Diving into Supervised Learning Classification Problems: A Comparative Analysis of K-Nearest Neighbor Algorithm, Decision Tree Algorithm, and Naïve Bayes Algorithm

Suyash Mehra
Deen Dayal Upadhyaya College,
University of Delhi

August 15, 2023

0.1 Introduction

This study contains a comparative analysis of performance of K-Nearest Neighbor Algorithm, Decision Tree Algorithm, and Naïve Bayes Algorithm on the same dataset. The dataset is also treated with all the required preprocessing tools. The main objective here is to understand the performance of each algorithm by comparing the values of the evaluation metrics with the default values and then focus on tuning the parameters in order to try to achieve higher accuracy, and better results on all other metrics considered.

0.2 Methodology

The first and foremost step in predicting values from a dataset requires its preprocessing, which includes its cleaning, transforming, and integrating of data in order to make it ready for analysis. The dataset chosen here is treated with all the necessary preprocessing tools. Therefore, before going into studying the algorithms, we will take a look at the preprocessing methods. We will then shift our focus on studying the algorithms' performance on the cleaned dataset.

A. Data Preprocessing

1. Acquiring Data

We first require a dataset on which the Machine Learning model can work on. The data chosen here is 'Cardiovascular Diseases Risk Prediction Dataset'

(Source: https://www.cdc.gov/brfss/annual_data/annual_2021.html).

NOTE: There exists a Class Imbalance issue with the dataset (Heart Disease: No-283883, Yes-24971).

The dataset present here consists the data of patients, and whether they have a heart disease or not based on certain factors. We will perform Binary Classification on the same, using the three above mentioned algorithms.

The data is present in a CSV file, which stands for Comma-Separated Values. Sometimes, we may require to use data present in an HTML or xlsx file.

2. Importing Libraries

We need to import few predefined libraries to use the functionalities they provide. We will be using three major libraries for our use.

1. Numpy: The Numpy library in Python serves the purpose of incorporating diverse mathematical operations into the code. It stands as a foundational module for performing scientific computations within the Python environment. Moreover, it facilitates the manipulation of extensive, multi-dimensional arrays and matrices.

2. Pandas: Moving on to the subsequent library, we encounter Pandas, a widely recognized Python library extensively utilized for importing and overseeing datasets. This library, an open-source resource, holds the capacity for manipulating and analyzing data.

3. Matplotlib: The subsequent module is matplotlib, a Python library for generating 2D plots. Within this library, it's necessary to import a sub-component named pyplot. This toolkit is employed to create diverse charts within the Python code.

The libraries could be imported as follows:

```python
#Importing necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

3. Importing the Dataset

The next step is to load the database that we have acquired. We will use the pd.read_csv() function of Pandas to load our CSV file, and then load it into a variable named 'dataset'.

```python
#Loading the dataset
dataset = pd.read_csv('Cardiovascular_Disease.csv')
```

Extracting dependent and independent variables:

The independent variables are the variables which contain the data with the help of which we will be predicting the values of the dependent variables. In our data, the dependent variable is Heart_Disease, the value of which would be either yes or no, and the value of this variable would be determined on the basis of the other factors in independent variables.

```python
1 #Defining independent variables in x and dependent variable (target)
     in y
2 x = dataset.iloc[:,:18].values
3 y = dataset.iloc[:,-1].values
```

Line 2 of the code above means that x contains all rows, and 0-17 columns of the dataset. Line 3 of the code above means that y contains all rows, and the last column, number 18.

4. Handling Missing Data

There are various strategies to deal with missing data present in the datasets. Some of these strategies are:

1. Impute the missing values with the mean/median/mode: The missing values can be filled with mean, median, or mode. This strategy helps in preventing the loss of data. For categorical (textual) data, the most frequent value could be used to fill the missing entries.

2. Deleting rows with missing data: The rows and columns with missing data could be removed from the dataset. Generally, the columns with most values as null are dropped in the beginning. The rows containing multiple entries as null could be removed from the dataset too.

3. Using Algorithms which are trained to deal with missing data: Some algorithms, like KNN Algorithm and Naïve Bayes Algorithm can be used when the dataset contains null or missing values.

Let's run the following code and check the output to determine if our dataset contains any null or empty values:

```
1 #Checking for missing values
2 missing_values = dataset.isnull().sum()
3 print('Missing values count:', missing_values)
```

In the above code, *dataset.isnull().sum*() in line 2 calculates the sum of all the missing or null values for each column and returns the count. Let's have a look at the output of the code above on our dataset.

```
#Output:
Missing values count: General_Health                  0
Checkup                         0
Exercise                        0
Skin_Cancer                     0
Other_Cancer                    0
Depression                      0
Diabetes                        0
Arthritis                       0
Sex                             0
Age_Category                    0
Height_(cm)                     0
Weight_(kg)                     0
BMI                             0
Smoking_History                 0
Alcohol_Consumption             0
Fruit_Consumption               0
Green_Vegetables_Consumption    0
FriedPotato_Consumption         0
Heart_Disease                   0
dtype: int64
```

Looks like our dataset has no null or missing values, so let's move on to the next step.

5. Encoding Categorical Data

Categorical data, also referred to as qualitative data, represents distinct groups or labels devoid of numerical value or quantitative interpretation. Categorical data aids in classifying items into specific classes or categories based on shared attributes. It characterizes qualities that fall into categories without any inherent order or meaningful numerical value.

Categorical data is categorized into two primary types:

1. Nominal Data: Nominal data portrays categories without any inherent ranking or order among them. Examples include colors, animal types, gender classifications, or various kinds of fruits. Nominal data merely indicates the distinction or similarity between two categories.

2. Ordinal Data: Ordinal data signifies categories with a specific order or ranking, though the differences between categories might not be uniformly measurable. Examples include educational levels (e.g., high school, bachelor's, master's), customer satisfaction ratings (e.g., poor, fair, good, excellent), or socioeconomic status groups (e.g., low, medium, high).

In our dataset, the following columns contain the Nominal Data:

```
['Exercise','Skin_Cancer','Other_Cancer','Depression','Diabetes','
    Arthritis','Sex','Smoking_History', 'Checkup', 'Age_Category']
```

and the

```
['General_Health']
```

column contains Ordinal Data.

Encoding categorical data involves representing qualitative attributes or labels in a numerical format that can be processed by algorithms. There are various methods for encoding categorical data:

1. Label Encoding: Assigning a unique numerical value to each category. However, this may unintentionally introduce ordinal relationships that don't exist.

2. One-Hot Encoding: Creating binary columns for each category, where a "1" indicates the presence of that category and "0" otherwise. This method is suitable for nominal data.

3. Ordinal Encoding: Assigning numerical values to categories in a way that preserves their ordinal relationships.

We will use **Ordinal Encoding** for

```
General_Health
```

column with the help of the following code:

```
#Applying Ordinal Encoding on the 1st column

from sklearn.preprocessing import OrdinalEncoder
oe = OrdinalEncoder()

encoded_column = oe.fit_transform(dataset[['General_Health']])
dataset['General_Health'] = encoded_column

#viewing first 5 values of the Ordinal Encoded values
dataset[['General_Health']].head(5)
```

We will use **Label Encoding** on the following columns:

```
['Exercise','Skin_Cancer','Other_Cancer','Depression','Diabetes','
    Arthritis','Sex','Smoking_History']
```

with the help of the following code:

```
#Applying Label Encoding on all the required columns
from sklearn.preprocessing import LabelEncoder

#Initializing and fitting LabelEncoder for required columns

```

```
6  columns_to_LabelEncode = ['Exercise','Skin_Cancer','Other_Cancer','Depression','Diabetes','Arthritis','Sex','Smoking_History']
7
8  encoded_columns = {}
9
10 for column in columns_to_LabelEncode:
11     encoded_columns[column] = LabelEncoder().fit_transform(dataset[column])
12
13 #Updating the dataset with encoded values
14 for column in encoded_columns:
15     dataset[column] = encoded_columns[column]
16
17 #Seperately handling the target (dependent) variable
18 target_column = 'Heart_Disease'
19 dataset[target_column] = LabelEncoder().fit_transform(dataset[target_column])
20
21 #Viewing the LabelEncoded data
22 dataset[['Exercise','Skin_Cancer','Other_Cancer','Depression','Diabetes','Arthritis','Sex','Smoking_History','Heart_Disease']].head(5)
```

In line 6 above, 'columns_to_LabelEncode' contains all the columns on which Label Encoding is to be performed. In line 10, a loop is run which applies encoding on all the columns mentioned in the array. The 'encoded_columns' dictionary then contains all the column names as keys, and an array along with them which contain the encoded values.

Lastly, we need to apply **One Hot Encoding** on the following columns:

```
['Checkup', 'Age_Category']
```

with the help of the following code:

```
#Applying One Hot Encoding on all the required columns
from sklearn.preprocessing import OneHotEncoder

columns_to_OneHotEncode = ['Checkup', 'Age_Category']

#Initializign the OneHotEncoder
ohencoder = OneHotEncoder(drop='first', sparse=False)

#Fitting and transforming the selected columns
encoded_columns = ohencoder.fit_transform(dataset[columns_to_OneHotEncode])

#Creating a new dataset with the encoded column
encoded_dataset = pd.DataFrame(encoded_columns, columns=ohencoder.get_feature_names_out(columns_to_OneHotEncode))

#Displaying the encoded data
```

```
#print(encoded_dataset)

#Concatenating the original dataset with the encoded dataset
dataset = pd.concat([dataset, encoded_dataset], axis=1)
```

Now, we need to redefine the dependent and independent variables due to the following reasons:

1. The columns made by One Hot Encoder are required to be included in the set of independent variables as they are required to be considered while predicting values through the model.

2. The independent variables contain the categorical data columns right now, so we remove them as we don't need to consider them. We are also not considering 'Height_(cm)' and 'Weight_(kg)' columns, which are used to calculate BMI, so we directly take into consideration the BMI of a patient.

```
1 #Redefining independent and dependent variables by including the
     encoded data in independent variable
2 df1 = dataset.iloc[:,0]      # Considering 'General Health' column
3 df2 = dataset.iloc[:,2:9]    # Excluding 'Checkup' and 'Age Category'
      columns as they have categorical data
4 df3 = dataset.iloc[:,12:18] # Excluding 'Height_(cm)', 'Weight_(kg)'
      columns as we have BMI column
5 df4 = dataset.iloc[:,19:-1] # Including all One Hot Encoded columns
     after the independent variable
6
7 x = pd.concat([df1, df2, df3, df4], axis=1)
8 y = dataset.iloc[:,18]
9 data = pd.concat([x, y], axis=1) # 'data' now consists of the
     cleaned and final data.
```

In line 7 above, we have performed concatenation to bring in together all the considered columns, without dropping the columns we do not take into consideration. We have created different data frames to avoid the columns in continuation.

6. Feature Selection

Feature Selection involves selecting the most relevant and informative features that would affect the target variable more than the other features. The goal is to enhance the model's performance, reduce overfitting, and improve interpretability.

The process of feature selection begins with a comprehensive understanding of the dataset and its underlying characteristics. Various techniques exist to perform feature selection. The choice of method depends on the dataset's structure, the problem at hand, and the algorithm being used.

The type of method to choose for Feature Selection depends on the independent and dependent variables being either Numerical or Categorical. While there exist many feature selection methods, here **Mutual Information** is chosen, as it measures the dependency between two variables, regardless of whether they are categorical or numerical, which makes it near to perfect to use it for our dataset, which contains both types of variables. As we have encoded all the categorical variables, we can also use **SelectKBest with f_regression**.

Mutual Information is applied with the help of the following code:

```
#Performing Feature Selection with Mutual Information
from sklearn.feature_selection import SelectKBest, mutual_info_classif

#Selecting the top k features based on mutual information
selector = SelectKBest(score_func=mutual_info_classif, k=7)
x = selector.fit_transform(x, y) #Replacing the dependent variables with the top k features

#Getting the indices of selected features and their corresponding scores
selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_
print(data.columns[selected_indices])
```

The above code outputs the following result, which are the selected features.

```
Index(['General_Health', 'Exercise', 'Diabetes', 'Arthritis', 'Sex',
       'Smoking_History', 'Checkup_Within the past year'],
      dtype='object')
```

7. Splitting the dataset into Training and Test Sets

To enhance the learning of our Machine Learning model, it is crucial that we divide the dataset available into Training and Test data. Our model will not determine good results if we train it on one dataset and test it on a completely different dataset. We need to train and test the model through the same dataset, so that our model is able to draw correlations among the values, and learn better.

We will divide our dataset with the help of the following code:

```
1 #Splitting data into Training Data and Testing Data
2 from sklearn.model_selection import train_test_split
3 x_train, x_test, y_train, y_test = train_test_split(x,y, test_size
    =0.2, random_state=1)
```

In line 3 of the above code, the train_test_split() function is used to split the data, and takes the following parameters:

1. First 2 parameters, x and y, are the arrays containing the data.

2. test_size refers to the size of the test data, and its value could be 0.25, 0.3, or 0.5.

3. random_state is used to set a seed for random generator so that the result is always the same.

The four variables created for containing output contain the following:

1. x_train: has the training data features of dependent variable

2. x_test: has the testing data features of dependent variable. It is used to test the model.

3. y_train: has the training data features of independent variable

4. y_test: has the testing data features of independent variable. It is used to compare with the predicted values of the model to determine accuracy.

8. Feature Scaling: Standardisation and Normalisation

Feature scaling plays a pivotal role in preparing data for machine learning by transforming numerical attributes into a uniform range. This is crucial as it ensures that all features contribute evenly to the learning process, especially for algorithms sensitive to feature scales.

Two prevalent techniques for feature scaling are standardization and normalization.

1. Standardization (Z-score normalization): Standardization centers features to have a mean of 0 and a standard deviation of 1. This method is beneficial when features exhibit varying scales or when specific algorithms assume data follows a normal distribution. The resulting data has no fixed range and can encompass negative values.

The standardization formula:

$$z = (x - mean)/std\_dev$$

In this equation, x represents the original value, mean signifies the mean of the feature, and std_dev is the standard deviation.

2. Normalization (Min-Max scaling): Normalization scales features within a range of 0 to 1. It is useful when features exhibit distinct ranges and need to be standardized to a shared scale. The normalized data retains its original distribution while ensuring that all values lie within the specified range.

The normalization formula:

$$x\_normalized = (x - min)/(max - min)$$

Here, x is the original value, min stands for the minimum value of the feature, and max denotes the maximum value.

We will be applying Standardisation on our data with the help of the following code:

```
1 #Standardizing the data
2 from sklearn.preprocessing import StandardScaler
3 sc = StandardScaler()
4 x_train=sc.fit_transform(x_train)
5 x_test=sc.transform(x_test)
```

The process guarantees that feature means become 0 and standard deviations become 1, a preparation that proves advantageous for numerous machine learning algorithms.

The line 4 of the code initiates the standardization transformation on the training data (x_train). Through the fit_transform method, mean and standard deviation statistics are calculated for each feature within the training data. Subsequently, these statistics are used to rescale the features, resulting in standardized values within the x_train dataset.

Similarly in the line 5 of the code, the standardization transformation is applied to the test data (x_test). Notably, the transform method is employed here, rather than fit_-transform. This decision ensures that the same scaling calculated from the training data is applied consistently to the test data. This consistency in scaling maintains the integrity of the standardization process across both datasets.

All the steps above complete the Data Preprocessing. Now we will choose different models and evaluate their performance.

B. K-Nearest Neighbor (KNN) Algorithm

The KNN algorithm operates based on the principle that data points with similar attributes tend to belong to the same class or exhibit similar behavior. Given a new data point, KNN determines its class by analyzing the classes of its K nearest neighbors in the training dataset. The value of K is a hyperparameter that needs to be predefined before implementing the algorithm.

The process of classifying a new data point using KNN involves the following steps:

1. Calculating Distances: KNN measures the distance between the new data point and all data points in the training dataset. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance.

2. Selecting Neighbors: The algorithm selects the K nearest data points based on the calculated distances. These data points are the "neighbors" of the new point.

3. Majority or Averaging: For classification tasks, the class that appears the most among the K neighbors is assigned to the new data point. For regression tasks, the output value is determined by averaging the output values of the K neighbors.

Choosing the value of K is quite essential here, due to the following reasons:

1. If the value of K is too small, then it would be sensitive to the noisy points.

2. If the value of K is big, then it would include many values from other classes

The implementation for fitting the KNN Classifier with the training data is as follows:

```python
#Fitting the KNN Classifier with the training data
from sklearn.neighbors import KNeighborsClassifier
classifier= KNeighborsClassifier(n_neighbors=5, metric='minkowski',
   p=2 )
classifier.fit(x_train, y_train)
```

Understanding the parameters of KNeighborsClassifier:

1. n_neighbors: The number of neighbours to be considered by the model.

2. metric: The metric decides type of distance with other data points which should be calculated by the model in order to find the 'nearest' neighbors. Here, the metric is 'minkowski', which along with the value of p as 2, becomes Euclidean Distance.

3. p: When p = 1, this is equivalent to using Manhattan Distance, and Euclidean Distance for p = 2. For arbitrary p, Minkowski Distance is used.

To predict the results calculated by our model on the above mentioned values, we use the following code:

```python
#Predicting the results of the test set
y_pred = classifier.predict(x_test)
```

To evaluate the performance of the classifier we will be using four measures, Accuracy, Precision, Recall Score, and F1 Score.

$$Accuracy = (Number of Correct Predictions)/(Total Number of Predictions)$$

$$Precision = (TruePositives)/(TruePositives + FalsePositives)$$

$$Recall = (TruePositives)/(TruePositives + FalseNegatives)$$

$$F1\_score = 2 * (Precision * Recall)/(Precision + Recall)$$

Above measures can be calculated and implemented as follows:

```
#Calculating Accuracy Percent
from sklearn.metrics import accuracy_score
a = accuracy_score(y_test, y_pred)
print('Accuracy percent: '+str(a*100)+'%')
```

```
#Calculating Precision Percent
from sklearn.metrics import precision_score
p = precision_score(y_test, y_pred)
print(p)
```

```
#Calculating Recall Score
from sklearn.metrics import recall_score
recall_score(y_test, y_pred)
```

```
#Calculating F1 Score
from sklearn.metrics import f1_score
f1 = f1_score(y_test, y_pred)
print(f1)
```

Ideally,

1. The accuracy score should be as high as possible, but it is not the best measure.

2. A higher precision score indicates that the positive predictions made by your model are more accurate.

3. A higher recall score indicates that the model is effectively capturing a larger proportion of positive instances.

4. A high F1-score indicates that the classification model is achieving a balanced trade-off between precision and recall.

For our dataset, we need to take in consideration that the dataset has Class Imbalance issue. Also, we are predicting if a person has Heart Disease or not based on certain features. In our case, False Negative plays a more vital role than False positive. Therefore, in our case, False Negative should be lesser which means **Recall** should be **higher**. Precision too should be higher.

Analysing the performance of the model by tuning the parameters:

n_neighbors	**metric**	**p**	**AccuracyPercent**	**PrecisionScore**	**RecallScore**	**F1_Score**
5	Minkowski	2	90.59	0.27	0.103	0.149
10	Minkowski	2	91.86	0.41	0.029	0.055
15	Minkowski	2	91.64	0.33	0.037	0.066
5	Minkowski	1	90.59	0.27	0.103	0.149
10	Minkowski	1	91.86	0.41	0.029	0.055
15	Minkowski	1	91.64	0.33	0.037	0.066

It is no surprise that we get low values for Recall and Precision as they largely consider True Positives and in our dataset, due to Class Imbalance, we have quite low values resulting in 'yes' anyway.

Taking into consideration the values achieved, we can draw the following insights:

- The high accuracy score (91.86%) indicates that the model is performing well in terms of overall correctness.
- The low precision suggests out of all the values predicted by the model as yes (already low in number), what is the ability of the model to determine the actual and correct 'yes' members.
- The low recall indicates that out of all the actual 'yes' class members (again, already low in number), what is the ability of the model to correctly determine the 'yes' members.
- The low F1-score further confirms that the model's precision and recall are not well balanced.

C. Decision Tree Algorithm

This algorithm is used for classification as well as regression problems. This classifier follows a tree-like structure, where internal nodes symbolize dataset features, branches indicate decision criteria, and each terminal node signifies the result.

It serves as a visual depiction for deriving all potential solutions to a choice, provided circumstances. A decision tree initiates by asking a query, and depending on the response (yes/no), it proceeds to divide the tree into smaller subbranches.

We will select the best feature for the root node, which causes the best split among the records. The best feature can be chosen with the help of Information Gain, the values of which should be high. The first split should be done on the basis of feature providing max Information Gain.

$$InformationGain = Entropy(S) - [(WeightedAvg) * Entropy(eachfeature)]$$

For this we will fit the classifier with the training data as follows:

```
from sklearn.tree import DecisionTreeClassifier
# Create a DecisionTreeClassifier
tree_classifier = DecisionTreeClassifier(criterion='entropy',
    random_state=0)

# Fit the model on the training data
tree_classifier.fit(x_train, y_train)
```

We will then predict the values of the testing data set as follows:

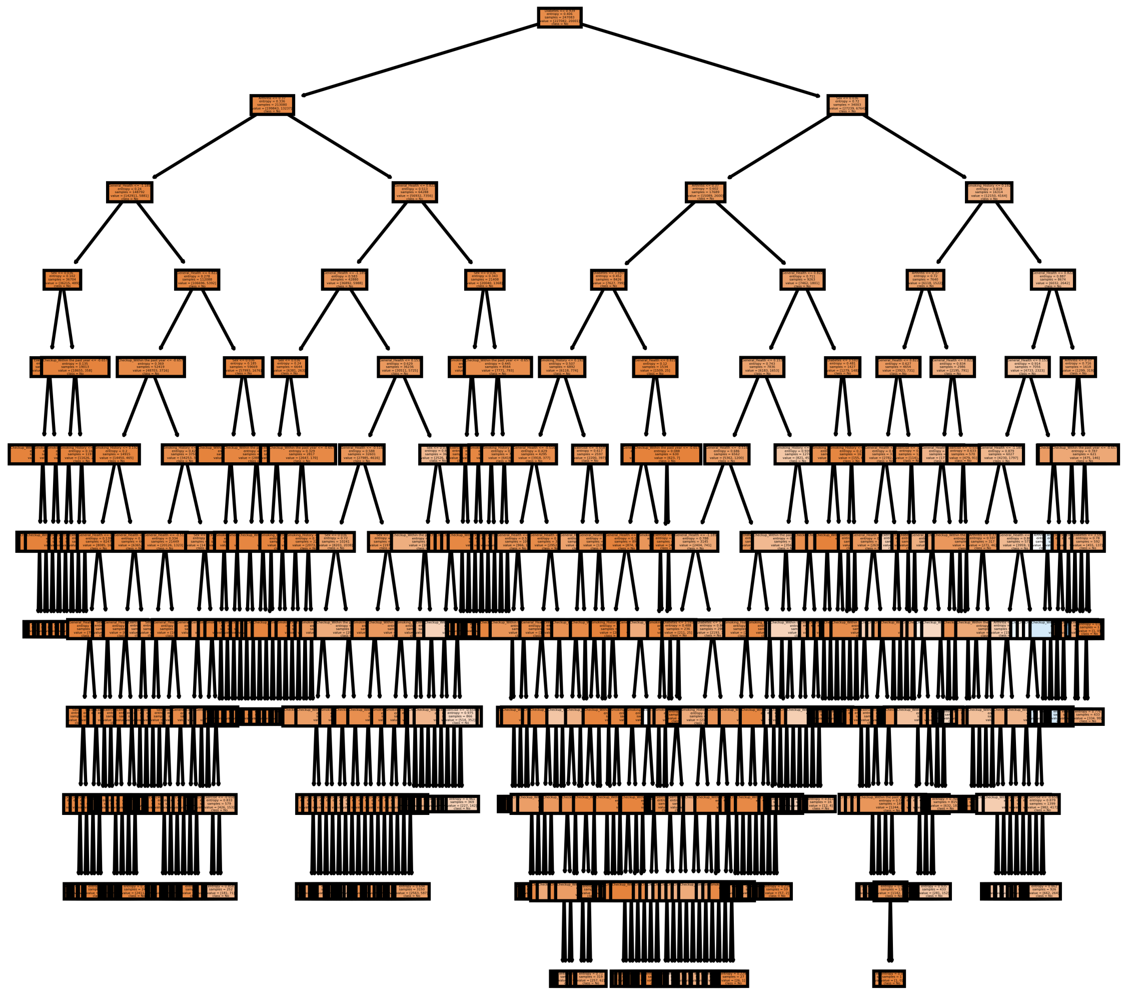
```
#Predicting the results of the test set
y_pred = tree_classifier.predict(x_test)
```

After calculating the accuracy percent, precision score, recall score, and F1 score, we get the following results:

criterion	**AccuracyPercent**	**PrecisionScore**	**RecallScore**	**F1_Score**
entropy	91.96	0.51	0.016	0.031

We visualise the Decision Tree using the following code:

```
#Visualizing the tree
from sklearn import tree
fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (6,6), dpi
    =800)
tree.plot_tree(tree_classifier, feature_names=data.columns[
    selected_indices], class_names=['No','Yes'], filled=True)
fig.savefig('DecisionTree_CVD_DTC.png')
```



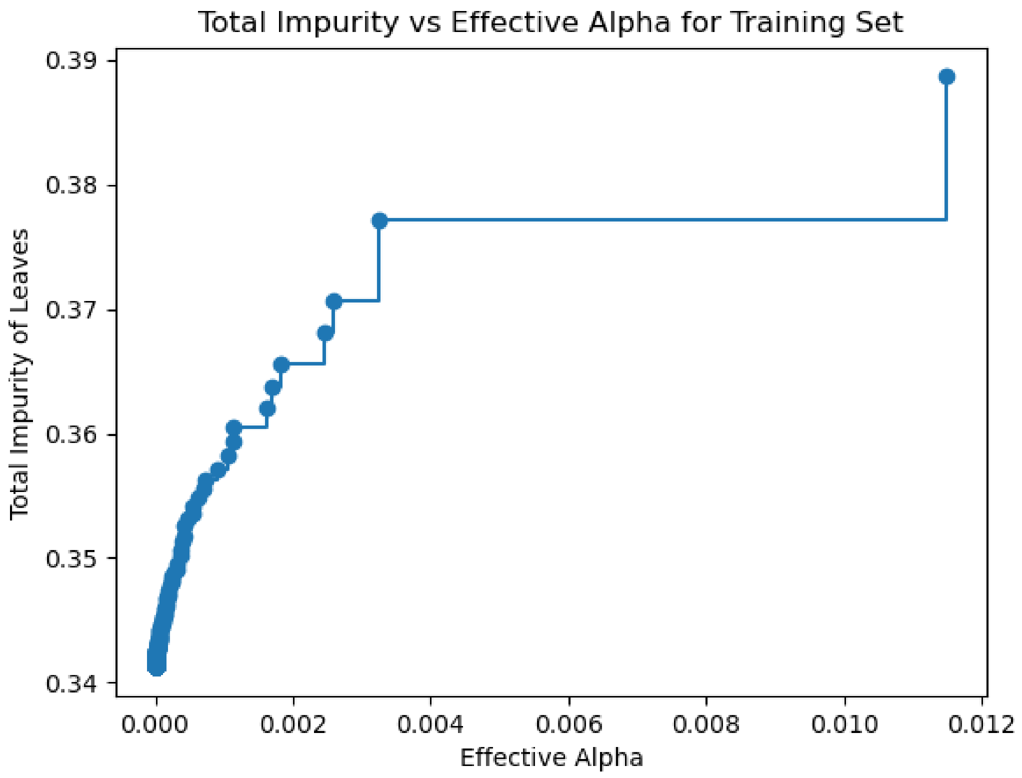
In order to try performing better than this, we will perform **Cost Complexity Pruning**, which works with the help of a Cost Complexity Parameter, alpha ($\alpha \geq 0$), the value of which is determined by sklearn's function *cost_complexity_pruning_path*(), which returns the effective alphas and the corresponding total leaf impurity when a particular alpha value is used to prune the tree.

```
#Obtaining the effective alphas and impurities on the training set
path = clf.cost_complexity_pruning_path(x_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

Now we plot a graph between the values of Alpha and Impurities, so that we can choose an alpha for which the impurity is of considerable and acceptable amount.

```
#Plotting Effective Alphas v/s Impurities
fig, ax = plt.subplots()
ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="
    steps-post")
ax.set_xlabel("Effective Alpha")
ax.set_ylabel("Total Impurity of Leaves")
ax.set_title("Total Impurity vs Effective Alpha for Training Set")
fig.savefig("ALvIMP.png")
```

The following output is received from the above code:



Next, we train a decision tree using the effective alphas. We will set these values of alpha and pass it to the *ccp_alpha* parameter of our *DecisionTreeClassifier*(). By looping over the alphas array, we will find the accuracy on both Train and Test parts of our dataset.

```
#Setting the values of ccp_alpha in the classifier and fitting with
    training data
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha
       )
    clf.fit(x_train, y_train)
    clfs.append(clf)
```

In lines 2 and 3 of the following code, we use the *score*() function which returns the mean accuracy on the given data and labels. After calculating the same, we will plot the graph between accuracy and alpha to determine the best value of alpha, providing an accuracy as maximum as possible.

```
1  #Calculating the mean accuracy on the given data
2  train_scores = [clf.score(x_train, y_train) for clf in clfs]
3  test_scores = [clf.score(x_test, y_test) for clf in clfs]
4
5  #Plotting Alpha v/s Accuracy
6  fig, ax = plt.subplots()
7  ax.set_xlabel("Alpha")
8  ax.set_ylabel("Accuracy")
9  ax.set_title("Accuracy vs Alpha for Training and Testing sets")
10 ax.plot(ccp_alphas, train_scores, marker="o", label="train",
       drawstyle="steps-post")
11 ax.plot(ccp_alphas, test_scores, marker="o", label="test", drawstyle
       ="steps-post")
12 plt.xlim(-0.00001, 0.0005)  #Setting x-axis limits for better
       visualisation into the graph
13 ax.legend()
14 plt.show()
15 fig.savefig('ACCvAL.png')
```

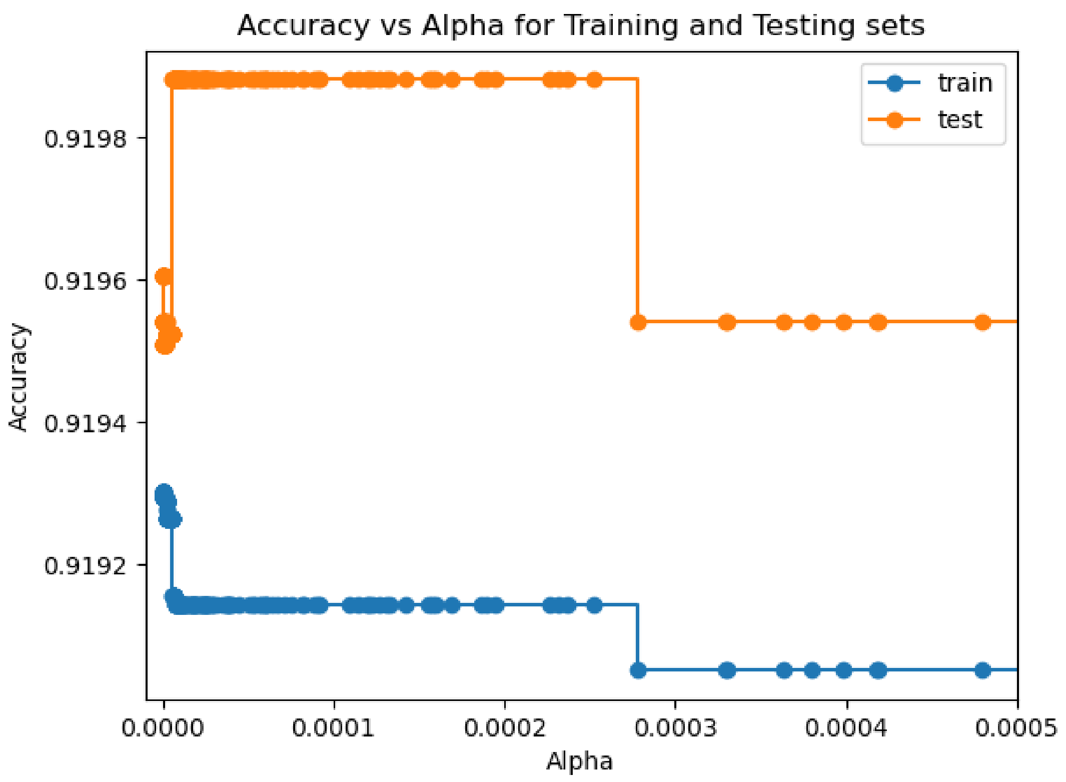
In line 11 of the above code, we have set limits on x-axis values just for better visualisation into the graph to obtain the value of the alpha easily. One can have a look at the graph by commenting this line out first.

After executing the above code, we get the following output. (See next page)

From the figure, we can see that we obtain highest accuracy when the value of alpha is either 0.0001 or 0.0002. We will now use this value in *ccp_alpha* parameter and see what we get as accuracy, and other metrics.

```
from sklearn.tree import DecisionTreeClassifier
# Create a DecisionTreeClassifier
tree_classifier = DecisionTreeClassifier(criterion='entropy',
    ccp_alpha=0.0002)

# Fit the model on the training data
tree_classifier.fit(x_train, y_train)
```

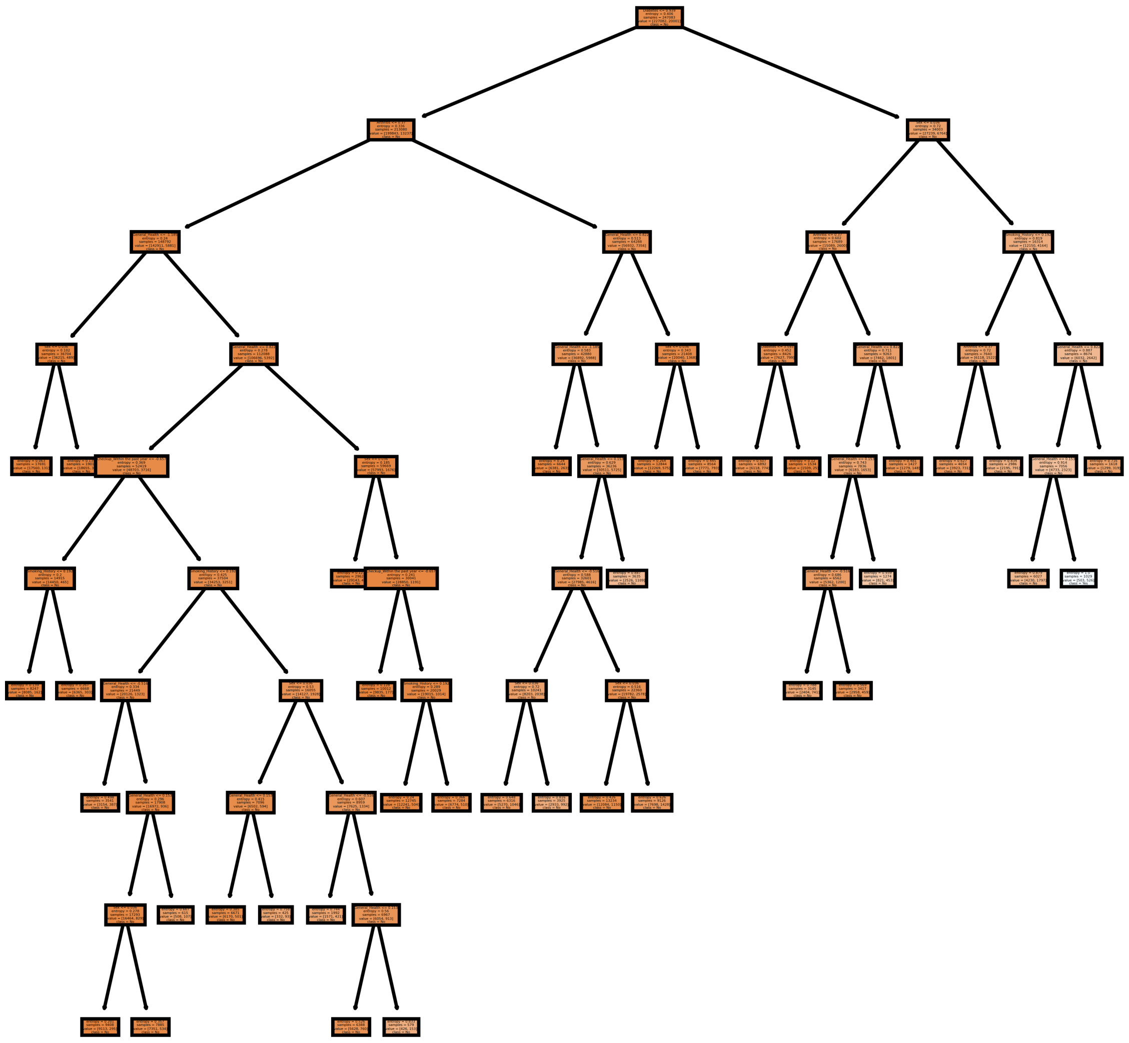


After pruning the tree, we get the following results:

criterion	AccuracyPercent	PrecisionScore	RecallScore	F1_Score
entropy	91.98	0.53	0.028	0.054

Due to the complex database, and class imbalance issues, we have still got low values for each metric, but we still observe that there has been an increase in the performance after performing pruning on the tree.

We visualise the tree post pruning: (See next page) After comparing the two trees, we can easily comprehend that the tree post pruning looks much less complex, as the weak links of the complex trees have been removed.



D. Naïve Bayes Algorithm

This algorithm works on Bayes' Theorem, in which the probability of a certain event happening is calculated based on prior knowledge and probabilities of related events. The algorithm determines an instance belongs to a particular class, given a set of features, with the help of the following:

$$P(Class|Features) = (P(Features|Class) * P(Class))/P(Features)$$

We will use a Gaussian Naïve Bayes Classifier:

```
from sklearn.naive_bayes import GaussianNB
#Creating a Gaussian Naive Bayes classifier
gnb = GaussianNB()

#Training the model using the training data
gnb.fit(x_train, y_train)

```

```
#Predicting the results of the test set
y_pred = gnb.predict(x_test)
```

After calculating the accuracy percent, precision score, recall score, and F1 score, we get the following results:

AccuracyPercent	**PrecisionScore**	**RecallScore**	**F1_Score**
87.99	0.23	0.21	0.22

There is not much we can do in terms of tuning a Gaussian Classifier. But in order to study the classifier further, we can see the probabilities considered by it for calculations.

First, we can find the probability of the class labels using the *class_prior_* and *classes_* methods :

```
#Accessing the estimated prior probabilities of each class
prior_probs = gnb.class_prior_
class_labels = gnb.classes_
print("Prior Class Label & Probabilities:", dict(zip(class_labels,
    prior_probs)))
```

Output:

```
Prior Class Label & Probabilities: {0: 0.9190514928182028, 1:
    0.0809485071817972}
```

This means the probability of 'No' is 0.91 and 'Yes' is 0.08, which is due to the class imbalance issue.

Next, we can see the probability of each instance belonging to a class with the help of *predict_proba* method:

```
#Calculating class probabilities for the test data
class_probs = gnb.predict_proba(x_test)

#Display the class probabilities for the first few instances
print("Class Probabilities:", class_probs[:5]) # Print probabilities
     for the first 5 instances
```

Output:

```
Class Probabilities: [[0.98777418 0.01222582]
 [0.90839878 0.09160122]
 [0.98641301 0.01358699]
 [0.84731306 0.15268694]
 [0.66861663 0.33138337]]
```

The above output shows the probabilities of an instance belonging to 'No' and 'Yes' respectively.

0.3 Experimental Results

The values obtained from the above three classifiers are:

Classifier	**AccuracyPercent**	**PrecisionScore**	**RecallScore**	**F1_Score**
KNNClassifier()	90.59	0.27	0.103	0.149
DecisionTreeClassifier()	91.98	0.53	0.028	0.054
GaussianNB()	87.99	0.23	0.219	0.227

By considering the above metrics to judge the performance of each classifier, we can comprehend the following:

- The highest overall accuracy percent is given by DecisionTreeClassifier(), but it is not the best measure to judge a classifier's performance.
- The highest precision score is given by DecisionTreeClassifier() too, but precision provides an analysis over True Positive and False positive, and here, we need to focus on False Negative.
- The highest recall score is given by GaussianNB() by quite a considerable difference than others, and high recall means that the model is making fewer incorrect false negative predictions, which is more important to us while predicting the possibility of a heart disease.
- The highest F1 score is given by GaussianNB() too, which indicates a good balance between precision and recall, suggesting that a classifier is performing well in terms of both minimizing false positives (precision) and false negatives (recall) simultaneously.

0.4 Conclusion

After analysing the experimental results, we can conclude that in this particular case and along with this dataset, the **Naïve Bayes Algorithm** provides the best results. We can further work to try achieving better results by changing the feature selection method used here, by addressing the class imbalance issue, by choosing a different pruning method other than cost complexity pruning, by studying other Naïve Bayes Classifiers and preparing the dataset according to their requirements.

Bibliography

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.