

Final Year Project Report

Full Unit - Final Report

Comparison of Machine Learning Algorithms

Janit Sudeesh

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dr Nicolo Colombo



Department of Computer Science
Royal Holloway, University of London

April 11, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 12,452

Student Name: Janit Sudeesh

Date of Submission:

Signature:

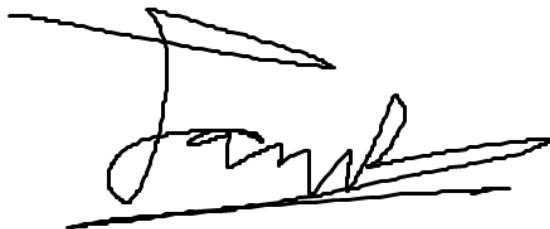
A handwritten signature in black ink, appearing to read 'Janit Sudeesh', written over a horizontal line.

Figure 1: Enter Caption

Table of Contents

Abstract	3
Project Specification	4
1 Introduction	5
1.1 Motivation	5
1.2 Aims and Objectives	6
2 Algorithms	7
2.1 K-Nearest Neighbours	7
2.2 Decision Trees	10
2.3 Support Vector Machine	16
2.4 Logistic Regression	22
2.5 Naive Bayes	26
3 Comparison and Results	31
3.1 Comparison and Analysis	31
3.2 Results	41

Abstract

This report presents a comparative analysis of classification machine learning algorithms., exploring their performance, strengths, and weaknesses across various domains. Classification tasks are integral to numerous applications, from healthcare to finance and beyond, making the selection of the most suitable algorithm a critical decision. Through empirical evaluation and theoretical examination, this study aims to provide insights into the effectiveness and applicability of different algorithms in diverse scenario. By synthesising findings from experiments and literature reviews, this report offers guidance to practitioners and researchers seeking to navigate the landscape of classification algorithms effectively.

Project Specification

Aims: To implement and compare on benchmark data sets various machine learning algorithms **Background:** Machine learning allows us to write computer programs to solve many complex problems: instead of solving a problem directly, the program can learn to solve a class of problems given a training set produced by a teacher. This project will involve implementing a range of machine learning algorithms, from the simplest to sophisticated, and studying their empirical performance using methods such as cross-validation and ROC analysis.

Chapter 1: Introduction

This project focuses primarily on classification, a fundamental task in machine learning which involves assigning input data points to predefined categories or classes. With the proliferation of data-drive applications across industries, the need for accurate and efficient classification algorithms has become paramount. However, the choice of algorithm greatly influences the performance and outcomes of classification tasks, necessitating a thorough understanding for their characteristics, strengths and limitations.

In this report, we embark on a comprehensive exploration of various classification machine learning algorithms focusing on their application in diagnosing heart problems. Our objective is to compare and contrast these algorithms, shedding light on their efficacy in different contexts and scenarios related to cardiac health. By analysing both empirical results and theoretical underpinnings, we aim to provide practitioners and researchers in the medical field with valuable insights to inform their algorithm selection process for heart diagnosis.

The report is structured as follows:

1. First provide complete explanations of all algorithms individually, including their mathematical background with relevant formulae, as well as describing in their implementation in code (for this context being in Python 3).
2. We will then delve into comparative analysis, evaluating the performance metrics, computational efficiency and robustness of each algorithm specifically.
3. Finally the report presents insights into the development process including tests and failures throughout the creation of this application.

Throughout this exploration we endeavour to elucidate the complexities of classification algorithms as they pertain to diagnosing heart problems, offering guidance to healthcare professionals and researchers seeking to leverage machine learning effectively in cardiac health assessment.

1.1 Motivation

Cardiovascular disease (CVD) is a type of heart disease that continues to be a major cause of death worldwide, accounting for over 30 percent of all deaths. If nothing is done, the total number of fatalities in the world is anticipated to rise to 22 million by 2030 [3].

Coronary Artery Disease (CAD), also known as ischemia heart disease (IHD) is the leading cause of death in adults over the age of 35 in different countries. During the same time span, it became China's biggest cause of death. When blood flow to the heart is reduced due to coronary artery stenosis, IHD occurs. Myocardial damage can have serious consequences including ventricular arrhythmia or even sudden cardiac death due to myocardial infarction.

Early identification of heart diseases on high risk individuals using a prediction model will provide improved diagnosis and can be recommended generally for fatality rate reduction as well as improved decision making for further treatment and prevention. Clinicians have already implemented prediction models and these are being utilised to support the clinicians in assessing the heart disease risk, and appropriate treatments are provided for managing the further risk. Additionally, numerous studies have also reported that implementation of

prediction models can improve decision quality, clinical decision making and preventive care respectively. [12]

Machine learning may be used to diagnose, detect and forecast many disorders in the medical industry. The primary purpose of this application is to give clinicians a tool to detect cardiac problems at an early stage as well as derive which algorithm is the most efficient and accurate. As a result, it will be easier to deliver appropriate treatment to patients while avoiding serious effects.

1.2 Aims and Objectives

The primary aim of this project is to implement and compare benchmark datasets various machine learning algorithms. The program will involve implementing a range of machine learning algorithms, from the simplest to sophisticated and studying their empirical performance using methods such as cross-validation and Receiver Operator Characteristic (ROC) curves analysis. The objectives of this project are as follows:

1. Compare and evaluate the performance of a range of classification algorithms including K-Nearest Neighbours, Decision Trees, Support Vector Machines, Logistic Regression and Naive Bayes in diagnosing heart problems.
2. Pre-process and prepare heart health related datasets, ensuring data quality, feature engineering, and appropriate handling of missing values and outliers.
3. Train each classification algorithm on the prepared datasets, optimising hyper-parameters through techniques such as cross-validation to achieve optimal performance.
4. Assess the performance of each algorithm using a variety of metrics such as accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC).
5. Generate detailed classification reports for each algorithm, providing insights into model performance.
6. Plot ROC Curves for each classification algorithm, visually comparing their performance in terms of true positive rate (sensitivity) versus false positive rate (1 - specificity)
7. Conduct a comparative analysis of the classification reports and ROC curves to determine the strengths, weaknesses, and suitability of each algorithm.
8. Provide insights and recommendations for healthcare professionals regarding the selection of appropriate classification algorithms for heart health diagnosis based on the analysis results.

Chapter 2: Algorithms

2.1 K-Nearest Neighbours

2.1.1 Introduction

K Nearest Neighbours (KNN) is a machine learning algorithm used for classification and regression tasks. It operates on the principle of similarity, where it classifies or predicts based on the majority class or average value of its nearest neighbours in the feature space. In classification, KNN assigns the class label of an input data point by considering the class labels of its K nearest neighbours. The algorithm does not require explicit training; instead, it memorises the entire training dataset. KNN is simple to understand and implement, making it a popular choice for various applications, although its performance may be affected by the choice of distance metric and the value of K.

At its core, KNN makes predictions based on the majority class (for classification) or the average value (for regression) of the K nearest neighbours to a given data point in the feature space. The "nearest" neighbours are determined based on a distance metric, commonly the Euclidean distance, although other metrics such as Manhattan distance or cosine similarity can also be used.

KNN does not involve a training phase in the traditional sense. Instead it stores the entire training dataset and makes predictions based on the proximity of new data points to the existing ones. This characteristic makes KNN a "lazy learner" or instance-based algorithm.

KNN is intuitive, easy to understand and applicable to a wide range of datasets. However, its main drawbacks include computational inefficiency, especially with large datasets, and sensitivity to the choice of distance metric and K value. [5]

2.1.2 Procedure

The K-Nearest Neighbours algorithm can be explained on the basis of this algorithm:

1. Select the K number of neighbours.
2. Calculate the Euclidean distance of K Number of Neighbours.
3. Take the K nearest neighbours as per the calculated Euclidean distance.
4. Among these K Neighbours, count the number of data points in each category.
5. Assign the new data points to that category for which the number of the neighbour is maximum.

The Euclidean distance previously mentioned can be defined as:

$$\text{Distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

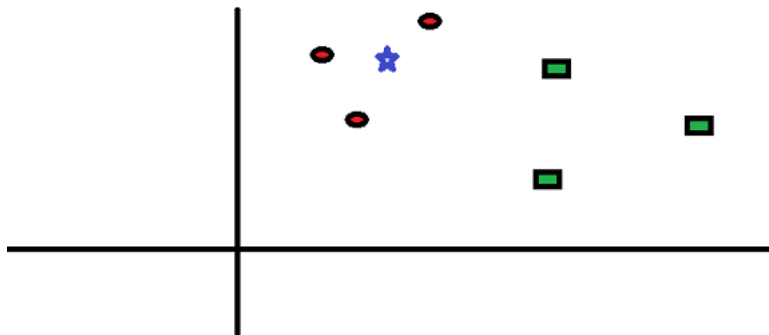
The Euclidean calculates the straight-line distance between two points in a Euclidean space. Given two points P and Q (with their respective x and y coordinates), the distance is

computed by calculating the length of the straight line connecting the two points in a n-dimensional space.

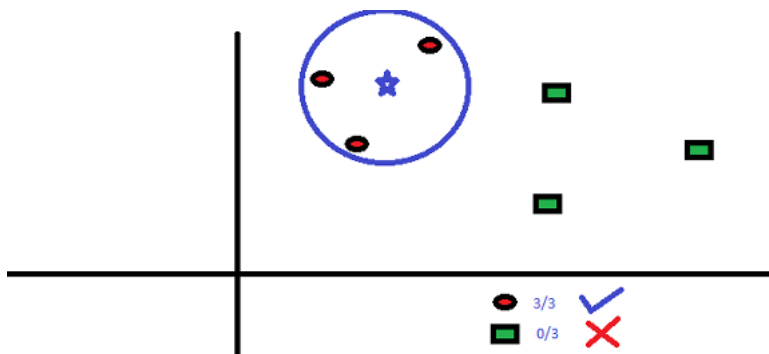
KNN as an algorithm can be used for both classification and regression tasks. Given the data-set with labeled points, to predict the label of a new, unseen data point, KNN looks at the 'k' closest labeled data points and assigns the label that is most common (in classification) or averages the labels (in regression) among its neighbors. The first step before executing the algorithm is preprocessing. Data preparation ensures the data-set is ready because KNN doesn't work well with categorical features unless they are appropriately encoded. Normalisation is also required to scale numerical features so that all features contribute equally to the distance calculation. Then comes the step of choosing 'k'. A value that will amount to the number of nearest neighbours to consider. The optimal 'k' value may vary depending on the data-set. Choosing 'k' too small may lead to over-fitting while choosing 'k' too large may lead to over-fitting.

For distance calculation we commonly use the Euclidean distance (previously mentioned) to measure the distance between data points. New data points are measured against all other data points in the training set. Sorting the distances obtained and selecting the 'k' data points with shortest distances will give us the nearest neighbours.

After obtaining the nearest neighbours the algorithm will decide whether to cast a majority vote or average based on whether the data-set works on classification or regression. If the problem requires classification, for each class among the 'k' neighbors, count the occurrences of each class label and assign the most frequent class label as the prediction for the new data point. If the problem requires regression, the algorithm will average the target value of the 'k' nearest neighbours and assign this average value as the prediction for the new data point. [1]



In the above example we have 3 classes, Red Circle (RC), Green Square (GS) and our unlabeled data point Blue Star (BS). We aim to classify the unlabeled data point (BS) as either one of the other 2 classes. Let's choose k as 3. Doing this will choose the 3 nearest neighbours of the BS point. We choose the points with the lowest Euclidean Distance.



The three closest points to BS are all RC. Therefore BS belongs to the class RC.

2.1.3 Development

The library of sci-kit has a package from sklearn called 'KNeighborsClassifier'. This implementation of K Nearest Neighbors has 2 phases. During the training phase, the algorithm stores the training data-set in memory without building an explicit model. It simply memorises the training examples and their associated class labels. When making predictions on new data points, the algorithm identifies the k-nearest neighbors to input the data point based on Euclidean distance. It calculates the majority class among these neighbours and assigns the class label to the new data point.

The custom implementation I have developed takes a lot of assets from this version made in sci-kit learn. Some obvious instances make an appearance such as, the 'n neighbors' variable which determines the number of neighbours to consider when making a prediction. For mathematical calculations, the 'NumPy' library allowed for the usage of quick form operations such as sorting arrays and square rooting. Other aspects such as fitting and testing the model remains consistent across the board as it remains more convenient to use these built in functions rather than creating my own that may produce inconsistent results across different algorithms.

The very first implementation of K Nearest Neighbours that was developed lacked the ability to read through a file and first calculated through built in data points as an example. Testing for this was done through changing the arguments that the values were tested for. Admittedly, a more effective Test Driven Development strategy could've been leveraged however the scale of the project at the time was minute and simple so the decision was made not to create full fledged test cases.

Eventually during my deployment of the first version of the Decision Trees algorithm (which reads from a data-set in a file), I transferred the file reading and encoding techniques and applied it to K Nearest Neighbours program. One issue that was prominent was that some data in the set could not be read as they did not fit the classification that the algorithm was calculating for. This was later fixed with the use of label encoding.

```
def __init__(self, k=3):
    self.k = k
```

The init method initialises the classifier with a parameter 'k', representing the number of neighbours to consider during classification. By default, 'k' is set to 3.

```
def fit(self, X, y):
    self.X_train = X
    self.y_train = y
```

The fit method is used to store the training data. It takes the input features 'X' and target labels 'y' as parameters and stores them in the 'X train' and 'y train' attributes, respectively.

```
def euclidean_distance(self, x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))
```

The Euclidean Distance method calculates the Euclidean distance between two data points 'x1' and 'x2'. It uses NumPy's vectorised operations to efficiently compute the distance.

```
def predict(self, X):
    y_pred = [self._predict(x) for x in X]
    return np.array(y_pred)
```

The predict method makes predictions using the KNN classifier for a set of input features 'X'. It iterates over each input data point and calls the other predict method to predict the class label.

```
def _predict(self, X):
    # Compute distances between the input data point and all training data
    distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]

    # Sort by distance and return indices of the first k neighbours
    k_indices = np.argsort(distances)[:self.k]

    # Extract the labels of the k nearest neighbour training samples
    k_nearest_labels = [self.y_train[i] for i in k_indices]

    # Return the most common class label among the k nearest neighbours

    most_common = Counter(k_nearest_labels).most_common(1)
    return most_common[0][0]
```

This predict method predicts the class label for a single data point 'x'. It computes the Euclidean distances between the input data point 'x' and all training data points. Then, it sorts the distances and selects the indices of the first 'k' neighbours. Next it extracts the labels of the 'k' nearest neighbour training samples. Finally, it returns the most common class label among the 'k' nearest neighbours as the predicted label for the input data point.

The 'Counter' class from the 'collections' module is used to count the occurrences of each class label among the 'k' nearest neighbours. This simplifies the process of determining the most common class label.

2.2 Decision Trees

2.2.1 Introduction

A decision tree is a non-parametric supervised learning algorithm for classification and regression tasks. They represent a tree-like structure where each internal node represents a feature or attribute, each branch represents a decision based on that feature, and each leaf node represents the outcome or prediction.

These models are particularly attractive due to their interpretability and simplicity, making them suitable for understanding and visualising decision-making processes. Decision Trees mimic human decision-making by breaking down a complex decision-making process into a series of simpler decisions.

In classification tasks, Decision Trees recursively partition the feature space into subsets, with each split aiming to maximise the homogeneity (or purity) of the resulting subsets with respect to the target variable. This process continues until a stopping criterion is met, such as reaching a maximum depth, achieving minimum samples per leaf, or when further splits do not improve the model's performance.

Decision Trees offer several advantages. Some include that it's easy to interpret and explain making them ideal for conveying insights to stakeholders, it can handle both numerical and categorical data without requiring extensive preprocessing. It is also able to capture non-linear relationships between features and the target variable and is robust to outliers and missing values.

However, Decision Trees are prone to overfitting, especially when the tree depth is not appropriately controlled. To address this, techniques such as pruning and limiting the tree depth are commonly employed. In summary, Decision Trees are versatile models that strike a balance between interpretability and predictive performance, making them valuable tools in various machine learning applications. [8]

2.2.2 Procedure

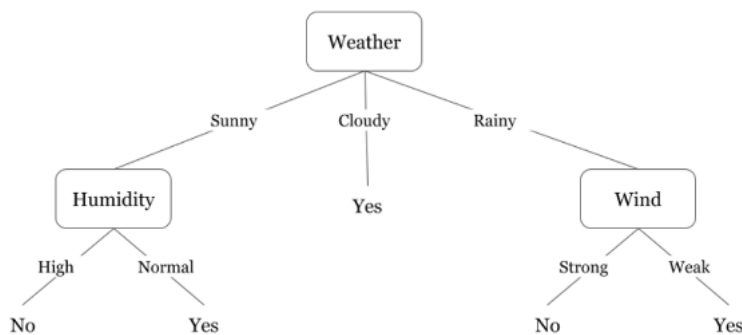
It is important to understand the various parts that construct a Decision Tree from the top down.

- The Root Node is the initial node at the beginning of a decision tree, where the entire population or dataset starts dividing based on various features or conditions.
- Decision Nodes are nodes that split from root nodes. These nodes represent intermediate decisions or conditions within the tree.
- Leaf Nodes are used when further splitting is not possible, often indicating the final classification or outcome. These can also be commonly referred to as terminal nodes.
- Sub trees are subsections of decision trees. It represents a specific portion.
- Pruning is the process of removing or cutting down specific nodes in a decision tree to prevent overfitting and simplify the model.
- Branches refer to a specific path of decisions and outcomes in a Decision Tree.
- Parent nodes refer to nodes that split into sub nodes. These sub nodes are named child nodes

These trees are constructed by recursively partitioning the input space into smaller regions, aiming to create subsets that are homogeneous as possible concerning the target variable. At each node of the tree, the algorithm selects the best feature to split the data, seeking to maximise the homogeneity or purity of the resulting subsets. The process continues until a stopping criterion is met, such as reaching a maximum tree depth, minimum number of samples in a node, or other pre-defined conditions.

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes
10	Rainy	Mild	High	Strong	No

In layman's terms, Decision Trees are nothing more than a bunch of if-else statements. The following table lists a set of conditions for whether it is suitable to go outside and play. These conditions can be mapped out onto a Decision Tree.



Several assumptions are made when building effective models for Decision Tree creation. These assumptions help guide the tree's construction and impact its performance. Some common assumptions include:

- Binary Splits are typically made in Decision Trees, meaning each node divides the data into two subsets based on a single feature or condition. This assumes that each decision can be represented as a binary choice.
- A Recursive Partitioning process is also utilised in Decision Trees where each node is divided into child nodes until a certain criterion is met. This assumption works only if the data can be effectively subdivided into smaller, more manageable subsets.
- Feature Independence is an attribute that we presume features that are used for splitting nodes are independent. While feature independence might not withhold in practice, Decision Trees can still perform well if features are correlated.
- Homogeneity refers to samples within a node that are as similar as possible regarding the target variable which creates homogeneous subgroups in each node. This assumption helps in achieving clear boundaries.
- A top down greedy approach is used when constructing Decision Trees, where each split is chosen to maximize information gain or minimize impurity at the current node. This may not always result in the globally optimal tree.

- Categorical and numerical features are both handled in Decision Trees. However, they may require different splitting strategies for each type.
- Overfitting can occur when they capture noise in the data. Pruning and setting appropriate stopping criteria are used to address this assumption.
- Impurity measures are used in Decision Trees such as Gini impurity or Entropy to evaluate how well a split separates classes. The choice of impurity measure can impact tree construction.
- Decision trees assume that there are no missing values in the dataset or that missing values have been appropriately handled through imputation or other methods.
- We assume equal importance for all features in Decision Trees unless feature scaling or weighting is applied to emphasize certain features.
- Decision trees are sensitive to outliers, and extreme values can influence their construction. Preprocessing or robust methods may be needed to handle outliers effectively.
- Small datasets may lead to overfitting, and large datasets may result in overly complex trees. The sample size and tree depth should be balanced.

Another key concept in Decision Trees is the concept of Entropy. Entropy is described as the uncertainty in a dataset or the measure of disorder. It can be defined as:

$H(X)$: Represents the entropy of a random variable X .

$P(x_i)$: Denotes the probability of the occurrence of the i th outcome or event x_i in a set of possible outcomes.

n : Represents the total number of possible outcomes or events of the random variable X .

\log_b : Denotes the logarithm to the base b .

$\sum_{i=1}^n$: Signifies the summation over all n possible outcomes or events of the random variable X .

$$H(X) = - \sum_{i=1}^n P(x_i) \cdot \log_b P(x_i)$$

Entropy serves as a metric to gauge the impurity of a node, where impurity signifies the level of randomness in the data. A perfectly pure sub-split would ensure a definitive outcome of either "yes" or "no". The goal of machine learning is to decrease uncertainty or impurity in the dataset. To counteract this, we bring in a new metric called "Information Gain" which tells us how much the entropy of a particular node has decreased after splitting it with some feature.

Information gain measures the reduction of uncertainty given some feature and it is also a deciding factor for which attribute should be selected as a decision node or root node. [7]

2.2.3 Development

My first implementation of Decision Trees used the sci-kit version named 'DecisionTreeClassifier'. Unlike my first iteration of K-Nearest Neighbours, this version started off with the ability to read from a file. Here I learned about the 'pandas' library which is actually quite ever-present in a lot of machine learning projects. I found one of its library functions to

be incredibly useful. There is a method to replace categorical data with 'dummy' variables through one hot encoding. In Pandas, one-hot encoding is a technique used to convert categorical variables into a numerical format suitable for machine learning algorithms. It creates binary dummy variables for each unique category in the categorical column. This solves the dilemma in datasets with mismatched feature sets. I followed the same standard procedure using the train test split function from the sci-kit library to develop the model. Eventually I started creating my own version of a Decision Tree class. The main basis forming this classifier was the creation of nodes and how the program will traverse this tree using these nodes. I created a basic implementation of a Decision Tree classifier however I opted to use Gini impurity over Entropy for the splitting criterion. The Gini impurity is calculated using the following formula:

$$\text{Gini Index} = 1 - \sum_j p_j^2$$

Where p_j is the probability of class j .

The Gini impurity measures the frequency at which any element of the dataset will be mis-labeled when it is randomly labeled.

The minimum value of the Gini Index is 0. This happens when the node is pure, meaning that all the elements contained in the node belong to one unique class. Therefore, this node will not be split again. Thus, the optimum split is chosen by the features with the lowest Gini Index. Moreover, it reaches the maximum value when the probabilities of the two classes are the same. The main difference between Gini index and Entropy is that Gini Index has values inside the interval $[0, 0.5]$ whereas the interval of the Entropy is $[0, 1]$. Computationally, Entropy is more complex since it makes use of logarithms and consequently, the calculation of the Gini index will be faster. This does not exclude the possibility of creating a second Decision Tree classifier using an entropy split instead of Gini. My Decision Tree classifier is grown recursively by splitting nodes based on the best features and threshold values to minimise Gini impurity. I created a split method that exhaustively searches for the best split among features to prevent overfitting. Additionally, I accomplished implementing a prediction method which uses the trained tree to predict the class labels for input data samples. It was necessary to include a 'max depth' parameter as well to control the tree depth and overfitting issue previously mentioned.

```
class Node:
    def __init__(self, predicted_class):
        self.predicted_class = predicted_class
        self.feature_index = 0
        self.threshold = 0
        self.left = None
        self.right = None
```

This class represents a Node in the decision tree. It holds information about the predicted class, feature index, threshold, and references to left and right child nodes. The 'predicted class' attribute holds the class that the node predicts. This value is determined during tree construction.

```
class CustomDecisionTreeClassifier:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
```

This class implements the decision tree classifier. The max depth parameter is used to control the maximum depth of the decision tree that will be grown during training.

```
def fit(self, X, y):
    self.n_classes_ = len(np.unique(y))
    self.n_features_ = X.shape[1]
    self.tree_ = self._grow_tree(X, y)
```

The fit method trains the decision tree classifier using input features 'X' and target labels 'y'. It computes the number of classes and features in the dataset and calls the grow tree method to build the decision tree.

```
def _grow_tree(self, X, y, depth=0):
    num_samples_per_class = [np.sum(y == i) for i in range(self.n_classes_)]
    predicted_class = np.argmax(num_samples_per_class)

    node = Node(predicted_class=predicted_class) # Creates node for predicted cla

    if self.max_depth is not None and depth < self.max_depth:
        feature_idx = np.random.choice(self.n_features_, self.n_features_, repla
        best_idx, best_thr = self._best_split(X, y, feature_idx) # Find the best
        if best_idx is not None:
            indices_left = X[:, best_idx] < best_thr
            X_left, y_left = X[indices_left], y[indices_left]
            X_right, y_right = X[~indices_left], y[~indices_left]
            node.feature_index = best_idx
            node.threshold = best_thr
            node.left = self._grow_tree(X_left, y_left, depth + 1) # Build left t
            node.right = self._grow_tree(X_right, y_right, depth + 1) # Build rig
    return node
```

The grow tree method recursively builds the decision tree. It starts with the root node and splits the data based on the best feature and threshold. The splitting process continues until the maximum depth is reached or further splits do not improve the impurity measure

```
def _best_split(self, X, y, feature_idx):
    best_idx, best_thr = None, None
    best_gini = 1.0

    for idx in feature_idx:
        thresholds = np.unique(X[:, idx])
        for thr in thresholds:
            indices_left = X[:, idx] < thr
            gini = self._gini_impurity(y[indices_left], y[~indices_left])
            if gini < best_gini:
                best_idx = idx
                best_thr = thr
                best_gini = gini
    return best_idx, best_thr
```

The best split method finds the best feature and threshold for splitting the data based on Gini impurity. It iterates over all features and thresholds to determine the split that minimises the impurity measure.


```

def _gini_impurity(self, left_labels, right_labels):
    p_left = len(left_labels) / (len(left_labels) + len(right_labels))
    p_right = len(right_labels) / (len(left_labels) + len(right_labels))
    gini_left = 1.0 - sum((np.sum(left_labels == c) / len(left_labels)) ** 2 for c in
    gini_right = 1.0 - sum((np.sum(right_labels == c) / len(right_labels)) ** 2 for c in
    gini = p_left * gini_left + p_right * gini_right
    return gini

```

The gini impurity method calculates the Gini impurity measure, which quantifies the impurity or disorder in a dataset. It computes the impurity for both left and right subsets and returns the weighted sum of impurities.

```

def predict(self, X):
    return [self._predict_tree(x, self.tree_) for x in X]

```

The predict method makes predictions using the trained decision tree. It iterates over each input feature vector and calls the predict tree method to traverse the decision tree and predict the class label.

```

def _predict_tree(self, x, node):
    if node.left is None and node.right is None:
        return node.predicted_class
    if x[node.feature_index] < node.threshold:
        return self._predict_tree(x, node.left)
    else:
        return self._predict_tree(x, node.right)

```

The predict tree method recursively traverses the decision tree to predict the class label for a given input feature vector. It follows the appropriate branches based on the feature index and threshold until reaching a leaf node, where it returns the predicted class label.

2.3 Support Vector Machine

2.3.1 Introduction

Support Vector Machines (SVM) are a class of powerful machine learning algorithms used primarily for classification tasks. At their core, SVMs aim to find the best possible separation between different classes of data points by maximising the margin between them. This concept of margin maximisation makes SVMs robust and effective in handling both linearly separable and non-linearly separable datasets. By leveraging sophisticated techniques such as the kernel trick and regularisation parameters, SVMs can capture complex decision boundaries, making them invaluable tools in various fields of study, including pattern recognition, bioinformatics, and finance.

SVMs operate through several key steps to classify data points effectively. Initially, SVMs analyse the labelled dataset to identify classes and features. Then, they determine the optimal hyperplane that maximises the margin, aiming to create the widest possible separation between different classes in the feature space. In this process, SVMs identify support vectors,

which are data points lying closest to the decision boundary. Depending on the dataset's linearity, SVMs may employ the kernel trick to transform the feature space, enabling the classification of non-linearly separable data. Once the hyperplane is established, SVMs can predict the class labels of new data points by determining which side of the hyperplane they fall on. This systematic approach enables SVMs to handle various classification tasks efficiently, making them widely used in machine learning applications.

SVMs offer several advantages that make them popular in the realm of machine learning. One significant advantage is their ability to handle high-dimensional data effectively. SVMs can efficiently process datasets with a large number of features, making them suitable for tasks such as image recognition and text classification where the input data may have numerous dimensions. Additionally, SVMs are memory efficient as they only utilise a subset of training data points, known as support vectors, in the decision-making process. This characteristic not only reduces computational complexity but also enables SVMs to scale well to large datasets, making them practical for real-world applications.

Another advantage of SVMs is their robustness to outliers. By maximising the margin between classes, SVMs inherently prioritise data points that lie closest to the decision boundary, known as support vectors. Consequently, outliers that are far from the decision boundary have minimal impact on the placement of the hyperplane. This resilience to outliers enhances the generalisation performance of SVMs, allowing them to produce reliable predictions even in the presence of noisy or skewed data.

Despite their many advantages, Support Vector Machines (SVMs) also have certain limitations that warrant consideration. One notable drawback is their computational complexity, particularly during training. As the size of the training dataset increases, the time and memory requirements of SVMs can become prohibitive. Additionally, SVMs involve parameter tuning, such as selecting the appropriate kernel and regularisation parameters, which can be challenging and time-consuming, especially for inexperienced users. Moreover, SVMs may not perform well on datasets with imbalanced class distributions, where one class significantly outnumbers the others. In such cases, the algorithm may prioritise accuracy on the majority class while neglecting the minority class, leading to biased predictions and suboptimal performance [6]

2.3.2 Procedure

SVMs perform classification tasks by finding the optimal hyperplane to separate different classes in a dataset. Here's the steps it takes:

1. SVMs start by preparing the dataset, which consists of labeled data points. Each data point is represented by a set of features (attributes) and belongs to one of two or more classes
2. The first step in SVMs is to identify the support vectors. These are the data points that lie closest to the decision boundary also known as the hyperplane between different classes. Support vectors are crucial as they define the hyperplane and determine the margin.
3. SVMs aim to find the hyperplane that maximises the margin, which is the distance between the hyperplane and the nearest support vectors from each class. By maximising the margin, SVMs achieve better generalisation and robustness to noise.
4. (Optional) If the classes are not linearly separable in the original feature space, SVMs can use the kernel trick to map the data into a higher-dimensional space where linear

separation is possible. This transformation allows SVMs to handle non-linear decision boundaries effectively.

5. Once the support vectors and hyperplane are identified, SVMs train the model by optimising parameters such as the regularisation parameter (C) and kernel parameters. Training involves finding the coefficients (weights) of the hyperplane that minimise a certain loss function, typically the hinge loss.
6. After training, SVMs evaluate the performance of the model using validation or testing data. Various evaluation metrics, such as accuracy, precision, recall, F1-score, and AUC-ROC, can be used to assess the model's performance.
7. Once trained and evaluated, SVMs use the learnt hyperplane to make predictions on new, unseen data points. SVMs classify the new data points based on which side of the hyperplane they fall on.
8. SVMs may fine-tune the model parameters, such as the choice of kernel and regularisation parameter, to optimise performance on the validation data and improve generalisation to unseen data.
9. Optionally, SVMs can iterate through steps 5-8 to further refine the model and improve classification accuracy.

The Support Vector Machine algorithm operates by finding the optimal hyperplane that separates different classes in the feature space. This hyperplane is determined by a linear equation that is expressed as follows:

$$f(x) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$

Here, \mathbf{w} represents the weights (coefficients) assigned to each feature, \mathbf{x} is the input feature vector, b is the bias term, and $f(x)$ is the decision function that predicts the class label of a data point x . The $\text{sign}(\cdot)$ function returns the sign of the expression, classifying the data point as belonging to one class (positive sign) or another class (negative sign).

The hyperplane divides the feature space into two regions, corresponding to the different classes. Data points lying on one side of the hyperplane are classified as belonging to one class, while data points on the other side are classified as belonging to the other class.

The decision function $f(x)$ can also be expressed as:

$$f(x) = \sum_{i=1}^n \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b$$

Here, α_i are the Lagrange multipliers, y_i are the class labels of the support vectors, \mathbf{x}_i are the support vectors, $K(\cdot, \cdot)$ is the kernel function that computes the similarity between two data points, and b is the bias term.

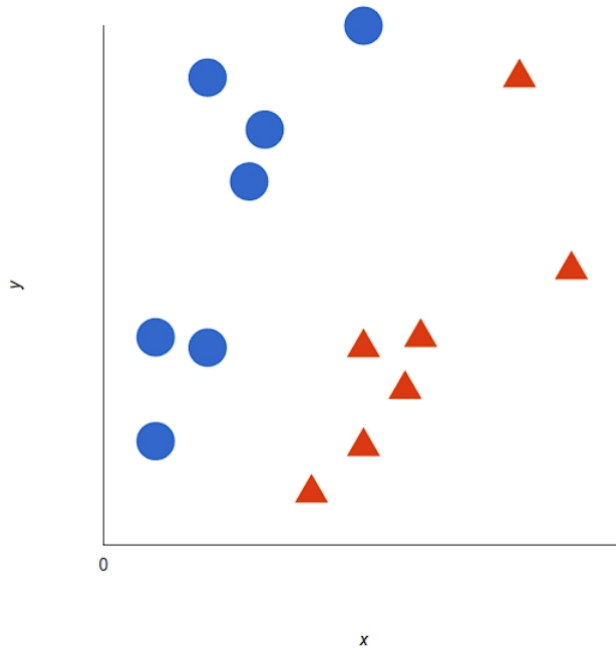
The choice of kernel function $K(\cdot, \cdot)$ is crucial in SVMs, as it determines the transformation of the input feature space. Commonly used kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid kernels. Each kernel function captures different types of relationships between features and can handle different types of data distributions.

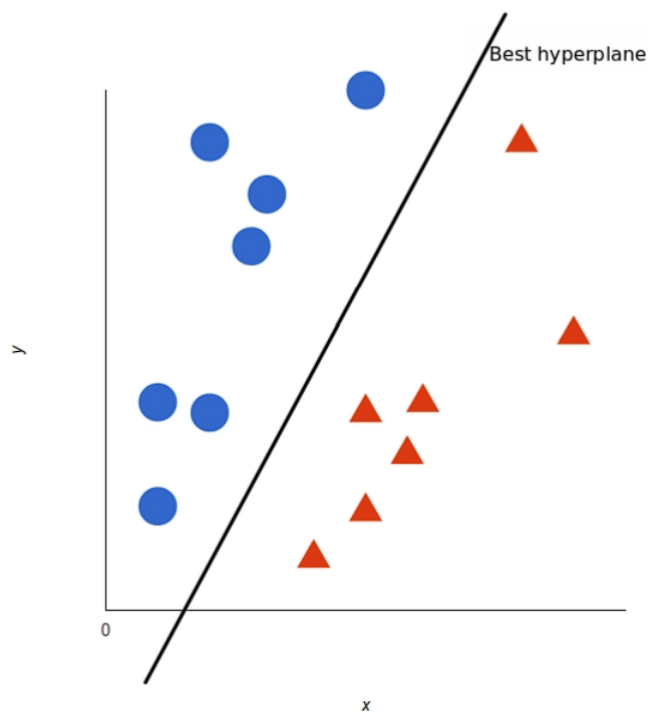
In summary, the SVM equation represents a decision function that assigns class labels to data points based on their position relative to the optimal hyperplane in the feature space. By

optimising the weights \mathbf{w} , bias term b , and kernel parameters, SVMs can effectively classify data points into different classes, making them powerful tools in various machine learning applications.

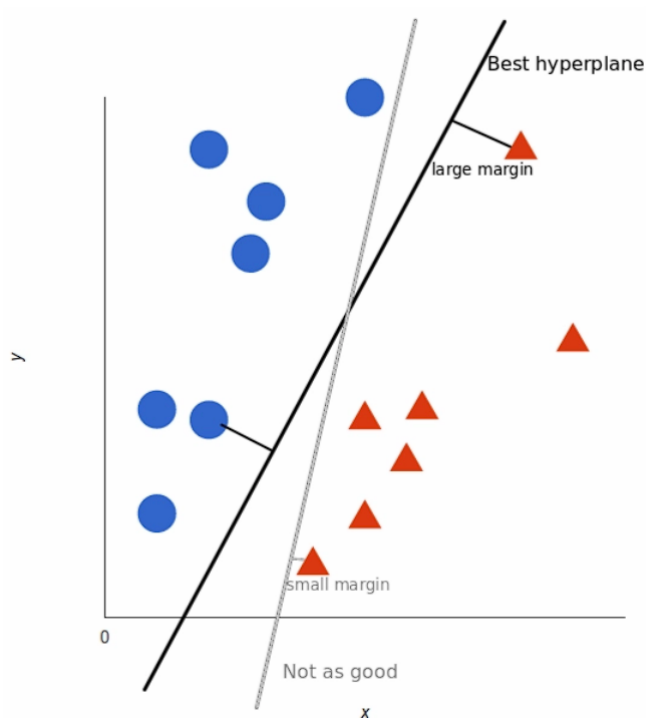
[9]

In the following example, we have two tags: red and blue, and our data has two features: x and y . We require a classifier such that given a pair (x, y) coordinates, it outputs if it's either red or blue.





The best hyperplane is the one that maximises the margins from both tags meaning the hyperplane whose distance to the nearest element of each tag is the largest.



2.3.3 Development

```
def __init__(self, learning_rate=0.0001, lambda_param=0.01, n_iters=1000):
    self.lr = learning_rate
    self.lambda_param = lambda_param
    self.n_iters = n_iters
```

```

self.w = None # Weight vector
self.b = None # Bias term

```

The constructor (init) initialises the SVM classifier with default values for parameters such as learning rate (learning rate), regularisation parameter (lambda param), and the number of iterations (n iters). These parameters are essential for controlling the learning process and optimising the SVM model. The decisions to include these parameters allow flexibility in adjusting the model's behaviour according to different datasets and optimisation requirements.

Adjusting the learning parameters in an SVM classifier can significantly impact its development and performance.

The learning rate (learning rate) determines the step size for updating the model parameters during training. A higher learning rate may lead to faster convergence, as the model parameters are adjusted more aggressively. However, excessively high learning rates can cause instability and overshooting, where the optimisation process oscillates or diverges, hindering convergence. On the other hand, a lower learning rate may result in slower convergence but can lead to more stable and precise updates. Finding the optimal learning rate often involves experimentation and balancing between convergence speed and stability.

The regularisation parameter (lambda param) controls the trade-off between margin maximisation and classification error. A higher regularisation parameter imposes stronger penalties on the model complexity, encouraging simpler decision boundaries and reducing overfitting. Conversely, a lower regularisation parameter allows the model to fit the training data more closely, potentially leading to overfitting. Adjusting the regularisation parameter requires consideration of the dataset's complexity and noise level. Cross-validation techniques can help determine the optimal regularisation parameter that balances bias and variance in the model.

The number of iterations (n iters) specifies the maximum number of training epochs or iterations performed during model training. Increasing the number of iterations allows the model to undergo more updates and refine its parameters further. However, excessively large numbers of iterations may lead to overfitting, where the model memorises the training data instead of learning generalisable patterns. Conversely, insufficient iterations may result in underfitting, where the model fails to capture the underlying patterns in the data. Choosing an appropriate number of iterations involves monitoring the model's performance on validation data and stopping training when further improvement is marginal.

```

def fit(self, X, y):
    n_samples, n_features = X.shape

    # Convert class labels to +1 and -1
    y_ = np.where(y <= 0, -1, 1)

    if np.unique(y_).size == 1:
        print("Only one class present. Skipping training.")
        return

    # Initialize weight vector and bias term
    self.w = np.zeros(n_features)
    self.b = 0

    # Check for non-zero samples in each class
    if np.unique(y_).size == 1:

```

```

        print("Only one class present. Skipping training.")
        return

    # Training the SVM using the selected number of iterations
    for _ in range(self.n_iters):
        for idx, x_i in enumerate(X):
            # Decision rule for updating parameters
            condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
            if condition:
                # Update for correctly classified samples
                self.w -= self.lr * (2 * self.lambda_param * self.w)
            else:
                # Update for misclassified samples
                self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y_[
                self.b -= self.lr * y_[idx]

```

The fit method preprocesses the target labels (y) by converting them to binary labels (+1 and -1). This conversion simplifies the optimisation process and ensures consistency in handling class labels. Additionally, it checks for the presence of only one class in the dataset and skips training if such a scenario occurs. This decision is justified as training an SVM with only one class present would be redundant and ineffective.

Within the fit method, there is a nested loop that iterates over the training data (X). For each iteration, the SVM updates its parameters based on a decision rule. If a data point is correctly classified (satisfying the condition), only the weight vector (self.w) is updated to enforce the margin maximisation and regularisation. However, if a data point is misclassified, both the weight vector and the bias term (self.b) are updated to adjust the decision boundary and correct the misclassification. These decisions align with the core principles of SVMs, focusing on margin maximisation and classification accuracy.

```

def predict(self, X):
    approx = np.dot(X, self.w) - self.b
    return np.sign(approx)

```

The predict method computes the decision function for new data points (X) using the trained parameters (self.w and self.b). It calculates the dot product of input features with the weight vector and subtracts the bias term to obtain an approximation of the decision boundary. Finally, it applies the sign function to determine the predicted class labels based on whether the approximated decision value is positive or negative. This decision follows the standard procedure for making predictions in SVMs, where the sign of the decision function determines the class label.

2.4 Logistic Regression

2.4.1 Introduction

Logistic Regression is a fundamental and versatile statistical technique used for binary classification tasks. Despite its name, logistic regression is a classification algorithm rather than a regression algorithm. It's widely employed in various fields such as healthcare, finance, marketing, and social sciences due to its simplicity, interpretability, and effectiveness.

At its core, logistic regression models the relationship between a binary dependant variable (target variable) and one or more independent variables (predictors) by estimating the probability of occurrence of a certain event. Unlike linear regression, logistic regression uses a logistic function (also known as the sigmoid function) to transform the output of a linear equation into a probability value between 0 and 1.

The logistic regression model outputs probabilities and predicts the class with the highest probability. Typically, if the predicted probability is greater than a predefined threshold (often 0.5), the observation is classified into one category, and if it's less than the threshold, it's classified into the other category.

Logistic regression offers simplicity and interpretability, making it a popular choice for binary classification tasks. Its coefficients provide direct insights into the relationship between predictors and the target variable, making it suitable for both beginners and experts in data analysis. Logistic regression performs efficiently with small to moderate-sized datasets and provides probabilistic outputs, allowing users to assess prediction confidence. However, logistic regression assumes linear relationships between predictors and the target variable and may struggle with complex, non-linear decision boundaries. It's sensitive to outliers and requires careful feature engineering to capture complex relationships adequately. Additionally, logistic regression is primarily designed for binary classification tasks and may not generalise well to multi-class problems. Despite its limitations, logistic regression remains a valuable tool in various domains due to its simplicity, interpretability, and computational efficiency. [2]

2.4.2 Procedure

Logistic Regression predicts dichotomous variables (1 or 0). These steps outline the procedure for training and using the logistic regression classifier to make predictions on new data points. By iteratively updating the model parameters through gradient descent and applying the sigmoid activation function, the classifier learns to separate the two classes based on the input features.

1. Set up the initial conditions and parameters required for the model, including the learning rate and the number of iterations for training. Initialise the model's parameters, such as weights and bias, to appropriate starting values, often zeros or random values.
2. Normalise or standardise the input features to ensure that they are on a similar scale. This step is essential for improving the convergence speed and stability of the optimisation algorithm, as features with different scales can affect the learning process disproportionately.
3. Implement the gradient descent algorithm to optimise the model parameters iteratively. In each iteration, compute the gradients of the cost function with respect to each parameter, indicating the direction and magnitude of the steepest descent. Update the parameters in the opposite direction of the gradients to minimise the cost function and improve the model's performance.
4. Apply an activation function to transform the linear combination of input features and model parameters into a non-linear output. Common activation functions used in logistic regression include the sigmoid function, which maps the input to a value between 0 and 1, representing the probability of belonging to a certain class.
5. Make predictions based on the output of the activation function. Depending on the problem's nature, apply a threshold to the predicted probabilities to classify data points

into different classes. For binary classification tasks, a threshold of 0.5 is often used, where predictions above the threshold belong to one class, and predictions below the threshold belong to the other class.

Logistic regression utilises the logistic function (also known as the sigmoid function) to model the relationship between the input features and the binary target variable. The logistic regression equation models the probability p of a binary outcome (usually coded as 1 or 0) based on one or more predictor variables:

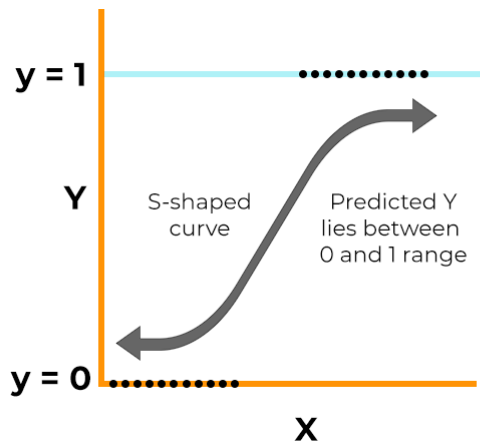
$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

Here, $p(y = 1|\mathbf{x})$ is the probability of the outcome y being 1 given the input features \mathbf{x} . \mathbf{w} represents the weight vector (coefficients) associated with each input feature, \mathbf{x} is the input feature vector, and b is the bias term (intercept). The sigmoid function is a key component of logistic regression, defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid function transforms the linear combination z of input features and model parameters ($\mathbf{w} \cdot \mathbf{x} + b$) into a probability value between 0 and 1. By adjusting the model parameters, logistic regression learns to discriminate between the two classes based on the input features. The decision boundary separating the two classes is determined by the threshold (usually 0.5) applied to the predicted probabilities. Logistic regression equations, along with the sigmoid function, provide a powerful framework for binary classification tasks, allowing for probabilistic predictions and interpretable results.

[4]



Logistic regression relies on the assumption that the dependant or response variable is binary or dichotomous, meaning it can only have two possible outcomes such as pass/fail, male/female, or malignant/benign. This assumption can be verified by examining the unique outcomes of the dependant variable. If more than two outcomes are observed, it suggests a violation of this assumption.

Another key assumption is the absence of multicollinearity among the predictor or explanatory variables. Multicollinearity occurs when independent variables are highly correlated,

which can distort the interpretation of the regression coefficients. The variance inflation factor (VIF) can be used to assess the strength of correlations between independent variables in the regression model.

Logistic regression assumes a linear relationship between the independent variables and the log odds of the outcome. Log odds represent the logarithm of the odds, which differs from probabilities. The linearity assumption can be evaluated by examining the relationship between independent variables and the log odds of the outcome.

A larger sample size is preferred for logistic regression analysis to yield reliable and valid results. This assumption is validated by ensuring an adequate number of cases for each predictor variable, typically at least 10 times the number of cases for the least frequent outcome.

Logistic regression requires the absence of extreme outliers in the dataset, as outliers can disproportionately influence the model results. Cook's distance can be computed for each observation to identify influential data points, and appropriate actions can be taken, such as removing outliers or adjusting the analysis accordingly.

Lastly, logistic regression assumes that observations in the dataset are independent of each other. This means that observations should not be related or derived from repeated measurements of the same individual. Residual plots against time can help assess the presence of a random pattern, indicating whether this assumption is violated.

2.4.3 Development

```
def __init__(self, learning_rate=0.01, num_iterations=1000):
    self.learning_rate = learning_rate
    self.num_iterations = num_iterations
    self.weights = None
    self.bias = None
    self.scaler = StandardScaler()
```

The constructor (init method) initialises the logistic regression model with two essential hyperparameters: the learning rate (learning rate) and the number of iterations for gradient descent (num iterations). By allowing users to specify these parameters, the model becomes flexible and adaptable to different datasets and optimisation requirements. Additionally, the weights and bias attributes are initialised to zeros, which is a common practise to start with an initial state that simplifies the optimisation process.

```
def sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```

The sigmoid method defines the sigmoid activation function, which maps any real-valued input to the range $[0, 1]$. This function is crucial in logistic regression because it converts the output of the linear regression (the log odds) into probabilities. The sigmoid function ensures that the predicted probabilities are bounded and interpretable as class probabilities. By encapsulating the sigmoid function as a separate method, the code maintains modularity and readability.

```
def fit(self, X, y):
```

```

num_samples, num_features = X.shape
self.weights = np.zeros(num_features)
self.bias = 0

# Feature scaling
X_scaled = self.scaler.fit_transform(X)

# Gradient Descent
for _ in range(self.num_iterations):
    linear_model = np.dot(X_scaled, self.weights) + self.bias
    predictions = self.sigmoid(linear_model)

    # Gradient calculations
    dw = (1 / num_samples) * np.dot(X_scaled.T, (predictions - y))
    db = (1 / num_samples) * np.sum(predictions - y)

    # Update weights and bias
    self.weights -= self.learning_rate * dw
    self.bias -= self.learning_rate * db

```

The fit method trains the logistic regression model using gradient descent, an iterative optimisation algorithm. It begins by scaling the input features using StandardScaler to standardise their distribution, which is important for gradient-based optimisation algorithms to converge effectively. The method then iterates through the specified number of iterations, computing predictions, gradients, and updating the model parameters accordingly. The decision to use gradient descent enables the model to learn optimal parameters by minimising the binary cross-entropy loss between predicted probabilities and actual labels.

```

def predict(self, X):
    # Feature Scaling
    X_scaled = self.scaler.transform(X)

    linear_model = np.dot(X_scaled, self.weights) + self.bias
    predictions = self.sigmoid(linear_model)
    return np.where(predictions > 0.5, 1, 0)

```

The predict method makes predictions using the trained logistic regression model. It first scales the input features using the same StandardScaler object used during training to ensure consistency. Then, it computes the linear combination of features and model parameters (weights), applies the sigmoid activation function to obtain probabilities, and converts these probabilities into binary class labels using a threshold of 0.5. This threshold-based decision rule is a standard practise for binary classification tasks.

2.5 Naive Bayes

2.5.1 Introduction

Naive Bayes is a straightforward yet powerful classification algorithm that relies on the principles of Bayes' theorem. It's particularly useful for solving classification problems, where the

goal is to assign a class label to a given input based on its features. What makes Naive Bayes "naive" is its assumption of feature independence, meaning that it assumes each feature contributes independently to the probability of the class label. Despite this simplification, Naive Bayes often performs remarkably well, especially in scenarios with high-dimensional data or when computational resources are limited.

The algorithm works by calculating the probability of each class label given the input features using Bayes' theorem. This involves estimating the likelihood of observing the input features given each class label and combining it with prior probabilities of the class labels to compute posterior probabilities. The class label with the highest posterior probability is then assigned to the input.

One of the key advantages of Naive Bayes is its efficiency and scalability, as it requires minimal training data and can handle large feature sets with ease. It's also robust to noisy data and works well even with incomplete or missing information. Additionally, Naive Bayes is well-suited for text classification tasks, such as spam detection or sentiment analysis, where features represent words or phrases.

One disadvantage of Naive Bayes is its "naive" assumption of feature independence. While this assumption simplifies the model and makes computations more tractable, it may not hold true in real-world scenarios where features are correlated. In cases where features are dependant on each other, Naive Bayes may produce suboptimal results as it fails to capture these dependencies. This can lead to a decrease in predictive accuracy, especially when dealing with complex and highly correlated data. To mitigate this disadvantage, preprocessing techniques such as feature selection or dimensionality reduction can be employed, or more sophisticated algorithms that can handle feature dependencies may be considered. [10]

2.5.2 Procedure

The Naive Bayes classifier is a probabilistic machine learning model based on Bayes' theroem. It assumes independence between features and calculates the probability of a given input belonging to a particular class.

1. Before applying Naive Bayes, the dataset undergoes preprocessing steps such as handling missing values, normalising or standardising numerical features, and encoding categorical variables. This ensures that the data is in a suitable format for training the model.
2. Naive Bayes calculates the likelihood of observing each feature given each class label in the training dataset. For continuous features, it typically assumes a probability distribution such as Gaussian (normal distribution). For categorical features, it estimates probabilities directly from the frequencies of each category within each class.
3. The algorithm computes the prior probability of each class label based on their frequencies in the training dataset. This is simply the proportion of instances belonging to each class in the dataset.
4. Using Bayes' theorem, Naive Bayes calculates the posterior probability of each class label given the observed features. This is done by multiplying the prior probability of each class by the product of the likelihoods of observing the features given that class.
5. Once the posterior probabilities are computed for each class label, Naive Bayes assigns the class label with the highest posterior probability to the input instance. In binary classification, this is often a simple comparison of the posterior probabilities of the

two classes. In multiclass classification, the class label with the maximum posterior probability is chosen.

Bayes' theorem is a fundamental principle in probability theory, named after the Reverend Thomas Bayes. It provides a way to update our beliefs about the probability of an event occurring based on new evidence or information. The theorem is expressed mathematically as:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

Where: - $P(A|B)$ is the posterior probability of event A occurring given that event B has occurred. - $P(B|A)$ is the likelihood of observing event B given that event A has occurred. - $P(A)$ is the prior probability of event A occurring before observing any evidence. - $P(B)$ is the marginal probability of observing event B , also known as the normalising constant.

In simpler terms, Bayes' theorem allows us to update our beliefs about the probability of an event (A) occurring, given new evidence or information (B). It incorporates our prior beliefs about the event ($P(A)$) and how likely we think the evidence would be if the event were true ($P(B|A)$), to calculate the probability of the event occurring given the evidence ($P(A|B)$).

Bayes' theorem is widely used in various fields, including statistics, machine learning, and Bayesian inference. It forms the basis of Bayesian statistics, where it is used to update probabilities as new data becomes available, making it a powerful tool for decision-making and inference. [11]

The following example illustrates a training data set of weather and the corresponding target variable 'Play' (which suggests the possibility of playing outside that day). For this we need to classify whether players will play or not based on the weather condition. We first convert the dataset into a frequency table.

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
All	5	9
	$\approx 5/14$	$\approx 9/14$
	0.36	0.64

Using the Naive Bayesian equation, we can calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of the prediction.

If the weather is sunny, players can go outside:

$$P(\text{Yes} \mid \text{Sunny}) = P(\text{Sunny} \mid \text{Yes}) * P(\text{Yes}) / P(\text{Sunny})$$

Here $P(\text{Sunny} \mid \text{Yes}) * P(\text{Yes})$ is in the numerator, and $P(\text{Sunny})$ is in the denominator.

Here we have $P(\text{Sunny} \mid \text{Yes}) = 3/9 = 0.33$, $P(\text{Sunny}) = 5/14 = 0.36$, $P(\text{Yes}) = 9/14 =$

0.64

Now, $P(\text{Yes} \mid \text{Sunny}) = 0.33 * 0.64 / 0.36 = 0.60$, which has higher probability.

2.5.3 Development

```
def fit(self, X, y):
    self.classes = np.unique(y) # Extract unique class labels
    # Initialise arrays to store class probabilities, means, and standard deviations
    self.classes_prob = np.zeros(len(self.classes))
    self.means = np.zeros((len(self.classes), X.shape[1]))
    self.stds = np.zeros((len(self.classes), X.shape[1]))

    # Calculate class probabilities, means and standard deviations
    for idx, c in enumerate(self.classes):
        X_c = X[y == c]
        self.classes_prob[idx] = len(X_c) / len(X)
        self.means[idx, :] = X_c.mean(axis = 0)
        self.stds[idx, :] = X_c.std(axis = 0)
```

The fit method initialises arrays to store class probabilities, means, and standard deviations. This decision is essential for efficiently computing and storing class-wise statistics during the training phase. Utilising NumPy's functions like unique, mean, and std enables efficient computation of class probabilities, means, and standard deviations for each feature. By leveraging these functions, the classifier ensures scalability and computational efficiency, crucial for handling large datasets.

```
def calcLikelihood(self, x, mean, std):
    exponent = np.exp(-((x - mean) ** 2) / (2 * (std ** 2)))
    return np.prod((1 / (np.sqrt(2 * np.pi) * std)) * exponent)
```

The calcLikelihood method calculates the likelihood of a feature given a class using the Gaussian probability density function. This decision aligns with the assumption of normality in Naive Bayes and allows the classifier to model complex data distributions effectively. By taking advantage of NumPy's vectorised operations for efficient computation, this method enhances the classifier's computational efficiency, especially when dealing with large datasets and multiple features. The use of vectorised operations minimises the need for explicit looping, resulting in faster computation times.

```
def predict(self, X):
    for x in X:
        posteriors = []

        # Calculate the posterior probability for each class
        for idx, c in enumerate(self.classes):
            prior = np.log(self.classes_prob[idx])
            likelihood = np.log(self.calcLikelihood(x, self.means[idx, :], self.stds[
                idx, :]))
            posterior = prior + np.sum(likelihood)
            posteriors.append(posterior)

        # Choose the class with the highest posterior probability
```

```
predictions.append(self.classes[np.argmax posteriors]))  
  
return predictions
```

The predict method calculates the posterior probability for each class and selects the class with the highest posterior probability as the predicted class label. This decision enables the classifier to make informed decisions based on both class probabilities and feature likelihoods. The use of logarithms of probabilities prevents underflow when dealing with small probabilities, enhancing numerical stability and computational efficiency. By employing logarithms, the method mitigates the risk of numerical instability, particularly when dealing with very small probabilities in high-dimensional feature spaces.

Chapter 3: Comparison and Results

All the algorithms listed above are all then loaded into the main module which provided with datasets can output accuracy scores, classification reports and a Receiver Operating Characteristic (ROC) Curve. In this chapter, we will delve into the theoretical background of how the machine learning process actually makes its calculations and displays its results. It's worth noting that not all results here posted here are as efficient as can be due to the algorithm implementation being less than optimal.

3.1 Comparison and Analysis

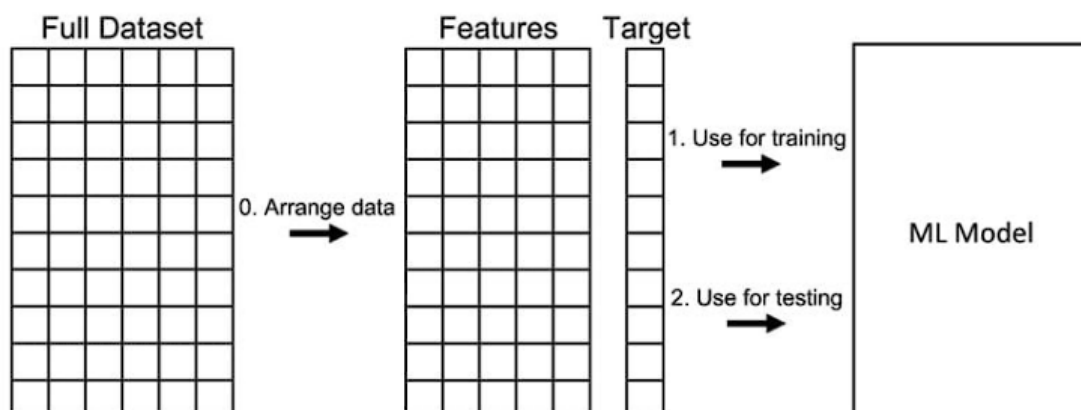
3.1.1 Train-Test splitting of data

In machine learning, we create models that learn from data to make predictions or classifications. However, it's crucial to know how well these models will perform on new, unseen data. Train-test splitting is a method that helps us evaluate our models effectively by dividing our dataset into two parts: one for training the model and one for testing its performance.

Train-test splitting serves several important purposes. First, it lets us see how well our model can predict outcomes on data it hasn't seen before. This helps us understand if our model can generalise its learnings to new situations. Additionally, it allows us to compare different models and choose the best one for our task. Train-test splitting also helps us fine-tune our models by adjusting parameters to improve their performance.

To ensure reliable results, we follow some best practises when splitting our data. We randomly shuffle the dataset before splitting to prevent any biases. If our dataset has different classes or categories, we use a technique called stratified sampling to make sure each category is represented equally in both the training and test sets. It's also important to divide the data in a consistent ratio, such as using 70 to 80 percent for training and 20 to 30 percent for testing.

Train-test splitting is essential for building trustworthy machine learning models. By evaluating models on unseen data and following best practises, we can develop models that not only perform well on the data they were trained on but also generalise effectively to new situations.



In the process of train-test splitting, we partition our dataset into two distinct subsets: one for training the model and the other for testing its performance. This procedure ensures that our model is evaluated on data it hasn't seen during training, enabling us to assess its ability to generalise to new instances effectively. The steps involved in train-test splitting are as follows:

1. Before splitting the data, we typically perform data preparation steps such as cleaning, preprocessing, and feature engineering to ensure that our dataset is ready for analysis. This may involve handling missing values, encoding categorical variables, and scaling numerical features.
2. To prevent any biases that may arise from the dataset's order, we randomly shuffle the data. Randomisation helps ensure that the training and test sets are representative of the overall dataset and that each subset captures the full range of variation present in the data..
3. Once the data is randomised, we partition it into two subsets: the training set and the test set. The training set typically comprises a larger portion of the data, often around 70 to 80 percent, while the test set contains the remaining portion, usually around 20 to 30 percent. The exact ratio may vary depending on the size of the dataset and the specific requirements of the task.
4. We split the randomised data into the training set and the test set according to the predetermined ratio. The training set is used to train the machine learning model, allowing it to learn patterns and relationships in the data. Meanwhile, the test set remains untouched during the training process and is reserved solely for evaluating the model's performance.
5. With the training set in hand, we proceed to train the machine learning model using various algorithms and techniques. The model learns from the input features in the training data and their corresponding target labels, adjusting its parameters to minimise the prediction errors.
6. Once the model is trained, we evaluate its performance using the test set. We input the features from the test set into the trained model and compare its predictions to the actual target labels. This allows us to assess how well the model generalises to new, unseen data and provides valuable insights into its effectiveness in real-world scenarios.
7. Finally, we calculate performance metrics such as accuracy, precision, recall, and F1-score to quantitatively measure the model's performance on the test set. These metrics provide valuable feedback on the model's predictive accuracy, helping us understand its strengths and weaknesses and guiding further improvements if necessary.

It is essential the model learns to fit training data sufficiently otherwise it will run into the common problem of overfitting.

Overfitting is a common problem in machine learning where a model learns to fit the training data too closely, capturing noise or random fluctuations in the data rather than underlying patterns or relationships. This phenomenon occurs when a model becomes overly complex or flexible, allowing it to memorise the training data rather than generalise well to new, unseen data. Several factors contribute to overfitting, including:

- Complex models with a large number of parameters have a higher capacity to fit the training data precisely. While this may result in low training error, it can lead to poor generalisation performance on unseen data.

- When the training dataset is small relative to the complexity of the model, there is a higher risk of overfitting. In such cases, the model may learn to memorise the training examples rather than learn the underlying patterns.
- If the training data contains noise or irrelevant features, the model may mistakenly learn these patterns, leading to overfitting. Noise can arise from various sources, including measurement errors, outliers, or sampling variability.
- Overly complex or redundant features in the dataset can also contribute to overfitting. Including irrelevant features or creating new features based on the training data can introduce noise and lead to overfitting.

Overfitting poses significant challenges to the performance and reliability of machine learning models due to various reasons. Firstly, overfitted models exhibit poor generalisation capabilities, excelling in performance on training data but failing to generalise well to new, unseen data. This limitation compromises their predictive accuracy and reliability in real-world applications, where robustness and adaptability are crucial. Additionally, overfitting renders models highly vulnerable to noise and outliers present in the training data. Even minor fluctuations or outliers can lead to substantial deviations in predictions, undermining the model's stability and robustness. Moreover, overly complex models resulting from overfitting often lack interpretability, making it challenging to derive meaningful insights or explanations from their predictions.

3.1.2 Datasets and Preprocessing

I chose to use 3 datasets for this application, all of which are related to cardiac health. All the datasets used compose of binary classification cases with the final column in each of the datasets being a 1 or 0 (yes or no). Since not all the datasets have the same amount of variables, preprocessing is required.

Preprocessing is a fundamental aspect of machine learning that encompasses a series of techniques aimed at transforming raw data into a format suitable for model training. It plays a pivotal role in enhancing the performance, accuracy, and interpretability of machine learning algorithms by addressing data quality issues, transforming features, and reducing dimensionality.

One of the primary objectives of preprocessing is to ensure the quality and integrity of the data by identifying and handling missing values, outliers, and errors. Data cleaning techniques involve imputing missing data, correcting errors, and removing outliers to eliminate inconsistencies and biases in the dataset. By cleaning the data, preprocessing enhances the reliability and robustness of ML models, preventing them from learning from erroneous or incomplete information.

In many ML algorithms, features are required to be on similar scales to facilitate optimal model performance. Feature scaling techniques, such as standardisation or normalisation, ensure that all features have a comparable range of values, preventing certain features from dominating others during model training. By scaling the features, preprocessing improves the convergence and stability of ML algorithms, leading to more efficient and accurate model predictions.

Preprocessing involves feature engineering techniques aimed at creating new features or transforming existing features to extract relevant information and improve model performance. Feature engineering encompasses tasks such as encoding categorical variables, creating polynomial features, and extracting features from text or images. These techniques enable ML

models to capture complex relationships within the data and make more informed predictions, thereby enhancing their predictive accuracy and effectiveness.

High-dimensional datasets with a large number of features pose challenges for ML algorithms, leading to increased computational complexity and overfitting. Preprocessing techniques such as principal component analysis (PCA) or feature selection help reduce the dimensionality of the data while preserving important information. By reducing the number of features, preprocessing improves model efficiency, generalisation, and interpretability, making it easier to derive meaningful insights from the data.

Preprocessing ensures that the data is normalised or standardised to remove biases and inconsistencies that may exist in the dataset. Normalisation techniques adjust the scale and distribution of the data, making it easier for ML algorithms to converge during training. By normalising the data, preprocessing enhances the stability and consistency of ML models, leading to more reliable and robust predictions.

In the main module, preprocessing is handled in this section:

```
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Encode categorical variables
label_encoder = LabelEncoder()
for i in range(X.shape[1]):
    if isinstance(X[0, i], str):
        X[:, i] = label_encoder.fit_transform(X[:, i])

# Convert to numeric type
X = X.astype(float)

# Convert class labels to +1 and -1
y_binary = np.where(y <= 0, -1, 1)
```

`X = df.iloc[:, :-1].values` extracts the features from the DataFrame `df`, excluding the last column which is assumed to be the target variable. The `.values` attribute converts the DataFrame columns to a NumPy array. `y = df.iloc[:, -1].values` extracts the target variable from the DataFrame.

`label_encoder = LabelEncoder()` creates an instance of the `LabelEncoder` class from `scikit-learn`. A loop iterates over each column of the feature matrix `X`. For each column, it checks if the data type of the first element is a string (`isinstance(X[0, i], str)`). If it is a string, it implies that the column contains categorical variables. For categorical columns, the `LabelEncoder` is used to transform the categorical labels into numerical values. This step is crucial because most machine learning algorithms require numerical inputs.

`X = X.astype(float)` converts all elements in the feature matrix `X` to float type. This step ensures that all feature values are numeric, which is necessary for many machine learning algorithms to operate effectively.

`y_binary = np.where(y != 0, -1, 1)` converts the original class labels stored in `y` to binary format (+1 or -1). This step is common in binary classification tasks, where the target variable often represents two classes. The conversion simplifies the classification task, making it easier for certain algorithms like Support Vector Machines (SVM) to classify the data.

We then need to split the dataset into training and testing subsets. This is done utilising

scikit-learn's 'train test split' function.

```
# Split the data with a consistent random state
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size=0.2, random_state=42)
```

X train: This variable stores the features of the training set. After splitting the dataset, it contains a subset of the original features that will be used to train the machine learning model.

X test: Similarly, this variable holds the features of the testing set. It contains a separate subset of the original features that will be used to evaluate the performance of the trained model.

y train: This variable represents the target labels corresponding to the training set. It contains the class labels or target values associated with the instances in the training data.

y test: Likewise, this variable stores the target labels for the testing set. It includes the true class labels or target values for the instances in the testing data.

train test split(): This function is called to perform the actual splitting of the dataset. It takes several parameters:

X: The feature matrix containing the input data to be split.

y binary: The target variable or class labels associated with the input data.

test size: The proportion of the dataset to include in the testing set. Here, it is set to 0.2, indicating that 20 percent of the data will be allocated to the testing set, while the remaining 80 percent will be used for training.

random state: This parameter ensures reproducibility by setting the random seed. By providing a specific integer value (in this case, 42), the random splitting of the data will be consistent across multiple runs of the code.

Obviously since you can't test on more than one dataset at once, I've implemented a while loop to handle decision making for the user selected dataset. It's pretty self explanatory how this works.

```
while True:
    # User choice for dataset
    dataset_choice = input("Select dataset: \n 1. Heart Failure dataset\n 2. Heart Attack dataset\n 3. Iris dataset\n 4. Heart Disease dataset\n Type 'exit' to quit \n")

    if dataset_choice.lower() == 'exit':
        print("Exiting program...")
        break

    if dataset_choice == '1':
        file_path = 'heartfailure.csv'
        df = pd.read_csv(file_path)
    elif dataset_choice == '2':
```

```

        file_path = 'heartattack.csv'
        df = pd.read_csv(file_path)
    elif dataset_choice == '3':
        # Load the Iris dataset
        iris = load_iris()
        df = pd.DataFrame(data=np.c_[iris['data'], iris['target']], columns=iris['feature
    elif dataset_choice == '4':
        file_path = 'heartdisease.csv'
        df = pd.read_csv(file_path)
    else:
        print("Invalid dataset choice")
        continue

```

The iris dataset was actually the first dataset I used to test all my algorithms as a benchmark beforehand before importing new datasets.

For the algorithm choice I opted for a dictionary over using if else statements.

```

# Create a dictionary to map user choices to classifiers
classifiers = {
    '1': KNNClassifier(k=3),
    '2': SVMClassifier(),
    '3': CustomDecisionTreeClassifier(),
    '4': LogisticRegressionClassifier(),
    '5': NaiveBayesClassifier()
}

# Algorithm selection
algorithm_choice = input("Select algorithm \n 1. K Nearest Neighbours
                        \n 2. SVM
                        \n 3. Decision Trees
                        \n 4. Logistic Regression
                        \n 5. Naive Bayes
                        \n Type 'exit' to quit \n")

if algorithm_choice.lower() == 'exit':
    print("Exiting program...")
    break

# Use the dictionary to get the selected algorithm
algorithm = classifiers.get(algorithm_choice)

if algorithm is None:
    print("Invalid choice")
    continue

# Redirect standard output to null device to suppress output
stdout_original = sys.stdout
sys.stdout = open(os.devnull, 'w')

# Fit the selected algorithm
algorithm.fit(X_train, y_train)

# Restore standard output

```

```
sys.stdout = stdout_original
```

Dictionaries provide cleaner and concise code especially when handling dynamic or changing conditions. Initially I didn't know how many classifiers I was going to develop so updating a dictionary to accommodate new conditions is easier and more adaptable over ruining the code structure with sequential if else statements which are also slower compared dictionaries with their faster look up times.

The selected choice is used to retrieve the corresponding classifier instance from the classifiers dictionary using the `get()` method. If the choice is invalid or not recognised, the program prints a message indicating an invalid choice and continues to the next iteration of the loop. Standard output is redirected to the null device to suppress any output generated during the fitting process of the selected algorithm. This is achieved by temporarily assigning `sys.stdout` to an open file object representing the null device. After fitting the algorithm, the standard output is restored to its original state by assigning `sys.stdout` back to its original value. This ensures that any subsequent output will be displayed normally.

The results outputted by my program output training accuracy, test accuracy alongside their respective classification reports which contains detailed statistics such as precision, recall and F1-score for each class in addition to a cross validation score created with a custom implementation and a ROC curve plot.

```
# Training accuracy
y_train_pred = algorithm.predict(X_train)
train_accuracy = accuracy_score(y_train, y_train_pred)
print(f"Training Accuracy: {train_accuracy * 100:.2f}%")

# Test accuracy
y_test_pred = algorithm.predict(X_test)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

# Generate training classification report
train_class_report = classification_report(y_train, y_train_pred, zero_division=1)
print("Training Classification Report:")
print(train_class_report)

# Generate test classification report
test_class_report = classification_report(y_test, y_test_pred, zero_division=1)
print("Test Classification Report:")
print(test_class_report)

# Cross-validation
cv_accuracy = custom_cross_val_score(algorithm, X, y_binary)
print(f"Cross validation scores: {cv_accuracy * 100:.2f}%")

# Plot ROC Curve for the test set
y_test_score = algorithm.predict(X_test)
plot_roc_curve(y_test, y_test_score, title='ROC Curve for Test Set')
```

The training accuracy provides an indication of how well the model fits the training data whereas the test accuracy assesses the model's performance on unseen data providing insights on its generalisation ability beyond the training set. The classification reports give a more nuanced understanding of the model's performance across different classes.

Precision measures the proportion of correctly predicted positive cases (true positives) out of all instances predicted as positive (true positives + false positives). It represents the model's ability to avoid falsely classifying negative instances as positive. Mathematically, precision is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

A high precision indicates that the model produces few false positive predictions, making it useful in scenarios where false positives are costly or undesirable.

Recall, also known as sensitivity or true positive rate, measures the proportion of correctly predicted positive cases (true positives) out of all actual positive instances (true positives + false negatives). It represents the model's ability to identify all positive instances without missing any. Mathematically, recall is calculated as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

A high recall indicates that the model captures most of the positive instances, which is crucial in scenarios where detecting all positive cases is essential, such as in medical diagnosis or fraud detection.

The F1 score is the harmonic mean of precision and recall, providing a balanced measure that considers both false positives and false negatives. It combines precision and recall into a single metric, making it useful for assessing overall model performance. Mathematically, the F1 score is calculated as:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score reaches its best value at 1 (perfect precision and recall) and its worst at 0. It balances precision and recall, making it suitable for situations where there is an uneven class distribution or when false positives and false negatives have similar importance.

Cross-validation is a technique used to evaluate the performance of a machine learning model by dividing the dataset into multiple subsets and using each subset as both a training and testing set. The cross-validation score represents the average performance of the model across all subsets.

```
def custom_cross_val_score(classifier, X, y, cv=5):
    skf = StratifiedKFold(n_splits=cv, shuffle=True, random_state=42)
    scores = []

    for train_index, test_index in skf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        classifier.fit(X_train, y_train)
        y_pred = classifier.predict(X_test)
```

```

    accuracy = accuracy_score(y_test, y_pred)
    scores.append(accuracy)

return np.mean(scores)

```

The function takes four parameters:

classifier: The machine learning classifier to be evaluated.

X: The input features.

y: The target labels.

cv: The number of folds for cross-validation (default is 5).

Inside the function, a StratifiedKFold object skf is created with n splits=cv, indicating the number of folds for cross-validation. The shuffle=True parameter ensures that the data is shuffled before splitting, and random state=42 sets the random seed for reproducibility.

An empty list scores is initialised to store the accuracy scores obtained from each fold of cross-validation.

The function then iterates over each fold of the cross-validation using a for loop:

For each iteration, skf.split(X, y) generates indices for the training and testing sets. X train, X test, y train, and y test are extracted based on the indices. The classifier is trained (fit) on the training data (X train, y train). Predictions (y pred) are made on the testing data (X test). The accuracy score is computed using accuracy score by comparing the actual labels (y test) with the predicted labels (y pred). The accuracy score is appended to the scores list. After all folds are processed, the function calculates the mean of the accuracy scores stored in the scores list using np.mean(scores).

The mean accuracy score is returned as the cross-validation score for the given classifier.

The Receiver Operating Characteristic (ROC) curve is a graphical representation that illustrates the performance of a binary classification model across different threshold settings. It plots the true positive rate (TPR) against the false positive rate (FPR) for various threshold values. The area under the ROC curve (AUC) quantifies the overall performance of the model, where a higher AUC indicates better discrimination ability.

```

def plot_roc_curve(y_true, y_score, title='ROC Curve'):
    fpr, tpr, thresholds = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC Curve (area = {:.2f})'.format(
    plt.plot([0,1], [0,1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

```


The function takes three parameters:

`y true`: The true binary labels of the data.

`y score`: The predicted probabilities or scores generated by the model.

`title`: The title of the ROC curve plot (default is 'ROC Curve').

Inside the function, the `roc curve` function from `sklearn.metrics` is called to compute the FPR, TPR, and thresholds based on the true labels (`y true`) and predicted scores (`y score`).

The `auc` function calculates the area under the ROC curve (ROC AUC) using the computed FPR and TPR values.

A new matplotlib figure is created using `plt.figure()`.

The ROC curve is plotted using `plt.plot(fpr, tpr, ...)`, where:

`fpr` (False Positive Rate) is plotted on the x-axis.

`tpr` (True Positive Rate or Recall) is plotted on the y-axis.

The line colour is set to 'darkorange', line width to 2, and label includes the computed ROC AUC.

The diagonal line representing random guessing ($FPR = TPR$) is plotted using `plt.plot([0,1], [0,1], ...)`. This line serves as a reference for comparison.

The x-axis and y-axis limits are set to `[0.0, 1.0]` to ensure the entire ROC curve is visible.

Labels for the x-axis ('False Positive Rate') and y-axis ('True Positive Rate') are added using `plt.xlabel` and `plt.ylabel`.

The title of the plot (title) is set using `plt.title`.

A legend is added to the plot, displaying the ROC curve area under the curve value, using `plt.legend`.

Finally, the ROC curve plot is displayed using `plt.show()`.

Regarding feedback given in my interim report, I was suggested to implement the accuracy score function myself instead of using an imported one.

```
def accuracy_score(y_true, y_pred):  
    return np.mean(y_true == y_pred)
```

Accuracy score is a common evaluation metric used to assess the performance of a classification model. It measures the proportion of correctly classified instances out of the total number of instances in the dataset.

Here's how it works:

- True Positive (TP): The model correctly predicts the positive class.
- True Negative (TN): The model correctly predicts the negative class.

- False Positive (FP): The model incorrectly predicts the positive class when the actual class is negative (Type I error).
- False Negative (FN): The model incorrectly predicts the negative class when the actual class is positive (Type II error).

Accuracy is calculated as the ratio of the number of correctly classified instances ($TP + TN$) to the total number of instances in the dataset ($TP + TN + FP + FN$):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The provided code defines a function `accuracy_score` that computes the accuracy score given the true labels (`y_true`) and predicted labels (`y_pred`). Here's what the code does:

1. It compares each element in the `y_true` array with the corresponding element in the `y_pred` array to check if they are equal (`y_true == y_pred`).
2. This comparison results in a boolean array where `True` indicates a correct prediction and `False` indicates an incorrect prediction.
3. The `np.mean` function calculates the mean of this boolean array, which represents the proportion of correct predictions in the dataset.
4. The mean value is returned as the accuracy score.

3.2 Results

Here are all the results I received for every dataset including the Iris dataset as a benchmark.

3.2.1 Figures

3.2.2 Performance

Upon analysing the performances of various machine learning algorithms, it is evident that each algorithm has its strengths and weaknesses. Naive Bayes consistently demonstrates robust performance across different datasets, showcasing its effectiveness in classification tasks. Its ability to handle both numerical and categorical data with relatively simple assumptions makes it a versatile choice for various applications. Support Vector Machine (SVM) also exhibits commendable performance, particularly on the Heart Attack dataset, indicating its capability to handle complex classification problems with high-dimensional data.

However, some algorithms like K Nearest Neighbours (KNN) show moderate performance, which suggests that their effectiveness may depend on the dataset characteristics and parameter settings. Decision Trees and Logistic Regression, on the other hand, often struggle to achieve satisfactory results, highlighting their limitations in capturing complex relationships within the data.

In conclusion, while Naive Bayes and SVM emerge as strong contenders for classification tasks, it is essential to consider the specific requirements of each problem and the nature

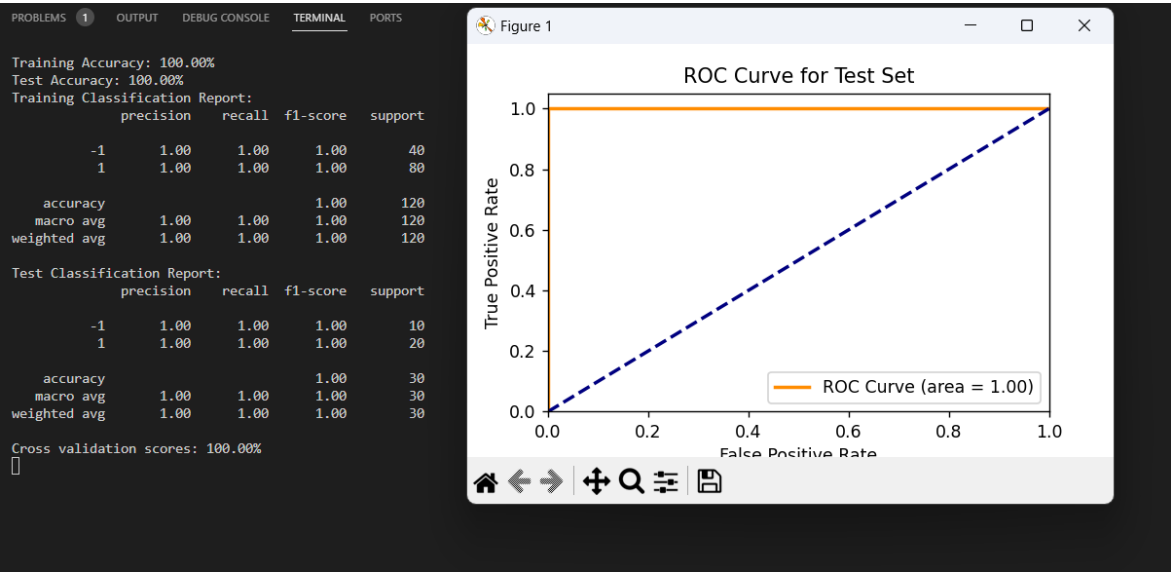


Figure 3.1: Iris - K Nearest Neighbours

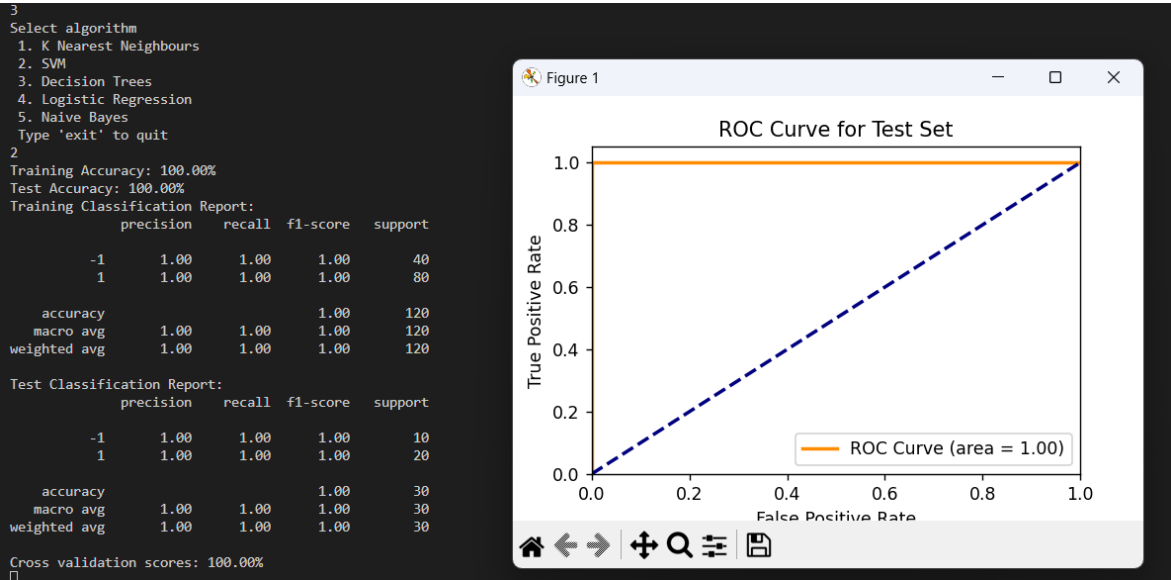


Figure 3.2: Iris - Support Vector Machine

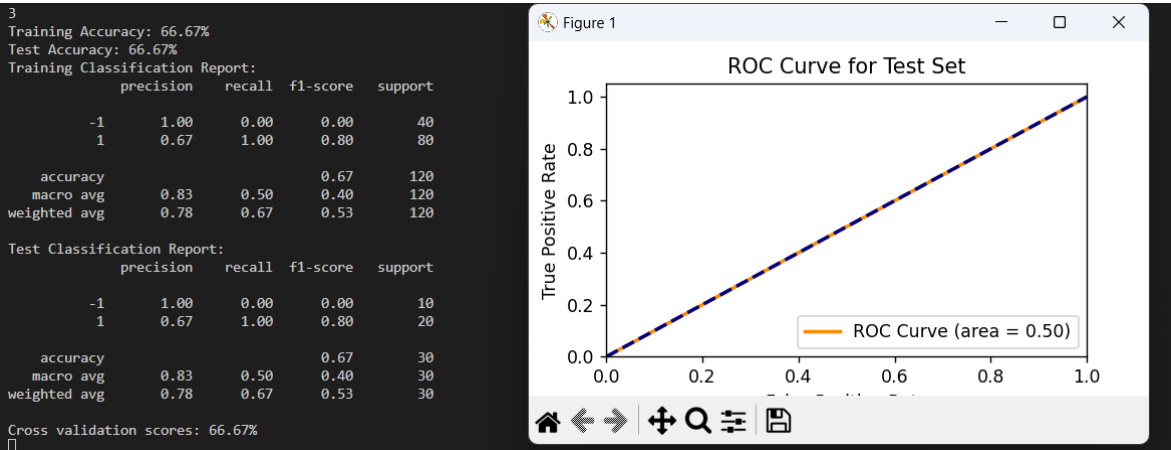


Figure 3.3: Iris - Decision Trees

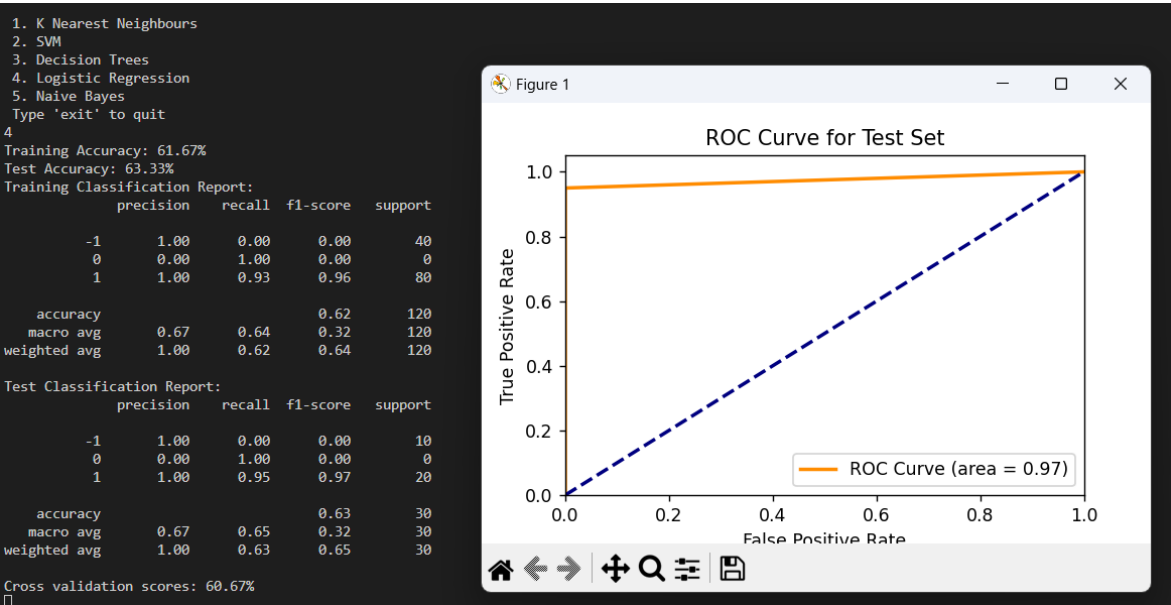


Figure 3.4: Iris - Logistic Regression

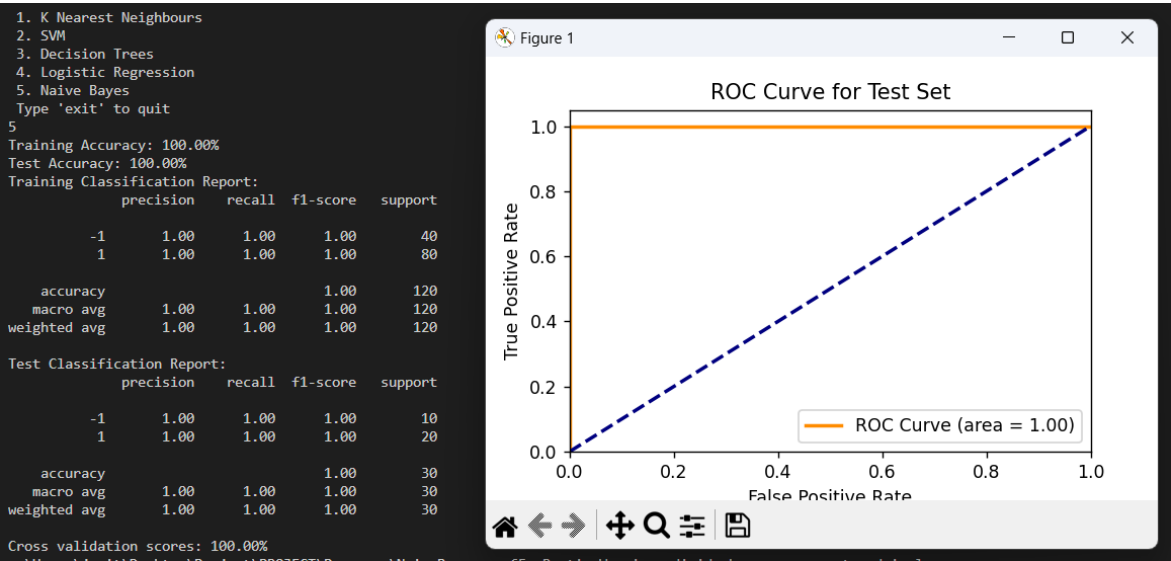


Figure 3.5: Iris - Naive Bayes

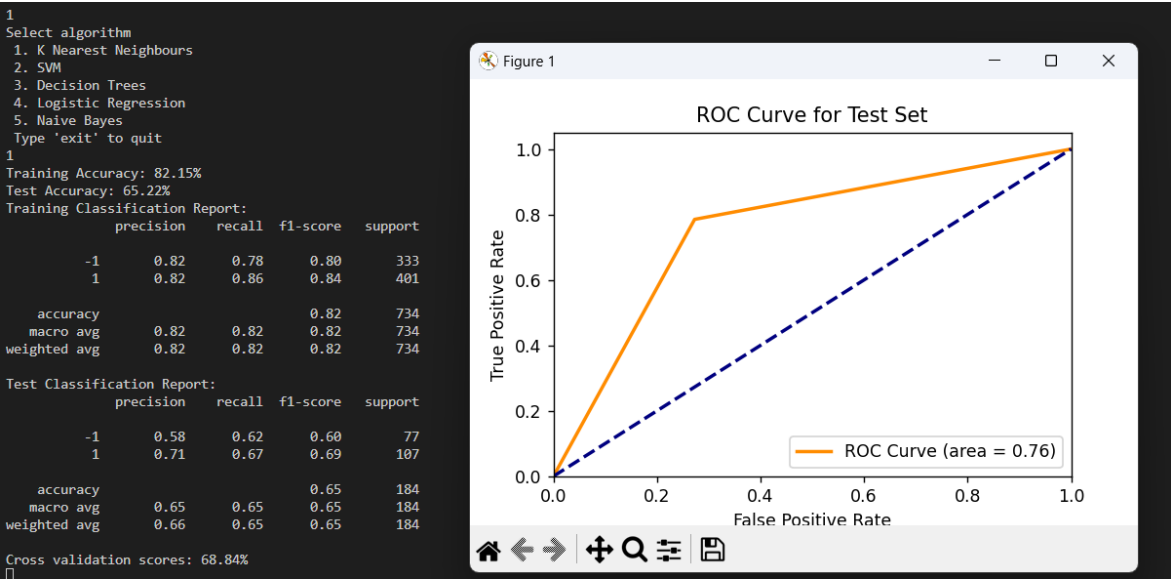


Figure 3.6: Heart Failure - K Nearest Neighbours

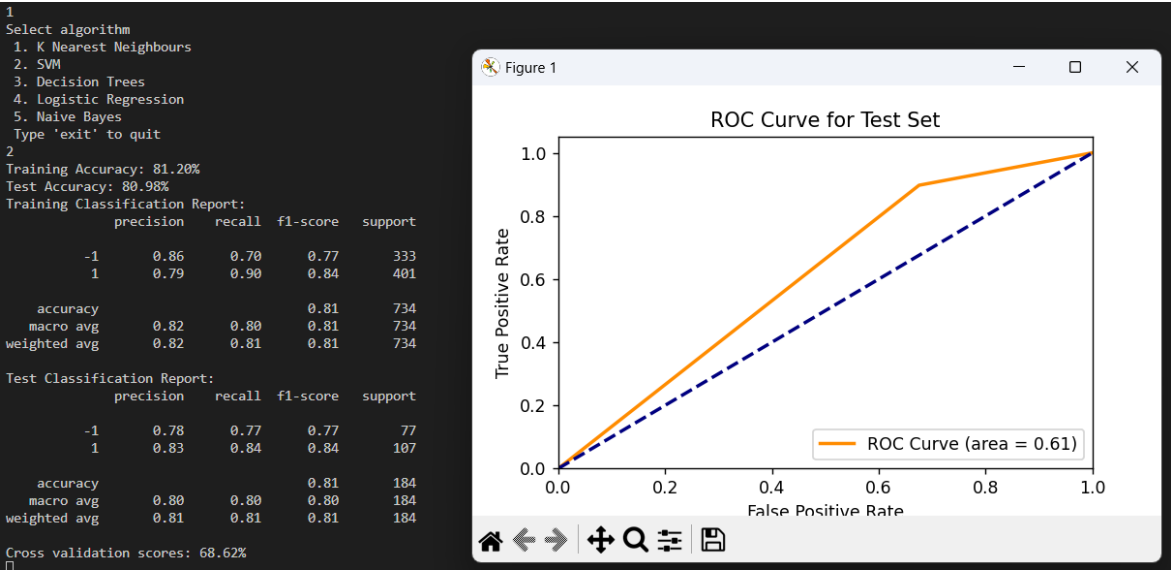


Figure 3.7: Heart Failure - Support Vector Machine

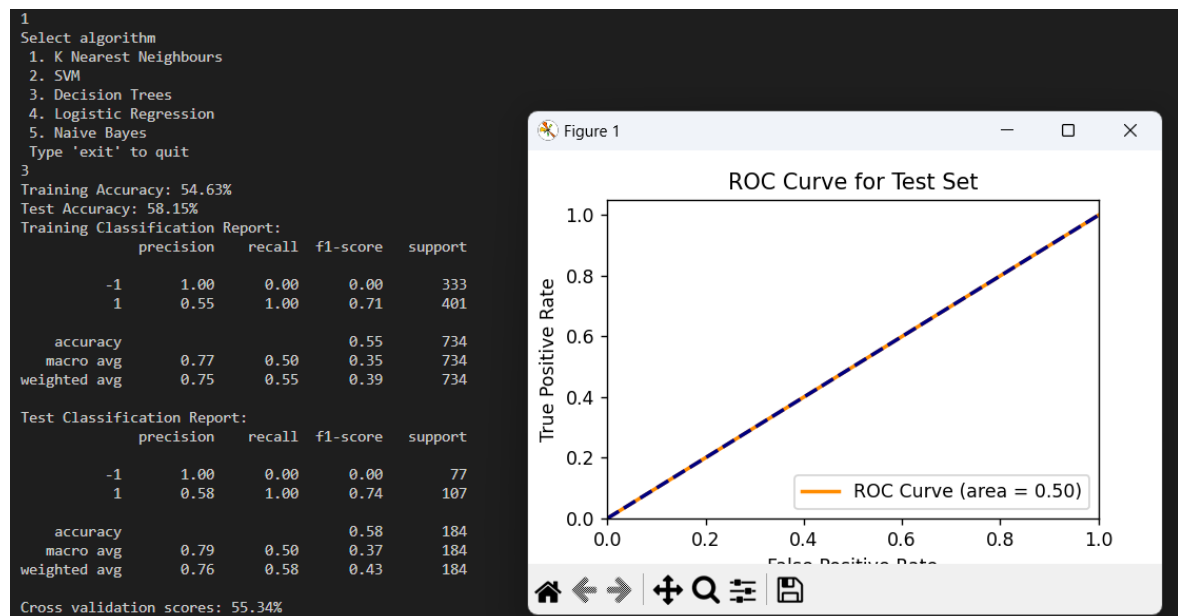


Figure 3.8: Heart Failure - Decision Trees

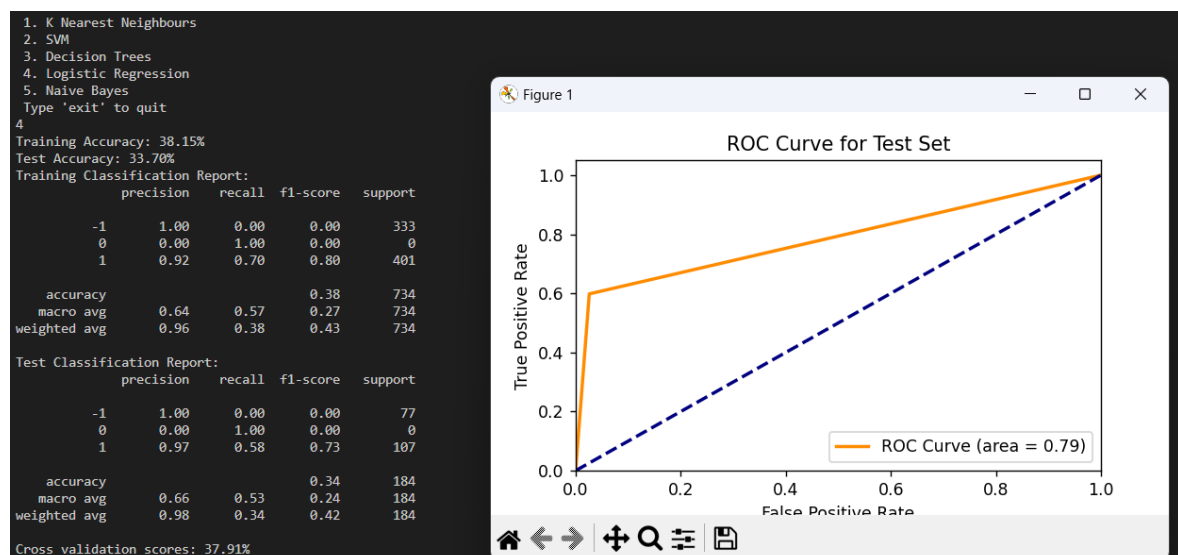


Figure 3.9: Heart Failure - Logistic Regression

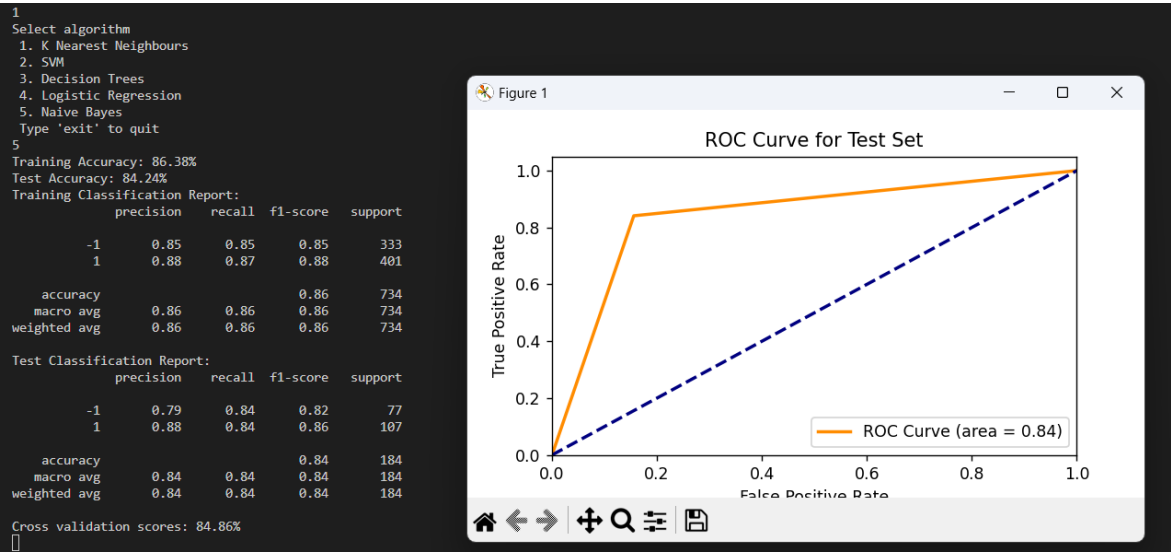


Figure 3.10: Heart Failure - Naive Bayes

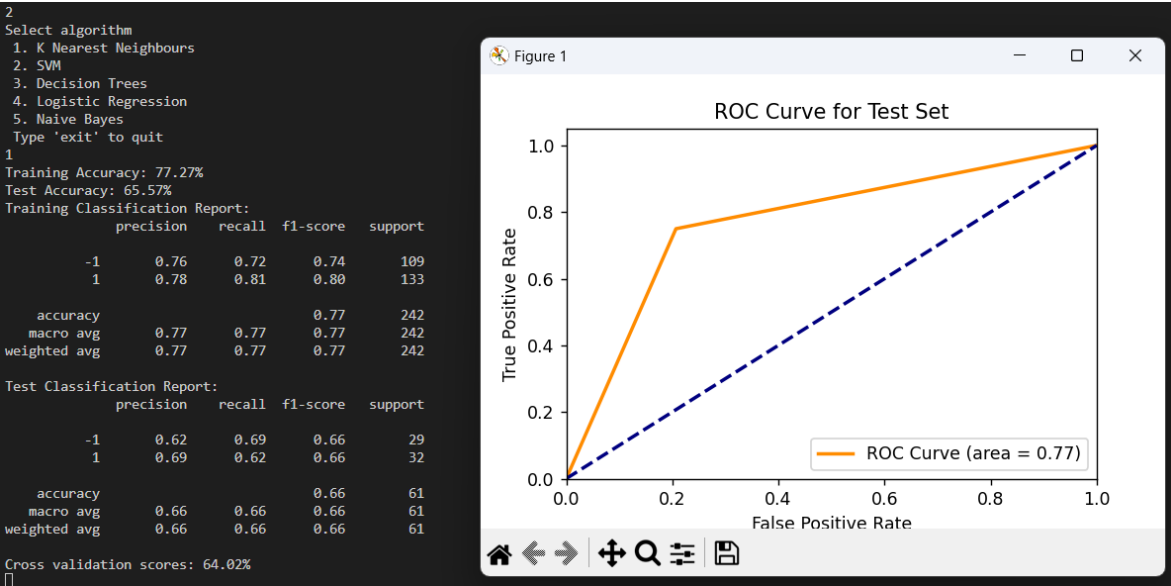


Figure 3.11: Heart Attack - K Nearest Neighbours

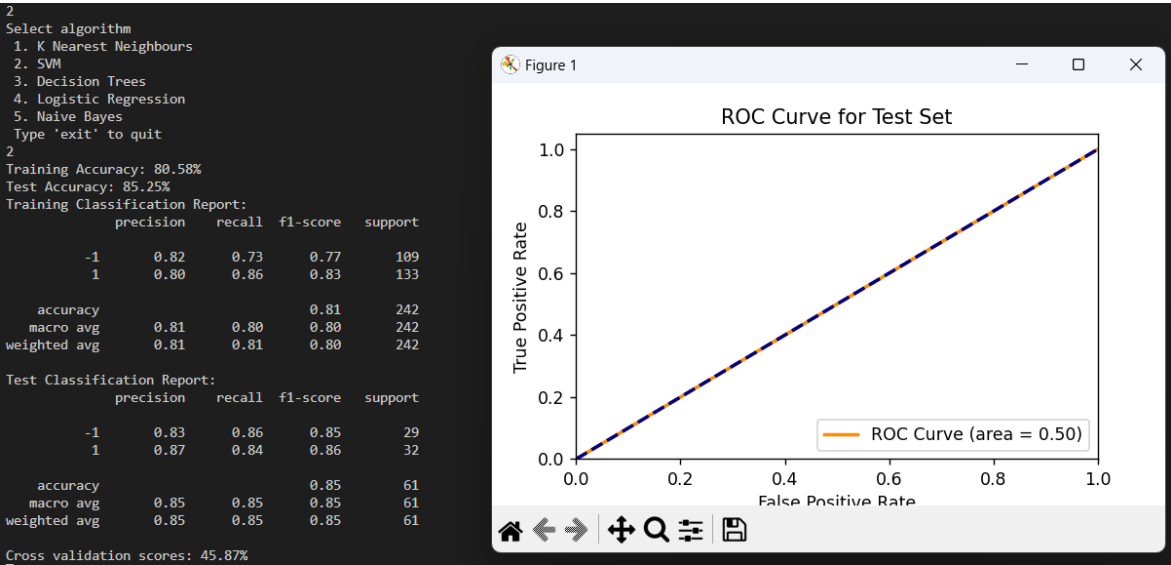


Figure 3.12: Heart Attack - Support Vector Machines

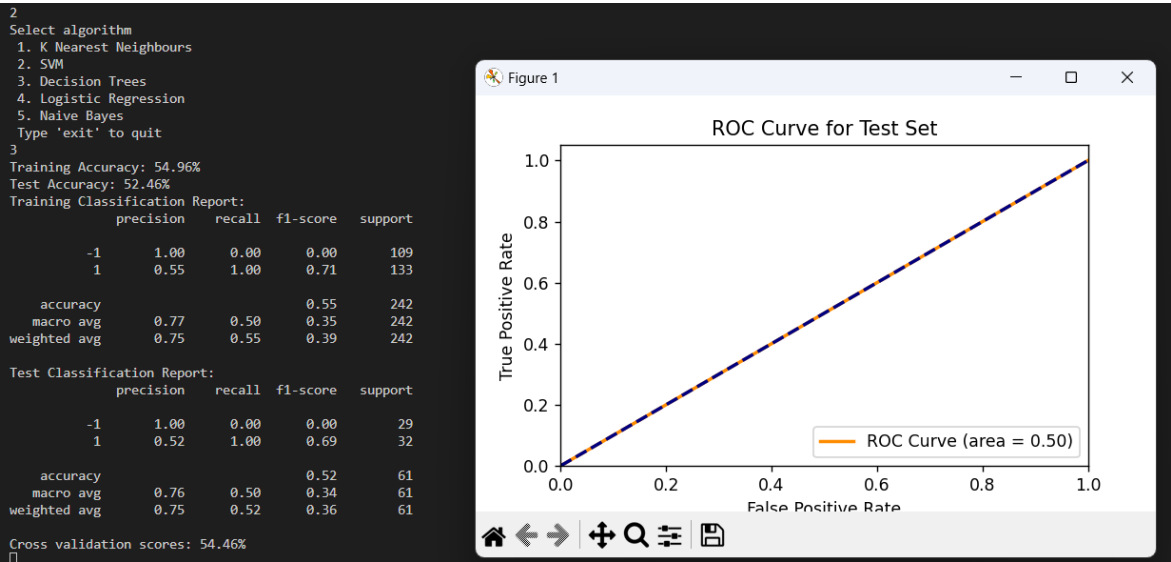


Figure 3.13: Heart Attack - Decision Trees

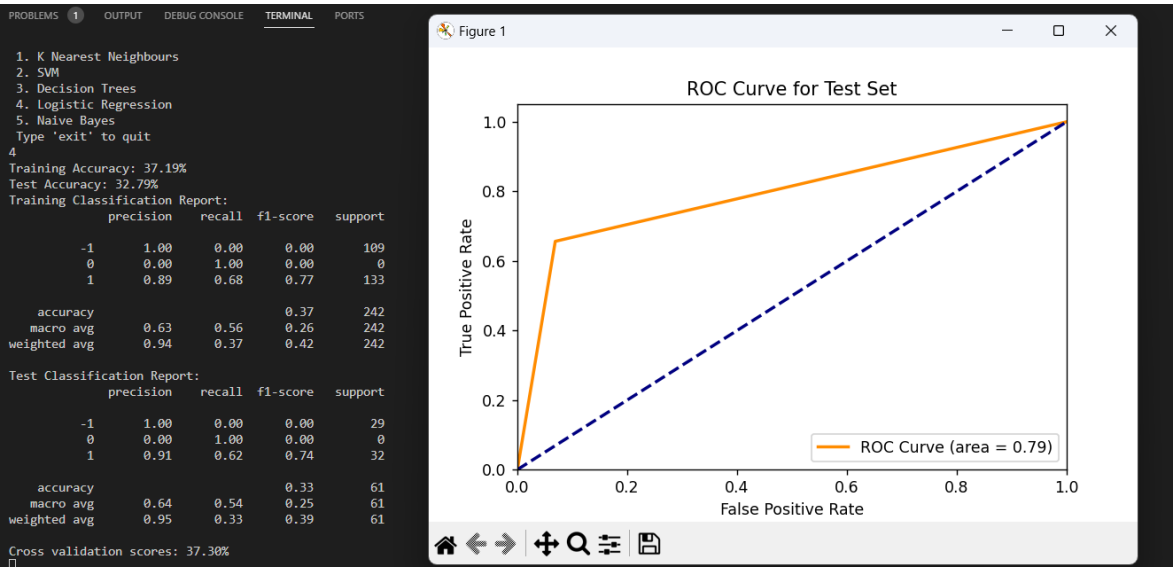


Figure 3.14: Heart Attack - Logistic Regression

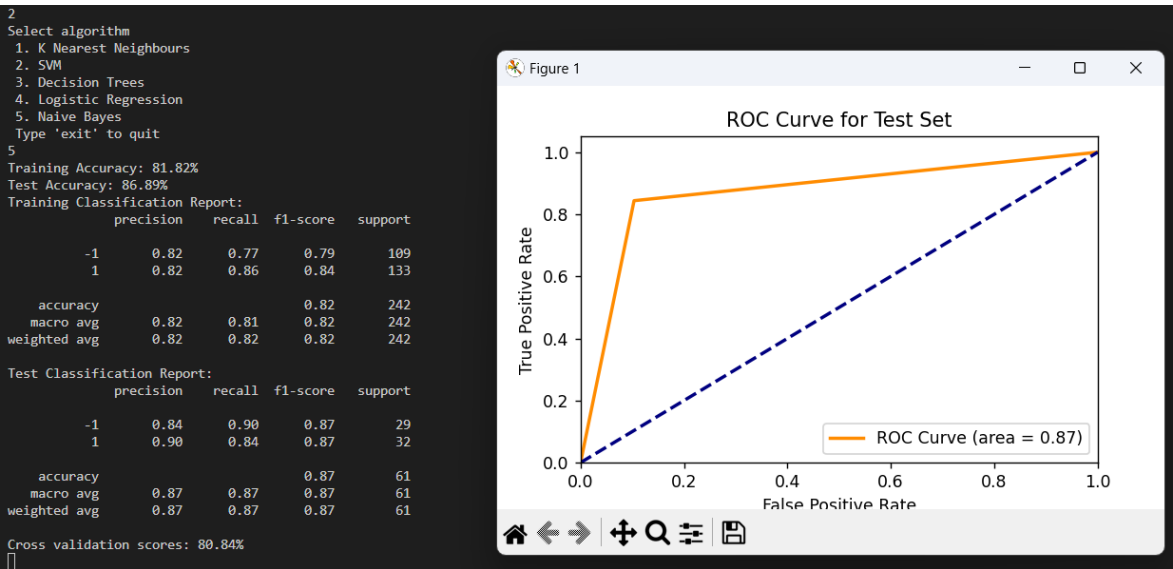


Figure 3.15: Heart Attack - Naive Bayes

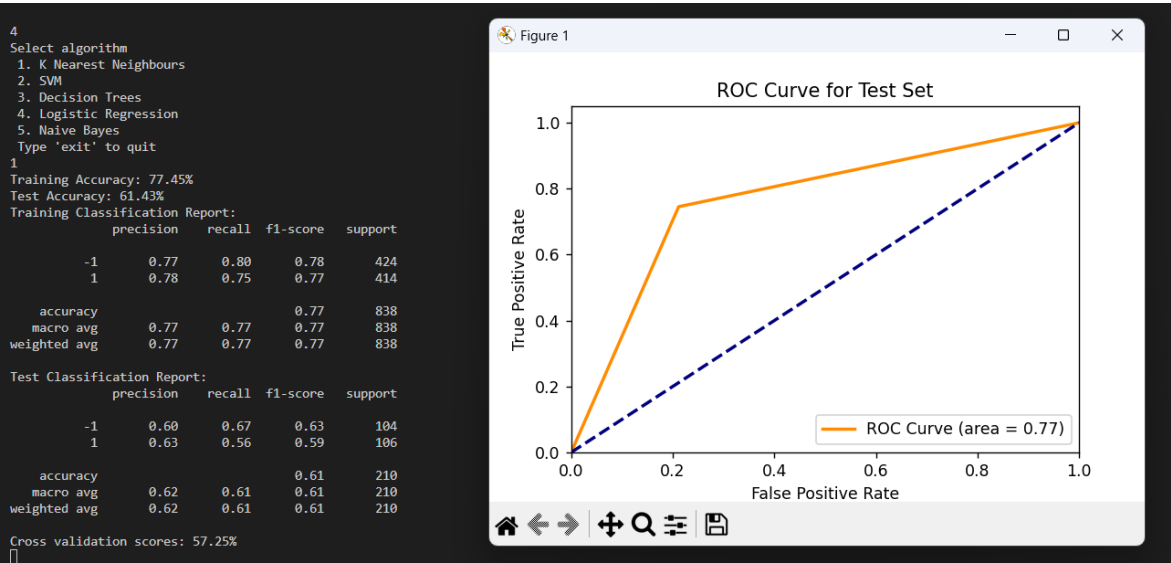


Figure 3.16: Heart Disease - K Nearest Neighbours

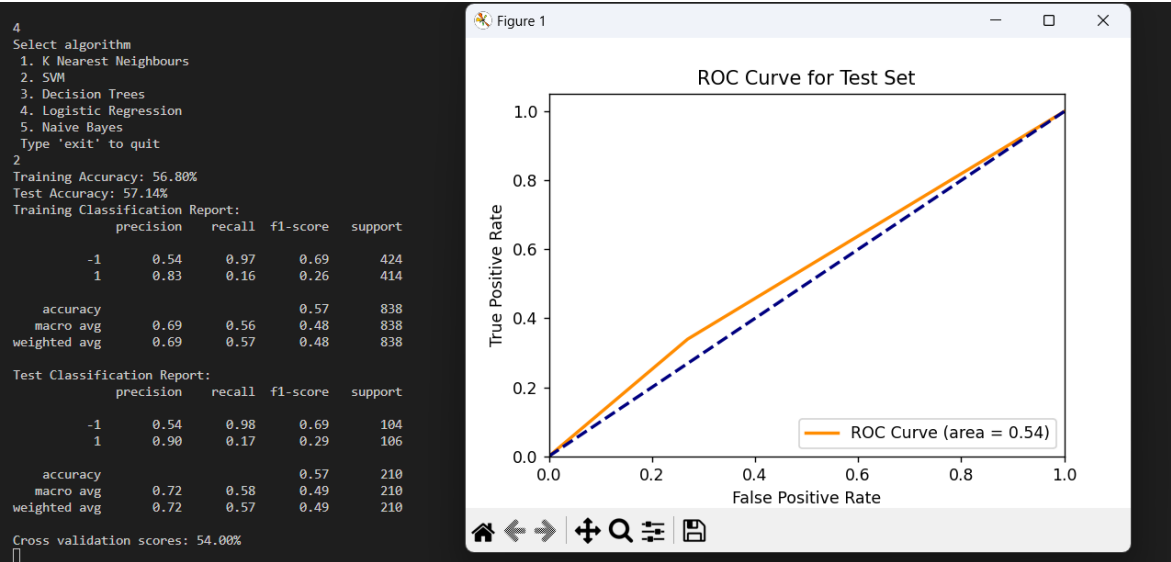


Figure 3.17: Heart Disease - Support Vector Machine

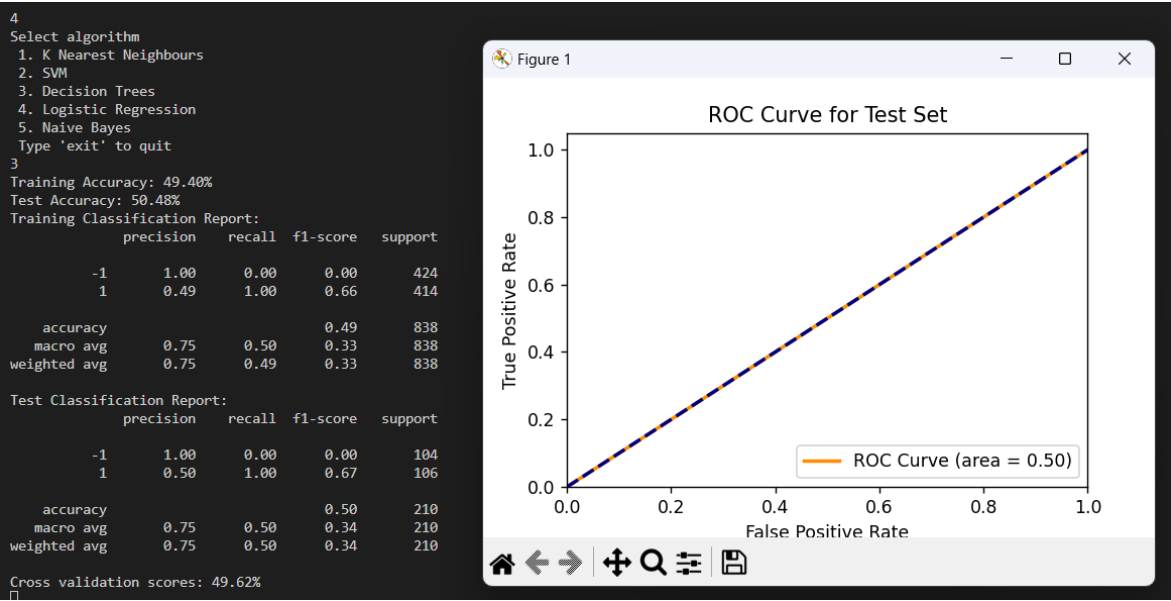


Figure 3.18: Heart Disease - Decision Trees

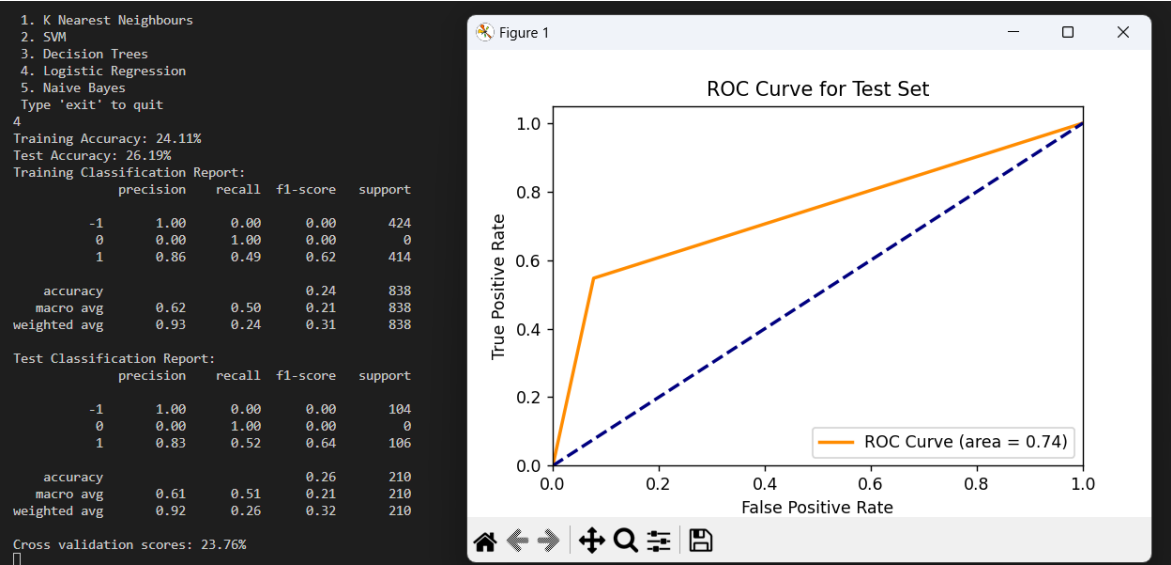


Figure 3.19: Heart Disease - Logistic Regression

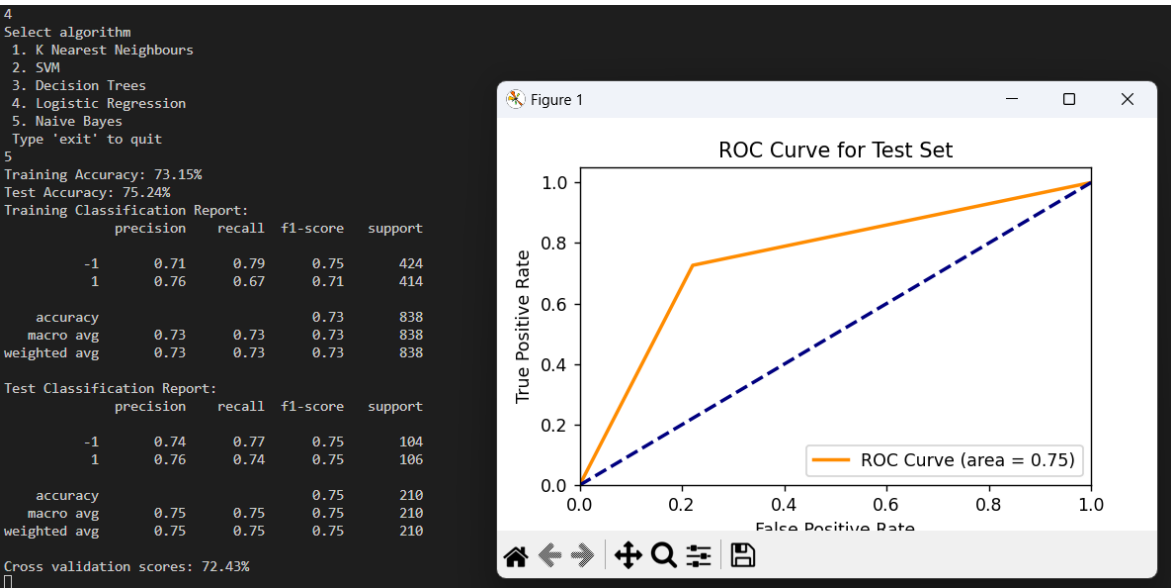


Figure 3.20: Heart Disease - Naive Bayes

of the dataset when selecting an algorithm. Furthermore, fine-tuning hyperparameters and exploring ensemble methods could potentially enhance the performance of algorithms that exhibit subpar results. Overall, this analysis underscores the importance of thorough experimentation and evaluation to identify the most suitable algorithm for a given task.

3.2.3 Professional Issues

Utilising machine learning in cardiac health presents a myriad of professional issues that necessitate careful consideration and ethical adherence. Here are some key areas of concern:

Data Privacy and Security: Handling sensitive patient data raises significant privacy and security concerns. Healthcare professionals and data scientists must ensure compliance with regulations such as HIPAA (Health Insurance Portability and Accountability Act) to safeguard patient confidentiality and prevent unauthorised access or breaches.

Bias and Fairness: Machine learning algorithms trained on biased or incomplete datasets may perpetuate existing inequalities or disparities in healthcare outcomes. Healthcare providers must strive to address biases in data collection, algorithm design, and model evaluation to ensure fair and equitable treatment for all patients, regardless of demographic factors.

Interpretability and Transparency: The black-box nature of some machine learning algorithms poses challenges in understanding how predictions are made. Healthcare professionals need interpretable models that provide insights into decision-making processes, enabling them to trust and validate the algorithm’s recommendations.

Accountability and Liability: As machine learning algorithms play an increasingly prominent role in clinical decision-making, questions of accountability and liability arise. Healthcare providers must clarify responsibilities and establish protocols for addressing errors, ensuring that patients receive appropriate care and recourse in the event of algorithmic failures.

Regulatory Compliance: Adherence to regulatory frameworks governing medical devices and software is paramount. Machine learning models used in cardiac health applications must undergo rigorous validation and obtain regulatory approval to ensure safety, efficacy, and compliance with industry standards.

Clinical Validation and Integration: Validating the effectiveness and reliability of machine learning models in real-world clinical settings is essential. Healthcare professionals should collaborate with data scientists to conduct rigorous clinical trials and integrate validated algorithms seamlessly into existing healthcare workflows.

Patient Autonomy and Informed Consent: Patients have the right to understand how their data is being used and to consent to its utilisation for research or clinical purposes. Transparent communication and informed consent processes are crucial for respecting patient autonomy and fostering trust in machine learning applications in cardiac health.

Continued Education and Training: Healthcare professionals must stay abreast of advances in machine learning and data science to effectively leverage these technologies in cardiac health. Ongoing education and training programmes can empower clinicians to harness the full potential of machine learning while mitigating risks and challenges.

Bibliography

- [1] Oliver Kramer and Oliver Kramer. K-nearest neighbors. *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23, 2013.
- [2] Michael P LaValley. Logistic regression. *Circulation*, 117(18):2395–2399, 2008.
- [3] Umarani Nagavelli, Debabrata Samanta, Partha Chakraborty, et al. Machine learning technology-based heart disease detection models. *Journal of Healthcare Engineering*, 2022, 2022.
- [4] Todd G Nick and Kathleen M Campbell. Logistic regression. *Topics in biostatistics*, pages 273–301, 2007.
- [5] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.
- [6] Derek A Pisner and David M Schnyer. Support vector machine. In *Machine learning*, pages 101–121. Elsevier, 2020.
- [7] Yan-Yan Song and LU Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130, 2015.
- [8] Shan Suthaharan and Shan Suthaharan. Decision tree learning. *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*, pages 237–269, 2016.
- [9] Shan Suthaharan and Shan Suthaharan. Support vector machine. *Machine learning models and algorithms for big data classification: thinking with examples for effective learning*, pages 207–235, 2016.
- [10] Geoffrey I Webb, Eamonn Keogh, and Risto Miikkulainen. Naïve bayes. *Encyclopedia of machine learning*, 15(1):713–714, 2010.
- [11] Feng-Jen Yang. An implementation of naive bayes classifier. In *2018 International conference on computational science and computational intelligence (CSCI)*, pages 301–306. IEEE, 2018.
- [12] Qi Zhenya and Zuoru Zhang. A hybrid cost-sensitive ensemble for heart disease prediction. *BMC medical informatics and decision making*, 21:1–18, 2021.

- MonkeyLearn, AnalyticsVidhya, KDNuggets, Spicellearn, sci-kit