Heart Disease Prediction Task

Anthony Reidy

Nov 07, 2022

CONTENTS

Heart disease, alternatively known as cardiovascular disease, encases various conditions that impact the heart and is the primary basis of death worldwide over the span of the past few decades. Approximately 10,000 people die in Ireland from Cardiovascular Disease each year, accounting for 36% of deaths per annum [*Worrying statistics about heart disease*, 2020]. That's despite the fact that 80% of all heart disease is deemed preventable. This investigations aims to explore a kaggle dataset and build a model that can predict the likelihood of a patient having heart disease. More specifically, with this dataset, we would like to see if we can develop a good model to predict if a person has heart disease and what *factors* can be attributed to heart disease most directly. We will be tackling this question with the usage of different regression techniques and algorithms.

Description of Dataset

We uses an existing dataset from Kaggle. The dataset comprises approx 320,0000 instances and 14 attributes.

Note: No personal identifiable information of the patients are recorded in the dataset.

The table below summarizes the multiple columns used in this investigation.

Name	Description
HeartDisease	Respondents that have ever reported having coronary heart disease (CHD) or myocardial infarction (MI).
BMI	Body Mass Index.
Smoking	Have you smoked at least 100 cigarettes in your entire life?
AlcoholDrinking	Heavy drinkers (adult men having more than 14 drinks per week and adult women having more than 7 drinks per week
Stroke	(Ever told) (you had) a stroke?
PhysicalHealth	Now thinking about your physical health, which includes physical illness and injury, for how many days during the past 30 days was your physical health not good? (0-30 days).
MentalHealth	Thinking about your mental health, for how many days during the past 30 days was your mental health not good? (0-30 days).
DiffWalking	Do you have serious difficulty walking or climbing stairs?
Sex	Are you male or female?
AgeCategory	Fourteen-level age category. (then calculated the mean)
Race	Imputed race/ethnicity value.
Diabetic	(Ever told) (you had) diabetes?
PhysicalActivity	Adults who reported doing physical activity or exercise during the past 30 days other than their regular job.
GenHealth	Would you say that in general your health is…
SleepTime	On average, how many hours of sleep do you get in a 24-hour period?
Asthma	(Ever told) (you had) asthma?
KidneyDisease	Not including kidney stones, bladder infection or incontinence, were you ever told you had kidney disease?
SkinCancer	(Ever told) (you had) skin cancer?

Methodology

Methodology will follows a typical data science project: from understanding the dataset through exploratory data analysis, data preparation, model buildings and finally model evaluation. We seek to build a model that predicts heart disease, a binary outcome In this investigation, we seek to use the K-means clustering approach to segment the patients into well-defined groups. To start, we perform an initial data exploration to perform transformations & data sanitization checks; acquire rudimentary statistics of the datasets; perform data augmentation; create exploratory visualizations. Next, we perform cluster analysis and evaluate our clusters using metrics such as Silhouette Coefficient and an Elbow curve. These clusters represent participants that exhibit similar risk factors for heart disease and may have similar underlying determinants of health such as their age, BMI, whether the smoke or have asthma. Next, we envision the probability of developing heart disease in the patients. Finally, we conclude with the most important outcomes of our work.

CHAPTER

ONE

INITIAL DATA EXPLORATION

The purpose of our initial data exploration is to:

1.1 Importing required libraries

Data processing

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import numpy as np
import copy
import random
import pickle
import json
```

Data Visualization

```
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.patches as mpatches
import plotly.graph_objects as go
import squarify
plt.style.use('ggplot')
```

Code Styling

```
from typing import List, Dict
```

Read in dataframe and bried inspection of the data.

```
heart_disease: pd.DataFrame = pd.read_csv("data/heart_2020_cleaned.csv")
```

1.2 Data Cleaning

Check for missing values

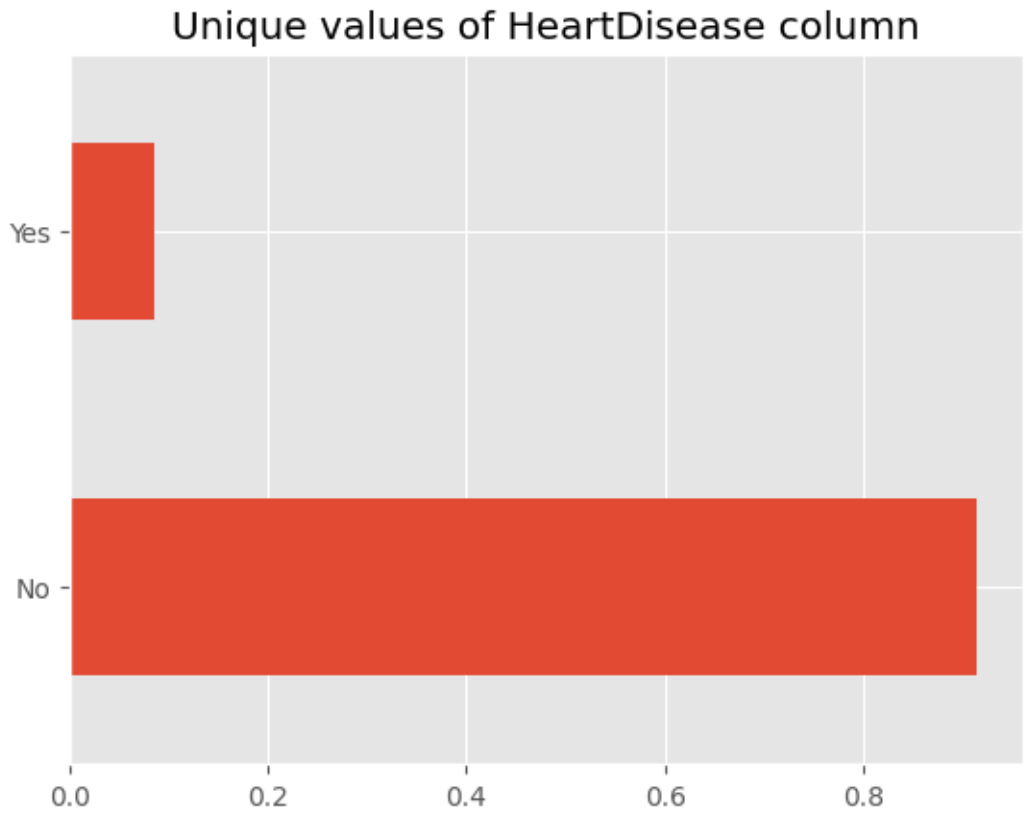
```
print(f'Does the heart disease dataset contain any null values? {heart_disease.
 ↪isnull().any().any()}')
```

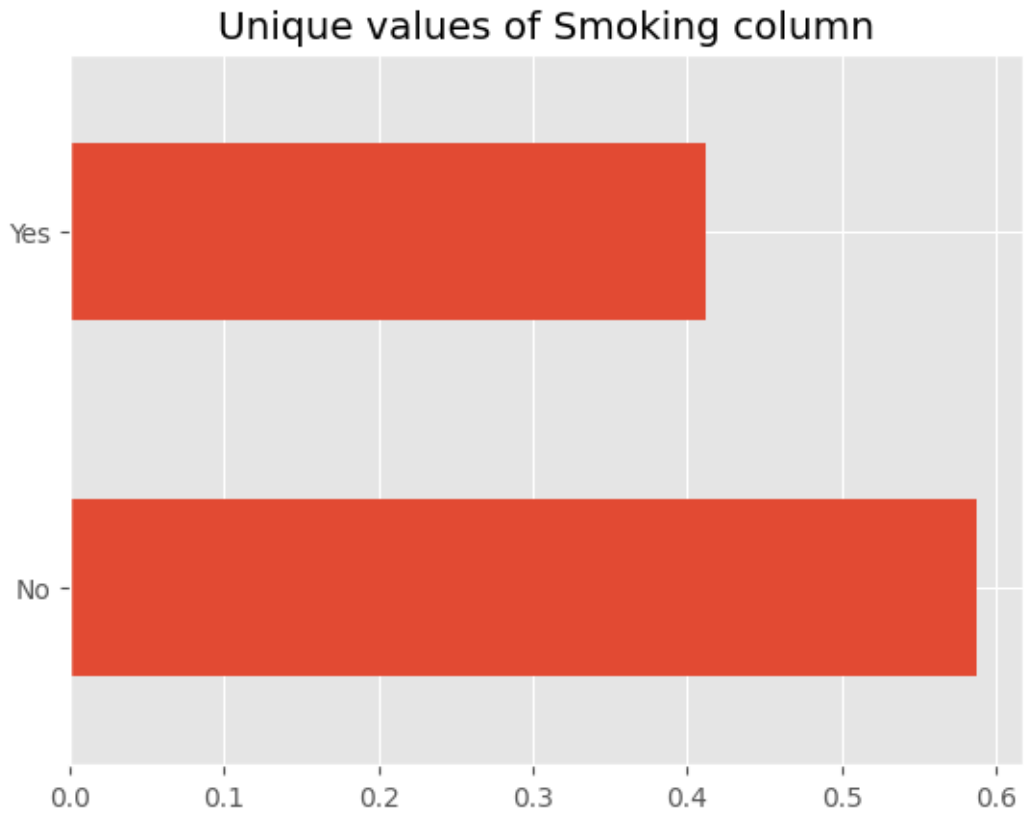
```
Does the heart disease dataset contain any null values? False
```

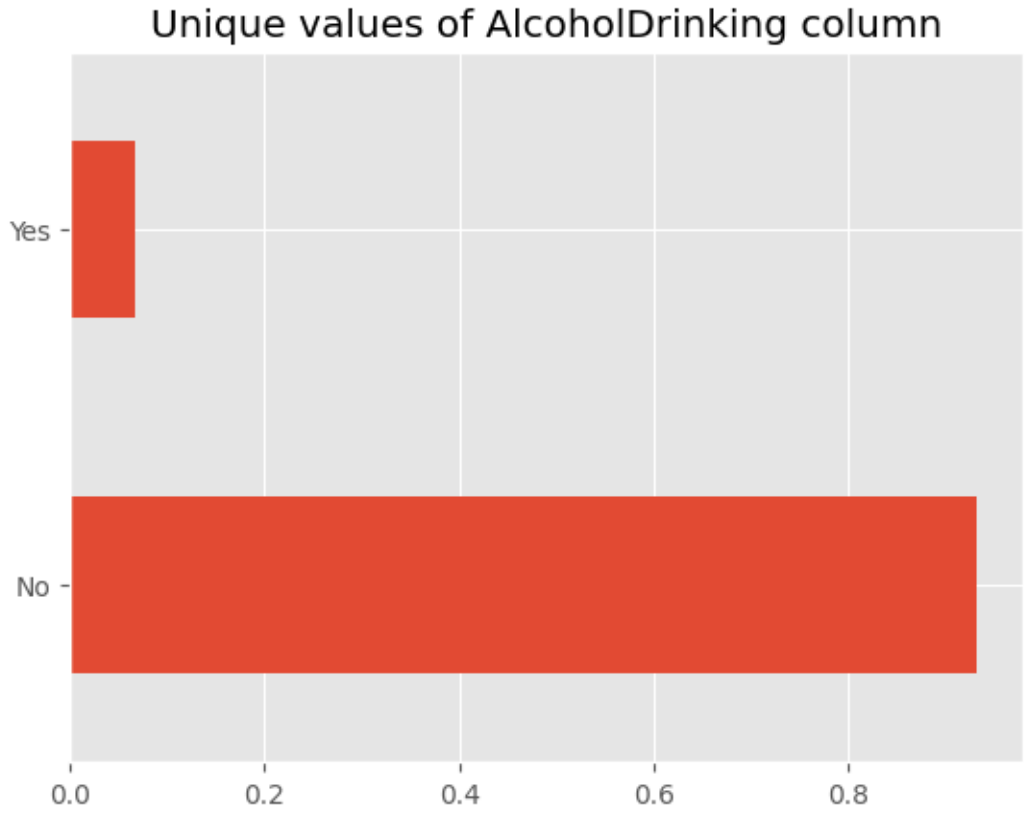
Next, we calculate the basic statistics of each data set. This is a trivial step, and it is designed to increase understanding of the computational problem.

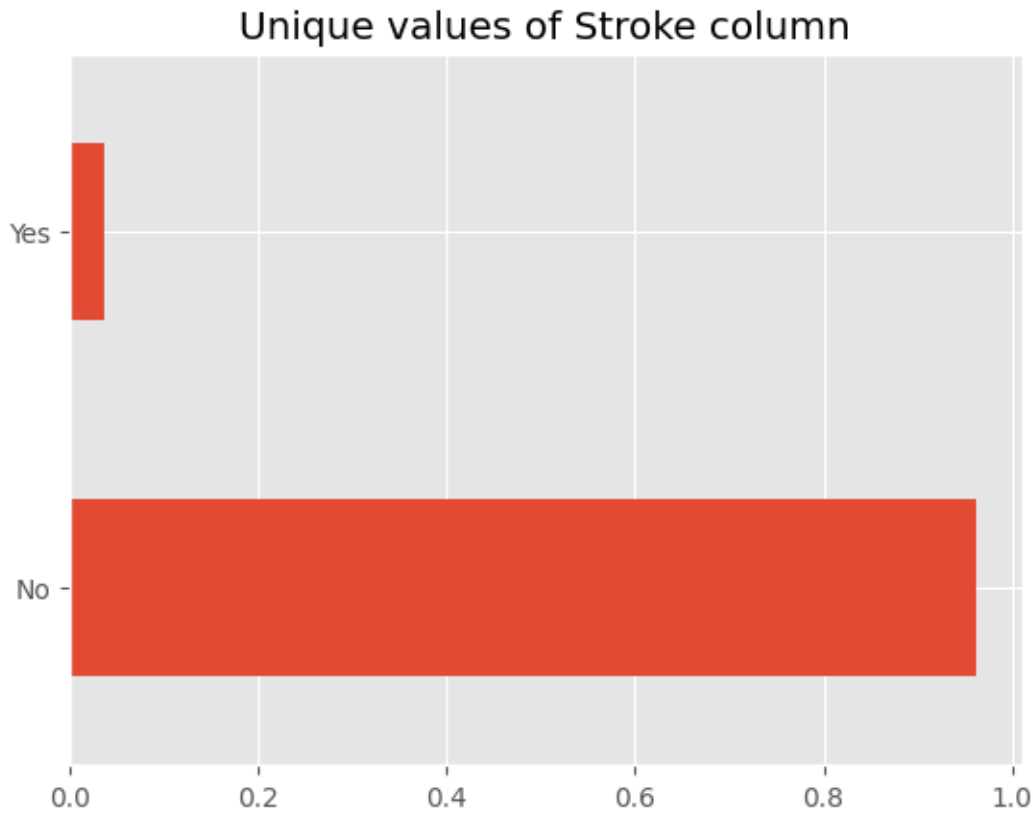
```
print("Unique values in the heart disease dataset, stratified by column:", sep='')
print("----------------------------------------------------------------\n")
for col in heart_disease.columns:
    if heart_disease[col].nunique() < 20:
        if heart_disease[col].nunique() < 3:
            heart_disease[col].value_counts(normalize=True).plot(kind='barh', title=f
 ↪"Unique values of {col} column", )
            plt.show()
```

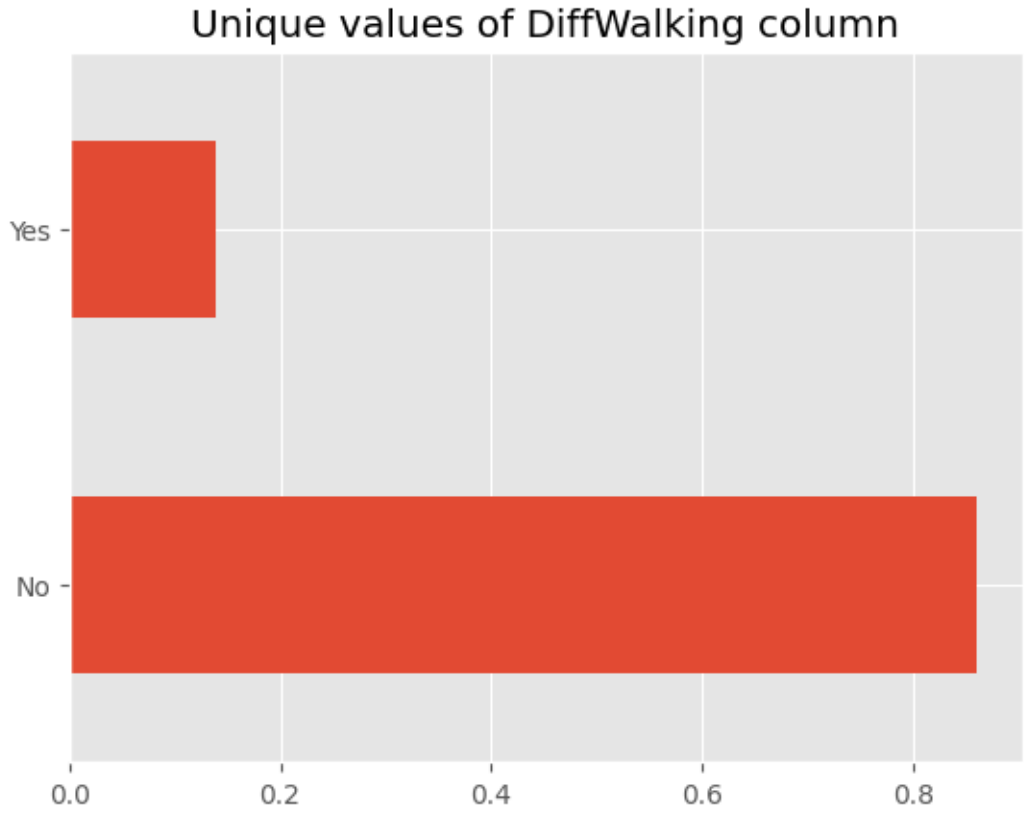
```
Unique values in the heart disease dataset, stratified by column:
----------------------------------------------------------------
```

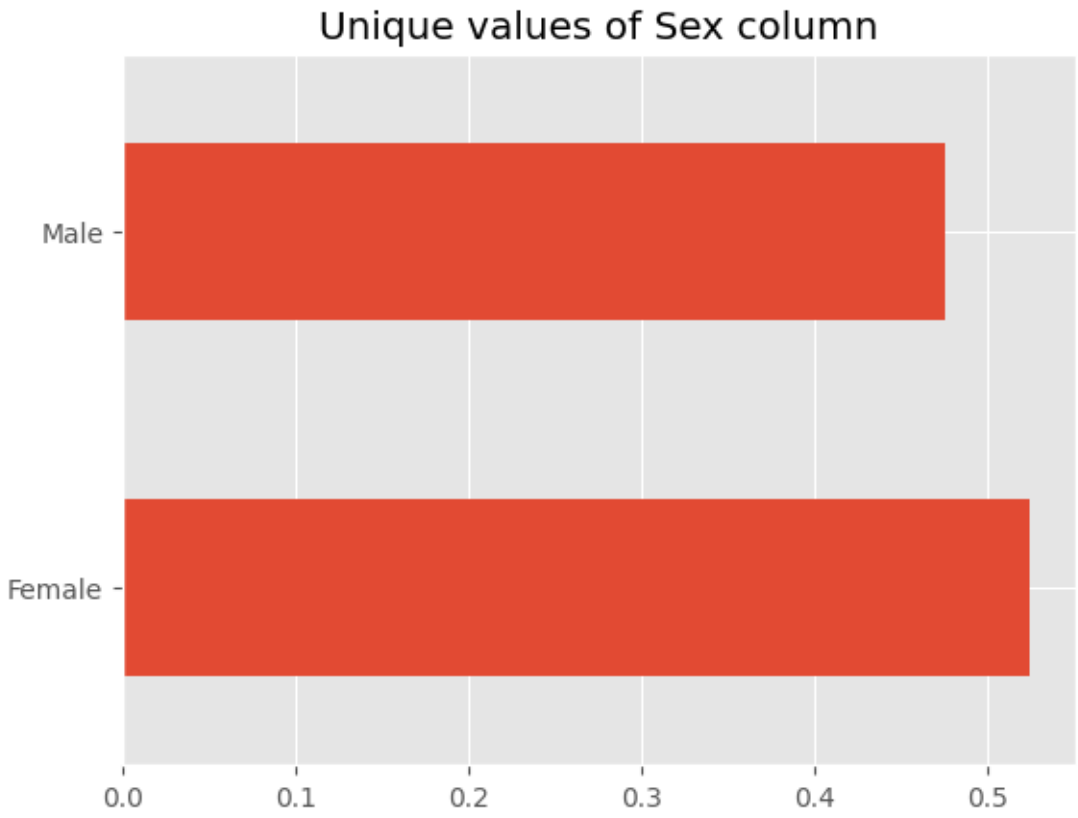


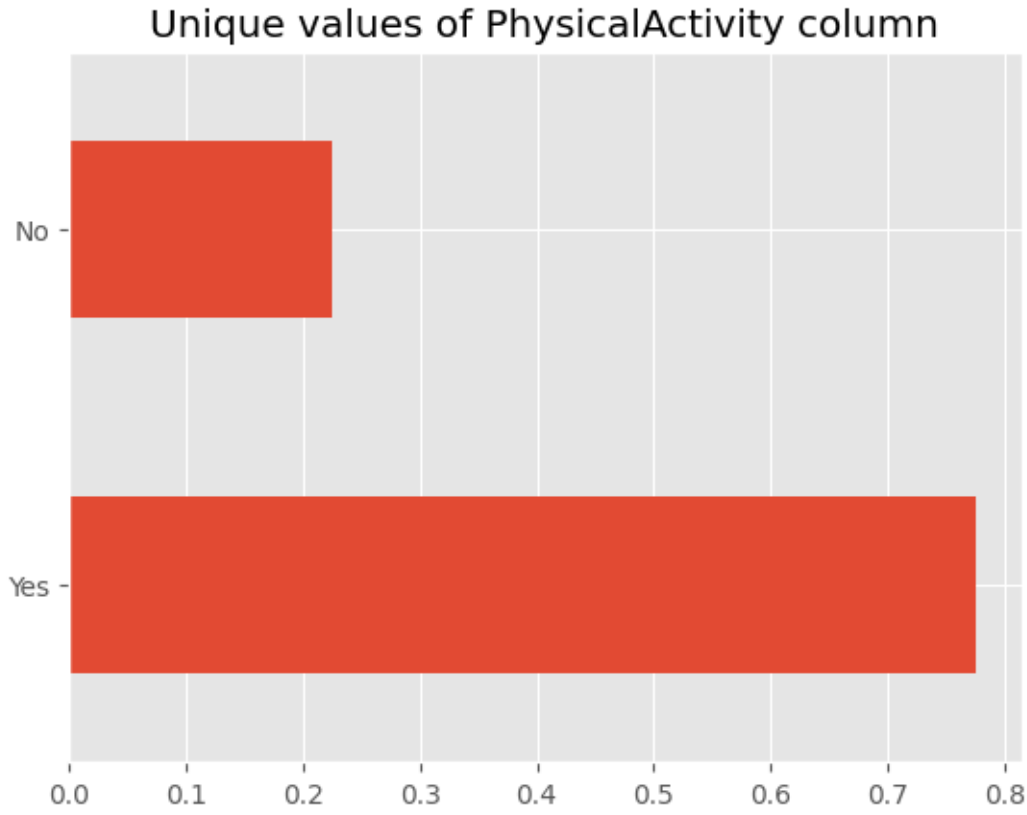

Unique values of Smoking column
Yes
No
0.0
0.1
0.2
0.3
0.4
0.5
0.6

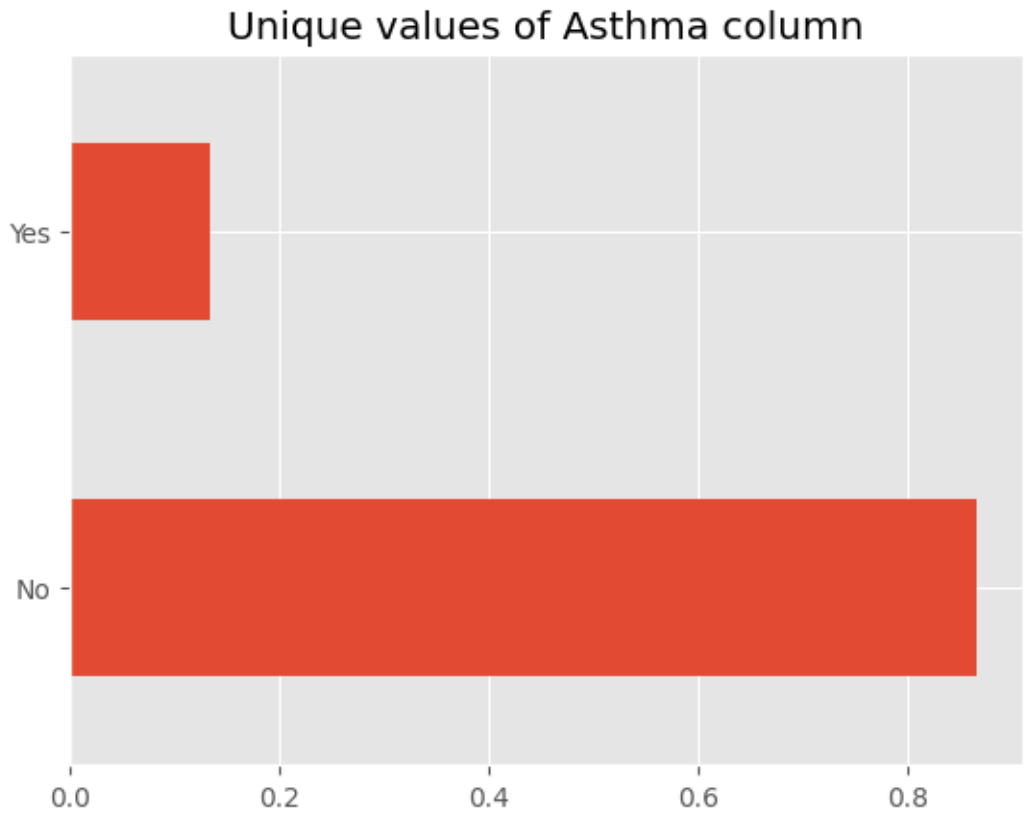

Unique values of AlcoholDrinking column
Yes
No
0.0
0.2
0.4
0.6
0.8

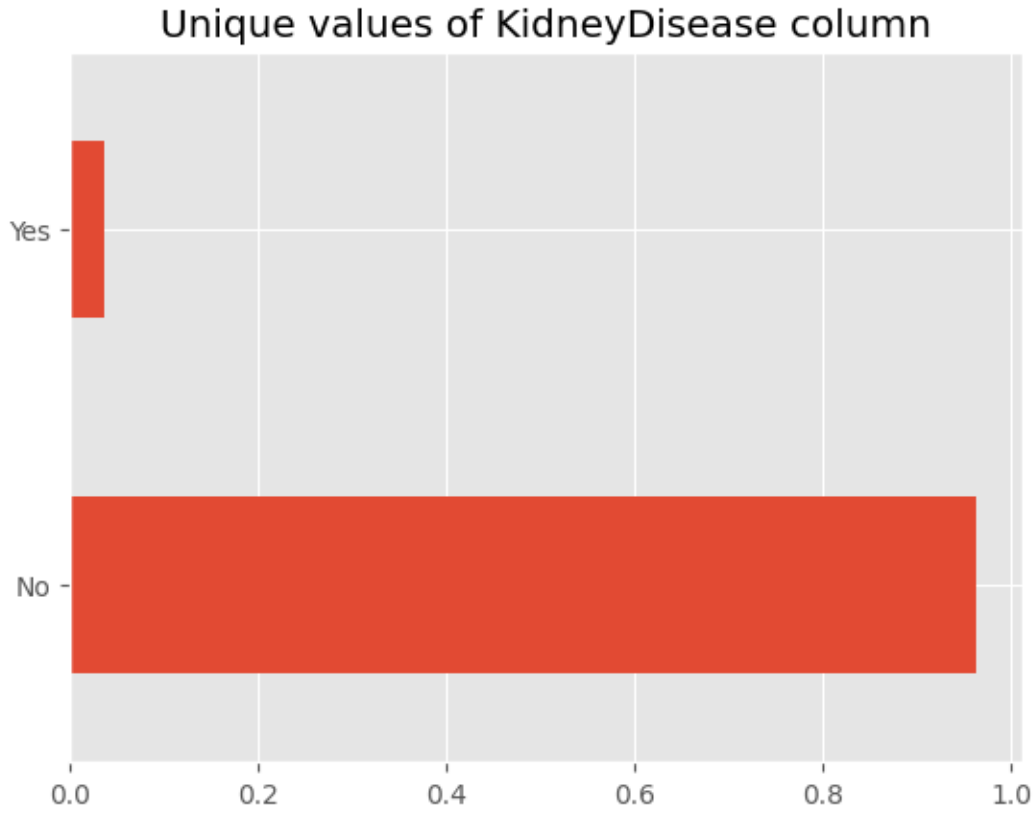

Unique values of Stroke column
Yes
No
0.0
0.2
0.4
0.6
0.8
1.0

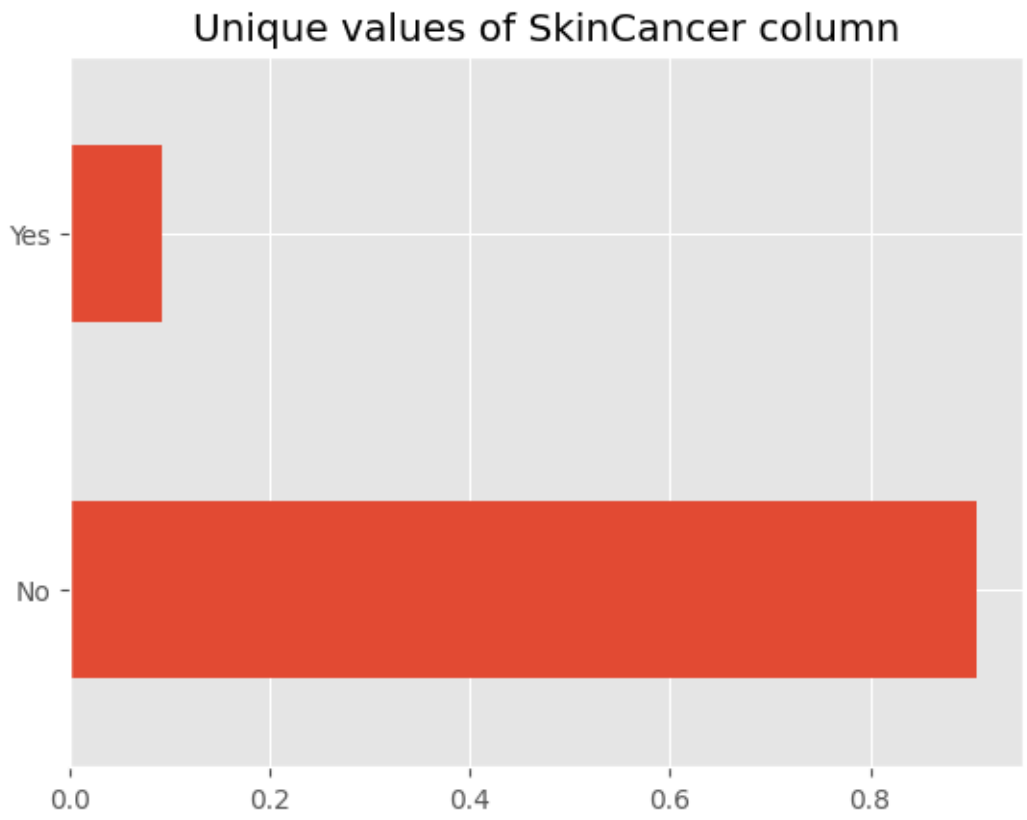

Unique values of DiffWalking column
Yes
No
0.0
0.2
0.4
0.6
0.8


Unique values of Sex column
Male
Female
0.0
0.1
0.2
0.3
0.4
0.5


Unique values of PhysicalActivity column
No
Yes
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


Unique values of Asthma column
Yes
No
0.0
0.2
0.4
0.6
0.8


Unique values of KidneyDisease column
Yes
No
0.0
0.2
0.4
0.6
0.8
1.0



From this we note the this dataset is imbalanced .i.e, data set with skewed class proportions. This is a common problem in classification problems. We will address this issue later in the notebook.

```
heart_disease.describe()
```

	BMI	PhysicalHealth	MentalHealth	SleepTime
count	319795.000000	319795.00000	319795.000000	319795.000000
mean	28.325399	3.37171	3.898366	7.097075
std	6.356100	7.95085	7.955235	1.436007
min	12.020000	0.00000	0.000000	1.000000
25%	24.030000	0.00000	0.000000	6.000000
50%	27.340000	0.00000	0.000000	7.000000
75%	31.420000	2.00000	3.000000	8.000000
max	94.850000	30.00000	30.000000	24.000000

```
sns.pairplot(heart_disease[
    ["HeartDisease", "BMI", "PhysicalHealth", "MentalHealth", "SleepTime"]
    ], hue="HeartDisease", plot_kws=dict(s=80, edgecolor="white", linewidth=2.5))
plt.show()
```

Surprisingly, the distribution of BMI, mental health and sleep time don't seem to be significantly different when comparing with people who **do or do not** have heart disease. However, the mental health do seem to have a strong correlation with heart disease.

CHAPTER

TWO

DATA VISUALIZATION

To familiarize ourself with the data, we generate plots that seek to illuminate some research questions.

1. Are people from certain backgrounds more likely to have heart disease, given their ethnicity?
2. Is the percentage of females and males with heart disease similar?
3. Do peoples *reported* physical activity correlate with their physical health?
4. Which risk factors are most correlated with heart disease?
5. What is the flow of patients from their age to substance misuse to if they have an aliment listed in the dataset to their likelihood of contracting heart disease?

```
def create_stacked_bar_hart(heart_disease: pd.DataFrame) -> plt.figure:
    """
    Args:
        heart_disease (pd.DataFrame): the original heart disease kaggle dataset

    Returns:
        plt.figure: a stacked bar chart of the percentage of people
            with heart disease and without heart disease, stratified by race
    """

    # top bar -> sum all values(HeartDisease=No and HeartDisease=Yes) to find y␣
↪position of the bars
    total: pd.DataFrame = heart_disease.groupby('Race').count().reset_index()

    # bar chart 1 -> top bars (group of 'HeartDisease=No')
    bar1: sns.barplot = sns.barplot(x="Race",  y="HeartDisease", data=total, color=
↪'red')

    # bottom bar ->  take only HeartDisease=Yes values from the data
    HeartDisease: pd.DataFrame = heart_disease[heart_disease.HeartDisease=='Yes'].
↪groupby('Race').count().reset_index()


    # bar chart 2 -> bottom bars (group of 'HeartDisease=Yes')
    bar2: sns.barplot = sns.barplot(x="Race", y="HeartDisease", data=HeartDisease,␣
↪errorbar=None,  color='green')

    # add legend
    top_bar: mpatches.Patch = mpatches.Patch(color='red', label='HeartDisease = No')
    bottom_bar: mpatches.Patch = mpatches.Patch(color='green', label='HeartDisease =␣
↪Yes')
    plt.legend(handles=[top_bar, bottom_bar])
```

(continues on next page)

(continued from previous page)

```
    plt.ylabel('Number of Patients')
    plt.xticks(rotation=45,ha='right')
    # show the graph
    plt.show()
```

```
def create_tree_diagram(heart_disease:pd.DataFrame) -> squarify.plot:
    """
    Args:
        heart_disease (pd.DataFrame): the original heart disease kaggle dataset

    Returns:
        squarify.plot: A tree diagram displaying the number of males and females
            that do and don't have heart disease
    """
    heart_disease = heart_disease.copy()
    tree_diagram_df: pd.DataFrame = heart_disease.groupby(['Sex', 'HeartDisease']).
↪count().reset_index()[['Sex','HeartDisease','BMI']]
    tree_diagram_df['Gender_choice'] = tree_diagram_df[['Sex', 'HeartDisease']].agg(',
↪ '.join, axis=1)
    tree_diagram_df.rename(columns={'BMI': 'count'}, inplace=True)
    squarify.plot(sizes=tree_diagram_df['count'], label=tree_diagram_df['Gender_choice
↪'], alpha=.4)
    plt.axis('off')
    plt.show()

def create_histplot(heart_disease:pd.DataFrame) -> plt.figure:
    """
    Args:
        heart_disease (pd.DataFrame): the original heart disease kaggle dataset

    Returns:
        plt.figure: Two histograms displaying the number of days in the past 30 days
                    in which patients were physically active.
                    Stratified by people's reported physical health
    """
    heart_disease = heart_disease.copy()
    sns.set(style="darkgrid")
    heart_disease = heart_disease[heart_disease.PhysicalHealth > 0]
    # plotting both distibutions on the same figure
    sns.histplot(heart_disease[heart_disease.PhysicalActivity == 'Yes'][
↪'PhysicalHealth'], color="skyblue", label="People who report doing physical activity
↪")
    sns.histplot(heart_disease[heart_disease.PhysicalActivity == 'No']['PhysicalHealth
↪'], color="red", label="People who do not physical activity")
    plt.legend()
    plt.xlabel("People who report that their physical health (physical illness and␣
↪injury) was not good? (0-30 days).")
    plt.show()
```

```
def create_corr_plot(heart_disease: pd.DataFrame):
    """
    Args:
        heart_disease (pd.DataFrame): the original heart disease kaggle dataset
```

(continues on next page)

(continued from previous page)

```
    Returns:
        squarify.plot: A bar chart displaying the correlation of risk factors vs ␣
↪heart disease
    """
    heart_disease = heart_disease.copy()
    binary_vars: List[str] = ["HeartDisease",
                        "Smoking",
                        "AlcoholDrinking",
                        "Stroke",
                        "DiffWalking",
                        "Diabetic",
                        "PhysicalActivity",
                        "Asthma",
                        "KidneyDisease",
                        "SkinCancer"]
    for binary_var in binary_vars:
        heart_disease[binary_var] = heart_disease[binary_var].replace({"Yes": 1, "No
↪": 0})
    heart_disease.Diabetic = heart_disease.Diabetic.replace({'No, borderline diabetes
↪': 1, 'Yes (during pregnancy)' : 1}).astype(int)
    heart_disease.corr()[['HeartDisease']].plot.bar(legend=False)
    plt.title("Correlations of HeartDisease variable vs risk factors")
```

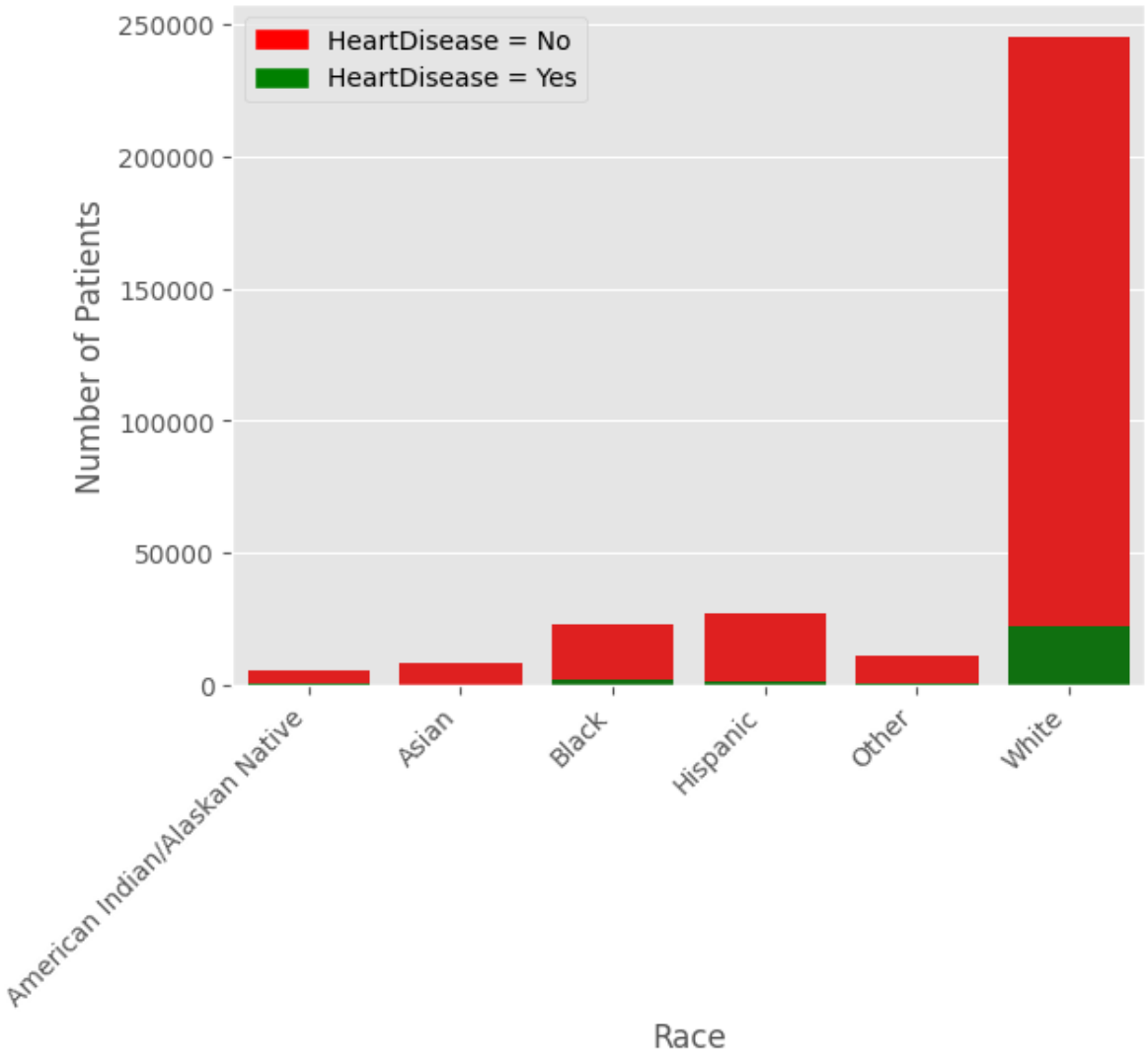
```
def create_sankey_chart(sankey_data:pd.DataFrame) -> go.Figure:
  """
  Args:
      heart_disease (pd.DataFrame): contains three columns; source target value.
                                    Indicates the amount of patients moving through␣
↪each stage of the sankey chart

  Returns:
      go.Figure: A sankey chart indicating the flow of patients in the following␣
↪pattern:
                    Age -> Substance Abuse -> Alinement -> Heart Disease outcome

  """
  sankey_data = sankey_data.copy()
  # Get unique values for both source and target columns
  unique_source_target_values  = list(pd.unique(sankey_data[['source', 'target']].
↪values.ravel('K')))

  source_target_mapping = {k:v for v,k in enumerate(unique_source_target_values)}
  with open("data/color_sankey_dict.json") as f:
      colour_dict= json.loads(f.read())
  links = sankey_data[['source', 'target', 'value']].copy()

  links["source"] = links["source"].map(source_target_mapping)
  links["target"] = links["target"].map(source_target_mapping)
  links_dict = links.to_dict(orient="list")

  with open('data/link_colours.pkl', 'rb') as f:
      link_colours = pickle.load(f)
```

(continues on next page)

(continued from previous page)

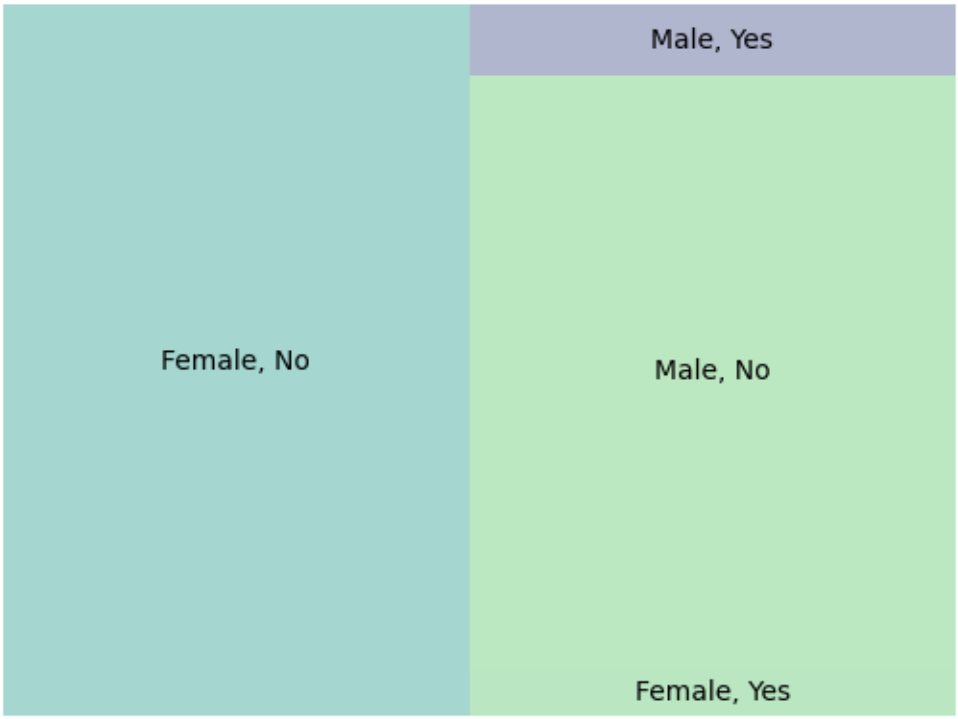
```
  fig = go.Figure(data = [go.Sankey(
      valueformat = ".0f",
      valuesuffix = "TWh",
      # Define nodes
      node = dict(
        pad = 15,
        thickness = 15,
        line = dict(color = "black", width = 0.2),
        label =  unique_source_target_values,
        color =  list(colour_dict.values())
      ),
      link = dict(
        source = links_dict["source"],
        target = links_dict["target"],
        value = links_dict["value"],
        color= link_colours,
      )

      )]
  )
  fig.update_layout(title_text="Patient Flow <br> Age -> Substance Abuse -> Alinement␣
↪-> Heart Disease outcome",
                    font_size=10)
  fig.show()
```

```
create_stacked_bar_hart(heart_disease)
```



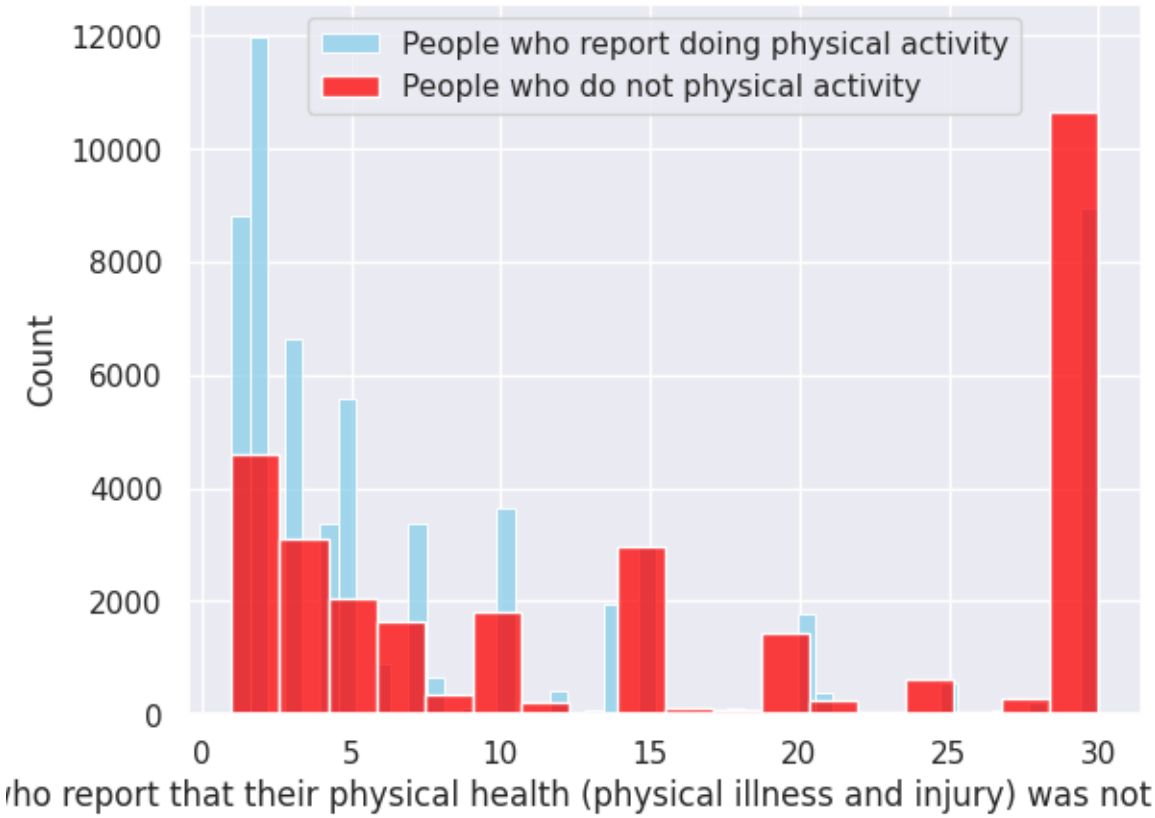
White people appear to both appear most often in this dataset and have the highest prevalence of heart disease in the their sub-population. The race with the second most heart disease prevalance is heart disease.

```
create_tree_diagram(heart_disease)
```



It is evident that relataive rate of heart disease is higher in men than in women.

```
create_histplot(heart_disease)
```



Surprisingly, it seems that peoples reported physical health does not necessarily align with their physical activity levels.

```
sankey_data: pd.DataFrame = pd.read_csv('data/full_sankey_data.csv')
create_sankey_chart(sankey_data)
```

2.1 Feature Engineering

A lot of our features contain categorical data. We will need to convert these to numerical values. Feature extraction projects the original high-dimensional features to a new feature space with low dimensionality, while feature selection directly selects a subset of relevant features. Both feature extraction and feature selection can improve learning performance, increase computational efficiency, decrease memory storage, and build better generalization models. However, as feature extraction creates a set of new features, further analysis is problematic as we cannot retain the physical meanings of these features. In contrast, by keeping some of the original features, feature selection maintains the physical meanings of the original features and gives models better readability and interpretability.

The columns broadly fall into three categories:

```
- Binary Categorical variables
  -  HeartDisease
  -  Smoking
  -  AlcoholDrinking
  -  Stroke
  -  DiffWalking
```

(continues on next page)

(continued from previous page)

```
  -  Diabet
  -  Physical Activity
  -  Asthma
  -  Kidney Disease
  -  Skin Cancer
  -  Sex
- Continuous variables
  - BMI
- Discrete variables
  - Sleeptime
  - PhsycialHealth
  - MentalHealth
- Polytomous variables; these are variables with more than two categories
  - AgeCategory
  - Race
  - GenHealth
```

Binary variables with yes or no values are converted to 1 and 0 respectively. The diabetes column currently has four categories: 'Yes', 'No', 'No, borderline diabetes', 'Yes (during pregnancy)'. Our earlier analysis revealed that the presence of heart dieasease and diabetes are weakly correlated. Therefore, we will combine the 'Yes (during pregnancy)' and 'No, borderline diabetes' categories into the 'Yes' category.

```
heart_disease['Diabetic'].unique()
```

```
array(['Yes', 'No', 'No, borderline diabetes', 'Yes (during pregnancy)'],
      dtype=object)
```

```
binary_vars: List[str] = ["HeartDisease",
                          "Smoking",
                          "AlcoholDrinking",
                          "Stroke",
                          "DiffWalking",
                          "Diabetic",
                          "PhysicalActivity",
                          "Asthma",
                          "KidneyDisease",
                          "SkinCancer"]
```

```
for binary_var in binary_vars:
    heart_disease[binary_var] = heart_disease[binary_var].replace({"Yes": 1, "No": 0})
heart_disease.Diabetic: pd.Series = heart_disease.Diabetic.replace({'No, borderline␣
↪diabetes': 1, 'Yes (during pregnancy)' : 1}).astype(int)
```

```
# One-hot encoding
sex_one_hot = pd.get_dummies(heart_disease['Sex'])
heart_disease = heart_disease.join(sex_one_hot)
heart_disease.drop(columns=['Sex'], inplace=True)
```

In the dataset, ordinal attributes are present. There is a natural ordering to the categories (i.e. GenHealth, AgeCategory). Thus, we use the `replace()` function to label encode them.

```
heart_disease["AgeCategory"].replace({
    '18-24':1,
    '25-29':2,
    '30-34':3,
    '35-39':4,
    '40-44':5,
    '45-49':6,
    '50-54':7,
    '55-59':8,
    '60-64':9,
    '65-69':10,
    '70-74':11,
    '75-79':12,
    '80 or older':13
}, inplace=True)
```

```
heart_disease["GenHealth"].replace({
    'Poor': 1,
    'Fair':2,
    'Good':3,
    'Very good':4,
    'Excellent':5},
    inplace=True)
```

Race has no order. Therefore, we decide to take two approaches.

- One-hot-encoding:Increase the dimensions to facilite a better feature selection space step later which produces a final lower dimension feature vector (i.e. every race is a binary feature)
- Frequency encoding: Frequency Encoding is an encoding technique that encodes categorical feature values to their respected frequencies. This will preserve the information about the values of distributions. We normalize the frequencies that result in getting the sum of unique values as 1. This is done to avoid the curse of dimensionality and reduce the sparsity in the dataset.

We will evaluate both approaches in our model evaluation.

```
# One-hot encoding
race_one_hot: pd.DataFrame = pd.get_dummies(heart_disease['Race'])
heart_disease: pd.DataFrame = heart_disease.join(race_one_hot)

# Frequency encoding using value_counts function
race_freq: pd.Series = heart_disease['Race'].value_counts(normalize=True)

# Mapping the encoded values with original data
heart_disease['Race_freq']: pd.Series = heart_disease['Race'].apply(lambda x : race_
 ↪freq[x])
heart_disease.drop(columns=['Race'], inplace=True)
```

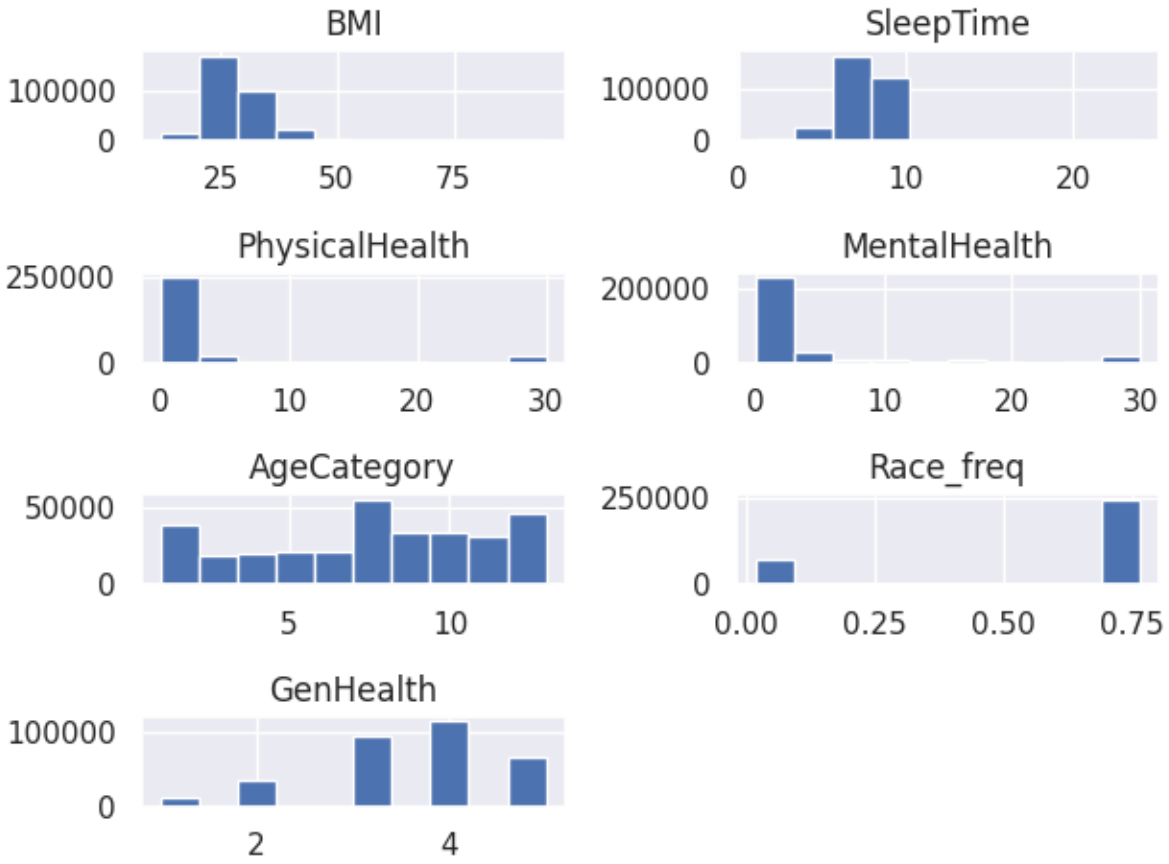
Outliers are extreme values. Outliers can skew the distribution but these are exceptional but genuine data points. Such distributions can impact certain algorithms such as regression type models (Lasso), k Nearest neighbors and Naive Bayes. However, decision trees and its ensemble (random forest) are not impacted.

```
heart_disease.hist(column=["BMI",
     "SleepTime",
     "PhysicalHealth",
     "MentalHealth",
```

(continues on next page)

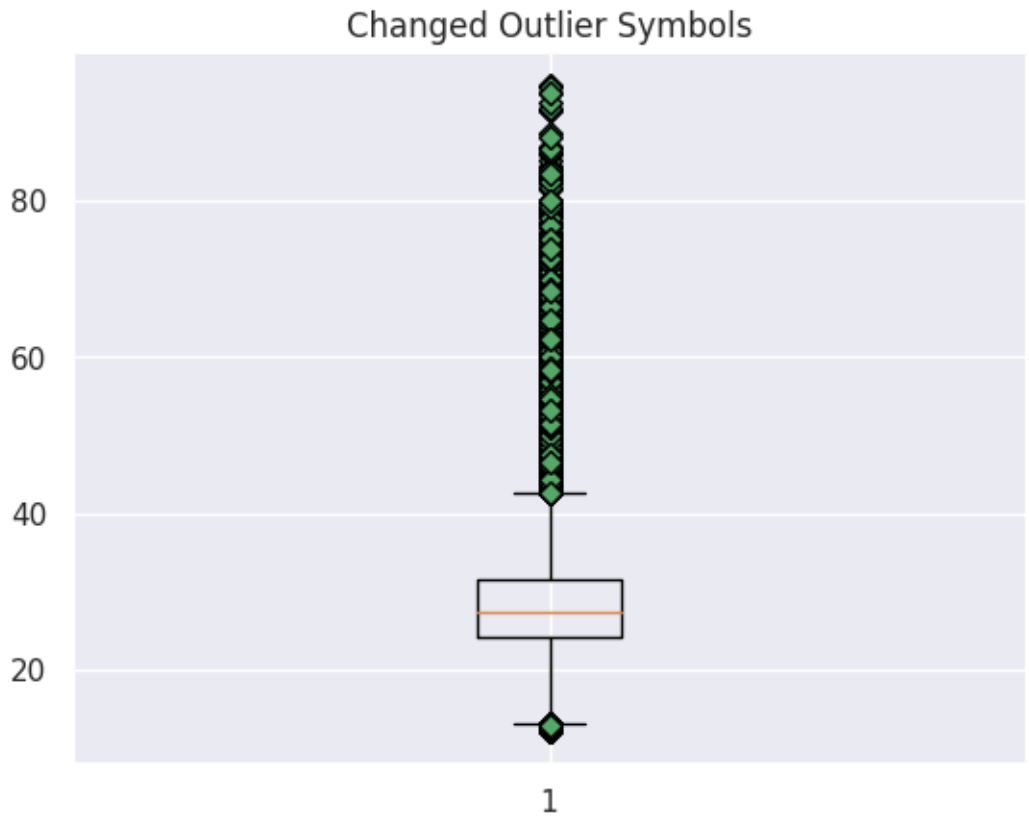
(continued from previous page)

```
     "AgeCategory",
     "Race_freq",
     "GenHealth"], layout=(4,2))
plt.tight_layout()
```




```
green_diamond: Dict[str, str] = dict(markerfacecolor='g', marker='D')
fig1, ax1 = plt.subplots()
ax1.set_title('Changed Outlier Symbols')
ax1.boxplot(heart_disease["BMI"], flierprops=green_diamond)
```

```
{'whiskers': [<matplotlib.lines.Line2D at 0x7f681586d130>,
  <matplotlib.lines.Line2D at 0x7f681579d100>],
 'caps': [<matplotlib.lines.Line2D at 0x7f68157f6460>,
  <matplotlib.lines.Line2D at 0x7f681575d460>],
 'boxes': [<matplotlib.lines.Line2D at 0x7f6815835eb0>],
 'medians': [<matplotlib.lines.Line2D at 0x7f6813023d00>],
 'fliers': [<matplotlib.lines.Line2D at 0x7f68109ae490>],
 'means': []}
```



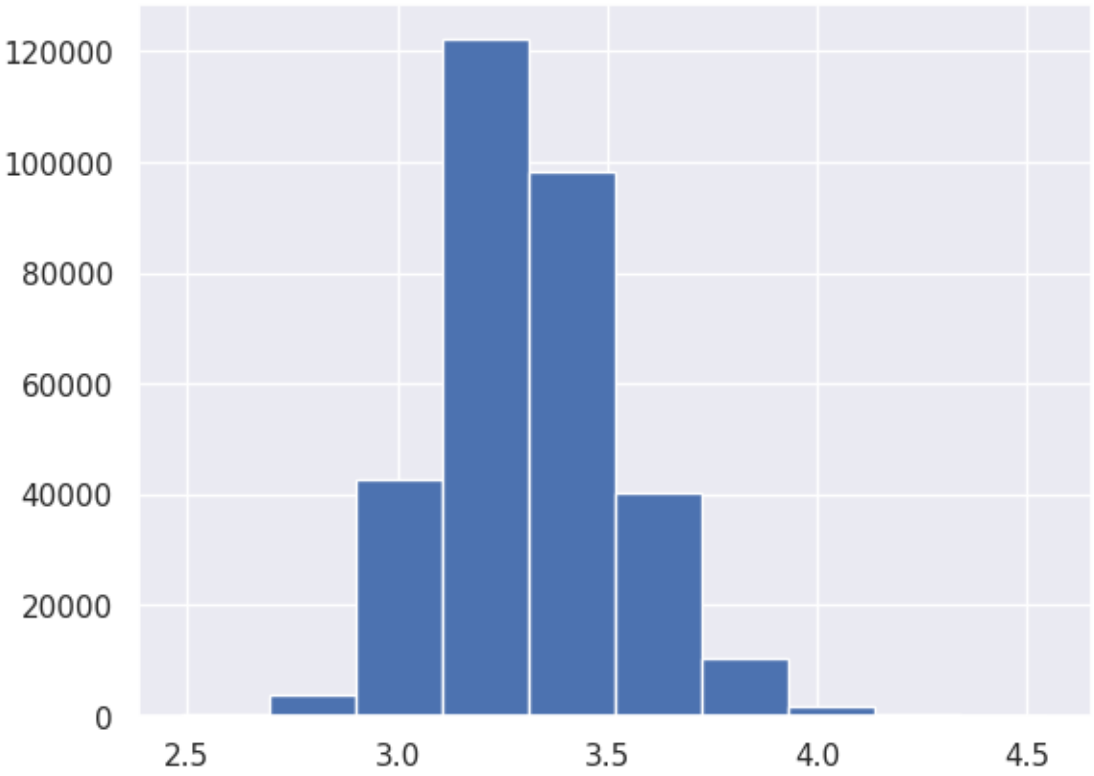
We note that many outliers exist in the BMI column. To rectify this, we take two approaches

- **Turn it into a ordinal variable**: The WHO uses BMI as a convenient rule of thumb used to broadly categorize a person as underweight, normal weight, overweight, or obese based on tissue mass (muscle, fat, and bone) and height. Major adult BMI classifications are underweight (under 18.5 kg/m2), normal weight (18.5 to 24.9), overweight (25 to 29.9), and obese (30 or more). We will encode these describe caegories into ordinal values.
- **Take log(x) of the BMI column**: Taking the log of a feature is a common trick to reduce the effect of outliers. This is because the log function is monotonically increasing. Therefore, the effect of outliers is reduced. Again, both aproaches will be evaluated in our model evaluation.

```
bins: List[int] = [0, 18.5, 24.9, 29.9, np.inf]
names: List[int] = [1, 2, 3, 4]
heart_disease['BMI_Bin']: pd.Series = pd.cut(heart_disease['BMI'], bins, labels=names)
```

```
heart_disease["LOG_BMI"]: pd.Series = np.log(heart_disease["BMI"])
heart_disease["LOG_BMI"].hist()
```

```
<AxesSubplot: >
```



2.1.1 PCA

Standardization is typically used for features of incomparable units. E.g. someone reporting the number of times they were physical active in the last thirty days and BMI. Standardisation will be applied to all features (both orginal and engineered) except the binary variables. We will also standardize the features due to k-means "isotropic" nature. In this case, if we left our variances unequal; we would inversely be putting more weight on features with high variance. In addition, we will perform principal component analysis due to avoid the curse of dimensionality that k-means can suffer from. The function of PCA is to reduce the dimensionality of a data set consisting of many variables correlated with each other, either heavily or lightly, while retaining the variation present in the data set to the maximum extent.

The same is done by transforming the variables (i.e. features) to a new set of variables, which are known as the principal components, ordered such that the retention of variation present decreases as we move down the order of components.

In addition, we will perform principal component analysis due to avoid the curse of dimensionality that some algorithims can suffer from. Initally, PCA is only perfoormed on the *features present within the orginal dataset*. Later on, we will perform A feature extraction method u by applying different subsets of training data to estimate the accuracy of these subsets for all used classifiers and measure the quality of the generated subsets per classification algorithm, and the results of the classifier are shown. We plan to use PCA again for those best performing subets to see if any improvement is made.

The function of PCA is to reduce the dimensionality of a data set consisting of many variables correlated with each other, either heavily or lightly, while retaining the variation present in the data set to the maximum extent.

The same is done by transforming the variables (i.e. features) to a new set of variables, which are known as the principal components, ordered such that the retention of variation present decreases as we move down the order of components.

The procedure of PCA involves five steps:

1. Standardize the data
2. Compute covariance matrix

3. Identify the eigenvalues and eigenvectors of the covariance matrix and order them according to the eigenvalues
4. Compute a feature vector
5. Recast the data

Standardisation

We now standardize the data using the following formulae:

$$X_i = X_i - \bar{X} \qquad X_i = \frac{X_i}{\sigma}$$

The standard deviation should equal 1 after standardization

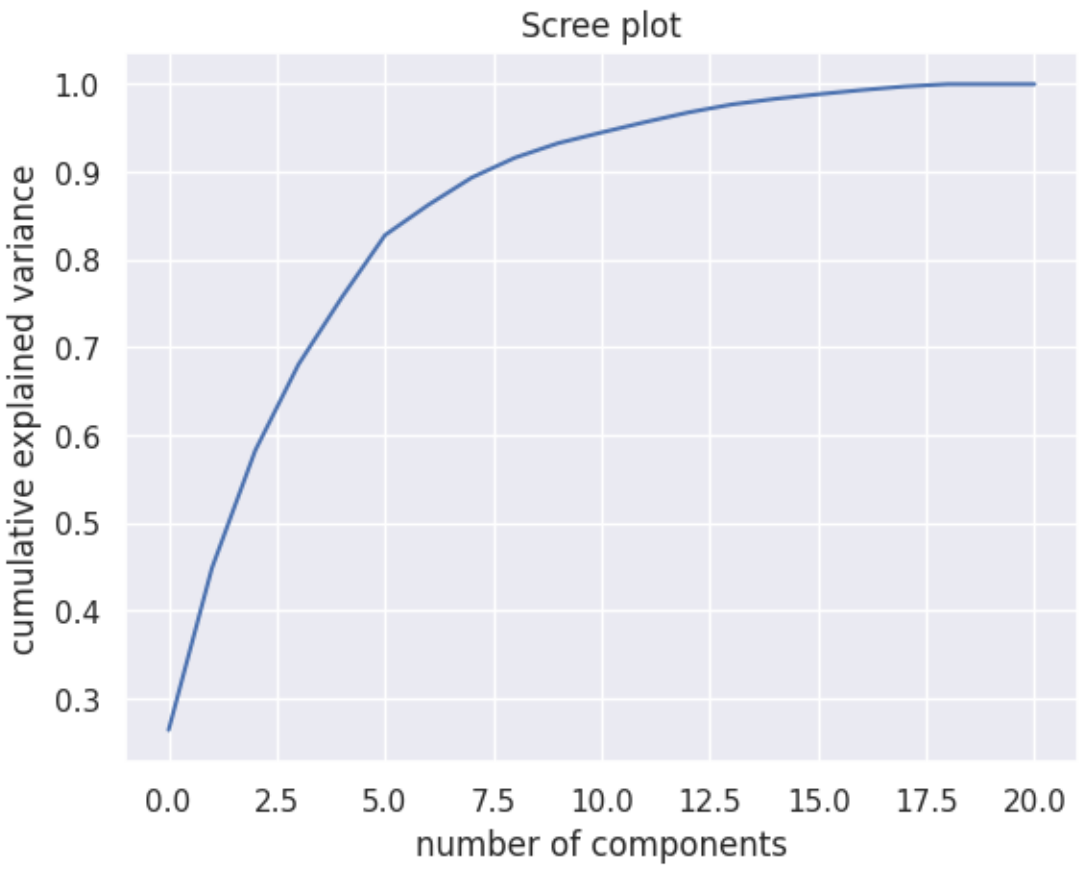
```
heart_disease[
    ['PhysicalHealth', 'MentalHealth', 'AgeCategory', 'GenHealth','SleepTime', 'BMI_
↪Bin']
    ] = StandardScaler().fit_transform(heart_disease[['PhysicalHealth', 'MentalHealth
↪', 'AgeCategory', 'GenHealth','SleepTime', 'BMI_Bin']])
```

```
heart_disease.to_csv('data/standardised_heart_disease.csv', index=False)
```

We will use the `PCA` function supplied by the `Scikit-learn` library for dimensionality reduction. But how do we find the optimal number of components? Which eigenvalues are important? The scree plot below describes the cumulative explained variance for each component. We reach 80% explained variance at the three component mark.

```
orginal_features_df: List[str] = heart_disease[[ 'Smoking', 'AlcoholDrinking', 'Stroke
↪',
       'PhysicalHealth', 'MentalHealth', 'DiffWalking', 'AgeCategory',
↪'PhysicalActivity', 'GenHealth', 'SleepTime', 'Asthma',
       'KidneyDisease', 'SkinCancer', 'Female', 'Male',
       'American Indian/Alaskan Native', 'Asian', 'Black', 'Hispanic', 'Other',
       'White']]
```

```
pca = PCA().fit(orginal_features_df)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')
plt.title('Scree plot')
plt.show()
```



We note a slight indent at the the 5th principal compent mark. According to the average-eigenvalue test (Kaiser-Guttman test) we should retain only those eigenvalues that are above the average which is 1.0. Jolliffe relaxes this criterium and suggest to retain eigenvalues greater than 0.7.

```
kasier_criterion: np.int64 = np.where(pca.explained_variance_ > 1)[-1][-1]
print(
        f'Kasier criterion optimal component number: {kasier_criterion}, explained␣
 ↪variance: {np.cumsum(pca.explained_variance_ratio_)[kasier_criterion]}'
    )
jolliffe_criterion: np.int64  = np.where(pca.explained_variance_ > 0.7)[-1][-1]
print(
    f'Jolliffe criterion optimal component number: {jolliffe_criterion} , explained␣
 ↪variance: {np.cumsum(pca.explained_variance_ratio_)[jolliffe_criterion]}'
    )
```

```
Kasier criterion optimal component number: 1, explained variance: 0.
 ↪4486201019244787
Jolliffe criterion optimal component number: 2 , explained variance: 0.
 ↪5825444541168093
```

For the purpose of this investigation, we decide to go with **both** the Jolaliffe criterion, we will retain the first two components.

Finally, we fit the `pca` model with the dataframes containing top 2 components , apply the dimensionality reduction on those respective dataframe and save the resulting dataframes.

```
pca_2d = PCA(n_components=2)
dim_reduced_2d: pd.DataFrame = pca_2d.fit_transform(orginal_features_df)
dim_reduced_2d = pd.DataFrame(data=dim_reduced_2d, columns=[f'component_{num}' for␣
↪num in range(1,3)])
dim_reduced_2d = pd.merge(
        heart_disease[['HeartDisease']], dim_reduced_2d, left_index=True, right_
↪index=True
        )
dim_reduced_2d.to_csv("data/dim_reduced_2d.csv", index=False)
```

Note, later on we will use PCA to identify the variables that contribute most to the variation in the dataset.

CHAPTER

THREE

CLUSTER ANALYSIS

The purpose of our cluster analysis is to:

- Measure clustering & central tendency.
- Perform k-means
- Evaluate the clusters, **particularly**:
  - does the dataset naturally cluster into people who do and do not have heart disease?
  - people with alchol abuse issues
  - underweight vs normal weight vs overweight vs obese

3.1 Import libaries

3.1.1 Data Processing

```
import pandas as pd
```

3.1.2 Scientific computing

```
from sklearn.neighbors import NearestNeighbors
from random import sample
from numpy.random import uniform
import numpy as np
from math import isnan
from sklearn.preprocessing import StandardScaler
```

3.1.3 Clustering

```
from sklearn.cluster import KMeans, MiniBatchKMeans
from sklearn.metrics import silhouette_score
from pyclustertend import ivat
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In [3], line 3
      1 from sklearn.cluster import KMeans, MiniBatchKMeans
      2 from sklearn.metrics import silhouette_score
----> 3 from pyclustertend import ivat

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/
 ↪pyclustertend/__init__.py:6
      1 from .hopkins import hopkins  # noqa: F401
      2 from .metric import (  # noqa: F401
      3     assess_tendency_by_mean_metric_score,
      4     assess_tendency_by_metric,
      5 )
----> 6 from .visual_assessment_of_tendency import (  # noqa: F401
      7     vat,
      8     compute_ordered_dissimilarity_matrix,
      9     ivat,
     10     compute_ivat_ordered_dissimilarity_matrix,
     11 )

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/
 ↪pyclustertend/visual_assessment_of_tendency.py:6
      4 import numpy as np
      5 from sklearn.metrics import pairwise_distances
----> 6 from numba import njit
      9 def vat(data: np.ndarray, return_odm: bool = False, figure_size: Tuple =␣
 ↪(10, 10)):
     10     """VAT means Visual assessment of tendency. basically, it allow to␣
 ↪asses cluster tendency
     11     through a map based on the dissimilarity matrix.
     12
   (...)
     32
     33     """

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/numba/__
 ↪init__.py:198
    195     return False
    197 _ensure_llvm()
--> 198 _ensure_critical_deps()
    200 # we know llvmlite is working as the above tests passed, import it now as␣
 ↪SVML
    201 # needs to mutate runtime options (sets the `-vector-library`).
    202 import llvmlite

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/numba/__
 ↪init__.py:138, in _ensure_critical_deps()
    136     raise ImportError("Numba needs NumPy 1.17 or greater")
    137 elif numpy_version > (1, 20):
--> 138     raise ImportError("Numba needs NumPy 1.20 or less")
    140 try:
    141     import scipy

ImportError: Numba needs NumPy 1.20 or less
```

3.1.4 Data Visualisation

```
import matplotlib.pyplot as plt
import matplotlib
from yellowbrick.cluster import SilhouetteVisualizer
import seaborn as sns
```

3.2 Measure Cluster Tendency

Clustering algorithms such as k-means are used to determine the structure of multi-dimensional data. Clusters are disjoint natural groups. However, K-means will find clusters in data even if none "actually" exist. Therefore, a fundamental question before applying any clustering algorithms is: Are clusters present at all? We will measure the clustering tendency of both datasets before subjecting it to k-means. These datasets contain the top **two principal components (2D)**. To do this, we employ

- Hopkins's statistic of randomness

3.2.1 Hopkins statistics

Hopkins statistics [Banerjee and Dave, 2004] tests the spatial randomness of a dataset i.e. it measures the probability that a given dataset aligns with a uniform distribution. It is based on the difference between the distance from a real point to its nearest neighbour, U, and the distance from a uniformly generated point within the data space to the nearest real data point, W.

- H_0: The dataset **is** uniformly distributed
- H_1: The dataset **is not** uniformly distributed

$$H = \frac{\sum_{i=1}^{m} u_i^d}{\sum_{i=1}^{m} u_i^d + \sum_{i=1}^{m} w_i^d}$$

If the value of the Hopkins statistic(H) is close to 1 (above 0.5), we reject H_0 and can conclude that the dataset is considered significantly clusterable. Otherwise, we fail to reject H_0 and can conclude that the dataset is considered significantly uniformly distributed.

```
def hopkins(X):
    """
    Hopkins statistic. Code snippet:
        https://matevzkunaver.wordpress.com/2017/06/20/hopkins-test-for-cluster-
↪tendency/
    """
    d = X.shape[1]
    n = len(X)
    m = int(0.1 * n)
    nbrs = NearestNeighbors(n_neighbors=1).fit(X.values)

    rand_X = sample(range(0, n, 1), m)

    ujd = []
    wjd = []
    for j in range(0, m):
        u_dist, _ = nbrs.kneighbors(uniform(np.amin(X,axis=0),np.amax(X,axis=0),d).
↪reshape(1, -1), 2, return_distance=True)
```

(continues on next page)

(continued from previous page)

```
        ujd.append(u_dist[0][1])
        w_dist, _ = nbrs.kneighbors(X.iloc[rand_X[j]].values.reshape(1, -1), 2,␣
↪return_distance=True)
        wjd.append(w_dist[0][1])

    H = sum(ujd) / (sum(ujd) + sum(wjd))
    if isnan(H):
        print(ujd, wjd)
        H = 0

    return H
```

```
dim_reduced_2d = pd.read_csv('data/dim_reduced_2d.csv')
# Get labels as defined in the markdown cell above to compare how k-means cluster␣
↪patients
original_heart_disease = pd.read_csv('data/heart_2020_cleaned.csv')
bins = [0, 18.5, 24.9, 29.9, np.inf]
names = ["UnderWeught", "NormalWeight", "Overweight", "Obese"]
dim_reduced_2d['BMI_Bin'] = pd.cut(original_heart_disease['BMI'], bins, labels=names)
dim_reduced_2d['AlcoholDrinking'] = original_heart_disease['AlcoholDrinking']
dim_reduced_2d['BMI_Bin'] = original_heart_disease['Smoking']
```

For both datasets, we reject H_0 and can conclude that the dataset has a significant tendency to cluster.

```
print(f"2D's hopkins statistic {hopkins(dim_reduced_2d.iloc[:,1:-2])}")
```

```
2D's hopkins statistic 0.9914319273681726
```

3.3 K-Means

K-means is a common clustering algorithm. Although a simple clustering algorithm, it has vast application areas, including customer segmentation and image compression. K-means is a centroid based algorithm that aims to minimize the sum of distances between the points and their respective cluster centroid. The main steps of this algorithm are:

- **Step 1**: Choose the number (k) of clusters
- **Step 2**: Select k random points, which will become the initial centroids
- **Step 3**: Assign all data points to the nearest centroid.
- **Step 4**: Compute the centroid of the newly formed clusters by taking the mean of data instances currently associated with that cluster.
- **Step 5**: Repeat steps 3 and 4 until either:
  - Centroids of newly formed clusters do not change
  - Points remain in the same cluster
  - Maximum number of iterations are reached

We utlise the `MiniBatchKMeans()` from the sckit-learn package, given the largness of this dataset.

But how do we find the optimal number of clusters?

- Elbow method

- Silhouette coefficient

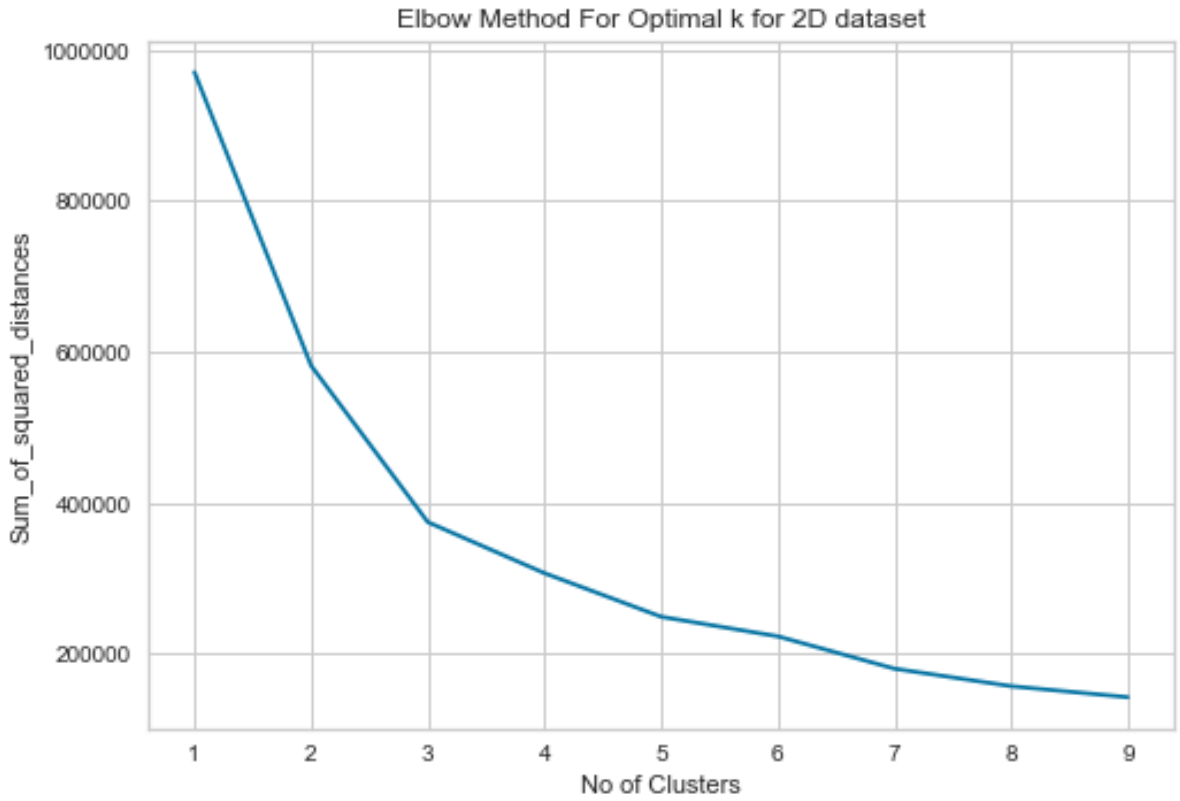
3.3.1 Elbow method

The Elbow method calculates the error or 'distortion' between the data points (y_i) and their corresponding centroid ($_i$) of N data points for k clusters where k ∈ {1…10}. The error metric used is the Sum of Squared Error (SSE):

$$SSE = \sum_{i=1}^{N} (y_i - {}_i)^2$$

We plot these values in an attempt to find an 'elbow' within the curve.

```
Sum_of_squared_distances = []
for k in range(1, 10):
    km_2d = MiniBatchKMeans(n_clusters=k, random_state=42)
    km_2d = km_2d.fit(dim_reduced_2d.iloc[:,1:-2])
    Sum_of_squared_distances.append(km_2d.inertia_)
plt.plot(range(1, 10), Sum_of_squared_distances, 'bx-')
plt.xlabel('No of Clusters')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k for 2D dataset')
plt.show()
```



We can see that the optimal number of clusters occur at k=2. A more suitable dip is noted at k-4.

3.4 Sillhoute method

This method is another method of finding the correct number of clusters(k). Silhouette coefficient for a particular data point (i) is defined as:

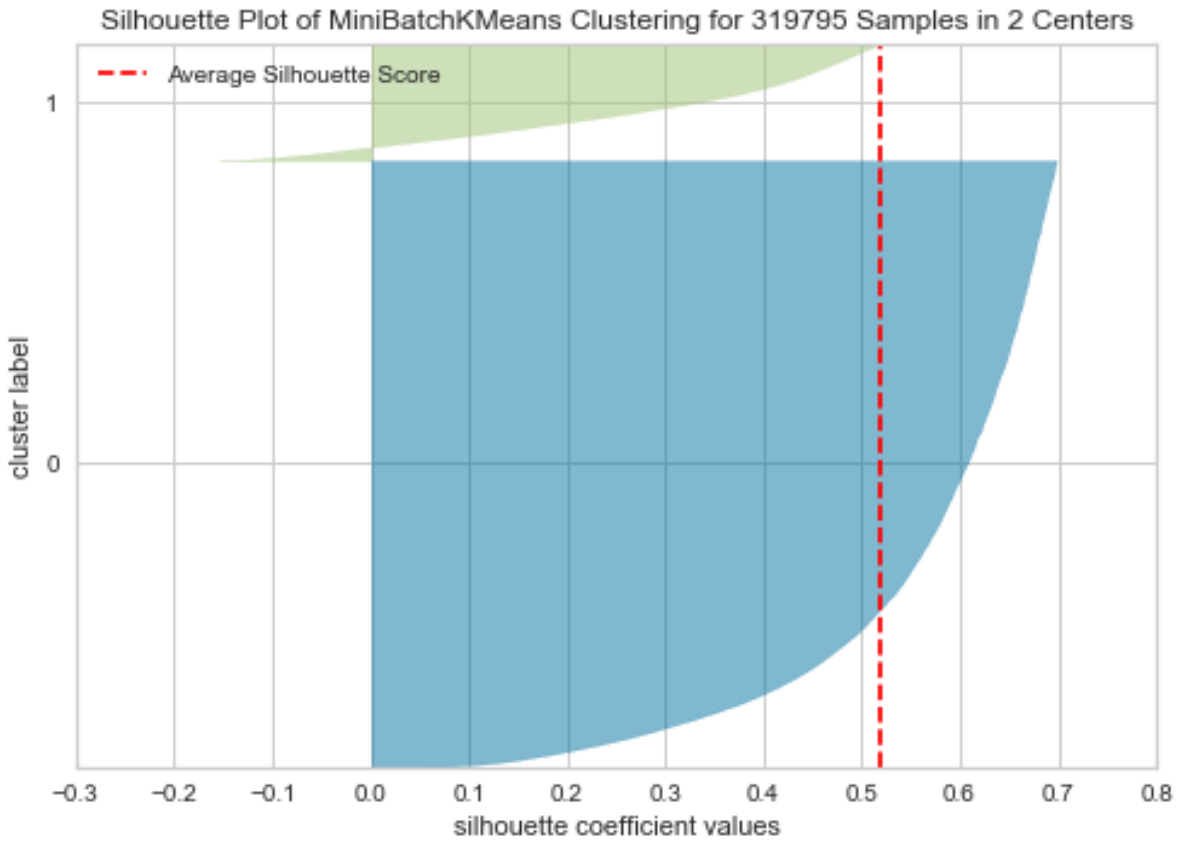
$$s_i = \frac{b_i - a_i}{max(b_i, a_i)}$$

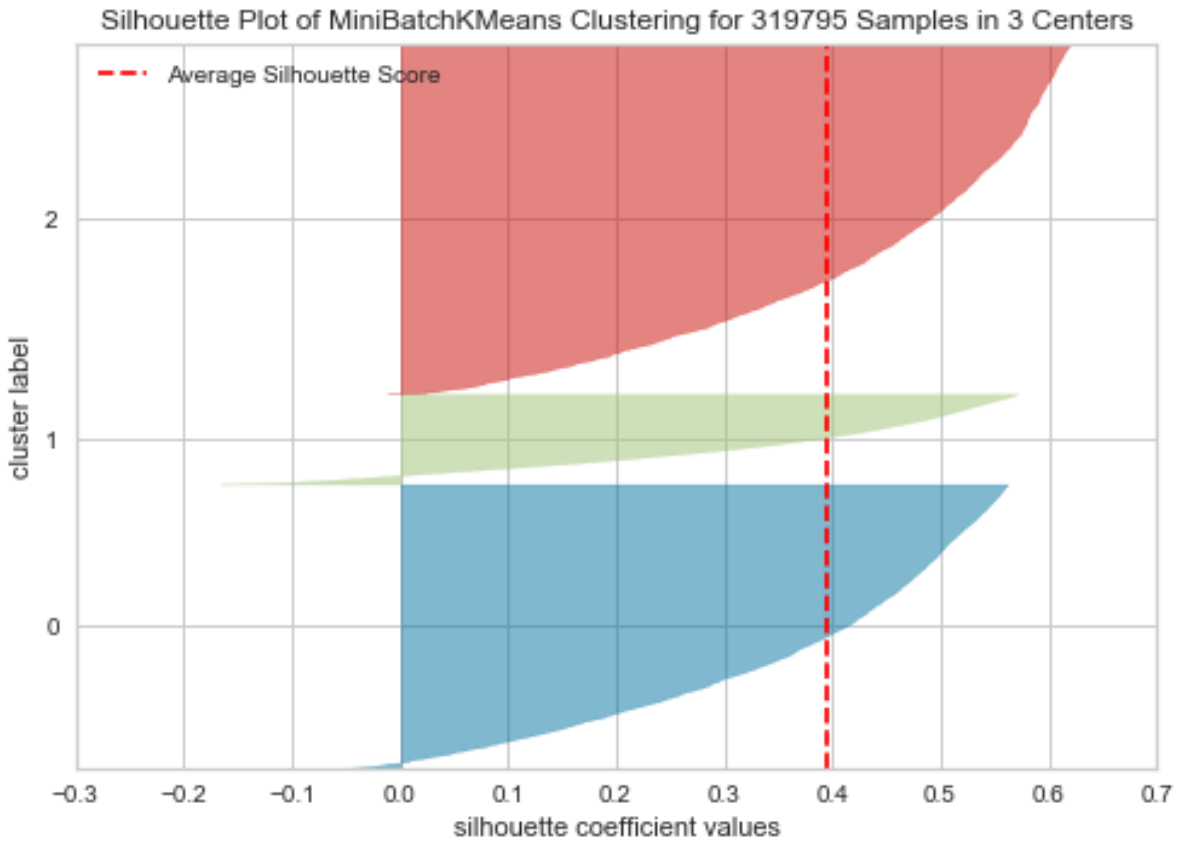
where:

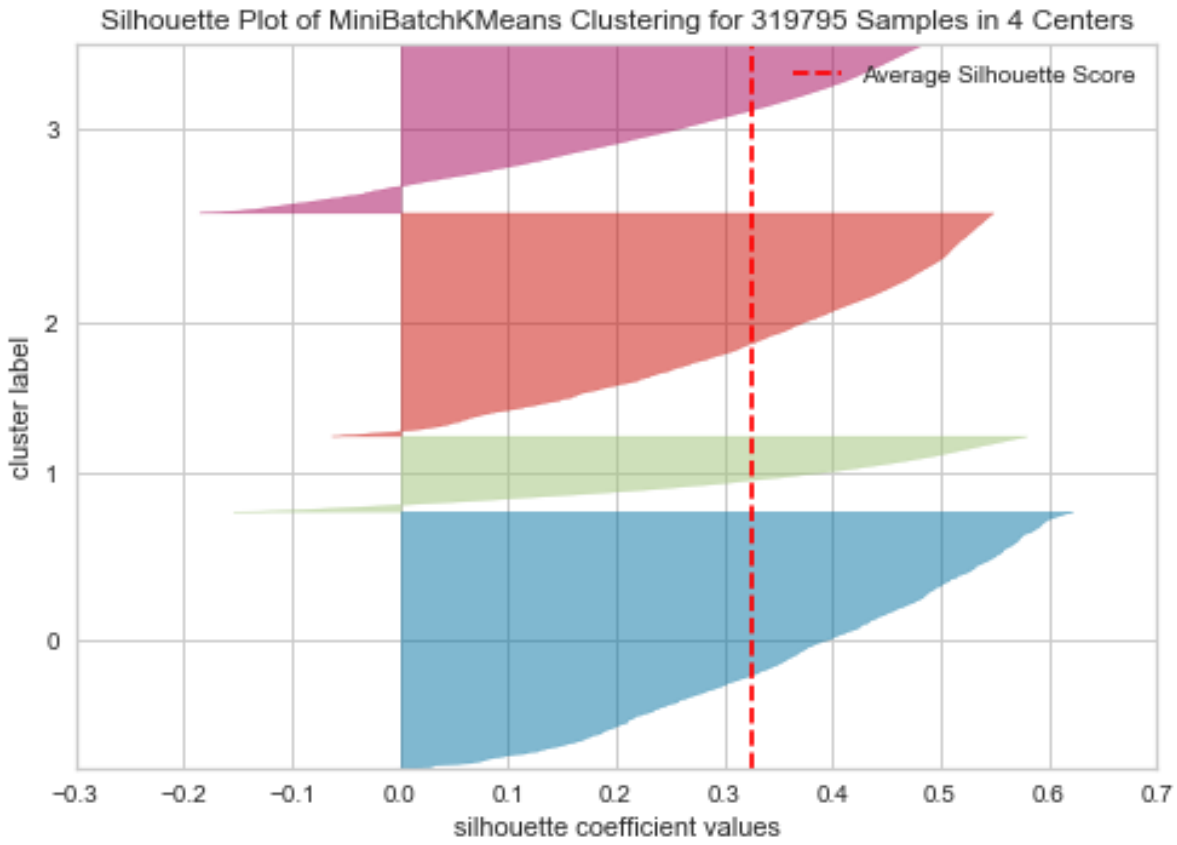
- s_i: the silhouette coefficient, ranging from -1 to 1. A score of 1 (the best) means that data point i is compact in its cluster and far away from other clusters. Conversely, the worst value is -1, while values near 0 denote overlapping clusters.
- b_i: average distance between i and all the other data points in its cluster.
- a_i: minimum average distance from i to all clusters to which i does not belong to

We evaluate using silhouette plots. These plots display how close each point in one cluster is to points in the neighbouring clusters.

```
from yellowbrick.cluster import SilhouetteVisualizer
for k in  range(2,5):
    km = MiniBatchKMeans(n_clusters=k, random_state=42)
    visualizer = SilhouetteVisualizer(km, colors='yellowbrick')
    visualizer.fit(dim_reduced_2d.iloc[:,1:-2])
    visualizer.show()
```




Silhouette Plot of MiniBatchKMeans Clustering for 319795 Samples in 3 Centers
Average Silhouette Score
cluster label
2
1
0
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
silhouette coefficient values



The results of the silhoute analysis are more ambiguous. K=2 seem to be sub-optimal due to wide fluctuations in size of the silhouette plot. However, the fluctuation at k=4 seems to be more uniform compared to 2. Thus, we select the optimal number of clusters as 4.

3.5 Findings

As mentioned previously, clusters can be considered as disjoint groups. In this context, these clusters seek to represent people with similar latent physiological processes and/or possible health statuses. We attempt to relate the groupings to the health disease status of individual pateints.

3.5.1 Top 2 principal component dataset

Healthy vs Unhealthy

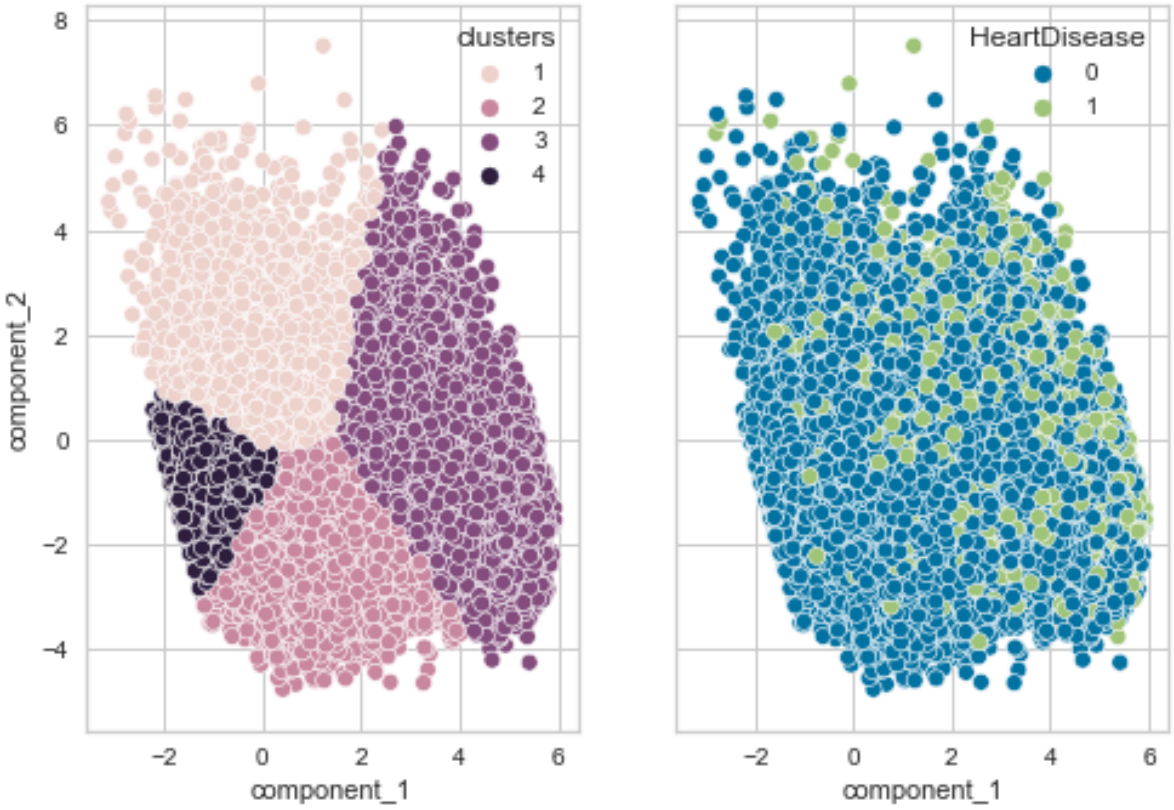
```
km_2d = KMeans(n_clusters=4, random_state=42).fit(dim_reduced_2d.iloc[:,1:-2])
dim_reduced_2d["clusters"] = km_2d.labels_ + 1
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
sns.scatterplot(x='component_1', y='component_2', data=dim_reduced_2d, hue='clusters',
 ↪ ax=ax1)
sns.scatterplot(x='component_1', y='component_2', data=dim_reduced_2d, hue=
 ↪'HeartDisease', ax=ax2)
results_df = dim_reduced_2d[['HeartDisease', 'clusters']].set_index('HeartDisease')
```

(continues on next page)

(continued from previous page)

```
results_df = results_df.apply(pd.Series.value_counts, axis=1).groupby(by=[
 ↪"HeartDisease"]).sum()
results_df
```

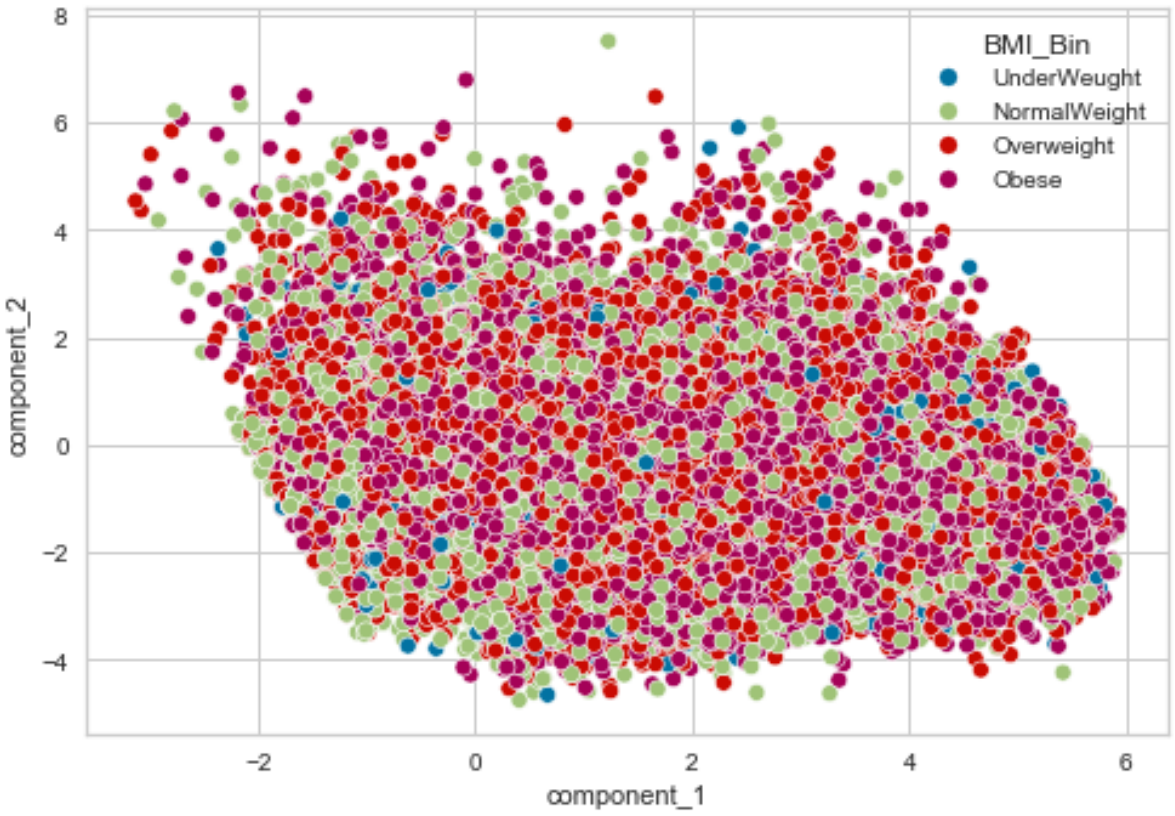
HeartDisease	1	2	3	4
0	116237.0	37359.0	22403.0	116423.0
1	16315.0	1832.0	7538.0	1688.0



The majority of people with heart disease fall in cluster 1. Although a significant proportion also fall in cluster 2. This lines with our hypothesis that people who suffer from herat disease exhibit similar risk factors.

```
sns.scatterplot(x='component_1', y='component_2', data=dim_reduced_2d, hue='BMI_Bin')
results_df = dim_reduced_2d[['BMI_Bin', 'clusters']].set_index('BMI_Bin')
results_df = results_df.apply(pd.Series.value_counts, axis=1).groupby(by=["BMI_Bin"]).
 ↪sum()
results_df
```

BMI_Bin	1	2	3	4
UnderWeught	1805.0	816.0	803.0	1690.0
NormalWeight	36996.0	10932.0	6614.0	40592.0
Overweight	50829.0	11545.0	8885.0	43493.0
Obese	42922.0	15898.0	13639.0	32336.0



Besides the majority of overweight and obese patients falling in clusters 1 and 4, participants in each BMI group seem to be equally distributed among the clusters. If patients could be accurately clustered by study, this would suggest that BMI would be a strong indicator of heart disease. However, this is not the case. It is possible that the clustering is not accurate enough to make such a conclusion, or that BMI has a mild influence on heart disease outcome.

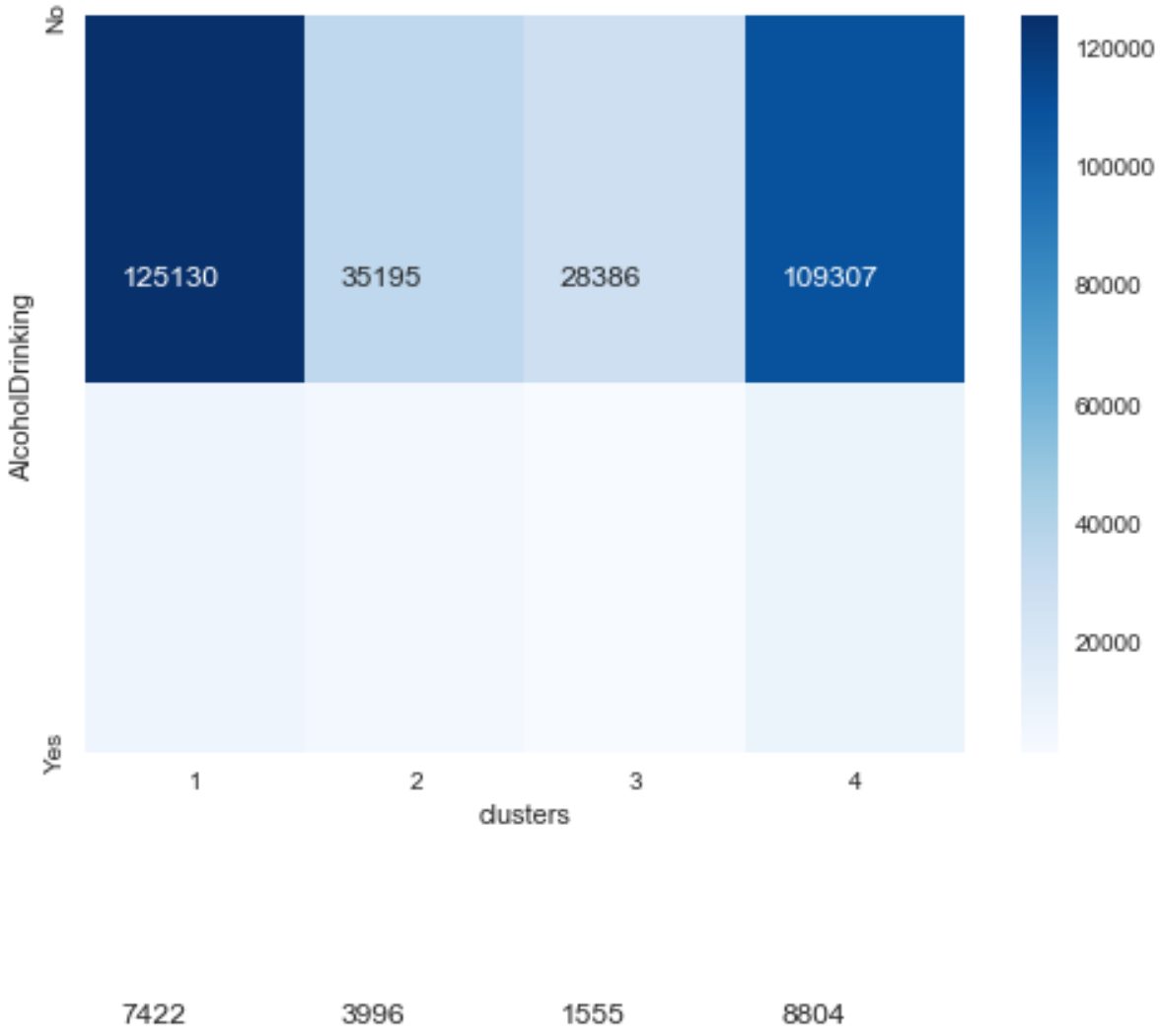
```
def visualise_cluster_heat_map(focus_feature, k, df, cmap=None):
    """
    Visualizes the clusters as heat maps , where the squares represents
        the number of particpants in k clusters grouped by the focus feature

    :param focus_feature: focus_feature respresents the feature we want to
        drill down by i.e. health status or study
    :param k: Number of desired clusters
    :param df: Dataframe containing study, health status and the top 3 principal
        components
    :param cmap: matplotlib colormap name or object, or list of colors, optional
    """


    km_3d = KMeans(n_clusters=k, random_state=42).fit(dim_reduced_2d[["component_1",
↪"component_2"]])
    df["clusters"] = km_3d.labels_ + 1
    df = df[[focus_feature, 'clusters']].set_index(focus_feature)
    df = df.apply(pd.Series.value_counts, axis=1).groupby(by=[focus_feature]).sum()
    akws = {"ha": 'left',"va": 'top'}
    ax = sns.heatmap(df, annot=True, cmap=cmap, fmt='g', annot_kws=akws)

    for t in ax.texts:
```

(continues on next page)

(continued from previous page)

```
        trans = t.get_transform()
        offs = matplotlib.transforms.ScaledTranslation(-0.34, 0.34,
                        matplotlib.transforms.IdentityTransform())
        t.set_transform( offs + trans )

    plt.xlabel('clusters')
    plt.show()
```

```
visualise_cluster_heat_map('AlcoholDrinking', 4, dim_reduced_2d, 'Blues')
```



The healthy participants seem to be evenly spread among the two clusters. Unhealthy participants have a tendency to appear in cluster 1 and 4. K-means seems to be reasonable able to cluster alcoholic patients.

Finally, we decide to append the clusters to our standardized dataset. Our feature selection method later on, will tell us if they were of use or not.

```
standradised_dataset = pd.read_csv('data/standardised_heart_disease.csv')
standradised_dataset["clusters"] = (dim_reduced_2d["clusters"]-dim_reduced_2d[
↪"clusters"].mean())/dim_reduced_2d["clusters"].std()
```

(continues on next page)

(continued from previous page)

```
standradised_dataset["clusters"].to_csv('data/standardised_heart_disease.csv',␣
↪index=False)
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3: FutureWarning:␣
↪The signature of `Series.to_csv` was aligned to that of `DataFrame.to_csv`, and␣
↪argument 'header' will change its default value from False to True: please pass␣
↪an explicit value to suppress this warning.
  This is separate from the ipykernel package so we can avoid doing imports until
```

CHAPTER

FOUR

MODEL BUILDING, EVALUATION & SENSITIVITY ANALYSIS

In this section, we will present a comparative analysis of the heart disease classification problem using different classification algorithms. We use the 80:20 train-test split rule to evaluate our models. **Note**, a small amount of patients were used as as hold-out set for hyper-parameter sensitivity analysis.

4.1 Models

We choose numerous shallow predictive methods to predict heart disease.

- Logistic Regression
- Gradient Boost Classifier
- Decision Tree Classifier
- Random Forest Classifier
- Naive Bayes

For more information on these algorithms, please click on the relevant links

4.2 Evaluation metrics

One of the key requirements in developing any algorithm is to measure it's effectiveness. Accuracy is the most simple measure. It tells us the he number of correctly classified examples over the total number of examples. More formally, $\$Accuracy = \frac{TruePositive+TrueNegative}{TruePositive+TrueNegative+FalsePositive+FalseNegative}\$$ But is accuracy telling the whole picture ? Well, let's consider those two examples:

- A classifier which, if a person has the heart disease, will always correctly diagnose it, but gets half of the healthy people wrong. You can see that announcing to a healthy person that he or she has the disease could lead to adverse consequences.
- A classifier that gets the diagnose right for every healthy person, but also miss half of the disease cases. That wouldn't be a very good algorithm would it?

Depending on the distribution of sick to healthy patients those two classifiers could have high accuracy while not being considered very good. Therefore, we decide to employ three further metrics

- *Precision:* dertimnes what proportion of the negative class got correctly classified. $\$\frac{TruePositive}{TruePositive+FalsePositive}\$$
- *Recall*: determine what proportion of the actual sick people were correctly detected by the model. $\$\frac{TruePositive}{TruePositive+FalseNegative}\$$

4.3 Import libaries

4.3.1 Data Processing

```
import pandas  as pd
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.decomposition import PCA
import pandas as pd
from sklearn.preprocessing import StandardScaler
import warnings
from sklearn.feature_selection import mutual_info_regression, mutual_info_classif
warnings.filterwarnings('ignore')
```

4.3.2 Model building and evaluation

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import plot_confusion_matrix

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.feature_selection import mutual_info_regression, mutual_info_classif
from sklearn.naive_bayes import GaussianNB
import numpy as np
from sklearn.metrics import classification_report, roc_curve, auc, roc_auc_score


from imblearn.under_sampling import (
    RandomUnderSampler,
    CondensedNearestNeighbour,
    TomekLinks,
    OneSidedSelection,
    EditedNearestNeighbours,
    RepeatedEditedNearestNeighbours,
    AllKNN,
    NeighbourhoodCleaningRule,
    NearMiss
)
from imblearn.over_sampling import (
    RandomOverSampler,
    SMOTE,
    ADASYN,
    BorderlineSMOTE,
    SVMSMOTE,
)
```

4.3.3 Model Serialisation

```
import pickle
```

4.3.4 Data Visualisation

```
import matplotlib.pyplot as plt
```

4.3.5 Code Type Hints

```
from typing import List
```

```
dict_classifiers = {
    "Logistic Regression ": LogisticRegression(),
    "Gradient Boost Classifier": GradientBoostingClassifier(),
    "Decision Tree Classifier": DecisionTreeClassifier(),
    "Random Forest Classifier": RandomForestClassifier(),
    "Naive Bayes": GaussianNB(),
}

heart_disease_dataset_standardized = pd.read_csv("data/standardised_heart_disease.csv
 ↪")

# Initial Attempt
train_heart_disease_df = heart_disease_dataset_standardized.drop('HeartDisease',axis=
 ↪'columns')
test_heart_diease_df = heart_disease_dataset_standardized.HeartDisease.astype(np.
 ↪float32)
X_train, X_test, y_train, y_test = train_test_split(train_heart_disease_df, test_
 ↪heart_diease_df, test_size=0.2, random_state=15, stratify=test_heart_diease_df)
```

```
print(f"Ratio of classes in training set:\n{y_train.value_counts(normalize=True)}")
print()
print(f"Ratio of classes in test set:\n{y_test.value_counts(normalize=True)}")
```

```
Ratio of classes in training set:
0.0    0.914406
1.0    0.085594
Name: HeartDisease, dtype: float64

Ratio of classes in test set:
0.0    0.914398
1.0    0.085602
Name: HeartDisease, dtype: float64
```

We balance the test dataset, to ensure accuracy is a fair measure of model performance.

```
test_df = X_test.copy()
test_df['HeartDisease'] = y_test
class_0 = test_df[test_df['HeartDisease'] == 0]
class_1 = test_df[test_df['HeartDisease'] == 1]

class_1 = class_1.sample(len(class_0),replace=True)
test_df = pd.concat([class_0, class_1], axis=0)
print('Data in Test:')
print(test_df['HeartDisease'].value_counts())
X_test = test_df.drop('HeartDisease',axis='columns')
y_test = test_df.HeartDisease.astype(np.float32)
```

```
Data in Test:
0.0    58484
1.0    58484
Name: HeartDisease, dtype: int64
```

```
def run_exps(
             X_train, X_test, y_train, y_test) -> pd.DataFrame:
    '''
    Lightweight script to test many models and find winners
    :param X_train: training split
    :param y_train: training target vector
    :param X_test: test split
    :param y_test: test target vector
    :return: None
    '''

    results = pd.DataFrame()
    for model_name, model in dict_classifiers.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        class_report = pd.DataFrame(classification_report(y_test, y_pred, target_
↪names=['No Heart Disease', 'Heart Disease'],output_dict=True)).transpose().reset_
↪index()
        class_report['model'] = [model_name] * class_report.shape[0]
        class_report['auc'] = [roc_auc_score(y_test, y_pred)] * class_report.shape[0]
        class_report = pd.concat([class_report], keys=['Model'], names=['Firstlevel'])
        results = pd.concat([results, class_report], ignore_index=True)
    results['dummy'] = None

    fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15,10))

    for cls, ax in zip(dict_classifiers.values(), axes.flatten()):
        plot_confusion_matrix(cls,
                              X_test,
                              y_test,
                              ax=ax,
                              cmap='Blues',
                              display_labels=['No Heart Disease', 'Heart Disease'])
        ax.title.set_text(type(cls).__name__)
    plt.tight_layout()
    plt.show()

    print(f"""Algorithm with the highest accuracy: {
```

(continues on next page)

(continued from previous page)

```
        results[results['index'] == 'accuracy'].sort_values(
            ['support'], ascending=False).head(1)[['model', 'support']].values[0].
↪tolist()}""")

    print(f"""Algorithm with the highest macro recall:
        {results[results['index'] == 'macro avg'].sort_values(
            ['recall'], ascending=False).head(1)[['model', 'recall']].values[0].
↪tolist()}""")

    print(f"""Algorithm with the highest macro precision:
        {results[results['index'] == 'macro avg'].sort_values(
            ['precision'], ascending=False).head(1)[['model', 'precision']].values[0].
↪tolist()}""")

    print(f"""Algorithm with the highest AUC:
        {results.sort_values(['auc'], ascending=False).head(1)[
            ['model', 'auc']].values[0].tolist()}""")

    print(results.groupby(
        ['model', 'index', 'precision', 'recall', 'f1-score', 'support',  'auc']
        )['dummy'].count())

    return results
```

4.4 Imbalanced data

As you can see above, our data is extremely imbalanced. Imbalanced datasets are those where there is a severe skew in the class distribution, such as 1:100 or 1:1000 examples in the minority class to the majority class.

This bias in the training dataset can influence many machine learning algorithms, leading some to ignore the minority class entirely. This is a problem as it is typically the minority class on which predictions are most important (i.e. predicting heart disease in our case).

One approach to addressing the problem of class imbalance is to randomly resample the training dataset. The two main approaches to randomly resampling an imbalanced dataset are to delete examples from the majority class, called under-sampling, and to duplicate examples from the minority class, called oversampling.

4.5 Undersampling

The follwoing undersampling methods were choosen:

- `RandomUnderSampler`: Random undersampling consists in extracting at random samples from the majority class, until they reach a certain proportion compared to the minority class, typically 50:50.
- `CondensedNearestNeighbour`: The algorithms works as follows:
  - Put all minority class observations in a group, typically group O
  - Add 1 sample (at random) from the majority class to group O
  - Train a KNN with group O
  - Take a sample of the majority class that is not in group O yet
  - Predict its class with the KNN from point 3

  - If the prediction was correct, go to 4 and repeat
  - If the prediction was incorrect, add that sample to group O, go to 3 and repeat
  - Continue until all samples of the majority class were either assigned to O or left out
  - Final version of Group O is our undersampled dataset

  This algorithm tends to pick points near the fuzzy boundary between the classes, and transfer those to the group O, in our example. If the classes are similar, group O will contain a fair amount of both classes. If the classes are very different, group O would contain mostly 1 class, the minority class.

- `TomekLinks`: Tomek links are 2 samples from a different class, which are nearest neighbours to each other. In other words, if 2 observations are nearest neighbours, and from a different class, they are Tomek Links. This procedures removes either the sample from the majority class if it is a Tomek Link, or alternatively, both observations, the one from the majority and the one from the minority class.
- `OneSidedSelection`: First finds the hardest instances to classify correctly from the majority class. Then removes noisy observations with Tomek Links.
- `EditedNearestNeighbours`: Train a KNN algorithm on the data (user defines number of neighbours, typically 3)
  - Find the 3 nearest neighbour to each observation (or the number defined by the user in 1)
  - Find the label of each of the neighbours (we know it, is the target in the dataset)
  - if the majority of the neighbours show the same label as the observation, then we keep the observation
- `RepeatedEditedNearestNeighbours`: Extends Edited Nearest neighbours in that it repeats the procedure over an over, until no further observation is removed from the dataset, or alternatively until a maximum number of iterations is reached.
- `AllKNN`: Adapts the functionality of Edited Nearest Neighbours in that, at each round, it increases the number of neighbours utilised to exclude or retain the observations. It starts by looking at the 1 closest neighbour. It finishes at a maximum number of neighbours to examine, determined by the user it stops prematurely if the majority class becomes the minority
- `NeighbourhoodCleaningRule`: The Neighbourhood Cleaning Rule works as follows:
  1. Remove noisy observations from the majority class with ENN:
  - explores the 3 closest neighbours\n
  - uses majority vote of neighbours to retain observations
  1. Remove observations from the majority class if:,
  - they are 1 of the 3 closest neighbours to a minority sample, and,
  - most / all of those 3 closest neighbours are not minority, and,
  - the majority class has at least half as many observations as those in the minority (this can be regulated)
- `NearMiss`: This procedures aims to select samples that are somewhat similar to the minority class, using 1 of three alternative procedures:
  - Select observations closer to the closest minority class
  - Select observations closer to the farthest minority class
  - Select observations furthest from their nearest neighbours

Note: We train the models on a portion of the data that is under-sampled We evaluate the model performance on another portion of the data that was not resampled, and thus contains the original class distribution.

Note: In addition a verbose output of the models performance will be generated.

```
undersampler_dict = {

    'random': RandomUnderSampler(
        sampling_strategy='auto',
        random_state=0,
        replacement=False),

    # 'cnn': CondensedNearestNeighbour(
    #     sampling_strategy='auto',
    #     random_state=0,
    #     n_neighbors=1,
    #     n_jobs=4),

    'tomek': TomekLinks(
        sampling_strategy='auto',
        n_jobs=4),

    'oss': OneSidedSelection(
        sampling_strategy='auto',
        random_state=0,
        n_neighbors=1,
        n_jobs=4),

    # 'enn': EditedNearestNeighbours(
    #     sampling_strategy='auto',
    #     n_neighbors=3,
    #     kind_sel='all',
    #     n_jobs=4),

    # 'renn': RepeatedEditedNearestNeighbours(
    #     sampling_strategy='auto',
    #     n_neighbors=3,
    #     kind_sel='all',
    #     n_jobs=4,
    #     max_iter=100),

    # 'allknn': AllKNN(
    #     sampling_strategy='auto',
    #     n_neighbors=3,
    #     kind_sel='all',
    #     n_jobs=4),

    # 'ncr': NeighbourhoodCleaningRule(
    #     sampling_strategy='auto',
    #     n_neighbors=3,
    #     kind_sel='all',
    #     n_jobs=4,
    #     threshold_cleaning=0.5),
```

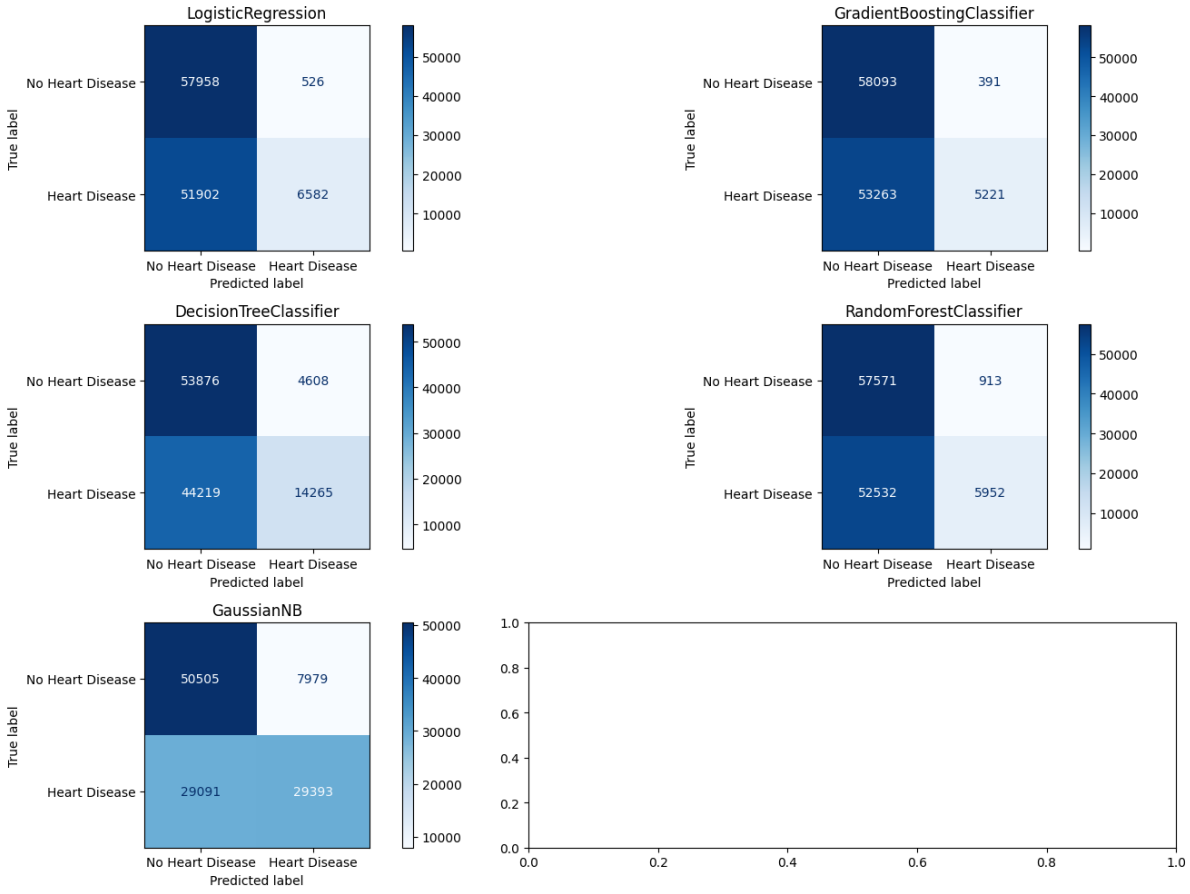
(continues on next page)

(continued from previous page)

```
    'nm1': NearMiss(
        sampling_strategy='auto',
        version=1,
        n_neighbors=3,
        n_jobs=4),

    'nm2': NearMiss(
        sampling_strategy='auto',
        version=2,
        n_neighbors=3,
        n_jobs=4),
}
```

```
# train a model on the original data without under-sampling
# and determine model performance
print("No UnderSampling")
print("-------------------")
run_exps(X_train, X_test, y_train, y_test)

print("UnderSampling Methods")
print("-------------------")
print()

# now, we test the different under-samplers, 1 at a time
for undersampler in undersampler_dict.keys():

    print(undersampler)
    print("-------------------")

    # resample the train set only
    X_resampled, y_resampled = undersampler_dict[undersampler].fit_resample(X_train.
 ↪copy(), y_train.copy())

    # train model and evaluate performance

    # Note the performance returned is using the
    # test set, which was not under-sampled

    run_exps(X_resampled, X_test, y_resampled, y_test)

    print()

print()
```

```
No UnderSampling
-------------------
```

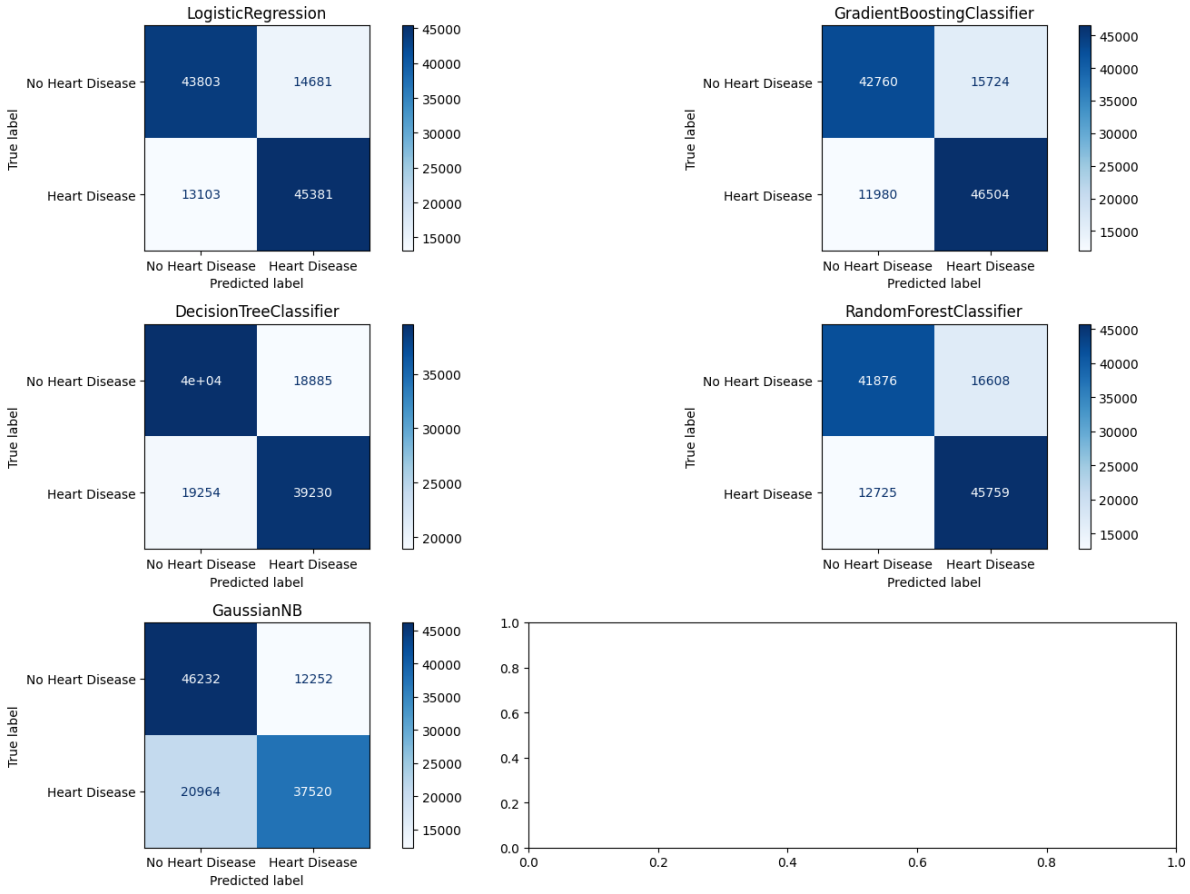



```
Algorithm with the highest accuracy: ['Naive Bayes', 0.6830757130155256]
Algorithm with the highest macro recall:
        ['Naive Bayes', 0.6830757130155256]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7267806132960342]
Algorithm with the highest AUC:
        ['Naive Bayes', 0.6830757130155256]
model                       index            precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.755842   0.243913  0.368810  58484.
 ↪000000   0.582561    0
                            No Heart Disease  0.549223   0.921209  0.688164  58484.
 ↪000000   0.582561    0
                            accuracy          0.582561   0.582561  0.582561  0.
 ↪582561       0.582561    0
                            macro avg         0.652532   0.582561  0.528487  116968.
 ↪000000  0.582561    0
                            weighted avg      0.652532   0.582561  0.528487  116968.
 ↪000000  0.582561    0
Gradient Boost Classifier   Heart Disease     0.930328   0.089272  0.162912  58484.
 ↪000000   0.541293    0
                            No Heart Disease  0.521687   0.993314  0.684091  58484.
 ↪000000   0.541293    0
                            accuracy          0.541293   0.541293  0.541293  0.
 ↪541293       0.541293    0
                            macro avg         0.726008   0.541293  0.423501  116968.
 ↪000000  0.541293    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.726008   0.541293  0.423501  116968.
 ↪000000  0.541293    0
Logistic Regression       Heart Disease     0.925999   0.112544  0.200695  58484.
 ↪000000   0.551775    0
                          No Heart Disease  0.527562   0.991006  0.688566  58484.
 ↪000000   0.551775    0
                          accuracy          0.551775   0.551775  0.551775  0.
 ↪551775       0.551775    0
                          macro avg         0.726781   0.551775  0.444631  116968.
 ↪000000  0.551775    0
                          weighted avg      0.726781   0.551775  0.444631  116968.
 ↪000000  0.551775    0
Naive Bayes               Heart Disease     0.786498   0.502582  0.613274  58484.
 ↪000000   0.683076    0
                          No Heart Disease  0.634517   0.863570  0.731532  58484.
 ↪000000   0.683076    0
                          accuracy          0.683076   0.683076  0.683076  0.
 ↪683076       0.683076    0
                          macro avg         0.710507   0.683076  0.672403  116968.
 ↪000000  0.683076    0
                          weighted avg      0.710507   0.683076  0.672403  116968.
 ↪000000  0.683076    0
Random Forest Classifier  Heart Disease     0.867007   0.101771  0.182160  58484.
 ↪000000   0.543080    0
                          No Heart Disease  0.522883   0.984389  0.682983  58484.
 ↪000000   0.543080    0
                          accuracy          0.543080   0.543080  0.543080  0.
 ↪543080       0.543080    0
                          macro avg         0.694945   0.543080  0.432572  116968.
 ↪000000  0.543080    0
                          weighted avg      0.694945   0.543080  0.432572  116968.
 ↪000000  0.543080    0
Name: dummy, dtype: int64
UnderSampling Methods
--------------------

random
--------------------
```

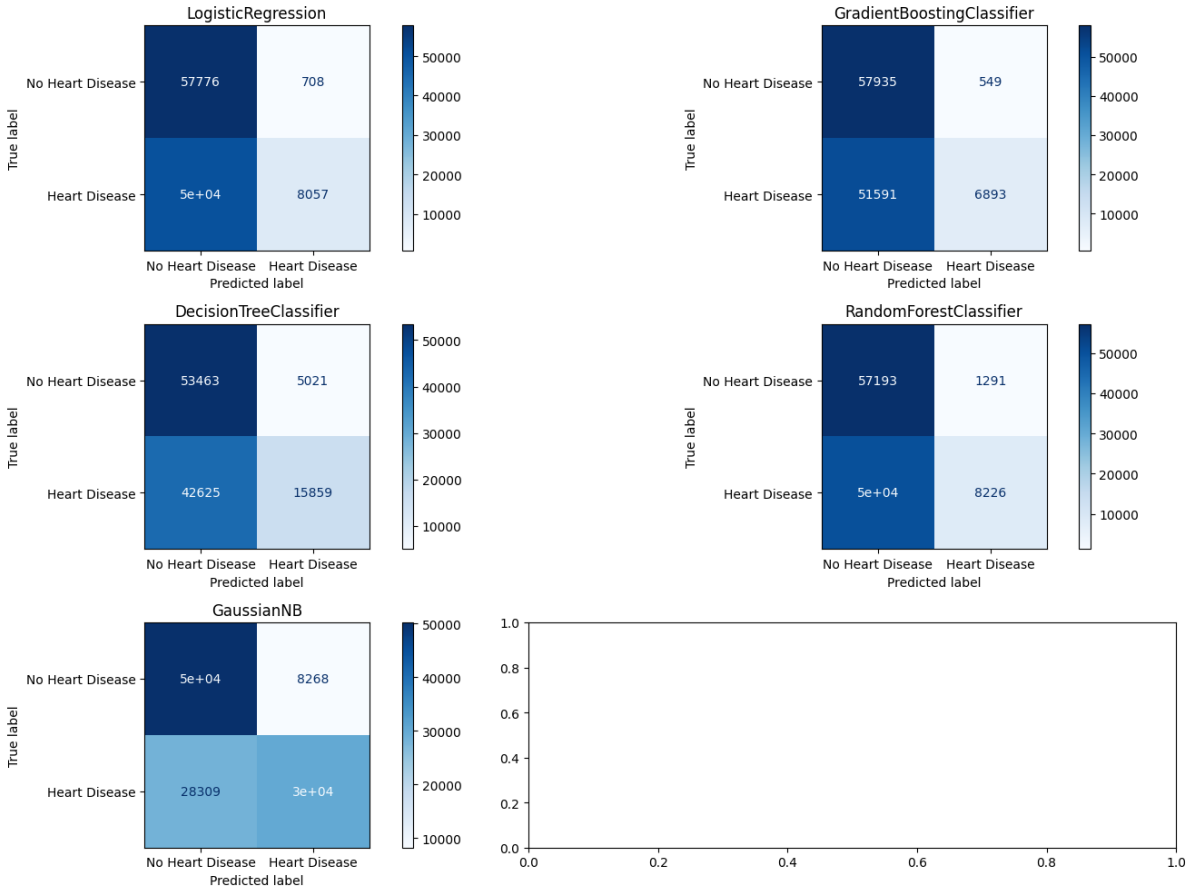



```
Algorithm with the highest accuracy: ['Gradient Boost Classifier', 0.
 ↪7631488954243896]
Algorithm with the highest macro recall:
        ['Gradient Boost Classifier', 0.7631488954243896]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.764231781066334]
Algorithm with the highest AUC:
        ['Gradient Boost Classifier', 0.7631488954243896]
model                       index              precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease      0.675041   0.670782  0.672905  58484.
 ↪000000   0.673936    0
                            No Heart Disease   0.672846   0.677091  0.674962  58484.
 ↪000000   0.673936    0
                            accuracy           0.673936   0.673936  0.673936  0.
 ↪673936       0.673936    0
                            macro avg          0.673943   0.673936  0.673933  116968.
 ↪000000  0.673936    0
                            weighted avg       0.673943   0.673936  0.673933  116968.
 ↪000000  0.673936    0
Gradient Boost Classifier   Heart Disease      0.747316   0.795158  0.770495  58484.
 ↪000000   0.763149    0
                            No Heart Disease   0.781147   0.731140  0.755317  58484.
 ↪000000   0.763149    0
                            accuracy           0.763149   0.763149  0.763149  0.
 ↪763149       0.763149    0
```

(continues on next page)

(continued from previous page)

```
                          macro avg         0.764232   0.763149  0.762906  116968.
 ↪000000  0.763149    0
                          weighted avg      0.764232   0.763149  0.762906  116968.
 ↪000000  0.763149    0
Logistic Regression       Heart Disease     0.755569   0.775956  0.765627  58484.
 ↪000000   0.762465    0
                          No Heart Disease  0.769743   0.748974  0.759217  58484.
 ↪000000   0.762465    0
                          accuracy          0.762465   0.762465  0.762465  0.
 ↪762465       0.762465    0
                          macro avg         0.762656   0.762465  0.762422  116968.
 ↪000000  0.762465    0
                          weighted avg      0.762656   0.762465  0.762422  116968.
 ↪000000  0.762465    0
Naive Bayes               Heart Disease     0.753837   0.641543  0.693172  58484.
 ↪000000   0.716025    0
                          No Heart Disease  0.688017   0.790507  0.735710  58484.
 ↪000000   0.716025    0
                          accuracy          0.716025   0.716025  0.716025  0.
 ↪716025       0.716025    0
                          macro avg         0.720927   0.716025  0.714441  116968.
 ↪000000  0.716025    0
                          weighted avg      0.720927   0.716025  0.714441  116968.
 ↪000000  0.716025    0
Random Forest Classifier  Heart Disease     0.733705   0.782419  0.757280  58484.
 ↪000000   0.749222    0
                          No Heart Disease  0.766946   0.716025  0.740611  58484.
 ↪000000   0.749222    0
                          accuracy          0.749222   0.749222  0.749222  0.
 ↪749222       0.749222    0
                          macro avg         0.750325   0.749222  0.748945  116968.
 ↪000000  0.749222    0
                          weighted avg      0.750325   0.749222  0.748945  116968.
 ↪000000  0.749222    0
Name: dummy, dtype: int64

tomek
--------------------
```

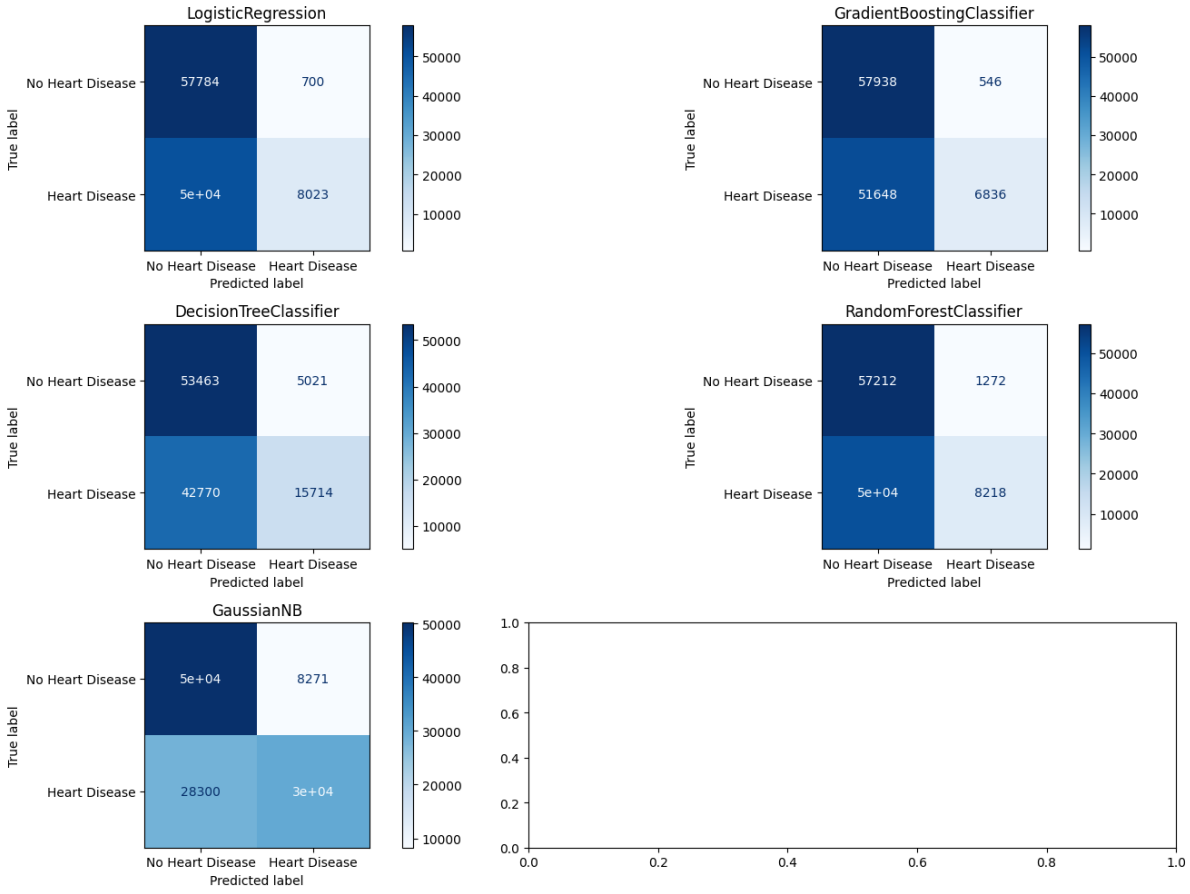



```
Algorithm with the highest accuracy: ['Naive Bayes', 0.6872905410026674]
Algorithm with the highest macro recall:
        ['Naive Bayes', 0.6872905410026674]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.727595334051979]
Algorithm with the highest AUC:
        ['Naive Bayes', 0.6872905410026673]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.759531   0.271168  0.399652  58484.
 ↪000000    0.592658    0
                            No Heart Disease  0.556396   0.914147  0.691755  58484.
 ↪000000    0.592658    0
                            accuracy          0.592658   0.592658  0.592658  0.
 ↪592658        0.592658    0
                            macro avg         0.657963   0.592658  0.545704  116968.
 ↪000000  0.592658    0
                            weighted avg      0.657963   0.592658  0.545704  116968.
 ↪000000  0.592658    0
Gradient Boost Classifier   Heart Disease     0.926230   0.117861  0.209113  58484.
 ↪000000    0.554237    0
                            No Heart Disease  0.528961   0.990613  0.689661  58484.
 ↪000000    0.554237    0
                            accuracy          0.554237   0.554237  0.554237  0.
 ↪554237        0.554237    0
                            macro avg         0.727595   0.554237  0.449387  116968.
 ↪000000  0.554237    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.727595   0.554237  0.449387  116968.
 ↪000000  0.554237    0
Logistic Regression       Heart Disease     0.919224   0.137764  0.239617  58484.
 ↪000000   0.562829    0
                          No Heart Disease  0.533959   0.987894  0.693227  58484.
 ↪000000   0.562829    0
                          accuracy          0.562829   0.562829  0.562829  0.
 ↪562829       0.562829    0
                          macro avg         0.726592   0.562829  0.466422  116968.
 ↪000000  0.562829    0
                          weighted avg      0.726592   0.562829  0.466422  116968.
 ↪000000  0.562829    0
Naive Bayes               Heart Disease     0.784928   0.515953  0.622634  58484.
 ↪000000   0.687291    0
                          No Heart Disease  0.639491   0.858628  0.733032  58484.
 ↪000000   0.687291    0
                          accuracy          0.687291   0.687291  0.687291  0.
 ↪687291       0.687291    0
                          macro avg         0.712209   0.687291  0.677833  116968.
 ↪000000  0.687291    0
                          weighted avg      0.712209   0.687291  0.677833  116968.
 ↪000000  0.687291    0
Random Forest Classifier  Heart Disease     0.864348   0.140654  0.241938  58484.
 ↪000000   0.559290    0
                          No Heart Disease  0.532271   0.977926  0.689342  58484.
 ↪000000   0.559290    0
                          accuracy          0.559290   0.559290  0.559290  0.
 ↪559290       0.559290    0
                          macro avg         0.698309   0.559290  0.465640  116968.
 ↪000000  0.559290    0
                          weighted avg      0.698309   0.559290  0.465640  116968.
 ↪000000  0.559290    0
Name: dummy, dtype: int64

oss
-------------------
```

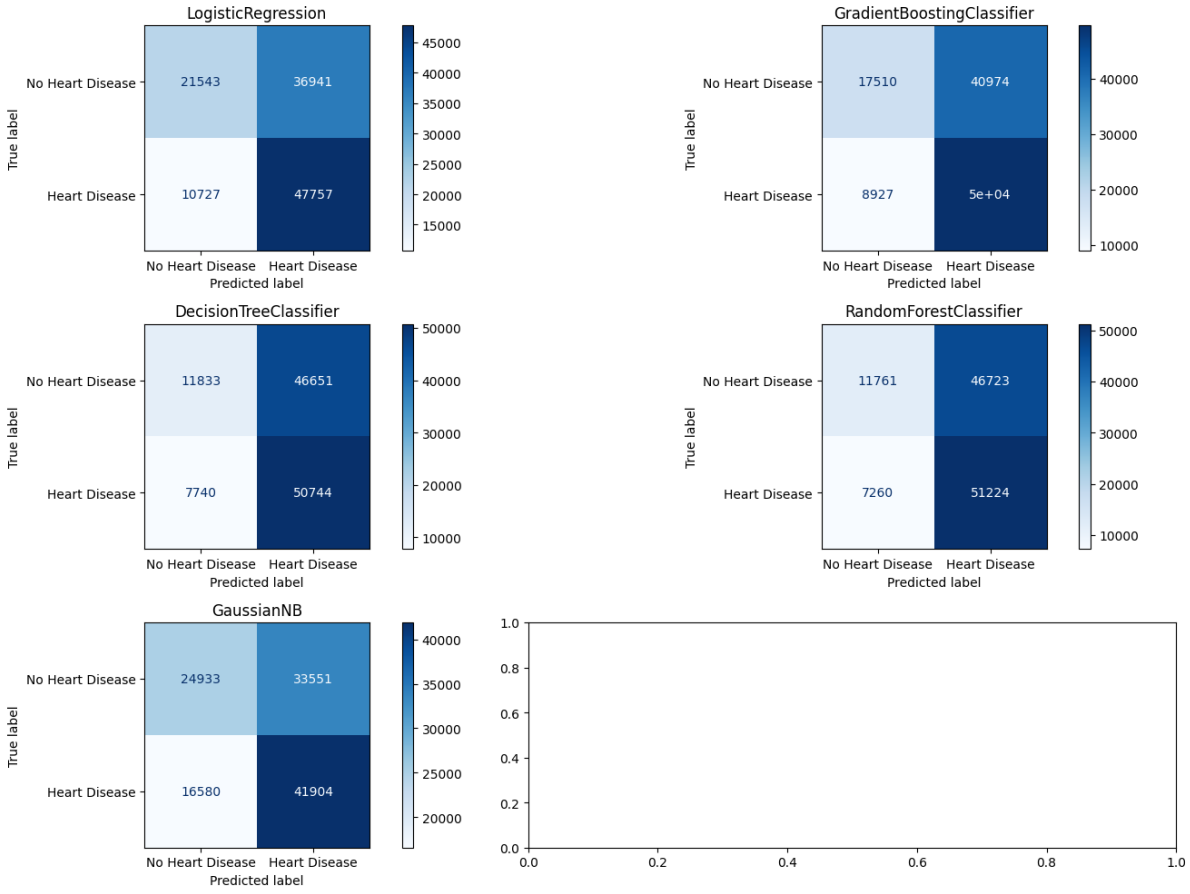



```
Algorithm with the highest accuracy: ['Naive Bayes', 0.6873418370836468]
Algorithm with the highest macro recall:
        ['Naive Bayes', 0.6873418370836468]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.7273676129597939]
Algorithm with the highest AUC:
        ['Naive Bayes', 0.6873418370836468]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.757849   0.268689  0.396723  58484.
 ↪000000   0.591418    0
                            No Heart Disease  0.555558   0.914147  0.691107  58484.
 ↪000000   0.591418    0
                            accuracy          0.591418   0.591418  0.591418  0.
 ↪591418       0.591418    0
                            macro avg         0.656703   0.591418  0.543915  116968.
 ↪000000  0.591418    0
                            weighted avg      0.656703   0.591418  0.543915  116968.
 ↪000000  0.591418    0
Gradient Boost Classifier   Heart Disease     0.926036   0.116887  0.207573  58484.
 ↪000000   0.553775    0
                            No Heart Disease  0.528699   0.990664  0.689451  58484.
 ↪000000   0.553775    0
                            accuracy          0.553775   0.553775  0.553775  0.
 ↪553775       0.553775    0
                            macro avg         0.727368   0.553775  0.448512  116968.
 ↪000000  0.553775    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.727368   0.553775  0.448512  116968.
 ↪000000  0.553775    0
Logistic Regression       Heart Disease     0.919752   0.137183  0.238755  58484.
 ↪000000   0.562607    0
                          No Heart Disease  0.533826   0.988031  0.693149  58484.
 ↪000000   0.562607    0
                          accuracy          0.562607   0.562607  0.562607  0.
 ↪562607       0.562607    0
                          macro avg         0.726789   0.562607  0.465952  116968.
 ↪000000  0.562607    0
                          weighted avg      0.726789   0.562607  0.465952  116968.
 ↪000000  0.562607    0
Naive Bayes               Heart Disease     0.784917   0.516107  0.622742  58484.
 ↪000000   0.687342    0
                          No Heart Disease  0.639550   0.858577  0.733053  58484.
 ↪000000   0.687342    0
                          accuracy          0.687342   0.687342  0.687342  0.
 ↪687342       0.687342    0
                          macro avg         0.712234   0.687342  0.677897  116968.
 ↪000000  0.687342    0
                          weighted avg      0.712234   0.687342  0.677897  116968.
 ↪000000  0.687342    0
Random Forest Classifier  Heart Disease     0.865964   0.140517  0.241798  58484.
 ↪000000   0.559384    0
                          No Heart Disease  0.532314   0.978250  0.689459  58484.
 ↪000000   0.559384    0
                          accuracy          0.559384   0.559384  0.559384  0.
 ↪559384       0.559384    0
                          macro avg         0.699139   0.559384  0.465629  116968.
 ↪000000  0.559384    0
                          weighted avg      0.699139   0.559384  0.465629  116968.
 ↪000000  0.559384    0
Name: dummy, dtype: int64

nm1
--------------------
```




```
Algorithm with the highest accuracy: ['Logistic Regression ', 0.5924697353122221]
Algorithm with the highest macro recall:
        ['Logistic Regression ', 0.5924697353122221]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.6157181896300579]
Algorithm with the highest AUC:
        ['Logistic Regression ', 0.5924697353122221]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.521012   0.867656  0.651069  58484.
 ↪000000   0.534992    0
                            No Heart Disease  0.604557   0.202329  0.303189  58484.
 ↪000000   0.534992    0
                            accuracy          0.534992   0.534992  0.534992  0.
 ↪534992       0.534992    0
                            macro avg         0.562785   0.534992  0.477129  116968.
 ↪000000  0.534992    0
                            weighted avg      0.562785   0.534992  0.477129  116968.
 ↪000000  0.534992    0
Gradient Boost Classifier   Heart Disease     0.547404   0.847360  0.665128  58484.
 ↪000000   0.573379    0
                            No Heart Disease  0.662329   0.299398  0.412383  58484.
 ↪000000   0.573379    0
                            accuracy          0.573379   0.573379  0.573379  0.
 ↪573379       0.573379    0
                            macro avg         0.604866   0.573379  0.538755  116968.
 ↪000000  0.573379    0
```

(continues on next page)

(continued from previous page)

```
                           weighted avg      0.604866   0.573379  0.538755  116968.
↪000000  0.573379    0
Logistic Regression        Heart Disease     0.563850   0.816582  0.667081  58484.
↪000000   0.592470    0
                           No Heart Disease  0.667586   0.368357  0.474756  58484.
↪000000   0.592470    0
                           accuracy          0.592470   0.592470  0.592470  0.
↪592470       0.592470    0
                           macro avg         0.615718   0.592470  0.570918  116968.
↪000000  0.592470    0
                           weighted avg      0.615718   0.592470  0.570918  116968.
↪000000  0.592470    0
Naive Bayes                Heart Disease     0.555351   0.716504  0.625718  58484.
↪000000   0.571413    0
                           No Heart Disease  0.600607   0.426322  0.498675  58484.
↪000000   0.571413    0
                           accuracy          0.571413   0.571413  0.571413  0.
↪571413       0.571413    0
                           macro avg         0.577979   0.571413  0.562196  116968.
↪000000  0.571413    0
                           weighted avg      0.577979   0.571413  0.562196  116968.
↪000000  0.571413    0
Random Forest Classifier   Heart Disease     0.522977   0.875863  0.654909  58484.
↪000000   0.538481    0
                           No Heart Disease  0.618317   0.201098  0.303490  58484.
↪000000   0.538481    0
                           accuracy          0.538481   0.538481  0.538481  0.
↪538481       0.538481    0
                           macro avg         0.570647   0.538481  0.479199  116968.
↪000000  0.538481    0
                           weighted avg      0.570647   0.538481  0.479199  116968.
↪000000  0.538481    0
Name: dummy, dtype: int64

nm2
-------------------
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
Cell In [11], line 18
     15 print("-------------------")
     17 # resample the train set only
---> 18 X_resampled, y_resampled = undersampler_dict[undersampler].fit_resample(X_
↪train.copy(), y_train.copy())
     20 # train model and evaluate performance
     21
     22 # Note the performance returned is using the
     23 # test set, which was not under-sampled
     25 run_exps(X_resampled, X_test, y_resampled, y_test)

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/imblearn/
↪base.py:83, in SamplerMixin.fit_resample(self, X, y)
     77 X, y, binarize_y = self._check_X_y(X, y)
     79 self.sampling_strategy_ = check_sampling_strategy(
     80     self.sampling_strategy, y, self._sampling_type
```

(continues on next page)

(continued from previous page)

```
     81 )
---> 83 output = self._fit_resample(X, y)
     85 y_ = (
     86     label_binarize(output[1], classes=np.unique(y)) if binarize_y else␣
 ↪output[1]
     87 )
     89 X_, y_ = arrays_transformer.transform(output[0], y_)

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/imblearn/
 ↪under_sampling/_prototype_selection/_nearmiss.py:233, in NearMiss._fit_
 ↪resample(self, X, y)
    224     index_target_class = self._selection_dist_based(
    225         X,
    226         y,
   (...)
    230         sel_strategy="nearest",
    231     )
    232 elif self.version == 2:
--> 233     dist_vec, idx_vec = self.nn_.kneighbors(
    234         X_class, n_neighbors=target_stats[class_minority]
    235     )
    236     index_target_class = self._selection_dist_based(
    237         X,
    238         y,
   (...)
    242         sel_strategy="nearest",
    243     )
    244 elif self.version == 3:

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/sklearn/
 ↪neighbors/_base.py:763, in KNeighborsMixin.kneighbors(self, X, n_neighbors,␣
 ↪return_distance)
    756 use_pairwise_distances_reductions = (
    757     self._fit_method == "brute"
    758     and PairwiseDistancesArgKmin.is_usable_for(
    759         X if X is not None else self._fit_X, self._fit_X, self.effective_
 ↪metric_
    760     )
    761 )
    762 if use_pairwise_distances_reductions:
--> 763     results = PairwiseDistancesArgKmin.compute(
    764         X=X,
    765         Y=self._fit_X,
    766         k=n_neighbors,
    767         metric=self.effective_metric_,
    768         metric_kwargs=self.effective_metric_params_,
    769         strategy="auto",
    770         return_distance=return_distance,
    771     )
    773 elif (
    774     self._fit_method == "brute" and self.metric == "precomputed" and␣
 ↪issparse(X)
    775 ):
    776     results = _kneighbors_from_graph(
    777         X, n_neighbors=n_neighbors, return_distance=return_distance
    778     )
```

(continues on next page)

(continued from previous page)

```
File sklearn/metrics/_pairwise_distances_reduction.pyx:679, in sklearn.metrics._
↪pairwise_distances_reduction.PairwiseDistancesArgKmin.compute()

File sklearn/metrics/_pairwise_distances_reduction.pyx:1060, in sklearn.metrics._
↪pairwise_distances_reduction.FastEuclideanPairwiseDistancesArgKmin.__init__()

File sklearn/metrics/_pairwise_distances_reduction.pyx:737, in sklearn.metrics._
↪pairwise_distances_reduction.PairwiseDistancesArgKmin.__init__()

File ~/optum/repos/HeartDiseaseAreidy1/venv/lib/python3.8/site-packages/numpy/core/
↪numeric.py:343, in full(shape, fill_value, dtype, order, like)
    341     fill_value = asarray(fill_value)
    342     dtype = fill_value.dtype
--> 343 a = empty(shape, dtype, order)
    344 multiarray.copyto(a, fill_value, casting='unsafe')
    345 return a

MemoryError: Unable to allocate 38.2 GiB for an array with shape (233938, 21898)␣
↪and data type int64
```

As we can see in the verbose model output,our best setting occurred with the `Random` undersampling strategy. Across most tracked metrics, we see a significant improvement in model performance compared with no undersupplying techniques.

Base (no under sampling techniques) **winners** for tracked metrics:

Metric	Metric	Value
Accuracy	Naive Bayes	0.684
Macro Recall	Naive Bayes	0.684
Macro precision	Gradient Boost Classifier	0.726
AUC	Naive Bayes	0.684

Random under sampling technique winners for tracked metrics:

Metric	Metric	Value
Accuracy	Gradient Boost Classifier	0.764
Macro Recall	Gradient Boost Classifier	0.763
Macro precision	Gradient Boost Classifier	0.765
AUC	Gradient Boost Classifier	0.763

4.6 Oversampling

The following undersampling methods were choosen:

- `Random Oversampling`: Random over-sampling consists in extracting at random samples from the minority class, until they reach a certain proportion compared to the majority class, typically 50:50, or in other words, a balancing ratio of 1.
- `SMOTE`: Creates new samples by interpolation of samples of the minority class and any of its k nearest neighbours (also from the minority class). K is typically 5.

- `ADASYN`: Creates new samples by interpolation of samples of the minority class and its closest neighbours. It creates more samples from samples that are harder to classify.
- `Borderline SMOTE`: Creates new samples by interpolation between samples of the minority class and their closest neighbours.
  - It does not use all observations from the minority class as templates, unllike SMOTE.
  - It selects those observations (from the minority) for which, most of their neighbours belong to a different class (DANGER group)
    - Variant 1 creates new examples, as SMOTE, between samples in the Danger group and their closest neighbours from the minority
    - Variant 2 creates new examples between samples in the Danger group and neighbours from minority and majority class
- `SVM SMOTE`: Creates new samples by interpolation of samples of the support vectors from minority class and its closest neighbours.

Note: We train the models on a portion of the data that is over-sampled We evaluate the model performance on another portion of the data that was not resampled, and thus contains the original class distribution.

```
oversampler_dict = {

    'random': RandomOverSampler(
        sampling_strategy='auto',
        random_state=0),

    'smote': SMOTE(
        sampling_strategy='auto',  # samples only the minority class
        random_state=0,  # for reproducibility
        k_neighbors=5,
        n_jobs=4),

    'adasyn': ADASYN(
        sampling_strategy='auto',  # samples only the minority class
        random_state=0,  # for reproducibility
        n_neighbors=5,
        n_jobs=4),

    'border1': BorderlineSMOTE(
        sampling_strategy='auto',  # samples only the minority class
        random_state=0,  # for reproducibility
        k_neighbors=5,
        m_neighbors=10,
        kind='borderline-1',
        n_jobs=4),

    'border2': BorderlineSMOTE(
        sampling_strategy='auto',  # samples only the minority class
        random_state=0,  # for reproducibility
        k_neighbors=5,
        m_neighbors=10,
        kind='borderline-2',
        n_jobs=4),
```

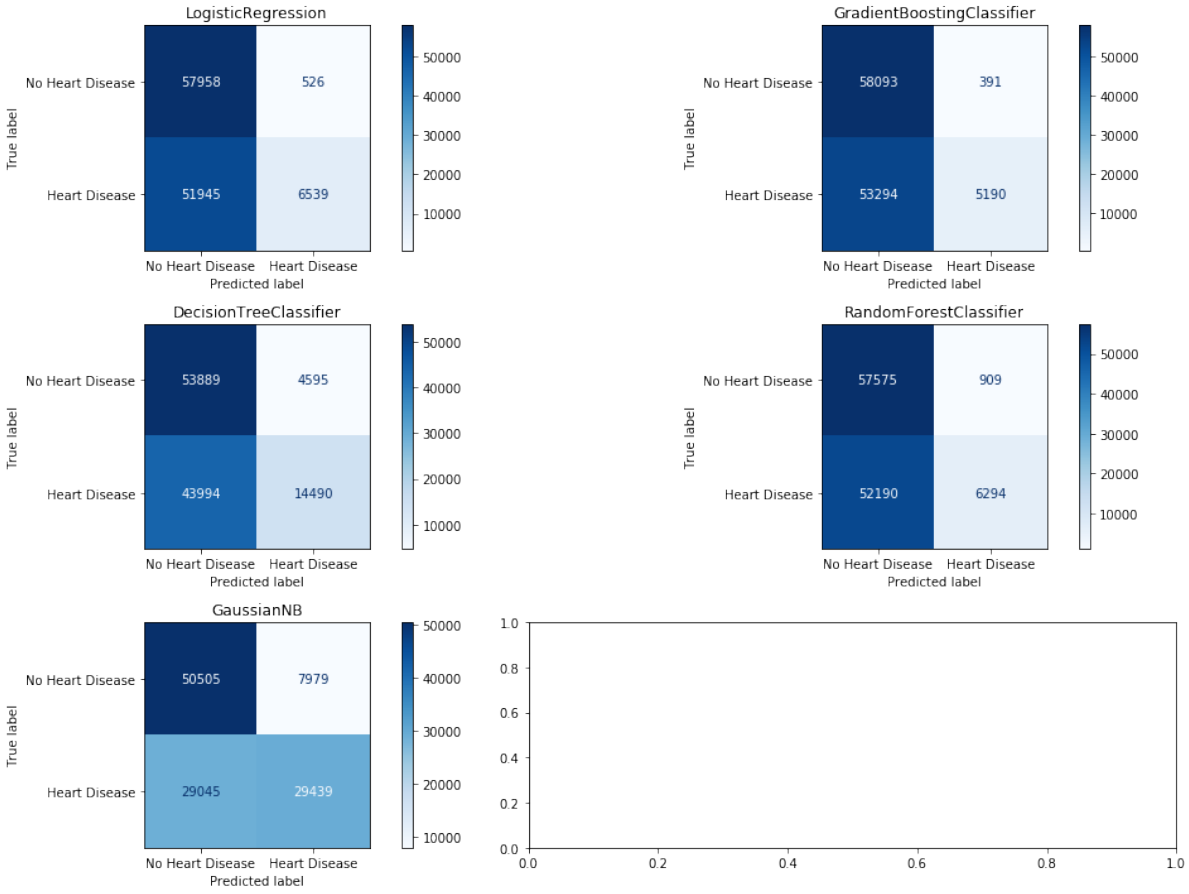
(continues on next page)

(continued from previous page)

```
    # 'svm': SVMSMOTE(
    #     sampling_strategy='auto',  # samples only the minority class
    #     random_state=0,  # for reproducibility
    #     k_neighbors=5,
    #     m_neighbors=10,
    #     n_jobs=4,
    #     svm_estimator=SVC(kernel='linear')),
}
```

```
# train a model on the original data without under-sampling
# and determine model performance
print("No OverSampling")
print("--------------------")
run_exps(X_train, X_test, y_train, y_test)

print()

print("OverSampling Methods")
print("--------------------")

# now, we test the different under-samplers, 1 at a time
for oversampler in oversampler_dict.keys():

    print(oversampler)
    print("--------------------")

    # resample the train set only
    X_resampled, y_resampled = oversampler_dict[oversampler].fit_resample(X_train.
 ↪copy(), y_train.copy())

    # train model and evaluate performance

    # Note the performance returned is using the
    # test set, which was not under-sampled

    run_exps(X_resampled, X_test, y_resampled, y_test)

    print()

print()
```

```
No OverSampling
--------------------
```

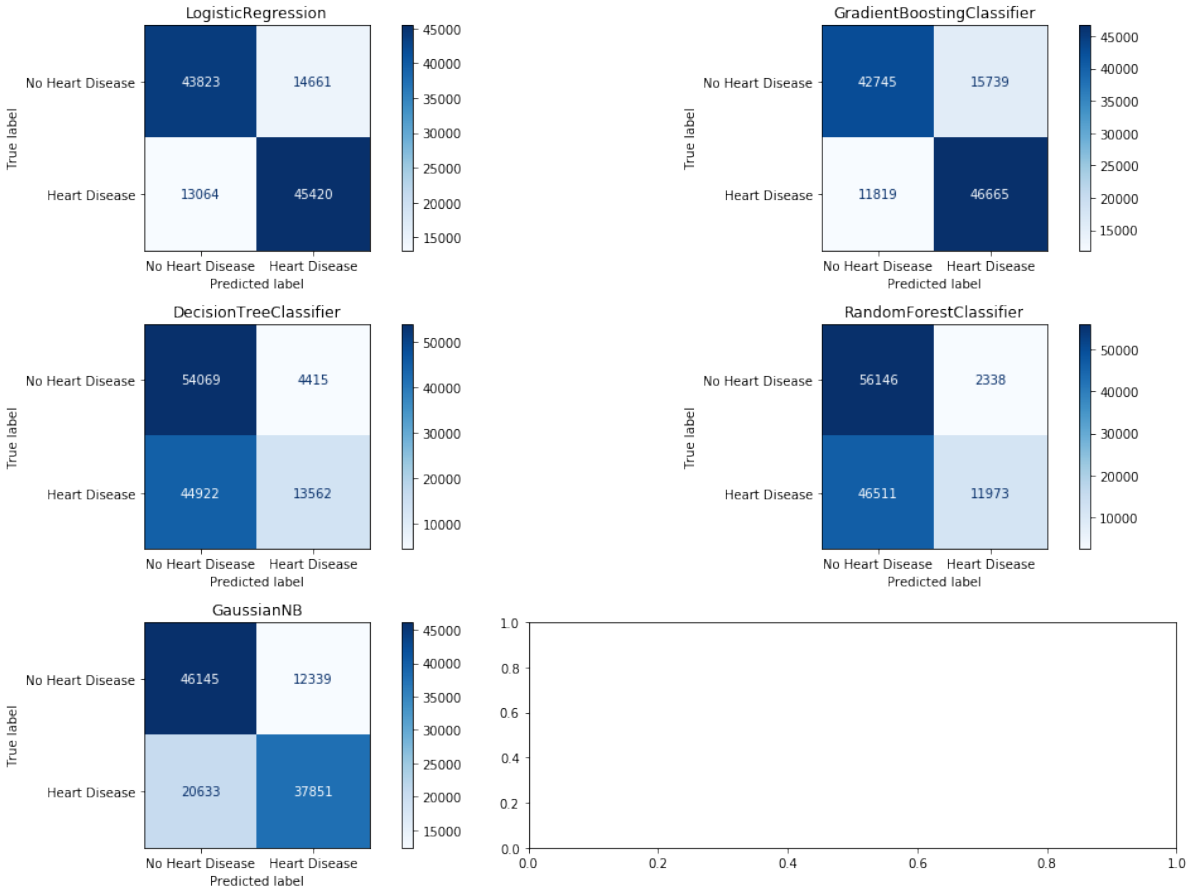



```
Algorithm with the highest accuracy: ['Naive Bayes', 0.6834689829697012]
Algorithm with the highest macro recall:
        ['Naive Bayes', 0.6834689829697012]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7264522097814119]
Algorithm with the highest AUC:
        ['Naive Bayes', 0.683468982969701]
model                       index            precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.759235   0.247760  0.373603  58484.
 ↪000000   0.584596    0
                            No Heart Disease  0.550545   0.921432  0.689263  58484.
 ↪000000   0.584596    0
                            accuracy          0.584596   0.584596  0.584596  0.
 ↪584596       0.584596    0
                            macro avg         0.654890   0.584596  0.531433  116968.
 ↪000000  0.584596    0
                            weighted avg      0.654890   0.584596  0.531433  116968.
 ↪000000  0.584596    0
Gradient Boost Classifier   Heart Disease     0.929941   0.088742  0.162023  58484.
 ↪000000   0.541028    0
                            No Heart Disease  0.521542   0.993314  0.683966  58484.
 ↪000000   0.541028    0
                            accuracy          0.541028   0.541028  0.541028  0.
 ↪541028       0.541028    0
                            macro avg         0.725741   0.541028  0.422995  116968.
 ↪000000  0.541028    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.725741   0.541028  0.422995  116968.
 ↪000000  0.541028    0
Logistic Regression       Heart Disease     0.925548   0.111808  0.199515  58484.
 ↪000000   0.551407    0
                          No Heart Disease  0.527356   0.991006  0.688390  58484.
 ↪000000   0.551407    0
                          accuracy          0.551407   0.551407  0.551407  0.
 ↪551407       0.551407    0
                          macro avg         0.726452   0.551407  0.443953  116968.
 ↪000000  0.551407    0
                          weighted avg      0.726452   0.551407  0.443953  116968.
 ↪000000  0.551407    0
Naive Bayes               Heart Disease     0.786760   0.503368  0.613939  58484.
 ↪000000   0.683469    0
                          No Heart Disease  0.634884   0.863570  0.731776  58484.
 ↪000000   0.683469    0
                          accuracy          0.683469   0.683469  0.683469  0.
 ↪683469       0.683469    0
                          macro avg         0.710822   0.683469  0.672858  116968.
 ↪000000  0.683469    0
                          weighted avg      0.710822   0.683469  0.672858  116968.
 ↪000000  0.683469    0
Random Forest Classifier  Heart Disease     0.873803   0.107619  0.191636  58484.
 ↪000000   0.546038    0
                          No Heart Disease  0.524530   0.984457  0.684402  58484.
 ↪000000   0.546038    0
                          accuracy          0.546038   0.546038  0.546038  0.
 ↪546038       0.546038    0
                          macro avg         0.699166   0.546038  0.438019  116968.
 ↪000000  0.546038    0
                          weighted avg      0.699166   0.546038  0.438019  116968.
 ↪000000  0.546038    0
Name: dummy, dtype: int64

OverSampling Methods
--------------------
random
--------------------
```

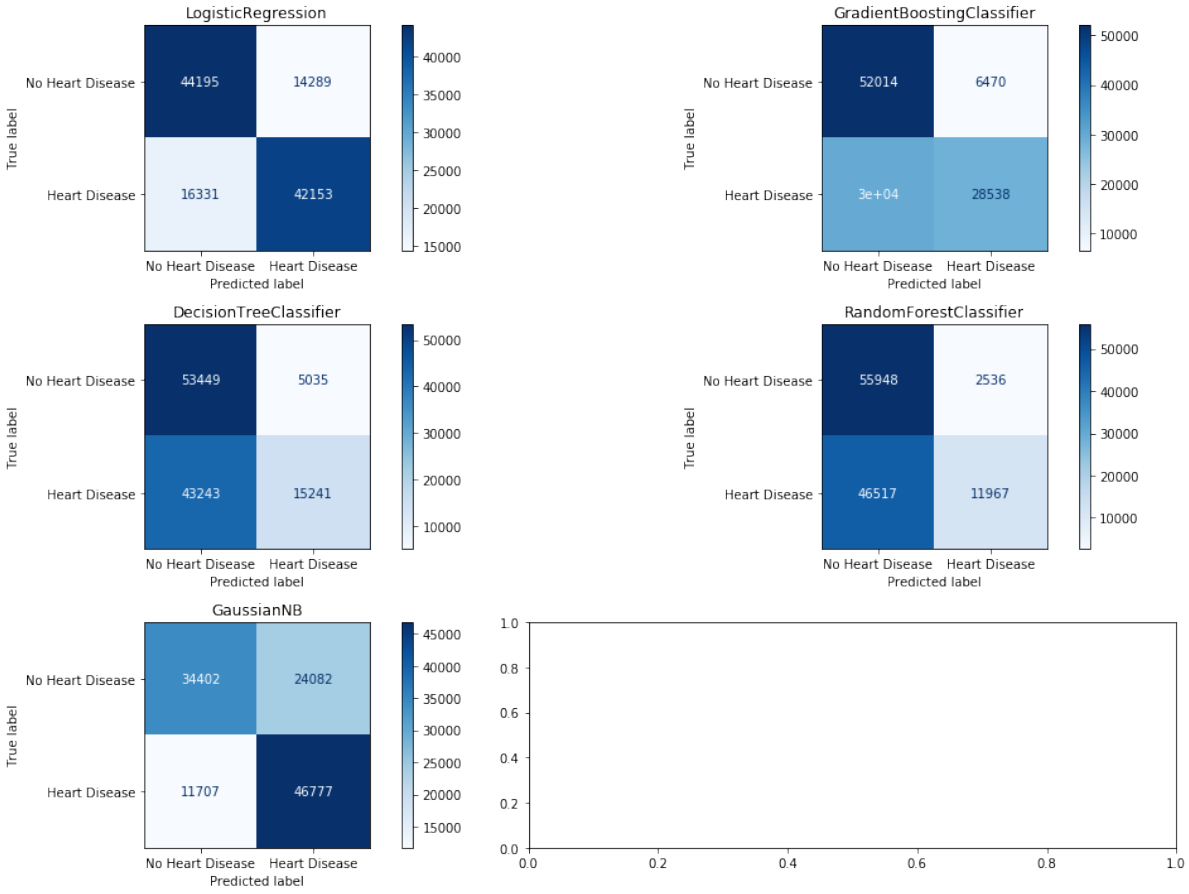



```
Algorithm with the highest accuracy: ['Gradient Boost Classifier', 0.
 ↪7643971000615553]
Algorithm with the highest macro recall:
        ['Gradient Boost Classifier', 0.7643971000615553]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.7655902916773867]
Algorithm with the highest AUC:
        ['Gradient Boost Classifier', 0.7643971000615553]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.754408   0.231892  0.354743  58484.
 ↪000000   0.578201    0
                            No Heart Disease  0.546201   0.924509  0.686699  58484.
 ↪000000   0.578201    0
                            accuracy          0.578201   0.578201  0.578201  0.
 ↪578201       0.578201    0
                            macro avg         0.650305   0.578201  0.520721  116968.
 ↪000000  0.578201    0
                            weighted avg      0.650305   0.578201  0.520721  116968.
 ↪000000  0.578201    0
Gradient Boost Classifier   Heart Disease     0.747789   0.797911  0.772037  58484.
 ↪000000   0.764397    0
                            No Heart Disease  0.783392   0.730884  0.756227  58484.
 ↪000000   0.764397    0
                            accuracy          0.764397   0.764397  0.764397  0.
 ↪764397       0.764397    0
```

(continues on next page)

(continued from previous page)

```
                          macro avg         0.765590   0.764397  0.764132  116968.
 ↪000000  0.764397    0
                          weighted avg      0.765590   0.764397  0.764132  116968.
 ↪000000  0.764397    0
Logistic Regression       Heart Disease     0.755979   0.776623  0.766162  58484.
 ↪000000   0.762969    0
                          No Heart Disease  0.770352   0.749316  0.759688  58484.
 ↪000000   0.762969    0
                          accuracy          0.762969   0.762969  0.762969  0.
 ↪762969       0.762969    0
                          macro avg         0.763166   0.762969  0.762925  116968.
 ↪000000  0.762969    0
                          weighted avg      0.763166   0.762969  0.762925  116968.
 ↪000000  0.762969    0
Naive Bayes               Heart Disease     0.754154   0.647203  0.696597  58484.
 ↪000000   0.718111    0
                          No Heart Disease  0.691021   0.789019  0.736776  58484.
 ↪000000   0.718111    0
                          accuracy          0.718111   0.718111  0.718111  0.
 ↪718111       0.718111    0
                          macro avg         0.722588   0.718111  0.716686  116968.
 ↪000000  0.718111    0
                          weighted avg      0.722588   0.718111  0.716686  116968.
 ↪000000  0.718111    0
Random Forest Classifier  Heart Disease     0.836629   0.204723  0.328951  58484.
 ↪000000   0.582373    0
                          No Heart Disease  0.546928   0.960023  0.696856  58484.
 ↪000000   0.582373    0
                          accuracy          0.582373   0.582373  0.582373  0.
 ↪582373       0.582373    0
                          macro avg         0.691779   0.582373  0.512903  116968.
 ↪000000  0.582373    0
                          weighted avg      0.691779   0.582373  0.512903  116968.
 ↪000000  0.582373    0
Name: dummy, dtype: int64

smote
--------------------
```

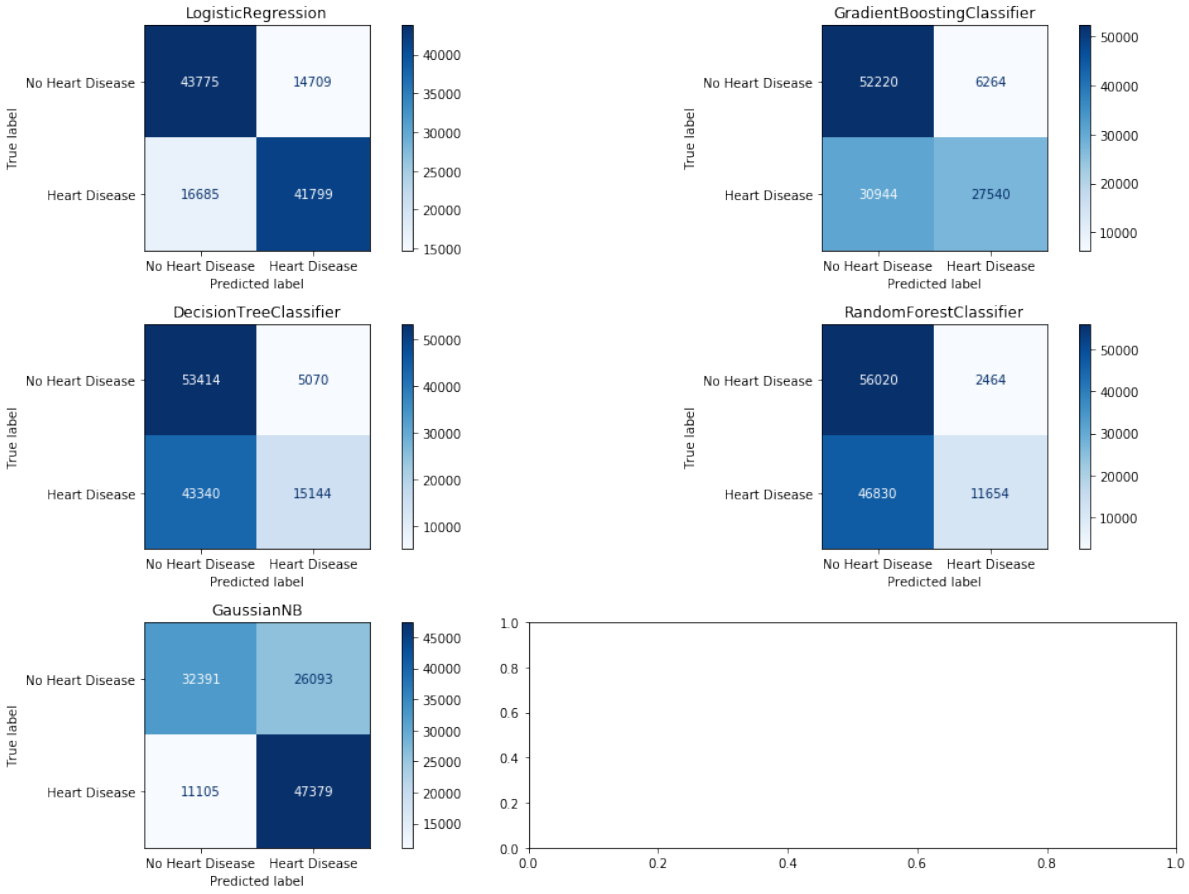



```
Algorithm with the highest accuracy: ['Logistic Regression ', 0.7382190000683948]
Algorithm with the highest macro recall:
        ['Logistic Regression ', 0.7382190000683948]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7385097659900086]
Algorithm with the highest AUC:
        ['Logistic Regression ', 0.7382190000683948]
model                       index              precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease      0.751677   0.260601  0.387024  58484.
 ↪000000   0.587255    0
                            No Heart Disease   0.552776   0.913908  0.688882  58484.
 ↪000000   0.587255    0
                            accuracy           0.587255   0.587255  0.587255  0.
 ↪587255       0.587255    0
                            macro avg          0.652226   0.587255  0.537953  116968.
 ↪000000  0.587255    0
                            weighted avg       0.652226   0.587255  0.537953  116968.
 ↪000000  0.587255    0
Gradient Boost Classifier   Heart Disease      0.815185   0.487963  0.610491  58484.
 ↪000000   0.688667    0
                            No Heart Disease   0.634627   0.889371  0.740708  58484.
 ↪000000   0.688667    0
                            accuracy           0.688667   0.688667  0.688667  0.
 ↪688667       0.688667    0
                            macro avg          0.724906   0.688667  0.675599  116968.
 ↪000000  0.688667    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.724906   0.688667  0.675599  116968.
 ↪000000  0.688667    0
Logistic Regression       Heart Disease     0.746837   0.720761  0.733568  58484.
 ↪000000   0.738219    0
                          No Heart Disease  0.730182   0.755677  0.742711  58484.
 ↪000000   0.738219    0
                          accuracy          0.738219   0.738219  0.738219  0.
 ↪738219       0.738219    0
                          macro avg         0.738510   0.738219  0.738139  116968.
 ↪000000  0.738219    0
                          weighted avg      0.738510   0.738219  0.738139  116968.
 ↪000000  0.738219    0
Naive Bayes               Heart Disease     0.660142   0.799826  0.723302  58484.
 ↪000000   0.694027    0
                          No Heart Disease  0.746102   0.588229  0.657826  58484.
 ↪000000   0.694027    0
                          accuracy          0.694027   0.694027  0.694027  0.
 ↪694027       0.694027    0
                          macro avg         0.703122   0.694027  0.690564  116968.
 ↪000000  0.694027    0
                          weighted avg      0.703122   0.694027  0.690564  116968.
 ↪000000  0.694027    0
Random Forest Classifier  Heart Disease     0.825140   0.204620  0.327921  58484.
 ↪000000   0.580629    0
                          No Heart Disease  0.546021   0.956638  0.695226  58484.
 ↪000000   0.580629    0
                          accuracy          0.580629   0.580629  0.580629  0.
 ↪580629       0.580629    0
                          macro avg         0.685580   0.580629  0.511574  116968.
 ↪000000  0.580629    0
                          weighted avg      0.685580   0.580629  0.511574  116968.
 ↪000000  0.580629    0
Name: dummy, dtype: int64

adasyn
-------------------
```

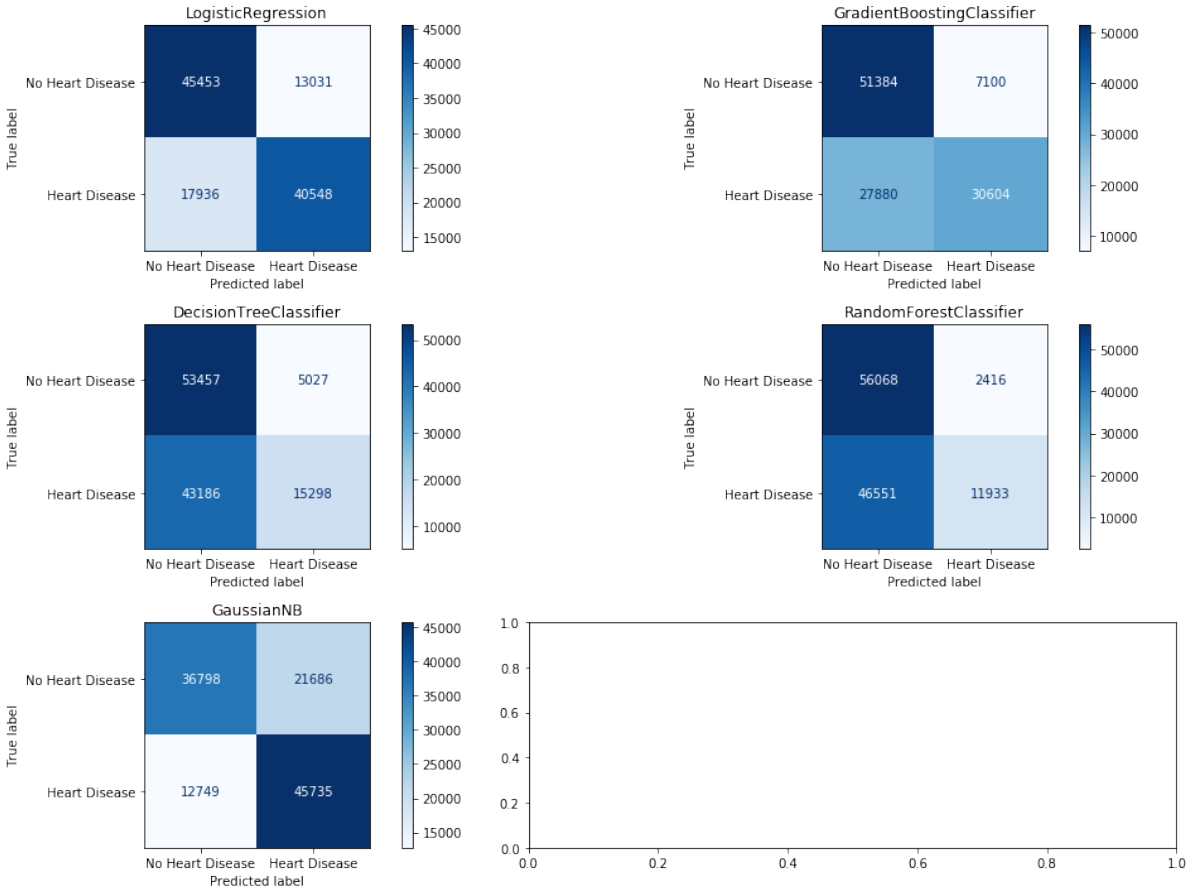



```
Algorithm with the highest accuracy: ['Logistic Regression ', 0.7316018056220505]
Algorithm with the highest macro recall:
        ['Logistic Regression ', 0.7316018056220505]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7318664957489149]
Algorithm with the highest AUC:
        ['Logistic Regression ', 0.7316018056220505]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.749184   0.258943  0.384864   58484.
 ↪000000    0.586126    0
                            No Heart Disease  0.552060   0.913310  0.688156   58484.
 ↪000000    0.586126    0
                            accuracy          0.586126   0.586126  0.586126   0.
 ↪586126        0.586126    0
                            macro avg         0.650622   0.586126  0.536510   116968.
 ↪000000  0.586126    0
                            weighted avg      0.650622   0.586126  0.536510   116968.
 ↪000000  0.586126    0
Gradient Boost Classifier   Heart Disease     0.814696   0.470898  0.596827   58484.
 ↪000000    0.681896    0
                            No Heart Disease  0.627916   0.892894  0.737321   58484.
 ↪000000    0.681896    0
                            accuracy          0.681896   0.681896  0.681896   0.
 ↪681896        0.681896    0
                            macro avg         0.721306   0.681896  0.667074   116968.
 ↪000000  0.681896    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.721306   0.681896  0.667074  116968.
 ↪000000  0.681896    0
Logistic Regression       Heart Disease     0.739701   0.714708  0.726990  58484.
 ↪000000   0.731602    0
                          No Heart Disease  0.724032   0.748495  0.736061  58484.
 ↪000000   0.731602    0
                          accuracy          0.731602   0.731602  0.731602  0.
 ↪731602       0.731602    0
                          macro avg         0.731866   0.731602  0.731525  116968.
 ↪000000  0.731602    0
                          weighted avg      0.731866   0.731602  0.731525  116968.
 ↪000000  0.731602    0
Naive Bayes               Heart Disease     0.644858   0.810119  0.718103  58484.
 ↪000000   0.681981    0
                          No Heart Disease  0.744689   0.553844  0.635242  58484.
 ↪000000   0.681981    0
                          accuracy          0.681981   0.681981  0.681981  0.
 ↪681981       0.681981    0
                          macro avg         0.694774   0.681981  0.676673  116968.
 ↪000000  0.681981    0
                          weighted avg      0.694774   0.681981  0.676673  116968.
 ↪000000  0.681981    0
Random Forest Classifier  Heart Disease     0.825471   0.199268  0.321038  58484.
 ↪000000   0.578568    0
                          No Heart Disease  0.544677   0.957869  0.694460  58484.
 ↪000000   0.578568    0
                          accuracy          0.578568   0.578568  0.578568  0.
 ↪578568       0.578568    0
                          macro avg         0.685074   0.578568  0.507749  116968.
 ↪000000  0.578568    0
                          weighted avg      0.685074   0.578568  0.507749  116968.
 ↪000000  0.578568    0
Name: dummy, dtype: int64

border1
-------------------
```

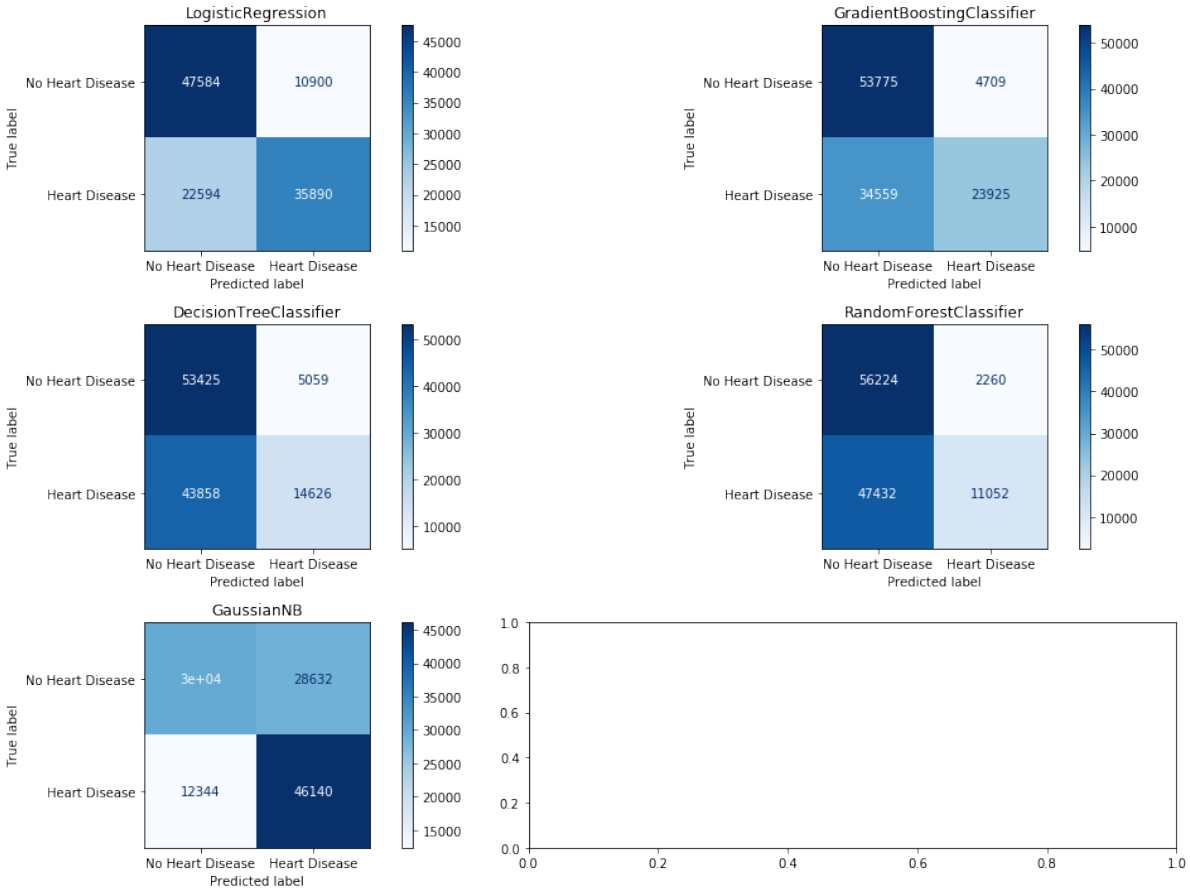



```
Algorithm with the highest accuracy: ['Logistic Regression ', 0.7352523767184187]
Algorithm with the highest macro recall:
        ['Logistic Regression ', 0.7352523767184187]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7369188698917095]
Algorithm with the highest AUC:
        ['Logistic Regression ', 0.7352523767184187]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.752669   0.261576  0.388230  58484.
 ↪000000   0.587810    0
                            No Heart Disease  0.553139   0.914045  0.689203  58484.
 ↪000000   0.587810    0
                            accuracy          0.587810   0.587810  0.587810  0.
 ↪587810       0.587810    0
                            macro avg         0.652904   0.587810  0.538716  116968.
 ↪000000  0.587810    0
                            weighted avg      0.652904   0.587810  0.538716  116968.
 ↪000000  0.587810    0
Gradient Boost Classifier   Heart Disease     0.811691   0.523288  0.636337  58484.
 ↪000000   0.700944    0
                            No Heart Disease  0.648264   0.878599  0.746058  58484.
 ↪000000   0.700944    0
                            accuracy          0.700944   0.700944  0.700944  0.
 ↪700944       0.700944    0
                            macro avg         0.729978   0.700944  0.691198  116968.
 ↪000000  0.700944    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.729978   0.700944  0.691198  116968.
 ↪000000  0.700944    0
Logistic Regression       Heart Disease     0.756789   0.693318  0.723664  58484.
 ↪000000   0.735252    0
                          No Heart Disease  0.717049   0.777187  0.745908  58484.
 ↪000000   0.735252    0
                          accuracy          0.735252   0.735252  0.735252  0.
 ↪735252       0.735252    0
                          macro avg         0.736919   0.735252  0.734786  116968.
 ↪000000  0.735252    0
                          weighted avg      0.736919   0.735252  0.734786  116968.
 ↪000000  0.735252    0
Naive Bayes               Heart Disease     0.678349   0.782009  0.726500  58484.
 ↪000000   0.705603    0
                          No Heart Disease  0.742689   0.629198  0.681249  58484.
 ↪000000   0.705603    0
                          accuracy          0.705603   0.705603  0.705603  0.
 ↪705603       0.705603    0
                          macro avg         0.710519   0.705603  0.703875  116968.
 ↪000000  0.705603    0
                          weighted avg      0.710519   0.705603  0.703875  116968.
 ↪000000  0.705603    0
Random Forest Classifier  Heart Disease     0.831626   0.204039  0.327681  58484.
 ↪000000   0.581364    0
                          No Heart Disease  0.546371   0.958690  0.696052  58484.
 ↪000000   0.581364    0
                          accuracy          0.581364   0.581364  0.581364  0.
 ↪581364       0.581364    0
                          macro avg         0.688998   0.581364  0.511866  116968.
 ↪000000  0.581364    0
                          weighted avg      0.688998   0.581364  0.511866  116968.
 ↪000000  0.581364    0
Name: dummy, dtype: int64

border2
-------------------
```




```
Algorithm with the highest accuracy: ['Logistic Regression ', 0.7136481772792559]
Algorithm with the highest macro recall:
        ['Logistic Regression ', 0.7136481772792558]
Algorithm with the highest macro precision:
        ['Logistic Regression ', 0.7225457457487992]
Algorithm with the highest AUC:
        ['Logistic Regression ', 0.7136481772792559]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.743002   0.250085  0.374215  58484.
 ↪000000   0.581792    0
                            No Heart Disease  0.549171   0.913498  0.685960  58484.
 ↪000000   0.581792    0
                            accuracy          0.581792   0.581792  0.581792  0.
 ↪581792       0.581792    0
                            macro avg         0.646087   0.581792  0.530088  116968.
 ↪000000  0.581792    0
                            weighted avg      0.646087   0.581792  0.530088  116968.
 ↪000000  0.581792    0
Gradient Boost Classifier   Heart Disease     0.835545   0.409086  0.549255  58484.
 ↪000000   0.664284    0
                            No Heart Disease  0.608769   0.919482  0.732540  58484.
 ↪000000   0.664284    0
                            accuracy          0.664284   0.664284  0.664284  0.
 ↪664284       0.664284    0
                            macro avg         0.722157   0.664284  0.640897  116968.
 ↪000000  0.664284    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.722157   0.664284  0.640897  116968.
 ↪000000  0.664284    0
Logistic Regression       Heart Disease     0.767044   0.613672  0.681840  58484.
 ↪000000   0.713648    0
                          No Heart Disease  0.678047   0.813624  0.739674  58484.
 ↪000000   0.713648    0
                          accuracy          0.713648   0.713648  0.713648  0.
 ↪713648       0.713648    0
                          macro avg         0.722546   0.713648  0.710757  116968.
 ↪000000  0.713648    0
                          weighted avg      0.722546   0.713648  0.710757  116968.
 ↪000000  0.713648    0
Naive Bayes               Heart Disease     0.617076   0.788934  0.692502  58484.
 ↪000000   0.649682    0
                          No Heart Disease  0.707460   0.510430  0.593008  58484.
 ↪000000   0.649682    0
                          accuracy          0.649682   0.649682  0.649682  0.
 ↪649682       0.649682    0
                          macro avg         0.662268   0.649682  0.642755  116968.
 ↪000000  0.649682    0
                          weighted avg      0.662268   0.649682  0.642755  116968.
 ↪000000  0.649682    0
Random Forest Classifier  Heart Disease     0.830228   0.188975  0.307872  58484.
 ↪000000   0.575166    0
                          No Heart Disease  0.542410   0.961357  0.693524  58484.
 ↪000000   0.575166    0
                          accuracy          0.575166   0.575166  0.575166  0.
 ↪575166       0.575166    0
                          macro avg         0.686319   0.575166  0.500698  116968.
 ↪000000  0.575166    0
                          weighted avg      0.686319   0.575166  0.500698  116968.
 ↪000000  0.575166    0
Name: dummy, dtype: int64
```

As we can see in the verbose model output,our best setting occurred with the `Random` over sampling strategy. Across most tracked metrics, we see a significant improvement in model performance compared with no oversampling techniques.

Base (no under sampling techniques) **winners** for tracked metrics:

Metric	Metric	Value
Accuracy	Naive Bayes	0.684
Macro Recall	Naive Bayes	0.684
Macro precision	Gradient Boost Classifier	0.726
AUC	Naive Bayes	0.684

Random over sampling technique winners for tracked metrics:

Metric	Metric	Value
Accuracy	Gradient Boost Classifier	0.764
Macro Recall	Gradient Boost Classifier	0.764
Macro precision	Gradient Boost Classifier	0.765
AUC	Gradient Boost Classifier	0.764

From now on, we choose the Gradient Boost Classifier with random oversampling as our base model. We will now try to

improve the performance of the model by using the feature selection. Although random undersampling achieves similar accuracy as random oversampling, we achieve marginally better auc, precision and recall scores.

4.7 Feature Selection

The idea of feature selection and extraction is to avoid the curse of dimensionality. This refers to the fact that as we move to higher dimension input feature spaces the volume of the space grows rapidly and we end up with very few instances per unit volume, i.e. we have very sparse sampling of the space of possible instances making modelling difficult.

Feature Selection: It is clear from what we have seen that a good feature engineering idea might be to choose a subset of the features available to reduce the dimension of the feature space. This act is called feature selection. One way of doing this is to try out different permutations of features increasing the numbers of features involved as you proceed and calculate machine learning performance. This is rarely practical though. More efficient approaches include wrapper, filter and embedded methods.

We decide to explore the following methods:

- Perform PCA anaylsis and idenitify the variables that most contribute to the
- A simple filter method:
  - Identify input features having high correlation with target variable.
  - Identify input features that have a low correlation with other independent variables
  - Find the information gain or mutual information of the independent variable with respect to a target variable

4.7.1 PCA

PCA is mathematically defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

We decide to get the top 5 features that contribute most to the first principal component and the top 5 features that contribute most to the second principal compoenent.

```
def pca_most_important_features(df: pd.DataFrame) -> List[str]:
    """
    Retrieve the top 10 features that contribute most
        variation to the top 2 principal components
    """
    model = PCA(n_components=2).fit(df)

    # number of components
    n_pcs: int = model.components_.shape[0]

    most_important_features_indicies: List[int] = []
    for i in range(n_pcs):
        top_5 = np.argpartition(np.abs(model.components_[i]), -5)[-5:].tolist()
        most_important_features_indicies.extend(top_5)

    most_important_features_indicies = list(set(most_important_features_indicies))
    initial_feature_names = df.columns
    most_important_names  = [initial_feature_names[i] for i in most_important_
↪features_indicies]
```

(continues on next page)

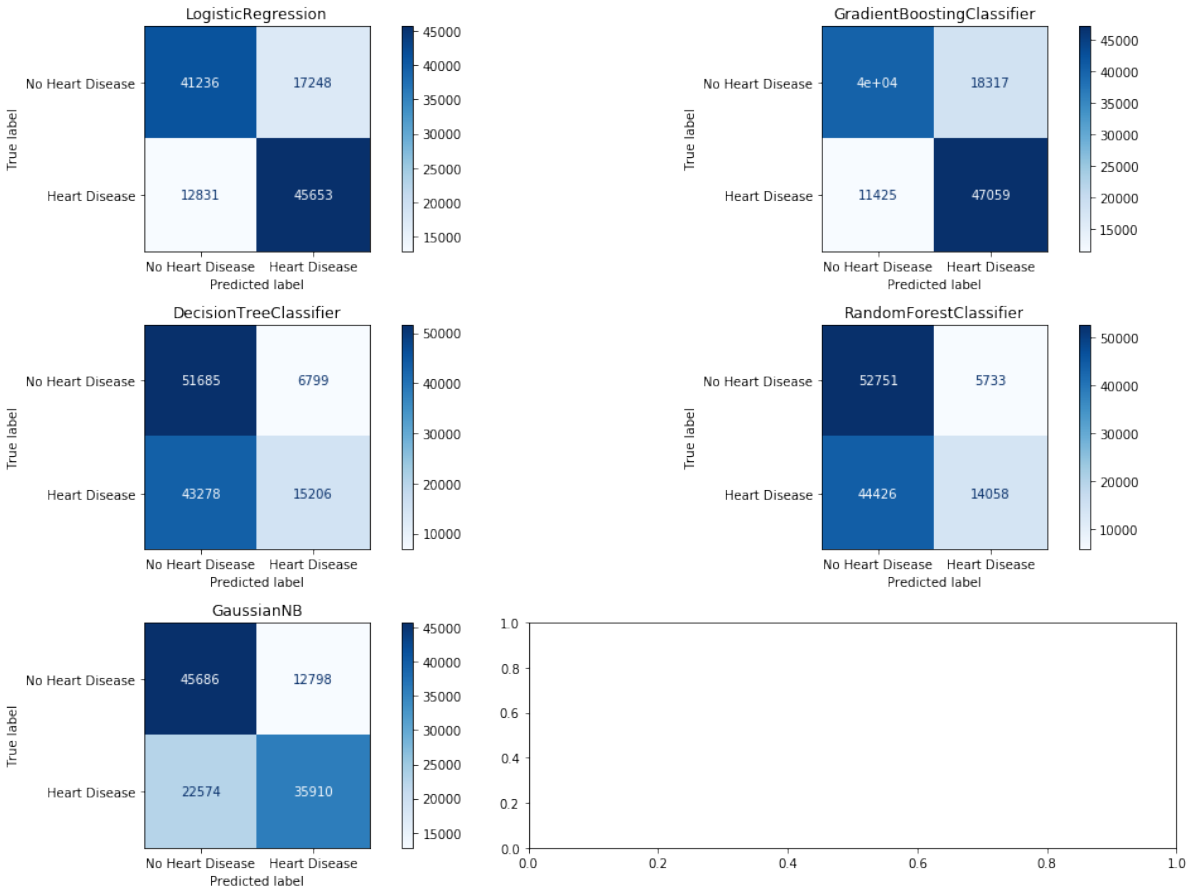
(continued from previous page)

```
    return most_important_names
```

```
pca_features = pca_most_important_features(heart_disease_dataset_standardized)
pca_features
```

```
['BMI',
 'PhysicalHealth',
 'MentalHealth',
 'AgeCategory',
 'GenHealth',
 'SleepTime',
 'BMI_Bin',
 'LOG_BMI']
```

```
X_resampled, y_resampled = oversampler_dict['random'].fit_resample(X_train.copy(), y_
↪train.copy())
run_exps(X_resampled[pca_features], X_test[pca_features], y_resampled, y_test)
```




```
Algorithm with the highest accuracy: ['Gradient Boost Classifier', 0.
↪7457253265850489]
```

(continues on next page)

(continued from previous page)

```
Algorithm with the highest macro recall:
        ['Gradient Boost Classifier', 0.7457253265850489]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.7491858337247853]
Algorithm with the highest AUC:
        ['Gradient Boost Classifier', 0.7457253265850489]
model                       index             precision  recall    f1-score ␣
 ↪support        auc
Decision Tree Classifier    Heart Disease     0.691025   0.260003  0.377840  58484.
 ↪000000   0.571874    0
                            No Heart Disease  0.544265   0.883746  0.673653  58484.
 ↪000000   0.571874    0
                            accuracy          0.571874   0.571874  0.571874  0.
 ↪571874       0.571874    0
                            macro avg         0.617645   0.571874  0.525747  116968.
 ↪000000  0.571874    0
                            weighted avg      0.617645   0.571874  0.525747  116968.
 ↪000000  0.571874    0
Gradient Boost Classifier   Heart Disease     0.719821   0.804647  0.759874  58484.
 ↪000000   0.745725    0
                            No Heart Disease  0.778551   0.686803  0.729805  58484.
 ↪000000   0.745725    0
                            accuracy          0.745725   0.745725  0.745725  0.
 ↪745725       0.745725    0
                            macro avg         0.749186   0.745725  0.744839  116968.
 ↪000000  0.745725    0
                            weighted avg      0.749186   0.745725  0.744839  116968.
 ↪000000  0.745725    0
Logistic Regression         Heart Disease     0.725791   0.780607  0.752202  58484.
 ↪000000   0.742844    0
                            No Heart Disease  0.762683   0.705082  0.732752  58484.
 ↪000000   0.742844    0
                            accuracy          0.742844   0.742844  0.742844  0.
 ↪742844       0.742844    0
                            macro avg         0.744237   0.742844  0.742477  116968.
 ↪000000  0.742844    0
                            weighted avg      0.744237   0.742844  0.742477  116968.
 ↪000000  0.742844    0
Naive Bayes                 Heart Disease     0.737251   0.614014  0.670013  58484.
 ↪000000   0.697593    0
                            No Heart Disease  0.669294   0.781171  0.720918  58484.
 ↪000000   0.697593    0
                            accuracy          0.697593   0.697593  0.697593  0.
 ↪697593       0.697593    0
                            macro avg         0.703272   0.697593  0.695465  116968.
 ↪000000  0.697593    0
                            weighted avg      0.703272   0.697593  0.695465  116968.
 ↪000000  0.697593    0
Random Forest Classifier    Heart Disease     0.710323   0.240373  0.359195  58484.
 ↪000000   0.571173    0
                            No Heart Disease  0.542834   0.901973  0.677768  58484.
 ↪000000   0.571173    0
                            accuracy          0.571173   0.571173  0.571173  0.
 ↪571173       0.571173    0
                            macro avg         0.626579   0.571173  0.518481  116968.
 ↪000000  0.571173    0
```

(continues on next page)

(continued from previous page)

```
                          weighted avg      0.626579   0.571173  0.518481  116968.
↪000000  0.571173    0
Name: dummy, dtype: int64
```

	index	precision	recall	f1-score	support
0	No Heart Disease	0.762683	0.705082	0.732752	58484.000000
1	Heart Disease	0.725791	0.780607	0.752202	58484.000000
2	accuracy	0.742844	0.742844	0.742844	0.742844
3	macro avg	0.744237	0.742844	0.742477	116968.000000
4	weighted avg	0.744237	0.742844	0.742477	116968.000000
5	No Heart Disease	0.778551	0.686803	0.729805	58484.000000
6	Heart Disease	0.719821	0.804647	0.759874	58484.000000
7	accuracy	0.745725	0.745725	0.745725	0.745725
8	macro avg	0.749186	0.745725	0.744839	116968.000000
9	weighted avg	0.749186	0.745725	0.744839	116968.000000
10	No Heart Disease	0.544265	0.883746	0.673653	58484.000000
11	Heart Disease	0.691025	0.260003	0.377840	58484.000000
12	accuracy	0.571874	0.571874	0.571874	0.571874
13	macro avg	0.617645	0.571874	0.525747	116968.000000
14	weighted avg	0.617645	0.571874	0.525747	116968.000000
15	No Heart Disease	0.542834	0.901973	0.677768	58484.000000
16	Heart Disease	0.710323	0.240373	0.359195	58484.000000
17	accuracy	0.571173	0.571173	0.571173	0.571173
18	macro avg	0.626579	0.571173	0.518481	116968.000000
19	weighted avg	0.626579	0.571173	0.518481	116968.000000
20	No Heart Disease	0.669294	0.781171	0.720918	58484.000000
21	Heart Disease	0.737251	0.614014	0.670013	58484.000000
22	accuracy	0.697593	0.697593	0.697593	0.697593
23	macro avg	0.703272	0.697593	0.695465	116968.000000
24	weighted avg	0.703272	0.697593	0.695465	116968.000000

	model	auc	dummy
0	Logistic Regression	0.742844	None
1	Logistic Regression	0.742844	None
2	Logistic Regression	0.742844	None
3	Logistic Regression	0.742844	None
4	Logistic Regression	0.742844	None
5	Gradient Boost Classifier	0.745725	None
6	Gradient Boost Classifier	0.745725	None
7	Gradient Boost Classifier	0.745725	None
8	Gradient Boost Classifier	0.745725	None
9	Gradient Boost Classifier	0.745725	None
10	Decision Tree Classifier	0.571874	None
11	Decision Tree Classifier	0.571874	None
12	Decision Tree Classifier	0.571874	None
13	Decision Tree Classifier	0.571874	None
14	Decision Tree Classifier	0.571874	None
15	Random Forest Classifier	0.571173	None
16	Random Forest Classifier	0.571173	None
17	Random Forest Classifier	0.571173	None
18	Random Forest Classifier	0.571173	None
19	Random Forest Classifier	0.571173	None
20	Naive Bayes	0.697593	None
21	Naive Bayes	0.697593	None
22	Naive Bayes	0.697593	None

(continues on next page)

(continued from previous page)

```
23                Naive Bayes  0.697593  None
24                Naive Bayes  0.697593  None
```

We note a marginal decrease in model performance when using the top 5 features from the 2 principal components. This is likely due to the fact that the top 5 features from the 2 principal components are not the most important features in the dataset.

4.7.2 Filter Methods

A simple filter method:

- `Identify input features having high correlation with target variable`: We want to keep features with only a high correlation with the target variable. This implies that the input feature has a high influence in predicting the target variable. We set the threshold to the absolute value of 0.2. We keep input features only if the correlation of the input feature with the target variable is greater than 0.2. Our analysis reveled most variables have little if all correlation to our target variable
- `Find the information gain or mutual information of the independent variable with respect to a target variable`

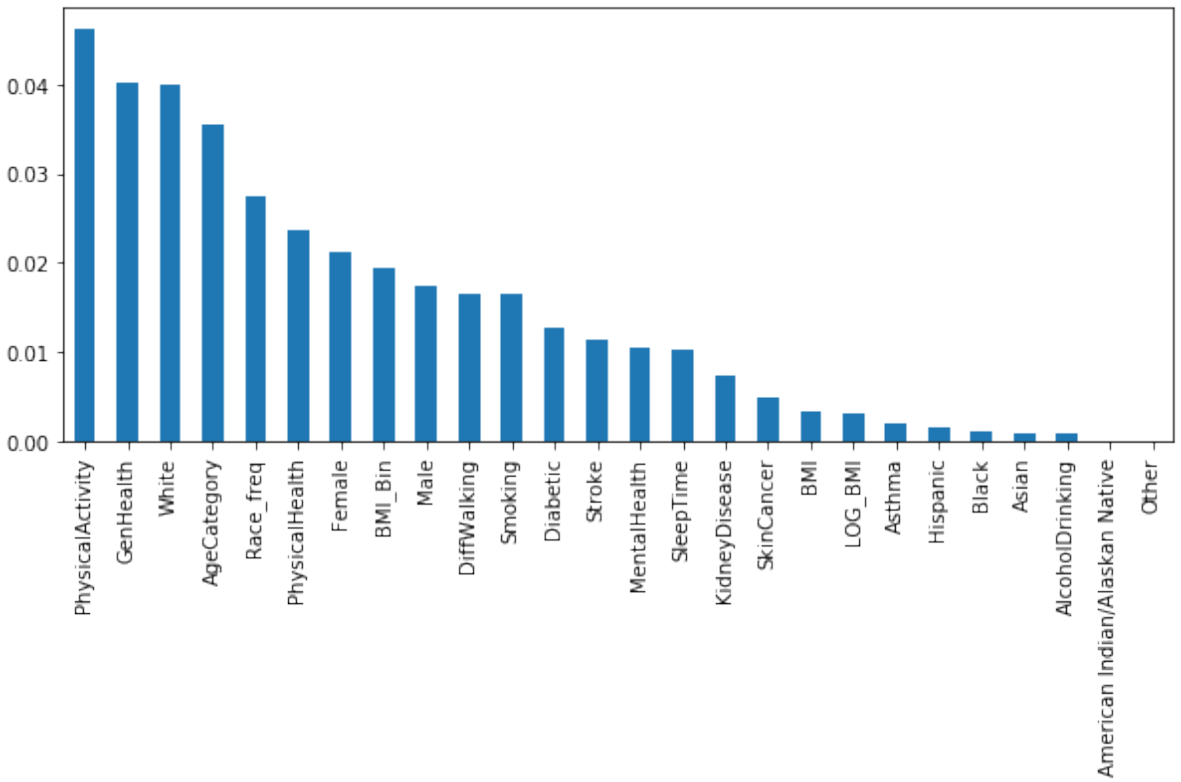
```
def idenity_high_corr_features(df: pd.DataFrame) -> List[str]:
    importances = df.drop(
        "HeartDisease", axis=1).apply(
            lambda x: x.corr(df.HeartDisease))

    indices = np.argsort(importances)
    important_feature_names = []
    for i in range(0, len(indices)):
        if np.abs(importances[i]) > 0.2:
            important_feature_names.append(
                df.columns[i])
    return important_feature_names
```

```
high_corr_features = idenity_high_corr_features(heart_disease_dataset_standardized)
high_corr_features
```

```
['MentalHealth', 'DiffWalking', 'PhysicalActivity']
```

```
mi = mutual_info_classif(heart_disease_dataset_standardized.drop("HeartDisease",␣
↪axis=1),
                         heart_disease_dataset_standardized["HeartDisease"])
mi = pd.Series(mi)
mi.index = heart_disease_dataset_standardized.drop("HeartDisease", axis=1).columns
mi.sort_values(ascending=False).plot.bar(figsize=(10, 4))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f932dce9090>
```




```
top_10_mi = mi.sort_values(ascending=False)[:10].index.tolist()
top_10_mi
```

```
['PhysicalActivity',
 'GenHealth',
 'White',
 'AgeCategory',
 'Race_freq',
 'PhysicalHealth',
 'Female',
 'BMI_Bin',
 'Male',
 'DiffWalking']
```

We choose to pick the top 10 features, ranked by mutual information, along with the high correlation features to use in our model.

```
filter_features = list(set(top_10_mi) | (set(high_corr_features)))
filter_features
```

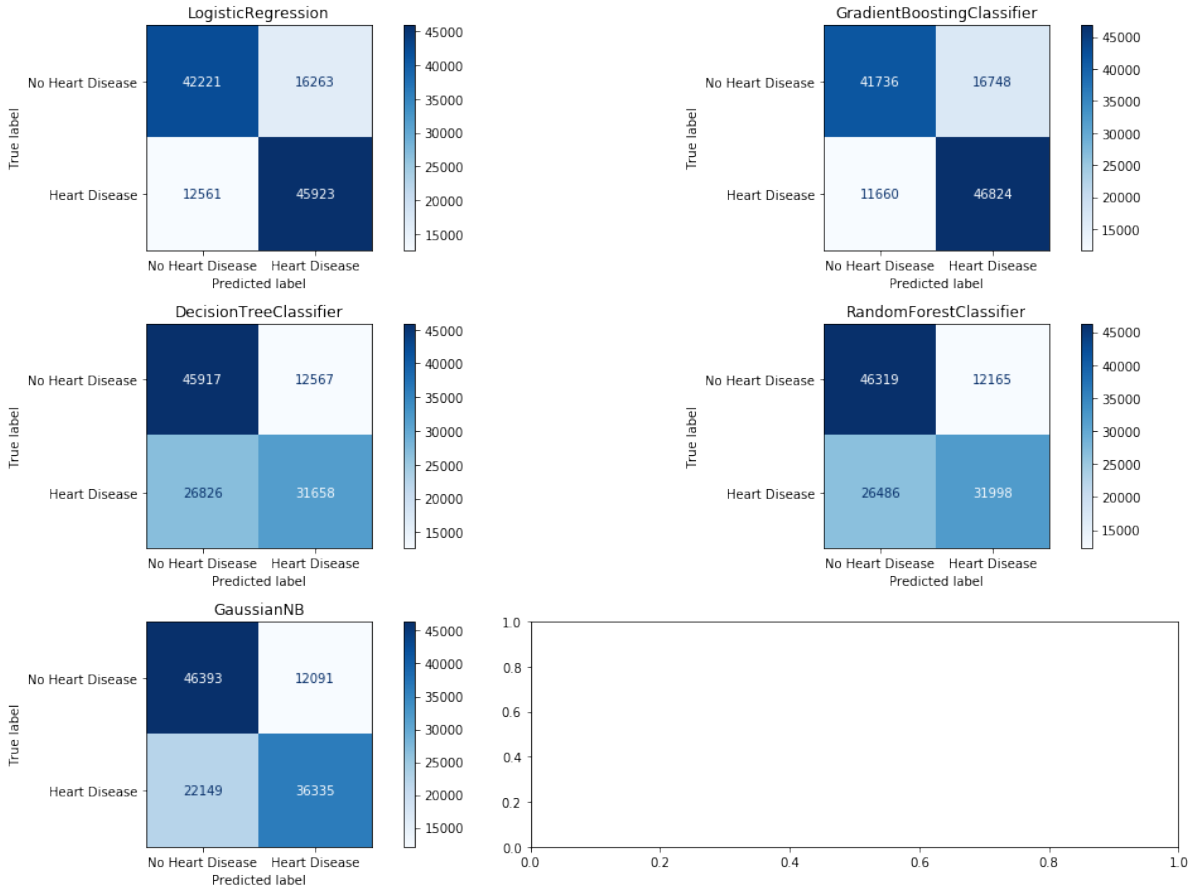
```
['Male',
 'GenHealth',
 'BMI_Bin',
 'PhysicalActivity',
 'Race_freq',
 'PhysicalHealth',
 'Female',
 'White',
```

(continues on next page)

(continued from previous page)

```
    'DiffWalking',
    'AgeCategory',
    'MentalHealth']
```

```
X_resampled, y_resampled = oversampler_dict['random'].fit_resample(X_train.copy(), y_
↪train.copy())
run_exps(X_resampled[filter_features], X_test[filter_features], y_resampled, y_test)
```




```
Algorithm with the highest accuracy: ['Gradient Boost Classifier', 0.
↪7571301552561385]
Algorithm with the highest macro recall:
        ['Gradient Boost Classifier', 0.7571301552561385]
Algorithm with the highest macro precision:
        ['Gradient Boost Classifier', 0.7590911327902984]
Algorithm with the highest AUC:
        ['Gradient Boost Classifier', 0.7571301552561385]
model                       index            precision  recall    f1-score ␣
↪support        auc
Decision Tree Classifier    Heart Disease     0.715839   0.541310  0.616460   58484.
↪000000   0.663216    0
                            No Heart Disease  0.631222   0.785121  0.699810   58484.
↪000000   0.663216    0
                            accuracy          0.663216   0.663216  0.663216   0.
↪663216       0.663216    0
```

(continues on next page)

(continued from previous page)

```
                          macro avg         0.673531   0.663216  0.658135  116968.
↪000000  0.663216    0
                          weighted avg      0.673531   0.663216  0.658135  116968.
↪000000  0.663216    0
Gradient Boost Classifier  Heart Disease     0.736551   0.800629  0.767254  58484.
↪000000   0.757130    0
                          No Heart Disease  0.781632   0.713631  0.746085  58484.
↪000000   0.757130    0
                          accuracy          0.757130   0.757130  0.757130  0.
↪757130       0.757130    0
                          macro avg         0.759091   0.757130  0.756670  116968.
↪000000  0.757130    0
                          weighted avg      0.759091   0.757130  0.756670  116968.
↪000000  0.757130    0
Logistic Regression        Heart Disease     0.738478   0.785223  0.761134  58484.
↪000000   0.753574    0
                          No Heart Disease  0.770709   0.721924  0.745519  58484.
↪000000   0.753574    0
                          accuracy          0.753574   0.753574  0.753574  0.
↪753574       0.753574    0
                          macro avg         0.754594   0.753574  0.753327  116968.
↪000000  0.753574    0
                          weighted avg      0.754594   0.753574  0.753327  116968.
↪000000  0.753574    0
Naive Bayes                Heart Disease     0.750320   0.621281  0.679731  58484.
↪000000   0.707270    0
                          No Heart Disease  0.676855   0.793260  0.730449  58484.
↪000000   0.707270    0
                          accuracy          0.707270   0.707270  0.707270  0.
↪707270       0.707270    0
                          macro avg         0.713588   0.707270  0.705090  116968.
↪000000  0.707270    0
                          weighted avg      0.713588   0.707270  0.705090  116968.
↪000000  0.707270    0
Random Forest Classifier   Heart Disease     0.724543   0.547124  0.623457  58484.
↪000000   0.669559    0
                          No Heart Disease  0.636206   0.791994  0.705604  58484.
↪000000   0.669559    0
                          accuracy          0.669559   0.669559  0.669559  0.
↪669559       0.669559    0
                          macro avg         0.680375   0.669559  0.664530  116968.
↪000000  0.669559    0
                          weighted avg      0.680375   0.669559  0.664530  116968.
↪000000  0.669559    0
Name: dummy, dtype: int64
```

```
              index  precision    recall  f1-score        support  \
0   No Heart Disease   0.770709  0.721924  0.745519   58484.000000
1      Heart Disease   0.738478  0.785223  0.761134   58484.000000
2           accuracy   0.753574  0.753574  0.753574       0.753574
3          macro avg   0.754594  0.753574  0.753327  116968.000000
4       weighted avg   0.754594  0.753574  0.753327  116968.000000
5   No Heart Disease   0.781632  0.713631  0.746085   58484.000000
6      Heart Disease   0.736551  0.800629  0.767254   58484.000000
7           accuracy   0.757130  0.757130  0.757130       0.757130
```

(continues on next page)

(continued from previous page)

```
8          macro avg   0.759091  0.757130  0.756670  116968.000000
9       weighted avg   0.759091  0.757130  0.756670  116968.000000
10  No Heart Disease   0.631222  0.785121  0.699810   58484.000000
11     Heart Disease   0.715839  0.541310  0.616460   58484.000000
12          accuracy   0.663216  0.663216  0.663216       0.663216
13         macro avg   0.673531  0.663216  0.658135  116968.000000
14      weighted avg   0.673531  0.663216  0.658135  116968.000000
15  No Heart Disease   0.636206  0.791994  0.705604   58484.000000
16     Heart Disease   0.724543  0.547124  0.623457   58484.000000
17          accuracy   0.669559  0.669559  0.669559       0.669559
18         macro avg   0.680375  0.669559  0.664530  116968.000000
19      weighted avg   0.680375  0.669559  0.664530  116968.000000
20  No Heart Disease   0.676855  0.793260  0.730449   58484.000000
21     Heart Disease   0.750320  0.621281  0.679731   58484.000000
22          accuracy   0.707270  0.707270  0.707270       0.707270
23         macro avg   0.713588  0.707270  0.705090  116968.000000
24      weighted avg   0.713588  0.707270  0.705090  116968.000000

                        model       auc dummy
0        Logistic Regression   0.753574  None
1        Logistic Regression   0.753574  None
2        Logistic Regression   0.753574  None
3        Logistic Regression   0.753574  None
4        Logistic Regression   0.753574  None
5   Gradient Boost Classifier  0.757130  None
6   Gradient Boost Classifier  0.757130  None
7   Gradient Boost Classifier  0.757130  None
8   Gradient Boost Classifier  0.757130  None
9   Gradient Boost Classifier  0.757130  None
10   Decision Tree Classifier  0.663216  None
11   Decision Tree Classifier  0.663216  None
12   Decision Tree Classifier  0.663216  None
13   Decision Tree Classifier  0.663216  None
14   Decision Tree Classifier  0.663216  None
15   Random Forest Classifier  0.669559  None
16   Random Forest Classifier  0.669559  None
17   Random Forest Classifier  0.669559  None
18   Random Forest Classifier  0.669559  None
19   Random Forest Classifier  0.669559  None
20               Naive Bayes  0.707270  None
21               Naive Bayes  0.707270  None
22               Naive Bayes  0.707270  None
23               Naive Bayes  0.707270  None
24               Naive Bayes  0.707270  None
```

Our filter method results in a marginal decrease in the evaluation metrics tracked when compared to with just the over-sampling method applied. Although, we do achieve a slight increase in the all scores when compared to our PCA analysis method.

4.8 Hyperparameter finetuning

Our best setting was the gradient booting classifier method with the random oversampling method applied. We will now try to improve the performance of the model by finetuning the hyperparameters of the model. We use cross validation on the trainig data to fine tune our model, focusing on the accuracy metric.

```
X_resampled, y_resampled = oversampler_dict['random'].fit_resample(X_train.copy(), y_
↪train.copy())
```

```
gbc = GradientBoostingClassifier()
parameters = {
    "n_estimators":[5,50,250,500],
    "max_depth":[1,3,5,7,9],
    "learning_rate":[0.01,0.1,1,10,100]
}
cv = GridSearchCV(gbc,parameters,cv=5, scoring='accuracy')
cv.fit(X_resampled,y_resampled)
```

```
def display(results):
    print(f'Best parameters are: {results.best_params_}')
    print("\n")
    mean_score = results.cv_results_['mean_test_score']
    std_score = results.cv_results_['std_test_score']
    params = results.cv_results_['params']
    for mean,std,params in zip(mean_score,std_score,params):
        print(f'{round(mean,3)} + or -{round(std,3)} for the {params}')
```

```
display(cv)
```

4.9 Pickle Model

Pickle is the standard way of serializing objects in Python.

You can use the pickle operation to serialize your machine learning algorithms and save the serialized format to a file.

Later we will load this file to deserialize your model and use it to make new predictions in our web app.

```
model = GradientBoostingClassifier()
```

```
model.fit(X_resampled, y_resampled)
```

```
GaussianNB()
```

```
pickle.dump(model, open('../app/model/finalized_model.sav', 'wb'))
```

CHAPTER

FIVE

CONCLUSION

In this investigation, we investigate the predictability of patients who have or did not have heart disease. Engineered features included a log transformed BMI variable to offset the effect of outliers in our model, a substance abuse binary variable (whether they were a heavy smoker or drinker) and two principal components of the dataset, in an attempt to reduce the dimensionality. In addition, we augment the existing data by adding participants general BMI classification (underweight, normal weight, overweight obese). This enables us, to examine not only the the risk factors for heart disease but also the risk factors for heart disease in the context of a patient's BMI. Furthermore, our method of using K-means clustering to segment patients into groups with similar risk factors for heart disease is a our approach to understanding the risk factors for heart disease. We propose that this method of segmenting patients into groups with similar risk factors for heart disease can be used to identify patients that are at higher risk of developing heart disease and thus, we added this feature into our final model

Our most significant results are the following:

1. Exploratory data analysis contradicts patients own self-reported health status. The majority of patients who have heart disease report having good or very good health, and the majority of patients who do not have heart disease report having fair or poor health. This suggests that the patients themselves are not aware of their heart disease status.
2. The most significant risk factors for heart disease for the general population are smoking, alcohol drinking, obesity, and diabetes.
3. Similarity, our findings suggest that patients who are obese and have a substance abuse problem are at a significantly higher risk of developing heart disease. This is not surprising as obesity is a well known risk factor for heart disease. However, the fact that patients who are obese and have a substance abuse problem are at a significantly higher risk of developing heart disease in the context of their BMI is more interesting. While this is not surprising, it does suggest that there are other factors that impact the risk of heart disease in obese patients.
4. Out of all the undersupplying and oversampling techniques we investigated to overcome the class imbalance problem in our dataset, random oversampling seemed to be most fruitful. This is because it oversampled the minority class (patients who have heart disease) to the same number of samples as the majority class (patients who do not have heart disease). This is important because it ensures that the model is not biased towards the majority class.
5. Our final model was serialized using the pickle library. This allowed us to create a responsive web application. More details on the web app can be found in the app directory.

5.1 Future work

As an extension to this work, and some sort of limitation to the work performed here, different types of classifiers can be included in the analysis and more in depth sensitivity analysis can be performed on these classifiers, also an extension can be made by applying same analysis to other bioinformatics diseases' datasets, and see the performance of these classifiers to classify and predict these diseases. In addition, we would like to investigate the use of deeper models. Similar endeavors have shown to be fruitful, albeit often decreasing the interoperability of results. Finally, we are interested in incorporating other features about the subjects such as socio-economic status, heart disease prevalence in their family (measured on some continuum), their blood pressure and cholesterol levels, and their dietary habitats. We hope such features might uncover specific genetic components patterns or behavioural aspects that might increase or decrease the likehood of heart disease.

CHAPTER

SIX

REFERENCES

@misc{laya_healthcare, title={Worrying statistics about heart disease}, url={https://www.layahealthcare.ie/pressandmedia/pressreleases/worrying-stats-highlight-heart-disease-risk-among-irish-adults-.html#_ftn3}, year={2020}, journal={Laya Healthcare}}

BIBLIOGRAPHY

[1] Worrying statistics about heart disease. 2020. URL: https://www.layahealthcare.ie/pressandmedia/pressreleases/worrying-stats-highlight-heart-disease-risk-among-irish-adults-.html#_ftn3.

[2] Amit Banerjee and Rajesh N Dave. Validating clusters using the hopkins statistic. In *2004 IEEE International conference on fuzzy systems (IEEE Cat. No. 04CH37542)*, volume 1, 149–153. IEEE, 2004.