:.2f}")

Decision Tree Classifier - Accuracy = 0.73

Decision Tree Classifier - Area Under ROC (AUC) = 0.59

Decision Tree Classifier - Area Under PR = 0.67
```

Figure-6.2: Evaluate the Model’s Accuracy, Area Under ROC and Area Under PR.

As illustrated in Figure-6.3’s Classification Report and Figure-6.4’s Confusion Matrix for the Decision Tree Classifier model, the accuracy is 73%. The precision for non-CVD cases is 71%, while for CVD cases, it is 76%. The recall rates are 80% and 66% for non-CVD and CVD cases, respectively.

When evaluating the confusion matrix, the model accurately predicted 27,392 out of 34,363 non-CVD cases. In contrast, it correctly classified 22,450 out of 33,763 CVD cases. However, the model misidentified 6,971 non-CVD cases as CVD and mistakenly recognized 11,313 CVD cases as non-CVD.

```
In [60]:
# Generate Classification report
print("\nDecision Tree Classifier - Classification Report: ")
print("-------------------------------------------------------")
classification_report(prediction)
print("-------------------------------------------------------")

Decision Tree Classifier - Classification Report:
-------------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.79    0.73       34363
           1  0.76         0.68    0.73       33763
    accuracy                         0.73       68126
   macro avg  0.73         0.73    0.73       68126
weighted avg  0.73         0.73    0.73       68126
-------------------------------------------------------
```

Figure-6.3: Classification Report

```
In [61]:
# Generate Confusion Matrix
print("\nDecision Tree Classifier - Confusion Matrix: ")
print("-------------------------------------------------------")
confusion_matrix(prediction)
print("-------------------------------------------------------")

Decision Tree Classifier - Confusion Matrix:
-------------------------------------------------------
TN: 26980       FP: 7383
FN: 10938       TP: 22825
-------------------------------------------------------
```

Figure-6.4: Confusion Matrix

As shown in Figure-6.5, the “Predictor Importance”, We can see that “ap_hi” is the highest importance and “ap_lo” is the least. “age” and “cholesterol” are followed as second and third important.

```
In [62]:
# Generate Predictor Feature Importance
print("\nDecision Tree Classifier - Predictor Feature Importance: ")
print("-------------------------------------------------------")
feature_importance(model.stages[-1], feature_cols)
print("-------------------------------------------------------")
```

```
Decision Tree Classifier - Predictor Feature Importance:
-------------------------------------------------------
Feature: ap_hi, Importance: 0.8015
Feature: age, Importance: 0.1111
Feature: cholesterol, Importance: 0.0754
Feature: gluc, Importance: 0.0050
Feature: Pulse_Rate, Importance: 0.0042
Feature: BMI, Importance: 0.0029
Feature: weight, Importance: 0.0000
Feature: ap_lo, Importance: 0.0000
-------------------------------------------------------
```

Figure-6.5: Predictor Feature Importance

6.1.2. 2nd Data Mining Objective: Random Forest Classifier Algorithm

As shown in Figure-6.6, we constructed the "Random Forest Classifier" model using its default settings. We then evaluated this model using the same process as the "Decision Tree Classifier".

```
In [63]:
# Build Random Forest Classifier
rfCls = RandomForestClassifier(labelCol='cardio',featuresCol='features')

pipeline = Pipeline(stages=[assembler, rfCls])
model = pipeline.fit(merge_df)
prediction = model.transform(merge_df)
```

Figure-6.5: Construct "Random Forest Classifier" model.

In Figure-6.6 outlines the model evaluation process for the Random Forest Classifier. Using the Multiclass Classification Evaluator, we assessed the accuracy rate, achieving a rate of 73%. When evaluating the Area Under the Receiver Operating Characteristic curve (ROC), the Binary Classification Evaluator indicated an AUC of 79%. Additionally, for the Area Under the Precision-Recall curve (PR), the evaluation yielded a result of 78%.

```
In [173]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under PR = {pr_auc:.2f}")
```

```
Random Forest Classifier - Accuracy = 0.73

Random Forest Classifier - Area Under ROC (AUC) = 0.79

Random Forest Classifier - Area Under PR = 0.78
```

Figure-6.6: Evaluate the Model's Accuracy, Area Under ROC and Area Under PR.

According to the Classification Report in Figure-6.7, the Random Forest Classifier achieved an overall accuracy of 73%. The model displayed a precision of 70% in predicting non-CVD cases and a notably higher precision of 76% for CVD cases. The recall values suggest that 79% of non-CVD and 66% of CVD cases were correctly identified. As further detailed in Figure-6.8's confusion matrix, out of the 34,363 non-CVD cases, 27,169 were accurately predicted. Meanwhile, of the 33,763 CVD cases, 22,308 were correctly classified. However, the model misclassified 7,196 non-CVD cases and 11,455 CVD cases.

```
In [65]:
# Generate Classification report
print("\nRandom Forest Classifier - Classification Report: ")
print("-----------------------------------------------------")
classification_report(prediction)
print("-----------------------------------------------------")
```

```
Random Forest Classifier - Classification Report:
-----------------------------------------------------

              precision    recall  f1-score   support
           0  0.70         0.79    0.73       34363
           1  0.76         0.66    0.73       33763
    accuracy                         0.73       68126
   macro avg  0.73         0.73    0.73       68126
weighted avg  0.73         0.73    0.73       68126
-----------------------------------------------------
```

Figure-6.7: Classification Report

```
In [66]:
# Generate Confusion Matrix
print("\nRandom Forest Classifier - Confusion Matrix: ")
print("-----------------------------------------------------")
confusion_matrix(prediction)
print("-----------------------------------------------------")
```

```
Random Forest Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 27167       FP: 7196
FN: 11455       TP: 22308
-----------------------------------------------------
```

Figure-6.8: Confusion Matrix

As shown in Figure-6.9, the "Predictor Importance", We can see that "ap_hi" is the highest importance and "gluc" is the least. "ap_lo" and "age" are followed as second and third important.

```
In [67]:
# Generate Predictor Feature Importance
print("\nRandom Forest Classifier - Predictor Feature Importance: ")
print("-----------------------------------------------------")
feature_importance(model.stages[-1], feature_cols)
print("-----------------------------------------------------")
```

```
Random Forest Classifier - Predictor Feature Importance:
-----------------------------------------------------
Feature: ap_hi, Importance: 0.3956
Feature: Pulse_Rate, Importance: 0.2100
Feature: ap_lo, Importance: 0.2090
Feature: age, Importance: 0.0942
Feature: cholesterol, Importance: 0.0733
Feature: BMI, Importance: 0.0133
Feature: weight, Importance: 0.0024
Feature: gluc, Importance: 0.0021
-----------------------------------------------------
```

Figure-6.8: Predictor Importance.

3rd Data Mining Objective: Gradient-Boosted Tree Classifier Algorithm

As shown in Figure-6.6, we built the "Gradient-Boosted Tree Classifier" model using its default parameters. Subsequently, we assessed this model following the same procedure used for the "Decision Tree Classifier" and "Random Forest Classifier".

```
In [246]:
# Build Gradient-Boosted Tree Classifier
gbtCls = GBTClassifier(labelCol='cardio',featuresCol='features')

pipeline = Pipeline(stages=[assembler, gbtCls])
model = pipeline.fit(merge_df)
prediction = model.transform(merge_df)
```

Figure-6.9: Built Gradient-Boosted Tree Classifier

Figure-6.1- outlines the model evaluation procedure for the Gradient-Boosted Tree Classifier. We used the Multiclass Classification Evaluator to determine the accuracy rate, which stood at 74%. In assessing the Area Under the Receiver Operating Characteristic curve (ROC), the Binary Classification Evaluator presented an AUC of 80%. Furthermore, the evaluation for the Area Under the Precision-Recall curve (PR) revealed a result of 79%.

```
In [247]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under PR = {pr_auc:.2f}")

Gradient-Boosted Tree Classifier - Accuracy = 0.74

Gradient-Boosted Tree Classifier - Area Under ROC (AUC) = 0.80

Gradient-Boosted Tree Classifier - Area Under PR = 0.79
```

Figure-6.10: Evaluate the Model's Accuracy, Area Under ROC and Area Under PR.

The Classification Report for the Gradient-Boosted Tree Classifier, displayed in Figure-6.11, indicates an overall accuracy of 74%. The model predicted non-CVD cases with a precision of 72% and achieved a recall rate of 78%. For CVD cases, the precision was 76%, and the recall stood at 69%. The confusion matrix, presented in Figure-6.12, provides additional insights: of the 34,363 non-CVD instances, 26,888 were correctly classified. Meanwhile, from the 26,888 CVD cases, 23,208 were accurately identified. However, the model misclassified 7,475 non-CVD and 10,555 CVD cases.

```
In [251]:
# Generate Classification report
print("\nGradient-Boosted Tree Classifier - Classification Report: ")
print("-----------------------------------------------------")
classification_report(prediction)
print("-----------------------------------------------------")

Gradient-Boosted Tree Classifier - Classification Report:
-----------------------------------------------------

              precision    recall  f1-score   support
           0  0.72         0.78    0.73       34363
           1  0.76         0.69    0.73       33763
    accuracy                       0.74       68126
   macro avg  0.74         0.73    0.73       68126
weighted avg  0.74         0.74    0.73       68126
-----------------------------------------------------
```

Figure-6.11: Classification Report

```
In [71]:
# Generate Confusion Matrix
print("\nGradient-Boosted Tree Classifier - Confusion Matrix: ")
print("-----------------------------------------------------")
confusion_matrix(prediction)
print("-----------------------------------------------------")

Gradient-Boosted Tree Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 26888       FP: 7475
FN: 10555       TP: 23208
-----------------------------------------------------
```

Figure-6.12: Confusion Matrix

As shown in Figure-6.13, the "Predictor Feature Importance" method analysis indicates that "ap_hi" is the most important predictor. This is followed closely by "age" and "BMI", which rank second and third in importance, respectively.

```
In [72]:  # Generate Predictor Feature Importance
          print("\nGradient-Boosted Tree Classifier - Predictor Feature Importance: ")
          print("-----------------------------------------------------")
          feature_importance(model.stages[-1], feature_cols)
          print("-----------------------------------------------------")

          Gradient-Boosted Tree Classifier - Predictor Feature Importance:
          -----------------------------------------------------
          Feature: ap_hi, Importance: 0.4909
          Feature: age, Importance: 0.1676
          Feature: BMI, Importance: 0.0859
          Feature: cholesterol, Importance: 0.0771
          Feature: ap_lo, Importance: 0.0733
          Feature: weight, Importance: 0.0663
          Feature: Pulse_Rate, Importance: 0.0200
          Feature: gluc, Importance: 0.0189
          -----------------------------------------------------
```

Figure-6.13: Predictor Importance

6.2. Select Data Mining Algorithms Based on Discussion

In the earlier sections of our data mining, we tried three algorithms with their default model-building parameters to measure initial performance. The results in Table-6.1 reveal significant insights regarding each algorithm's proficiency in handling our dataset.

S/N	Data Mining Algorithms	Accuracy	AUC
1.	Gradient-Boosted Tree Classifier	74%	0.80
2.	Random Forest Classifier	73%	0.79
3.	Decision Tree Classifier	73%	0.59

Table-6.1: Compare Algorithms

Analysing the table:

1. ***Gradient-Boosted Tree Classifier:*** This algorithm emerged as the top performer with an accuracy of 74%. More importantly, its AUC of 0.80, a critical metric indicating its ability to distinguish between the positive and negative classes, is commendable.
2. ***Random Forest Classifier:*** A close second, the Random Forest algorithm showcased an accuracy of 73%. Its AUC of 0.79 demonstrates its reliability in class distinction, making it another robust choice.
3. ***Decision Tree Classifier:*** While the Decision Tree achieved an accuracy like the Random Forest, its AUC lags at 0.59. While its general classification might be on par, its ability to differentiate between classes isn't as robust as the other two.

Accuracy alone doesn't help to select the algorithms. The AUC, which measures the true positive rate against the false positive rate, provides a more holistic view of an algorithm's performance, especially in datasets where class balance might be an issue. Given the results, the "Gradient-Boosted Tree Classifier" and the "Random Forest Classifier" stand out as our top picks. Their balanced precision, recall, and superior AUC scores indicate their strong capability to differentiate between CVD and non-CVD cases.

Our exploratory analysis makes the decision clear: the "Gradient-Boosted Tree Classifier" and "Random Forest Classifier" stand out. Their superior accuracy, outstanding AUC scores, and sharp discernment make them the top choices, perfectly tailored for insightful, data-driven decisions.

6.3. Build/Select Model with Algorithm/Model Parameters(s)

We intend to develop models using our selected algorithms: "Gradient-Boosted Tree Classifier" and "Random Forest Classifier" Our approach consists of the following steps:

- Firstly, we will segregate our dataset into training and test sets utilising the “random” function as shown in Figure-7.1. The process of this step will be explored in Section 7.1.
- After partitioning the data, we will utilise the “Gradient-Boosted Tree Classifier” and “Random Forest Classifier” algorithms to develop our predictive models based on the training set which we will address in following Section-7.2.
- For the hyperparameter tuning of these models, we will manually adjust various parameters and assess their performance using the Binary Classification Evaluator. We’ll compare the outcomes and determine the best parameter configurations for both algorithms. Specifically, we’ll adjust the “maxDepths” and “stepSize” for the Gradient-Boosted Tree Classifier, and for the Random Forest Classifier, we’ll fine-tune the “maxDepths” and “maxBins”. After this tuning, the best parameters identified for the Gradient-Boosted Tree Classifier are “maxDepths = 5” and “stepSize = 0.1”. The optimal parameters for the Random Forest Classifier are “maxDepths = 10” and “maxBins = 64”, as illustrated in Figure-6.14 and Figure-6.15.

```
In [255]:
# Gradient-Boosted Tree Classifier Hyperparameter tuning
maxDepths = [3, 5, 7]
learningRates = [0.01, 0.05, 0.1]
combinations = [(depth, rate) for depth in maxDepths for rate in learningRates]

evaluator = BinaryClassificationEvaluator(labelCol="cardio")

best_metric = float('-inf')
best_model = None
best_params = (None, None)

for depth, rate in combinations:
    cls = GBTClassifier(labelCol='cardio', featuresCol='features', maxDepth=depth, stepSize=rate)
    pipeline = Pipeline(stages=[assembler, cls])
    model = pipeline.fit(train_data)
    prediction = model.transform(test_data)
    metric = evaluator.evaluate(prediction)

    if metric > best_metric:
        best_metric = metric
        best_model = model
        best_params = (depth, rate)

print(f"Best Parameters: Max Depth = {best_params[0]}, Step Size = {best_params[1]}")

Best Parameters: Max Depth = 5, Learning Rate = 0.1
```

Figure-6.13: Hyperparameters tuning for Gradient-Boosted Tree Classifier

```
In [256]:
# Random Forest Classifier Hyperparameter tuning
maxDepths = [5, 10, 15]
maxBins = [32, 64]

combinations = [(depth, bins) for depth in maxDepths for bins in maxBins]

evaluator = BinaryClassificationEvaluator(labelCol="cardio")

best_metric = float('-inf')
best_model = None
best_params = (None, None, None)

for depth, bins in combinations:
    cls = RandomForestClassifier(labelCol='cardio', featuresCol='features', maxDepth=depth, maxBins=bins)
    pipeline = Pipeline(stages=[assembler, cls])
    model = pipeline.fit(train_data)
    prediction = model.transform(test_data)
    metric = evaluator.evaluate(prediction)

    if metric > best_metric:
        best_metric = metric
        best_model = model
        best_params = (depth, bins)

print(f"Best Parameters: Max Depth = {best_params[0]}, Max Bins = {best_params[1]}")

23/10/05 23:47:31 WARN DAGScheduler: Broadcasting large task binary with size 1268.4 KiB
23/10/05 23:47:32 WARN DAGScheduler: Broadcasting large task binary with size 2.0 MiB
23/10/05 23:47:35 WARN DAGScheduler: Broadcasting large task binary with size 1352.0 KiB
23/10/05 23:47:41 WARN DAGScheduler: Broadcasting large task binary with size 1231.1 KiB
23/10/05 23:48:29 WARN DAGScheduler: Broadcasting large task binary with size 6.1 MiB

Best Parameters: Max Depth = 10, Max Bins = 64
```

Figure-6.14: Hyperparameters tuning for Random Forest Classifier

- After the models' creation, we will evaluate and analysis the built model with test data with various ways such as classification report, confusion matrix, predictor feature importance, and tree model structure, which we will address in following Section-7

To summarise, we divide our data into training and test datasets, build a model from the training set using the best parameters, and evaluate each model with the test data. The detailed procedure will be addressed in following Section 7.

7. Data Mining

7.1. Creating Logical Test(s)

As illustrated in Figure-7.1, we will use the "randomSplit" method to split the entire dataset into two parts randomly: a training set, which is used to build the model and a test set, which is used to evaluate the performance. We are breaking 70% as training partition size and 30% as testing partition size to avoid overfitting.

```
In [253]:
# Split the data into training and test sets (70% train, 30% test)
train_data, test_data = merge_df.randomSplit([0.7, 0.3])
```

Figure-7.1: Splitting Datasets into "training" and "test"

7.2. Conducting Data Mining (The model must run)

After partitioning the data, we will construct models using two selected algorithms: the "K-Neighbours Classifier" and the "Gaussian Naive Bayes". Subsequently, in the following sections, we will explore both and run models with best hyperparameter and compare the results.

7.2.1. Gradient-Boosted Tree Classifier Model

As shown in Figure-7.2, we will construct the "Gradient-Boosted Tree Classifier" model using the best hyperparameter "maxDepths = 5" and "stepSize = 0.1". After constructing the model, we will evaluate the model with test data using the reports and analysis the Tree Model how it is structured the tree, as illustrated in Figures-7.3, 7.4, 7.5, 7.6 and 7.7.

```
In [258]:
# Build the Gradient-Boosted Tree Classifier with best hyperparameter
gbtCls = GBTClassifier(labelCol='cardio', featuresCol='features', maxDepth=5, stepSize=0.1)
pipeline = Pipeline(stages=[assembler, rfCls])
model = pipeline.fit(train_data)
prediction = model.transform(test_data)
metric = evaluator.evaluate(prediction)
```

Figure-7.2: Building the "Gradient-Boosted Tree Classifier" Model

```
In [259]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under PR = {pr_auc:.2f}")

Gradient-Boosted Tree Classifier - Accuracy = 0.72

Gradient-Boosted Tree Classifier - Area Under ROC (AUC) = 0.79

Gradient-Boosted Tree Classifier - Area Under PR = 0.78
```

Figure-7.3: Evaluate the Model's Accuracy, Area Under ROC and Area Under PR

```
In [79]:
# Generate Classification report
print("\nGradient-Boosted Tree Classifier - Classification Report: ")
print("-----------------------------------------------------")
classification_report(prediction)
print("-----------------------------------------------------")
```

```
Gradient-Boosted Tree Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.70         0.79    0.72       10388
           1  0.75         0.64    0.72       10055
    accuracy                        0.72       20443
   macro avg  0.73         0.72    0.72       20443
weighted avg  0.72         0.72    0.72       20443
-----------------------------------------------------
```

Figure-7.4: Classification Report

```
In [80]:
# Generate Confusion Matrix
print("\nGradient-Boosted Tree Classifier - Confusion Matrix: ")
print("-----------------------------------------------------")
confusion_matrix(prediction)
print("-----------------------------------------------------")
```

```
Gradient-Boosted Tree Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 8251        FP: 2137
FN: 3570        TP: 6485
-----------------------------------------------------
```

Figure-7.5: Confusion Matrix

```
In [81]:
# Generate Predictor Feature Importance
print("\nGradient-Boosted Tree Classifier - Predictor Feature Importance: ")
print("-----------------------------------------------------")
feature_importance(model.stages[-1], feature_cols)
print("-----------------------------------------------------")
```

```
Gradient-Boosted Tree Classifier - Predictor Feature Importance:
-----------------------------------------------------
Feature: ap_hi, Importance: 0.4058
Feature: Pulse_Rate, Importance: 0.2328
Feature: ap_lo, Importance: 0.1771
Feature: age, Importance: 0.1021
Feature: cholesterol, Importance: 0.0632
Feature: BMI, Importance: 0.0138
Feature: weight, Importance: 0.0026
Feature: gluc, Importance: 0.0026
-----------------------------------------------------
```

Figure-7.6: Predictor Feature Importance

7.2.2. Random Forest Classifier Model

In Figure-7.7, we will configure the “Random Forest Classifier” model using the optimal hyperparameters: “maxDepth = 10” and “maxBins = 64”. After this setup, we will assess its performance on the test data and examine the tree’s structure, as illustrated in Figures 7.8, 7.9, 7.10, and 7.11.

```
In [82]:
# Build the Random Forest Classifier with best hyperparameter
rfCls = RandomForestClassifier(labelCol='cardio', featuresCol='features', maxDepth=10, maxBins=64)
pipeline = Pipeline(stages=[assembler, rfCls])
model = pipeline.fit(train_data)
prediction = model.transform(test_data)
metric = evaluator.evaluate(prediction)
```

Figure-7.7: Building the “Random Forest Classifier” Model

```
In [83]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under PR = {pr_auc:.2f}")

Random Forest Classifier - Accuracy = 0.73

Random Forest Classifier - Area Under ROC (AUC) = 0.79

Random Forest Classifier - Area Under PR = 0.77
```

Figure-7.8: Evaluate the Model's Accuracy, Area Under ROC and Area Under PR

```
In [84]:
# Generate Classification report
print("\nRandom Forest Classifier - Classification Report: ")
print("-------------------------------------------------------")
classification_report(prediction)
print("-------------------------------------------------------")

Random Forest Classifier - Classification Report:
-------------------------------------------------------

              precision    recall  f1-score   support
           0  0.71         0.77    0.73       10388
           1  0.74         0.68    0.73       10055
    accuracy                        0.73       20443
   macro avg  0.73         0.73    0.73       20443
weighted avg  0.73         0.73    0.73       20443
-------------------------------------------------------
```

Figure-7.9: Classification Report

```
In [85]:
# Generate Confusion Matrix
print("\nRandom Forest Classifier - Confusion Matrix: ")
print("-------------------------------------------------------")
confusion_matrix(prediction)
print("-------------------------------------------------------")

Random Forest Classifier - Confusion Matrix:
-------------------------------------------------------
TN: 8026        FP: 2362
FN: 3228        TP: 6827
-------------------------------------------------------
```

Figure-7.10: Confusion Matrix

```
In [86]:
# Generate Predictor Feature Importance
print("\nRandom Forest Classifier - Predictor Feature Importance: ")
print("-------------------------------------------------------")
feature_importance(model.stages[-1], feature_cols)
print("-------------------------------------------------------")

Random Forest Classifier - Predictor Feature Importance:
-------------------------------------------------------
Feature: ap_hi, Importance: 0.3428
Feature: Pulse_Rate, Importance: 0.1903
Feature: ap_lo, Importance: 0.1575
Feature: age, Importance: 0.1322
Feature: cholesterol, Importance: 0.0752
Feature: BMI, Importance: 0.0514
Feature: weight, Importance: 0.0379
Feature: gluc, Importance: 0.0127
-------------------------------------------------------
```

Figure-7.11: Predictor Feature Importance

7.3. Searching for Patterns

After our previous section, where we observed the model being effectively trained and tested, we now want to find patterns using different methods to understand its predictions. These methods are:

- ***Classification Report Examination:*** We will delve into the Classification Report, offering insights into the nuances of the model's predictive capabilities.
- ***Performance Assessment via Confusion Matrix:*** Using the Confusion Matrix, we'll measure and dissect the performance metrics of our model.
- ***Precision-Recall and ROC Analysis:*** We seek a complete understanding of our model's precision and recall by analysing the Precision-Recall and ROC values.
- ***Input Features and Prediction Relationship:*** Our goal is to discern the underlying relationships between the model's input features and its predictions, clarifying influential factors.

7.3.1. Classification Report Examination

We developed a custom function, shown in Figure-7.12, that utilise the "MulticlassClassificationEvaluator" to evaluate and generate the reports. This function allows us to generate a report and examine the accuracy patterns of our model. The generated report for both models can be viewed in Figure-7.13 and Figure-7.14. We will examine both models' predictions.

```
In [93]:
# Classification Report Method
def classification_report(predictions, label_col="cardio"):
    # Create evaluators
    evaluator = MulticlassClassificationEvaluator(labelCol=label_col, predictionCol="prediction")

    # Compute metrics
    precision_0 = evaluator.evaluate(predictions, {evaluator.metricName: "precisionByLabel", evaluator.metricLabel: 0})
    recall_0 = evaluator.evaluate(predictions, {evaluator.metricName: "recallByLabel", evaluator.metricLabel: 0})
    f1_0 = evaluator.evaluate(predictions, {evaluator.metricName: "f1", evaluator.metricLabel: 0})

    precision_1 = evaluator.evaluate(predictions, {evaluator.metricName: "precisionByLabel", evaluator.metricLabel: 1})
    recall_1 = evaluator.evaluate(predictions, {evaluator.metricName: "recallByLabel", evaluator.metricLabel: 1})
    f1_1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1", evaluator.metricLabel: 1})

    accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})

    # Compute support (number of occurrences of each class)
    support_0 = predictions.filter(predictions[label_col] == 0).count()
    support_1 = predictions.filter(predictions[label_col] == 1).count()

    # Print report
    print('              precision    recall  f1-score   support')
    print(f'           0  {precision_0:.2f}         {recall_0:.2f}    {f1_0:.2f}       {support_0}')
    print(f'           1  {precision_1:.2f}         {recall_1:.2f}    {f1_1:.2f}       {support_1}')
    print(f'    accuracy                       {accuracy:.2f}       {support_0 + support_1}')
    print(f'   macro avg  {(precision_0 + precision_1) / 2:.2f}         {(recall_0 + recall_1) / 2:.2f}    {(f1_0 + f1_1) / 2
    print(f'weighted avg  {(precision_0 * support_0 + precision_1 * support_1) / (support_0 + support_1):.2f}         {(recal
```

Figure-7.12: Customise Function of Classification Report

```
Gradient-Boosted Tree Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.70         0.80    0.73       10335
           1  0.76         0.66    0.73       10126
    accuracy                        0.73       20461
   macro avg  0.73         0.73    0.73       20461
weighted avg  0.73         0.73    0.73       20461
-----------------------------------------------------
```

Figure-7.13: "Gradient-Boosted Tree Classifier" Model's Classification Report

```
Random Forest Classifier - Classification Report:
-------------------------------------------------------
              precision    recall  f1-score   support
           0       0.72      0.78      0.73     10335
           1       0.75      0.69      0.73     10126
    accuracy                           0.73     20461
   macro avg       0.73      0.73      0.73     20461
weighted avg       0.73      0.73      0.73     20461
-------------------------------------------------------
```

Figure-7.14: “Random Forest Classifier” Model’s Classification Report

7.3.2. Performance Assessment via Confusion Matrix

We have developed the customise method to extract matrix data from model prediction. And then we will print out the extracted result as illustrated in Figure-7.15. Both Model’s Confusion Matrix results can be viewed in Figure-7.16 and Figure-7.17. We will assess both models based on the confusion matrix.

```
In [94]:
# Confusion Matrix Method
def confusion_matrix(predictions, label_col="cardio", prediction_col="prediction"):
    # Count combinations of label and prediction
    confusion_matrix = predictions.groupBy(label_col, prediction_col).count().collect()

    # Extract counts from the DataFrame
    true_pos = true_neg = false_pos = false_neg = 0
    for row in confusion_matrix:
        if row[label_col] == 0 and row[prediction_col] == 0:
            true_neg = row['count']
        elif row[label_col] == 0 and row[prediction_col] == 1:
            false_pos = row['count']
        elif row[label_col] == 1 and row[prediction_col] == 0:
            false_neg = row['count']
        elif row[label_col] == 1 and row[prediction_col] == 1:
            true_pos = row['count']
    print(f"TN: {true_neg}\tFP: {false_pos}")
    print(f"FN: {false_neg}\tTP: {true_pos}")
```

Figure-7.15: “Confusion Matrix” customise function.

```
Gradient-Boosted Tree Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 8250        FP: 2085
FN: 3456        TP: 6670
-----------------------------------------------------
```

Figure-7.16: “Gradient-Boosted Tree Classifier” Model’s Confusion Matrix

```
Random Forest Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 8024        FP: 2311
FN: 3151        TP: 6975
-----------------------------------------------------
```

Figure-7.17: “Random Forest Classifier” Model’s Confusion Matrix

7.3.3. Precision-Recall and ROC Analysis

We have applied the “BinaryClassificationEvaluator” method to evaluate the value of Precision-Recall and ROC for both Models as shown in Figure-7.18 and Figure-7.19.

```
In [76]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nGradient-Boosted Tree Classifier - Area Under PR = {pr_auc:.2f}")

Gradient-Boosted Tree Classifier - Accuracy = 0.73

Gradient-Boosted Tree Classifier - Area Under ROC (AUC) = 0.79

Gradient-Boosted Tree Classifier - Area Under PR = 0.77
```

Figure-7.18: “Gradient-Boosted Tree Classifier” Precision-Recall and ROC analysis

```
In [97]:
# Evaluate model Accuracy
mulEvaluator = MulticlassClassificationEvaluator(labelCol="cardio", predictionCol="prediction", metricName="accuracy")
accuracy = mulEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Accuracy = {accuracy:.2f}")

# Evaluate model using areaUnderROC
binaryEvaluator = BinaryClassificationEvaluator(labelCol="cardio", rawPredictionCol="rawPrediction", metricName="areaUnderROC
roc_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under ROC (AUC) = {roc_auc:.2f}")

# Evaluate model using areaUnderPR
binaryEvaluator.setMetricName("areaUnderPR")
pr_auc = binaryEvaluator.evaluate(prediction)
print(f"\nRandom Forest Classifier - Area Under PR = {pr_auc:.2f}")

Random Forest Classifier - Accuracy = 0.73

Random Forest Classifier - Area Under ROC (AUC) = 0.79

Random Forest Classifier - Area Under PR = 0.78
```

Figure-7.19: “Random Forest Classifier” Model’s Precision-Recall and ROC analysis

7.3.4. Input Features and Prediction Relationship

We implemented a custom function, shown in Figure-7.20, that utilise the “featureImportances” property to get the importance features for model prediction. This function allows us to print out the list of importance features in descending order. The generated report for both models can be viewed in Figure-7.21 and Figure-7.22. We will examine the input features and prediction relationship.

```
In [95]:
# Featrue Importance Method
def feature_importance(model, feature_names):
    # Extract feature importances from the trained model
    importances = model.featureImportances

    # Map feature names to their importances and sort by importance
    features_with_importance = sorted(zip(feature_names, importances), key=lambda x: x[1], reverse=True)

    for feature, importance in features_with_importance:
        print(f"Feature: {feature}, Importance: {importance:.4f}")
```

Figure-7.20: “feature_importance” customise function.

```
Gradient-Boosted Tree Classifier - Predictor Feature Importance:
------------------------------------------------------
Feature: ap_hi, Importance: 0.4054
Feature: Pulse_Rate, Importance: 0.2252
Feature: ap_lo, Importance: 0.1760
Feature: age, Importance: 0.1089
Feature: cholesterol, Importance: 0.0685
Feature: BMI, Importance: 0.0117
Feature: weight, Importance: 0.0032
Feature: gluc, Importance: 0.0011
------------------------------------------------------
```

Figure-7.12: “Gradient-Boosted Tree Classifier” model’s Predictor Feature Importance

```
Random Forest Classifier - Predictor Feature Importance:
--------------------------------------------------------
Feature: ap_hi, Importance: 0.3395
Feature: Pulse_Rate, Importance: 0.1878
Feature: ap_lo, Importance: 0.1577
Feature: age, Importance: 0.1379
Feature: cholesterol, Importance: 0.0764
Feature: BMI, Importance: 0.0493
Feature: weight, Importance: 0.0389
Feature: gluc, Importance: 0.0126
--------------------------------------------------------
```

Figure-7.22: "Random Forest Classifier" model's Predictor Feature Importance

8. Interpretation

8.1. Study and Discuss Mined Patterns

8.1.1. Pattern-1: Classification Reports

Gradient-Boosted Tree Classifier - Classification Report:

	precision	recall	f1-score	support
0	0.70	0.80	0.73	10335
1	0.76	0.66	0.73	10126
accuracy			0.73	20461
macro avg	0.73	0.73	0.73	20461
weighted avg	0.73	0.73	0.73	20461

Random Forest Classifier - Classification Report:

	precision	recall	f1-score	support
0	0.72	0.78	0.73	10335
1	0.75	0.69	0.73	10126
accuracy			0.73	20461
macro avg	0.73	0.73	0.73	20461
weighted avg	0.73	0.73	0.73	20461

Figure-8.1: "Gardient-Boosted Tree Classifier" and "Random Forest Classifier" Classification Reports

As illustrated in Figure-8.1, which compares the classification reports of the Gradient-Boosted Tree Classifier and Random Forest Classifier models, we observed several distinctions between them:

- ***Accuracy**:* Both Gradient-Boosted Tree Classifier and Random Forest Classifier show an accuracy level of around 73% in predicting outcomes.
- ***Precision**:* In terms of predicting "cardio=0" outcomes, the Gradient-Boosted Tree Classifier shows a precision of 70%, while the Random Forest Classifier has a precision of 72%. For "cardio=1", the Gradient-Boosted Tree Classifier achieves a precision of 76%, and the Random Forest Classifier gets a precision of 75%.
- ***Recall**:* For the Gradient-Boosted Tree Classifier, the recall rate for "cardio=0" is 80%, and for "cardio=1" is 66%. For the Random Forest Classifier, the recall rate for "cardio=0" is 78% and for "cardio=1" is 69%.
- ***F1-Score**:* The F1-scores for "cardio=0" are 73% for both models. For "cardio=1" predictions, both classifiers show an F1-score of 73%.
- ***Averages**:* Examining both the macro and weighted averages indicates a consistent performance by both models, suggesting a harmonized proficiency in predicting the "cardio=0" and "cardio=1" outcomes.

8.1.2. Pattern-2: Confusion Matrix

```
Gradient-Boosted Tree Classifier - Confusion Matrix:
----------------------------------------------------
TN: 8250        FP: 2085
FN: 3456        TP: 6670
----------------------------------------------------

Random Forest Classifier - Confusion Matrix:
----------------------------------------------------
TN: 8024        FP: 2311
FN: 3151        TP: 6975
----------------------------------------------------
```

Figure-8.2: "Gardient-Boosted Tree Classifier" and "Random Forest Classifier" Confusion Matrix

As shown in Figure 8.2, we make a comparison of the confusion matrix result for the Gradient-Boosted Tree Classifier and Random Forest Classifier and discovered the following points:

- ***True Negatives:*** The Gradient-Boosted Tree Classifier exceeded the Random Forest Classifier in correctly discerning negative instances with counts of 8250 to 8024, respectively.
- ***False Positives:*** The Gradient-Boosted Tree Classifier recorded more false positives, noting 2085 compared to the Random Forest Classifier's 2311. It suggests that the Gradient-Boosted Tree Classifier has a slightly higher tendency to classify actual negative cases as positive than the Random Forest Classifier.
- ***False Negatives:*** In the context of false negatives, the Gradient-Boosted Tree Classifier showed a higher tally of 3456 versus the Random Forest Classifier's 3151. It indicates that the Gradient-Boosted Tree Classifier was more likely to overlook positive cases, mistakenly labelling them as unfavourable, compared to the Random Forest Classifier.
- ***True Positives:*** In correctly classifying positive cases, the Random Forest Classifier led with a count of 6975 against the Gradient-Boosted Tree Classifier's 6670.

8.1.3. Pattern-3: Precision-Recall & ROC Curve

For imbalanced datasets, the Precision-Recall Curve (PRC) is an important tool, highlighting the performance related to the positive class. The ROC Curve outlines the balance between the true and false positive rates.

```
Gradient-Boosted Tree Classifier - Accuracy = 0.73

Gradient-Boosted Tree Classifier - Area Under ROC (AUC) = 0.79

Gradient-Boosted Tree Classifier - Area Under PR = 0.77
```

Figure-8.3: "Gardient-Boosted Tree Classifier" Precision-Recall & ROC Curve Value

```
Random Forest Classifier - Accuracy = 0.73

Random Forest Classifier - Area Under ROC (AUC) = 0.80

Random Forest Classifier - Area Under PR = 0.78
```

Figure-8.4: "Random Forest Classifier" Precision-Recall & ROC Curve Value

As illustrated in Figures-8.3 and 8.4, the AUC values for the two models are as follows:

- ***Gradient-Boosted Tree Classifier***: PRC AUC: 0.77 / ROC AUC: 0.79 / Accuracy Rate: 73%
- ***Random Forest Classifier*****:** PRC AUC: 0.78 / ROC AUC: 0.80 / Accuracy Rate: 73%

Upon comparison, the performance of both classifiers is similar. The Random Forest Classifier shows marginally better performance in the PRC evaluation. However, both classifiers display nearly identical efficiency when assessed via the ROC curve, with the Random Forest Classifier having a slight edge in the ROC AUC value. Additionally, both models have the same accuracy rate of 73%.

8.1.4. Pattern-4: Predictor Importance Features

```
Gradient-Boosted Tree Classifier - Predictor Feature Importance:
--------------------------------------------------
Feature: ap_hi, Importance: 0.4054
Feature: Pulse_Rate, Importance: 0.2252
Feature: ap_lo, Importance: 0.1760
Feature: age, Importance: 0.1089
Feature: cholesterol, Importance: 0.0685
Feature: BMI, Importance: 0.0117
Feature: weight, Importance: 0.0032
Feature: gluc, Importance: 0.0011
--------------------------------------------------

Random Forest Classifier - Predictor Feature Importance:
--------------------------------------------------
Feature: ap_hi, Importance: 0.3395
Feature: Pulse_Rate, Importance: 0.1878
Feature: ap_lo, Importance: 0.1577
Feature: age, Importance: 0.1379
Feature: cholesterol, Importance: 0.0764
Feature: BMI, Importance: 0.0493
Feature: weight, Importance: 0.0389
Feature: gluc, Importance: 0.0126
--------------------------------------------------
```

Figure-8.5: "Gardient-Boosted Tree Classifier" and "Random Forest Classifier" Predictor Importance

As illustrated in Figure-8.5, for the "Gradient-Boosted Tree Classifier," the most impactful predictor is "ap_hi" with an importance score of 0.4054, followed closely by "Pulse_Rate" and "ap_lo" with importance scores of 0.2252 and 0.1760, respectively. On the other hand, for the "Random Forest Classifier", the primary predictor is also "ap_hi", but with a slightly lower importance of 0.3395. It is trailed by "Pulse_Rate" and "ap_lo" with significance values of 0.1878 and 0.1577, respectively. The feature "gluc" has the most negligible influence on the prediction for both classifiers, indicating its lesser relevance in these specific model constructs.

8.2. Visualising the Data, Results, Models and Patterns

In this section, we will try to use the PySpark's evaluators library to evaluate on the model prediction and the data between input features and target. And along with the detail model results, report and charts.

8.2.1. Visualise the Data

We will use the "groupBy" method from PySpark's SQL library to analysis the data relationship on the training dataset as shown in Figure-8.6, 8.7, 8.8, and 8.9.

Cardio vs Gender

cardio	gender	Count
1	2	11796
1	1	21487
0	1	22100
0	2	11774

Figure-8.6: Cardio vs Gender

Cardio vs Active

cardio	active	Count
1	0	7033
1	1	26250
0	0	6161
0	1	27713

Figure-8.7: Cardio vs Active

```
Cardio vs Smoke
-----------------------------------
+------+-----+-----+
|cardio|smoke|Count|
+------+-----+-----+
|     1|    0|30491|
|     1|    1| 2792|
|     0|    0|30710|
|     0|    1| 3164|
+------+-----+-----+
```

Figure-8.8: Cardio vs Smoke

```
Cardio vs Alco
-----------------------------------
+------+----+-----+
|cardio|alco|Count|
+------+----+-----+
|     1|   0|31553|
|     1|   1| 1730|
|     0|   0|31983|
|     0|   1| 1891|
+------+----+-----+
```

Figure-8.9: Cardio vs Alco

We will use the “seaborn” and “matplotlib.pyplot” libraries to plot the chart to visualise the data relationship and distribution on the training dataset as shown in Figure-8.10, 8.11, 8.12, 8.13, and 8.14.

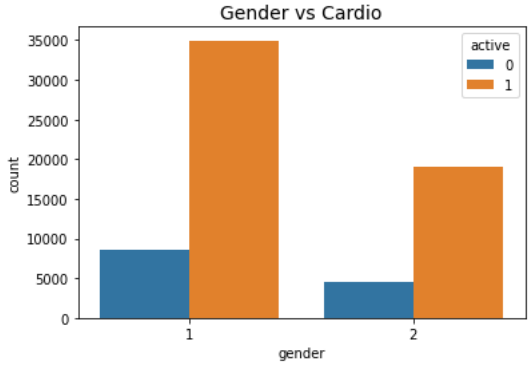


Figure-8.10: “gender” vs “cardio”

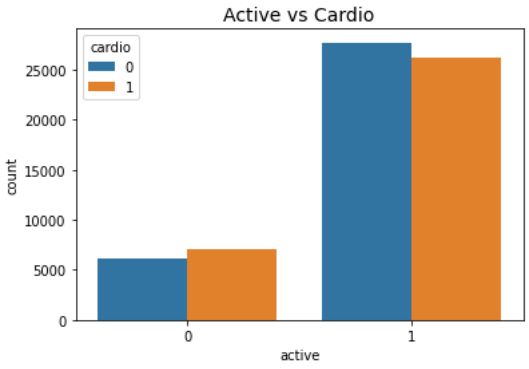


Figure-8.11: “active” vs “cardio”

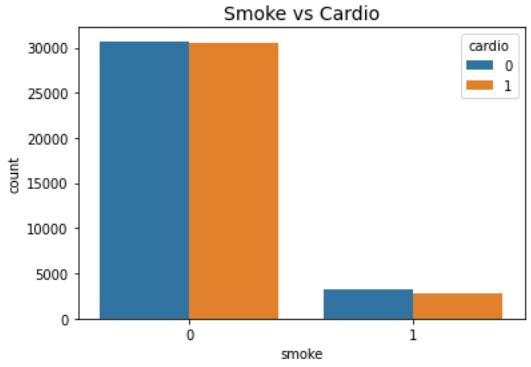


Figure-8.12: “smoke” vs “cardio”

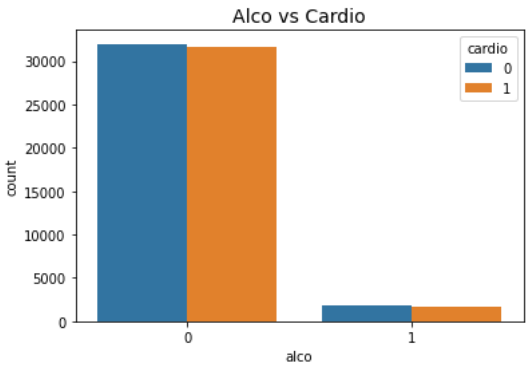


Figure-8.13: “alco” vs “cardio”

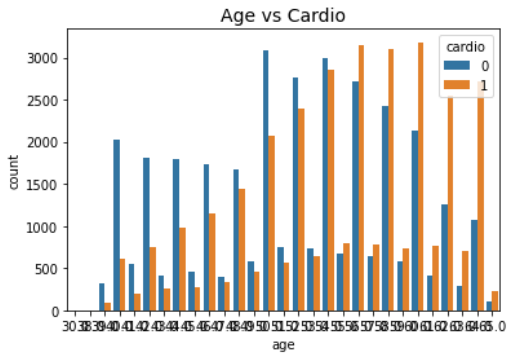


Figure:8.14: “age” vs “cardio”

We will utilise the “stat.corr” method to study the correlation between target and each featrue and also visualise the correlation result in heat-map diagram using “seaborn” and “matplotlib.pyplot” libraries as described in Figure-8.15.

```
Correlation Results
-----------------------------------
Correlation between "cardio" and "ap_hi": 0.4243
Correlation between "cardio" and "ap_lo": 0.3340
Correlation between "cardio" and "Pulse_Rate": 0.3182
Correlation between "cardio" and "age": 0.2386
Correlation between "cardio" and "cholesterol": 0.2212
Correlation between "cardio" and "BMI": 0.1900
Correlation between "cardio" and "weight": 0.1787
Correlation between "cardio" and "gluc": -0.0915
```

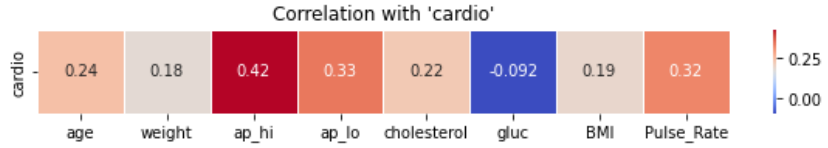


Figure-8.15: Correlation Results and Heat-map diagram

We will apply the “skewness” method to check the skewness value on each feature and also visualise the distribution chart using “seaborn” and “matplotlib.pyplot” libraries as shown in Figure-8.16 and Figure-8.17.

```
Skewness Values
-----------------------------------
Skewness value for age: -0.3042
Skewness value for weight: 1.0454
Skewness value for ap_hi: 0.9155
Skewness value for ap_lo: 0.6647
Skewness value for cholesterol: 1.5947
Skewness value for gluc: -2.0554
Skewness value for BMI: 1.2111
Skewness value for Pulse_Rate: 0.5382
```

Figure-8.16: Skewness Values for each feature

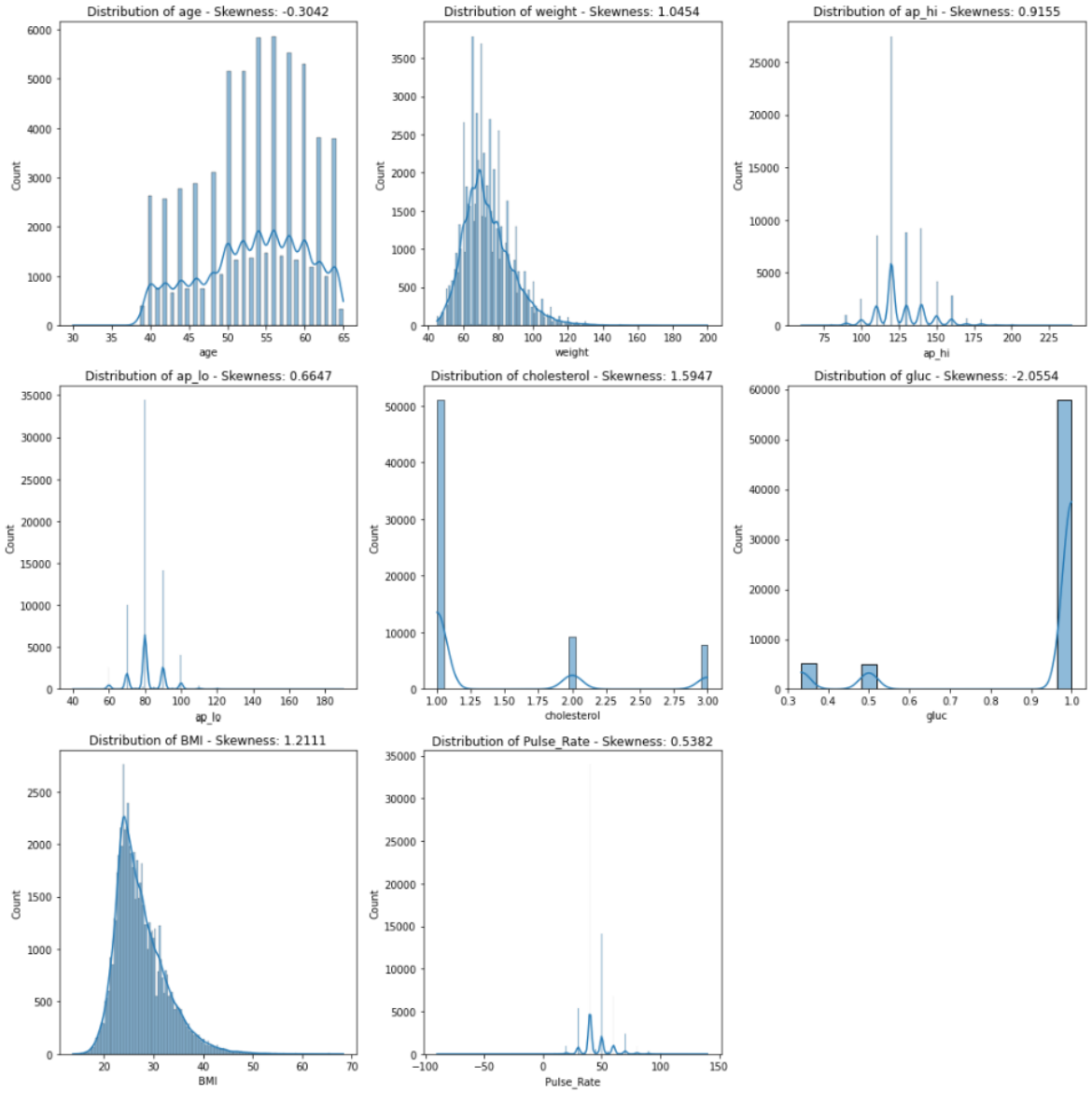


Figure-8.17: Distribution Charts

8.2.2. Visualise the "Classification Report" for Both Models

As shown in Figure-8.18 and Figure-8.19, We are analysing the classification report for both models.

```
Gradient-Boosted Tree Classifier - Classification Report:
------------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.78    0.73       10268
           1  0.75         0.67    0.73       10104
    accuracy                        0.73       20372
   macro avg  0.73         0.73    0.73       20372
weighted avg  0.73         0.73    0.73       20372
------------------------------------------------------
```

Figure-8.18: "Gradient-Boosted Tree Classifier" Model's Classification Report

```
Random Forest Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.78    0.73       10268
           1  0.75         0.67    0.73       10104
    accuracy                        0.73       20372
   macro avg  0.73         0.73    0.73       20372
weighted avg  0.73         0.73    0.73       20372
-----------------------------------------------------
```

Figure-8.19: “Random Forest Classifier” Model’s Classification Report

8.2.3. Visualise the “Confusion Matrix” for Both Models

In Figure-8.20 and Figure-8.21, we will visualise the heat-map of “Confusion Matrix” for both models.

```
Gradient-Boosted Tree Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 8057        FP: 2211
FN: 3327        TP: 6777
-----------------------------------------------------
```

Figure-8-20: “Gradient-Boosted Tree Classifier” Model’s Confusion Matrix

```
Random Forest Classifier - Confusion Matrix:
-----------------------------------------------------
TN: 8057        FP: 2211
FN: 3327        TP: 6777
-----------------------------------------------------
```

Figure-8.21: “Random Forest Classifier” Model’s Confusion Matrix

8.2.4. Visualise the “Precision-Recall” and “ROC” curve for Both Models.

We are analysing the model’s “Accuracy”, “Precision-Recall” and “ROC” care value as follows Figures-8.22, and 8.23.

```
Gradient-Boosted Tree Classifier - Accuracy = 0.73

Gradient-Boosted Tree Classifier - Area Under ROC (AUC) = 0.79

Gradient-Boosted Tree Classifier - Area Under PR = 0.77
```

Figrue-8.22: Gradient-Boosted Tree Classifier’s “Accuracy”, “Precision-Recall” and “ROC” curve value

```
Random Forest Classifier - Accuracy = 0.73

Random Forest Classifier - Area Under ROC (AUC) = 0.80

Random Forest Classifier - Area Under PR = 0.78
```

Figrue-8.23: Random Forest Classifier’s “Accuracy”, “Precision-Recall” and “ROC” curve value

8.2.5. Visualise the “Predictor Feature Importance” for Both Models.

We used the “featureImportances” property to evaluate the predictor importance as shown in Figure 8.24 and 8.25.

```
Gradient-Boosted Tree Classifier - Predictor Featrue Improtance:
--------------------------------------------------
Feature: ap_hi, Importance: 0.4784
Feature: ap_lo, Importance: 0.1450
Feature: age, Importance: 0.1227
Feature: Pulse_Rate, Importance: 0.0796
Feature: cholesterol, Importance: 0.0753
Feature: BMI, Importance: 0.0493
Feature: weight, Importance: 0.0377
Feature: gluc, Importance: 0.0121
--------------------------------------------------
```

Figure-8.24: “Gradient-Boosted Tree Classifier” Predictor Feature Importance

```
Random Forest Classifier - Predictor Feature Importance:
--------------------------------------------------
Feature: ap_hi, Importance: 0.3395
Feature: Pulse_Rate, Importance: 0.1878
Feature: ap_lo, Importance: 0.1577
Feature: age, Importance: 0.1379
Feature: cholesterol, Importance: 0.0764
Feature: BMI, Importance: 0.0493
Feature: weight, Importance: 0.0389
Feature: gluc, Importance: 0.0126
--------------------------------------------------
```

Figur-8.25: “Random Forest Classifier” Predictor Feature Importance

8.3. Interpreting the Results, Models and Patterns

Section 8.1.1, a detailed analysis of the classification reports for the two models underscored an intriguing observation: the Gradient-Boosted Tree Classifier and the Random Forest Classifier posted near-identical accuracy levels, rounding off to roughly 73%. In simple terms, for about every four predictions the models make, they get three correct, indicating good accuracy.

Section 8.1.2, a visual expedition into the Confusion Matrix revealed distinct strengths for each model. The Gradient-Boosted Tree Classifier demonstrated a pronounced aptitude in accurately identifying negative cases, denoted as True Negatives. Conversely, the Random Forest Classifier displayed marked prowess in pinpointing positive cases, termed True Positives.

Section 8.1.3, an examination of the ROC and Precision-Recall Curve values, showed that both classifiers offer closely matched performance. While they were nearly identical in their efficiency on the ROC curve, the Random Forest Classifier edged slightly ahead in the PRC evaluation. It's also significant to highlight that both models reported an equivalent accuracy rate of 73%.

Section 8.1.4, our exploration into feature importance, brought distinct preferences for each model. While the Gradient-Boosted Tree Classifier placed significant emphasis on the “ap_hi” feature, the Random Forest Classifier, although valuing “ap_hi” as well, also recognised the significance of other features such as “Pulse_Rate” and “ap_lo”. This divergence in feature prioritisation underscores the nuanced differences in how each model interprets and utilises the data for predictions.

Section 8.2.1, several visualisations like reports, results, heat-map and distribution charts help identify potential correlations and patterns. For instance, we can infer relationships between features such as “gender” and “cardio” or “cholesterol” and “cardio”, giving clues about potential risk factors.

8.4. Assessing and Evaluating Results, Models, and Patterns

After accessing and evaluating the results, models, and patterns, we found some important points and new areas to study.

- ***Model Performance:*** The Gradient-Boosted Tree Classifier and the Random Forest Classifier proved an accuracy rate of around 73%. While this is a respectable figure, it is vital to recognize that this implies a margin of error of about 27%, highlighting the room for improvement.
- ***Confusion Matrix Analysis:*** The Gradient-Boosted Tree Classifier exhibits a stronger ability to pinpoint negative instances. In contrast, the Random Forest Classifier excels in recognising positive cases. It is possible to explore an ensemble methodology that capitalises on the strengths of both models for potentially heightened accuracy.
- ***Feature Importance:*** It is important that while the Gradient-Boosted Tree Classifier places considerable weight on the "ap_hi" feature, the Random Forest Classifier balances its importance across "ap_hi", "Pulse_Rate", and "ap_lo".
- ***Precision-Recall vs. ROC:*** Both classifiers present similarly when evaluated using the ROC curve. However, the Random Forest Classifier exhibits a slight advantage in the Precision-Recall Curve assessment. Though subtle, this distinction underscores each model's varied capabilities, even as they maintain an identical accuracy rate of 73%.
- ***Visualizations:*** Various visualisations, including reports, heat-maps, and distribution charts, facilitate the identification of correlations and emerging patterns. Such visual aids enable insights into the relationships between specific features and the target variable, "cardio". For example, connections between "gender" and "cardio" or "cholesterol" and "cardio" are evident, providing indications of potential risk factors.

We observed the following limitations:

- ***Feature Engineering:*** The importance of "Pulse_Rate" suggests there might be other features or combinations of features that could be significant. Exploring more refined feature engineering techniques might enhance the models.
- ***Hyperparameter Tuning:*** While we used tuned hyperparameters, not all parameters were applied. We might need to consider tuning other possible parameters.

Following are the areas for further exploration:

- ***Model Ensembling:*** As mentioned earlier, combining both models could make a better overall model.
- ***Advanced Algorithms:*** Besides Gradient-Boosted Tree and Random Forest, looking into advanced model like Multilayer Perceptron Classifier[9] also known as Neural Networks could be helpful.
- ***Deeper Data Analysis:*** Working with experts, such as cardiologists, for this study could guide the modelling process more accurately.

8.5. Iterations

We have simultaneously created two models, which are "Gradient-Boosted Tree Classifier" and "Random Forest Classifier", and carried out data mining in the earlier sections, such as Section-7

[9] https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier

onwards, which can be seen as iteration. Additionally, we will continue with further iterations as follows:

1. To adjust the data partition size to 80/20 and rebuild both models ("Gradient-Boosted Tree Classifier" and "Random Forest Classifier") and analysis the accuracy with Classification Report.
2. To build the "Decision Tree Classifier" on partition data size 80/20 and evaluate the model accuracy with classification report, that classifier was left out due to lower rank when we pick the algorithm for our study in Section-6.

8.5.1. Iteration-1 – Partition Data Size 80% of Training and 20% of Testing

As shown in Figure-8.26, we will adjust the data partition size that we apply for training data size to 80% and testing to 20%. And then, we will rebuild both models with the best hyperparameter that we were tuned in Section-6 as illustrated in Figure-8.27 and Figure-9.28.

Iteration-1 – Partition Data Size 80% of Training and 20% of Testing

```
In [94]:
# Split the data into training and test sets (80% train, 20% test)
train_data, test_data = merge_df.randomSplit([0.8, 0.2])
```

Figure-8.26: Adjust the data partition size to 80/20.

```
In [95]:
# Build the Gradient-Boosted Tree Classifier with best hyperparameter
gbtCls = GBTClassifier(labelCol='cardio', featuresCol='features', maxDepth=5, stepSize=0.1)
pipeline = Pipeline(stages=[assembler, rfCls])
model = pipeline.fit(train_data)
prediction = model.transform(test_data)

# Generate Classification report
print("\nGradient-Boosted Tree Classifier - Classification Report: ")
print("-------------------------------------------------------")
classification_report(prediction)
print("-------------------------------------------------------")
```

Figure-8.27: Rebuilt "Gradient-Boosted Tree Classifier" Model with 80/20 data size

```
In [96]:
# Build the Random Forest Classifier with best hyperparameter
rfCls = RandomForestClassifier(labelCol='cardio', featuresCol='features', maxDepth=10, maxBins=64)
pipeline = Pipeline(stages=[assembler, rfCls])
model = pipeline.fit(train_data)
prediction = model.transform(test_data)

# Generate Classification report
print("\nRandom Forest Classifier - Classification Report: ")
print("-------------------------------------------------------")
classification_report(prediction)
print("-------------------------------------------------------")
```

Figure-8.28: Rebuilt "Random Forest Classifier" Model with 80/20 data size

As shown in Figure-8.29, both models maintained an accuracy of 73%, consistent with the previous results from the 70/30 data partition ratio model that we performed in Section-7.

```
Gradient-Boosted Tree Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.79    0.73       6900
           1  0.76         0.67    0.73       6690
    accuracy                        0.73       13590
   macro avg  0.74         0.73    0.73       13590
weighted avg  0.74         0.73    0.73       13590
-----------------------------------------------------


Random Forest Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.79    0.73       6900
           1  0.76         0.67    0.73       6690
    accuracy                        0.73       13590
   macro avg  0.74         0.73    0.73       13590
weighted avg  0.74         0.73    0.73       13590
-----------------------------------------------------
```

Figure-8.29: Classification Reports for Both

8.5.2. Iteration-2 – "Decision Tree Classifier" Model

As illustrated in Figure-8.30, we constructed the "Decision Tree Classifier" model with default hyper parameter. This model was built using 80/20 data partition size. Upon analysing the predictions with the classification report, we observed an accuracy rate of 73% which was same accuracy rate as pervious model that we explored in Section-6.1.2, as shown in Figure-8.31.

```
In [97]: # Build Decision Tree Classifier
dtCls = DecisionTreeClassifier(labelCol='cardio',featuresCol='features')
pipeline = Pipeline(stages=[assembler, dtCls])
model = pipeline.fit(train_data)
prediction = model.transform(test_data)

# Generate Classification report
print("\nDecision Tree Classifier - Classification Report: ")
print("-----------------------------------------------------")
classification_report(prediction)
print("-----------------------------------------------------")
```

Figure-8.30: Build "Decision Tree Classifier" Model

```
Decision Tree Classifier - Classification Report:
-----------------------------------------------------
              precision    recall  f1-score   support
           0  0.71         0.79    0.73       6900
           1  0.76         0.67    0.73       6690
    accuracy                        0.73       13590
   macro avg  0.74         0.73    0.73       13590
weighted avg  0.74         0.73    0.73       13590
-----------------------------------------------------
```

Figure-8.31: "Decision Tree Classification" Classification Report

Reference

World Health Organization (2021, June 11). Cardiovascular diseases (CVDs). Www.who.int. Retrieved July 19, 2023, from https://www.who.int/news-room/fact-sheets/detail/cardiovascular-diseases-(CVDs)

World Health Organization (2013, November 14). Global action plan for the prevention and control of NCDs 2013–2020. WHO. Retrieved July 19, 2023, from https://www.who.int/publications/i/item/9789241506236

Heart Research Institute NZ (n.d.). Cardiovascular Disease. Retrieved July 19, 2023, from https://www.hri.org.nz/health/learn/cardiovascular-disease/cardiovascular-disease-impacts-and-risks

JAHA (n.d.). Sustainable Development Goals and the Future of Cardiovascular Health: A Statement From the Global Cardiovascular Disease Taskforce. Journal of the American Heart Association. Retrieved July 20, 2023, from https://www.ahajournals.org/doi/full/10.1161/JAHA.114.000504

Smartsheet (n.d.). *Project Plan*. Retrieved July 20, 2023, from https://www.smartsheet.com

IBM (n.d.). *Acceptable Skewness Level*. IBM SPSS Documentaton. Retrieved October 1, 2023, from https://www.ibm.com/docs/en/spss-statistics/29.0.0?topic=summarize-statistics

PySpark (n.d.). *ML Algorithms*. PySpark Documentation. Retrieved October 1, 2023, from https://spark.apache.org/docs/latest/ml-guide.html

PySpark (n.d.). *Decision Tree Classifier*. PySpark Documentation. Retrieved October 1, 2023, from https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-classifier

PySpark (n.d.). *Random Forest Classifier*. PySpark Documentation. Retrieved October 1, 2023, from https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier

PySpark (n.d.). *Gardient-Boosted Tree Classifier*. PySpark Documentation. Retrieved October 1, 2023, from https://spark.apache.org/docs/latest/ml-classification-regression.html#gradient-boosted-tree-classifier

PySpark (n.d.). *Multilayer Perceptron Classifier*. PySpark Documentation. Retrieved October 1, 2023, from https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier

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.