



# MACHINE LEARNING

Submitted to:

**Dr. Mahmoud Khalil**

**Eng. Engy Ahmed**

Submitted by:

|                      |                |
|----------------------|----------------|
| <b>Lina Sameh</b>    | <b>21P0198</b> |
| <b>Malak Mahmoud</b> | <b>21P0278</b> |
| <b>Nouran Magdy</b>  | <b>2100688</b> |

## Table of Contents

|  |    |
|--|----|
| 1. Introduction .....                            | 3  |
| 2. Dataset Description.....                      | 3  |
| 2.1 Dataset Overview .....                       | 3  |
| 2.2 Columns Description .....                    | 3  |
| 3. Dataset Preprocessing .....                   | 5  |
| 3.1 Initial Data Inspection.....                 | 5  |
| 3.2 Handling Missing Values.....                 | 5  |
| 3.3 Encoding the Data .....                      | 6  |
| 3.4 Standardization .....                        | 6  |
| 3.5 Outlier Detection.....                       | 7  |
| 3.6 Duplicate Removal:.....                      | 7  |
| 3.7 Data Splitting: .....                        | 7  |
| 4. Dataset Visualization.....                    | 7  |
| 4.1 PCA Visualization: .....                     | 7  |
| 4.2 Outlier Analysis:.....                       | 9  |
| 5. Machine Learning Models.....                  | 9  |
| 5.1 Naïve Bayes .....                            | 9  |
| 5.1.1 Hyperparameter Tuning .....                | 10 |
| 5.1.2 Testing.....                               | 11 |
| 5.2 SVM .....                                    | 13 |
| 5.2.1 Hyperparameter Tuning .....                | 13 |
| 5.2.2 Testing.....                               | 15 |
| 5.3 KNN.....                                     | 15 |
| 5.3.1 Hyperparameter Tuning Parameter Grid:..... | 15 |

|       |  |    |
|-------|--|----|
| 5.2.1 | Testing.....                               | 17 |
| 5.4   | Decision Trees .....                       | 18 |
| 5.4.1 | Hyperparameter Tuning Parameter Grid:..... | 18 |
| 5.4.2 | Testing.....                               | 20 |
| 6.    | Histogram.....                             | 21 |
| 6.1   | Code:.....                                 | 21 |
| 6.2   | Results: .....                             | 23 |
| 7.    | Dendrogram .....                           | 23 |
| 8.    | Conclusion.....                            | 26 |

## 1. Introduction

Our approach makes use of the Heart Failure Prediction Dataset, which comprises a target variable that indicates the presence of heart disease as well as clinical and demographic characteristics.

In order to predict the risk of heart failure, we intend to analyze the dataset, visualize the patterns we find, and develop machine learning models. Classifiers like Naïve Bayes, SVM, KNN, and Decision Trees are trained as part of the analysis, which also includes data cleaning and preprocessing. Furthermore, we create a dendrogram to illustrate hierarchical clustering, offering information on patient groups according to their characteristics.

## 2. Dataset Description

### 2.1 Dataset Overview

This dataset focuses on **heart disease diagnostics** and includes clinical data about patients' demographics, symptoms, and test results. The primary objective is to predict the likelihood of a patient having heart disease based on various medical attributes.

- A) **Objective:** To identify patterns and relationships between patient attributes and heart disease occurrence. This can help in building predictive models to aid in early diagnosis and risk assessment.
- B) **Dataset Type:** Structured tabular data with a mix of **numerical** and **categorical** features.
- C) **Total Rows:** Approximately 918 **records**
- D) **Total Features:** 12 columns, including one target variable (HeartDisease)

### 2.2 Columns Description

- **Age**
  - **Type:** Numeric (integer)
  - **Description:** Age of the patient.
  - **Range:** 36 to 60

- **Sex**
  - **Type:** Categorical
  - **Description:** Gender of the patient.
  - **Values:** M (Male), F (Female)
- **ChestPainType**
  - **Type:** Categorical
  - **Description:** Type of chest pain.
  - **Values:** ATA (Atypical Angina), NAP (Non-Anginal Pain), ASY (Asymptomatic), TA (Typical Angina)
- **RestingBP (Resting Blood Pressure)**
  - **Type:** Numeric (integer)
  - **Description:** Resting blood pressure in mmHg.
  - **Range:** 100–160
- **Cholesterol**
  - **Type:** Numeric (integer)
  - **Description:** Cholesterol levels in mg/dL.
  - **Range:** 180–468
- **FastingBS (Fasting Blood Sugar)**
  - **Type:** Binary (0 or 1)
  - **Description:** Indicates if fasting blood sugar > 120 mg/dL.
  - **Values:** 0 (False), 1 (True)
- **RestingECG (Resting Electrocardiogram Results)**
  - **Type:** Categorical
  - **Description:** Results of the resting ECG.
  - **Values:** Normal, ST (ST-T wave abnormality), Left Ventricular Hypertrophy
- **MaxHR (Maximum Heart Rate Achieved)**
  - **Type:** Numeric (integer)
  - **Description:** Maximum heart rate achieved during exercise.
  - **Range:** 98–178
- **ExerciseAngina**
  - **Type:** Binary (Y/N)
  - **Description:** Indicates if exercise-induced angina occurred.
  - **Values:** Y (Yes), N (No)

- **Oldpeak**
  - **Type:** Numeric (float)
  - **Description:** ST depression induced by exercise relative to rest.
  - **Range:** 0–3
- **ST\_Slope**
  - **Type:** Categorical
  - **Description:** Slope of the peak exercise ST segment.
  - **Values:** Up (Upsloping), Flat, Down (Downsloping)
- **HeartDisease**
  - **Type:** Binary (0 or 1)
  - **Description:** Indicates if the patient has heart disease.
  - **Values:** 0 (No), 1 (Yes)

### 3. Dataset Preprocessing

In our project, we began by examining the *Heart Failure Prediction Dataset* to gain an initial understanding of its structure and content. This step involved exploring the data and preparing it for further analysis

#### 3.1 Initial Data Inspection

- We used `data.head()`, `data.info()`, and `data.describe()` to inspect the first few rows, understand the data types, and summarize the statistical properties of the dataset.
- Then we checked for missing values using `data.isnull()`, the dataset did not contain any missing values, simplifying the preprocessing process.

#### 3.2 Handling Missing Values

- In case missing values were found:

- A) **Numerical Columns:** Missing values would have been imputed using the **mean** strategy with SimpleImputer. This ensures that numerical features remain unbiased by replacing null values with the average of the column.
  - B) **Categorical Columns:** Missing values would have been filled using the **most frequent value (mode)** to maintain the integrity of the categorical data.
  - C) The imputed numerical and categorical columns would then be combined into a new DataFrame to ensure a clean and complete dataset
- As noted during the exploration phase, the dataset did not contain any missing values, so no imputation or removal was necessary.

### 3.3 Encoding the Data

- To prepare the categorical features for machine learning, we implemented label encoding to convert categorical variables into numeric labels.
  - A) **Data Separation:** We created two DataFrames, one for categorical data and one for numeric data, to facilitate appropriate preprocessing for each type.
  - B) **Label Encoder Creation:** An instance of a label encoder was created to convert categorical text data into numeric values.
  - C) **Applying Label Encoding:** The label encoder was applied to each categorical column, transforming unique categories into corresponding numeric labels.
  - D) **Data Combination:** After encoding, we reset the index of the categorical DataFrame and combined it with the numeric DataFrame to form a new, complete dataset, ready for analysis.

### 3.4 Standardization

- To prepare the dataset for modeling, we standardized the features to ensure they are on a similar scale.
  - A) **Feature and Target Separation:** The features were separated into a DataFrame X, while the target variable, HeartDisease, was stored in y.
  - B) **Standardization Process:** We used a standard scaler to transform the features, ensuring they have a mean of 0 and a standard deviation of 1.
  - C) **Creating a Scaled DataFrame:** The scaled features were stored in a new DataFrame, data\_scaled\_df, with the target variable added back in, resulting in a standardized dataset ready for analysis.

### 3.5 Outlier Detection

- Outliers in the dataset were identified using the Z-score method. This allowed us to pinpoint data points that significantly deviated from the mean, which could impact model training and evaluation.

### 3.6 Duplicate Removal:

- The dataset was checked for duplicate records to ensure the integrity of the data, Any duplicates found were removed using `drop_duplicates()`

### 3.7 Data Splitting:

- The dataset was prepared by separating the target variable, HeartDisease, from the features. We defined the target column and created X for features and y for the target. The shapes of X and y were displayed to confirm their dimensions.
- The data was then split into training, validation, and test sets: 70% for training and 30% for validation and testing, which were further divided equally. This approach ensures effective training and accurate evaluation of machine learning models.

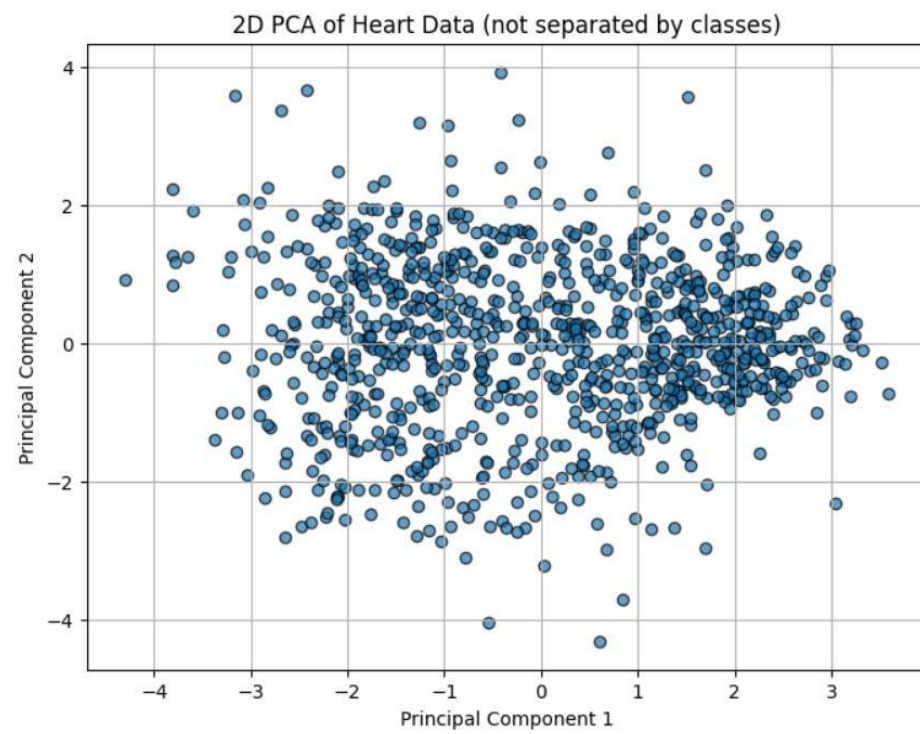
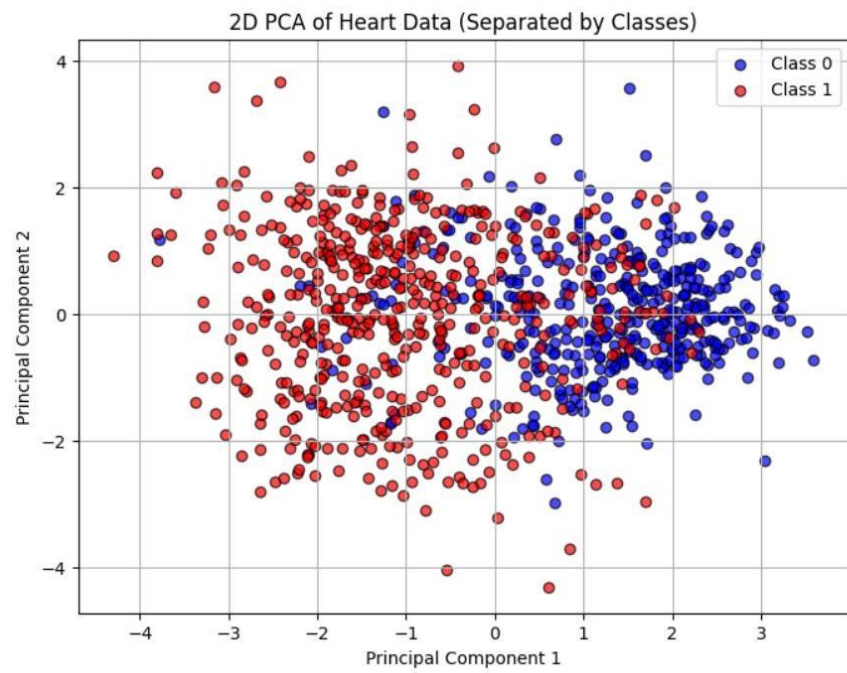
## 4. Dataset Visualization

In this section, we explored the *Heart Failure Prediction Dataset* to gain insights into the feature distributions, relationships between variables, and overall structure of the data. Visualization techniques were applied to better understand the dataset

### 4.1 PCA Visualization:

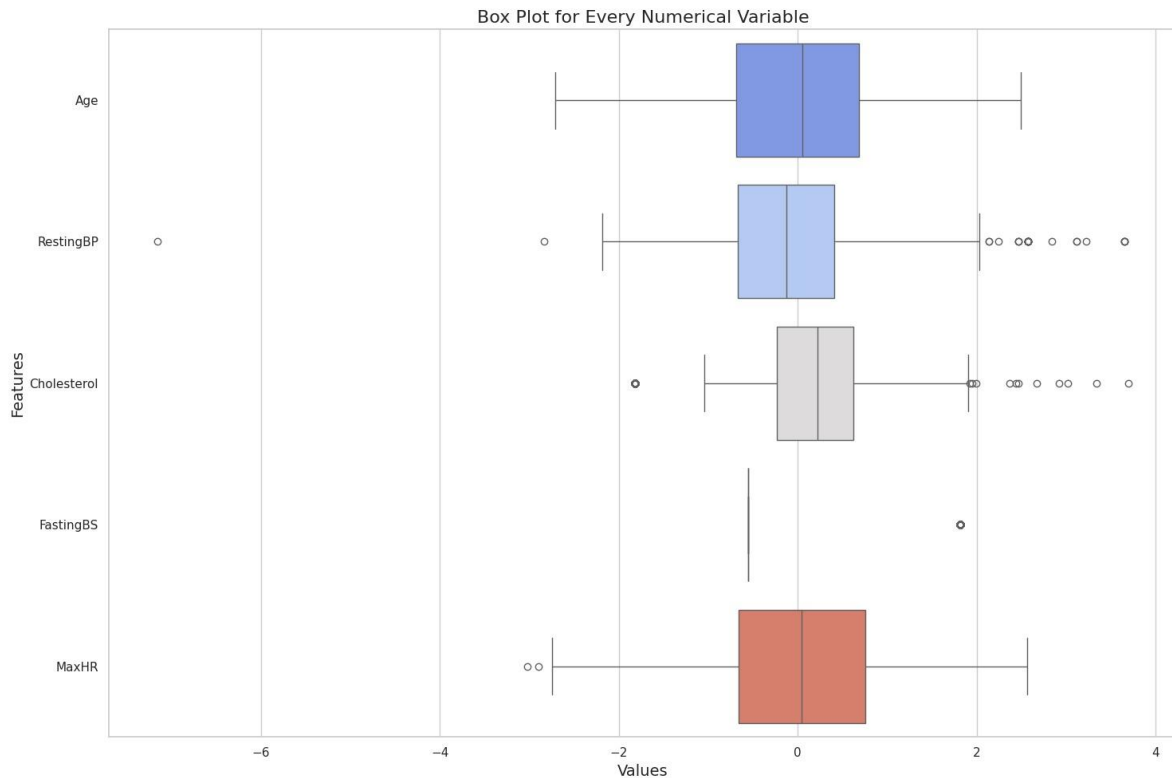
- To explore the data, **Principal Component Analysis (PCA)** was applied to reduce the dataset to two dimensions. The resulting scatter plot helped in visualizing patterns, clusters, or separations within the data.
- We did two plots, one with classes separated, and the other one wasn't separated, shown below:





## 4.2 Outlier Analysis:

- Outliers identified through the **Z-score method** were visualized using boxplots. This helped us understand their impact on the overall data distribution.



## 5. Machine Learning Models

### 5.1 Naïve Bayes

Training a Naïve Bayes classifier involves modeling the probability distribution of each class based on its features using a labeled dataset. During training, the prior probabilities of each class and the likelihood of each feature given the class are calculated, assuming feature independence (the "naive" assumption). These probabilities form the basis of the model, which predicts the class of new, unseen data by applying Bayes' theorem.

### 5.1.1 Hyperparameter Tuning

In this code, hyperparameter tuning is performed for a Naive Bayes classifier using GridSearchCV to optimize the `var_smoothing` parameter.

```
# Define the Naive Bayes model
nb_model = GaussianNB()

# Define the hyperparameter grid
param_grid = {'var_smoothing': np.logspace(-9, -3, 100)}

# Perform GridSearchCV
grid_search = GridSearchCV(estimator=nb_model, param_grid=param_grid, cv=5, scoring='accuracy', return_train_score=True)
grid_search.fit(X_train, y_train)

# Get the results
results = grid_search.cv_results_

# Extract the best hyperparameters and accuracies
best_params = grid_search.best_params_
mean_train_scores = results['mean_train_score'] # Now this key will exist
mean_test_scores = results['mean_test_score']
param_values = results['param_var_smoothing']

# Print the best hyperparameters
print(f"Best hyperparameters: {best_params}")

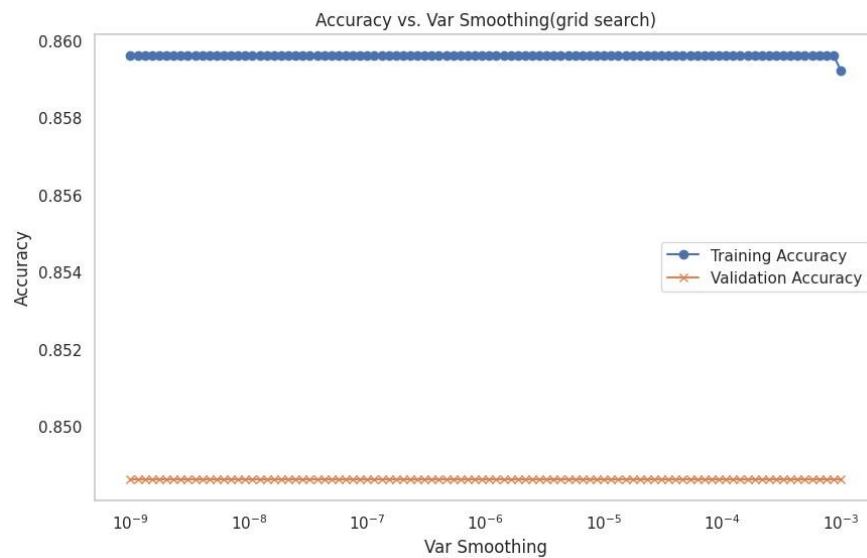
# Plot training and validation
plt.figure(figsize=(10, 6))
plt.plot(param_values, mean_train_scores, label="Training Accuracy", marker="o")
plt.plot(param_values, mean_test_scores, label="Validation Accuracy", marker="x")
plt.xscale('log')
plt.xlabel("Var Smoothing")
plt.ylabel("Accuracy")
plt.title("Accuracy vs. Var Smoothing(grid search)")
```

In this code, hyperparameter tuning is performed for a Naive Bayes classifier using **GridSearchCV** to optimize the **var\_smoothing** parameter.

- **Model Definition:**  
The GaussianNB model is initialized as `nb_model`. This classifier is suitable for continuous features and assumes the data follows a Gaussian distribution.
- **Hyperparameter Grid:**  
A grid of values for the `var_smoothing` parameter is defined using `np.logspace(-9, -3, 100)`. This parameter helps stabilize variance calculations by adding a small value to the variance, preventing numerical instability.
- **GridSearchCV Setup:**
  - GridSearchCV is used to find the optimal value of `var_smoothing` by evaluating different values from the grid.
  - `cv=5` indicates 5-fold cross-validation, splitting the dataset into five subsets for training and validation.
  - The scoring metric is set to `accuracy`, and `return_train_score=True` ensures training scores are captured for analysis.
- **Training:**  
The model is trained on the dataset (`X_train` and `y_train`) using `grid_search.fit()`. For each value of `var_smoothing`, the accuracy is calculated for both the training and validation sets.

- Results Extraction:
  - `grid_search.cv_results_` contains detailed results, including mean accuracy scores for training and validation for each parameter value.
  - `grid_search.best_params_` identifies the value of `var_smoothing` that yielded the best validation accuracy.

### Accuracy vs. var\_smoothing



#### 5.1.2 Testing

This code evaluates the performance of the best Naive Bayes model, obtained from hyperparameter tuning, on both the validation and test datasets using key classification metrics.

```

best_nb_model = random_search.best_estimator_
nb_y_val_pred = best_nb_model.predict(X_val)
nb_y_test_pred = best_nb_model.predict(X_test)

from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix, classification_report

# Evaluate on the validation set
print("Validation Metrics:")

# Recall, Precision, F1-Score for validation
nb_recall_val = recall_score(y_val, nb_y_val_pred, average='binary')
nb_precision_val = precision_score(y_val, nb_y_val_pred, average='binary')
nb_f1_val = f1_score(y_val, nb_y_val_pred, average='binary')

# Print validation metrics
print(f"Recall: {nb_recall_val:.2f}")
print(f"Precision: {nb_precision_val:.2f}")
print(f"F1-score: {nb_f1_val:.2f}")

# Evaluate on the test set
print("\nTest Metrics:")

# Recall, Precision, F1-Score for test
nb_recall_test = recall_score(y_test, nb_y_test_pred, average='binary')
nb_precision_test = precision_score(y_test, nb_y_test_pred, average='binary')
nb_f1_test = f1_score(y_test, nb_y_test_pred, average='binary')

# Print test metrics
print(f"Recall: {nb_recall_test:.2f}")

```

- The model, stored as `best_nb_model`, is used to predict labels for the validation (`X_val`) and test (`X_test`) sets.
- The predictions are compared to the true labels (`y_val` and `y_test`) to compute metrics such as recall, precision, and F1-score using the `recall_score`, `precision_score`, and `f1_score` functions from `sklearn.metrics`.
- These metrics provide insights into the model's ability to correctly identify positive cases (recall), the accuracy of its positive predictions (precision), and the harmonic mean of recall and precision (F1-score), which balances these two aspects.

```

Validation Metrics:
Recall: 0.84
Precision: 0.87
F1-score: 0.86

Test Metrics:
Recall: 0.81
Precision: 0.87
F1-score: 0.84

```

## 5.2 SVM

A Support Vector Machine (SVM) classifier is a supervised learning algorithm used for classification tasks, particularly effective in high-dimensional spaces. It works by finding the optimal hyperplane that separates data points of different classes with the maximum margin, ensuring robust generalization. SVM can handle both linear and non-linear classification problems by using kernel functions, such as linear, polynomial, or radial basis function (RBF), to map data into higher dimensions where a linear separation is possible.

### 5.2.1 Hyperparameter Tuning

This code performs hyperparameter tuning for a Support Vector Machine (SVM) classifier using GridSearchCV to optimize its performance.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Initialize the SVC model
SVMmodel = SVC()

# Define the hyperparameter grid
param_grid = {
    'C': [0.1, 1, 10, 100, 1000],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': [0.0001, 0.001, 0.01, 0.1, 1]
}

# Set up GridSearchCV
grid = GridSearchCV(SVMmodel, param_grid, refit=True, verbose=3, return_train_score=True)

# Fit the model
grid.fit(X_train, y_train)
```

- The SVM model is initialized using SVC() from sklearn.svm.
- A hyperparameter grid is defined to explore different combinations of C (regularization parameter), kernel (linear, RBF, polynomial), and gamma (kernel coefficient) values. GridSearchCV systematically evaluates each combination of these parameters through cross-validation, using refit=True to automatically fit the model with the best parameter set.
- The model is then trained on the training dataset (X\_train, y\_train) using grid.fit(), allowing the best-performing hyperparameters to be identified based on validation scores.
- The process ensures an optimal configuration of the SVM classifier for the given data.

#### Hyperparameter Tuning Results for SVM Classifier using GridSearchCV

```
[CV 5/5] END C=1000, gamma=0.1, kernel=linear, score=(train=0.860, test=0.873) total time= 3.2s
[CV 1/5] END C=1000, gamma=0.1, kernel=rbf, score=(train=1.000, test=0.795) total time= 0.0s
[CV 2/5] END C=1000, gamma=0.1, kernel=rbf, score=(train=1.000, test=0.780) total time= 0.0s
[CV 3/5] END C=1000, gamma=0.1, kernel=rbf, score=(train=1.000, test=0.811) total time= 0.0s
[CV 4/5] END C=1000, gamma=0.1, kernel=rbf, score=(train=1.000, test=0.795) total time= 0.0s
[CV 5/5] END C=1000, gamma=0.1, kernel=rbf, score=(train=1.000, test=0.810) total time= 0.0s
[CV 1/5] END C=1000, gamma=0.1, kernel=poly, score=(train=1.000, test=0.772) total time= 0.0s
[CV 2/5] END C=1000, gamma=0.1, kernel=poly, score=(train=1.000, test=0.787) total time= 0.0s
[CV 3/5] END C=1000, gamma=0.1, kernel=poly, score=(train=1.000, test=0.795) total time= 0.0s
[CV 4/5] END C=1000, gamma=0.1, kernel=poly, score=(train=1.000, test=0.795) total time= 0.0s
[CV 5/5] END C=1000, gamma=0.1, kernel=poly, score=(train=1.000, test=0.794) total time= 0.0s
[CV 1/5] END C=1000, gamma=1, kernel=linear, score=(train=0.884, test=0.883) total time= 2.6s
[CV 2/5] END C=1000, gamma=1, kernel=linear, score=(train=0.850, test=0.898) total time= 2.9s
[CV 3/5] END C=1000, gamma=1, kernel=linear, score=(train=0.872, test=0.843) total time= 2.6s
[CV 4/5] END C=1000, gamma=1, kernel=linear, score=(train=0.862, test=0.874) total time= 3.3s
[CV 5/5] END C=1000, gamma=1, kernel=linear, score=(train=0.860, test=0.873) total time= 3.2s
[CV 1/5] END C=1000, gamma=1, kernel=rbf, score=(train=1.000, test=0.724) total time= 0.0s
[CV 2/5] END C=1000, gamma=1, kernel=rbf, score=(train=1.000, test=0.803) total time= 0.0s
[CV 3/5] END C=1000, gamma=1, kernel=rbf, score=(train=1.000, test=0.740) total time= 0.0s
[CV 4/5] END C=1000, gamma=1, kernel=rbf, score=(train=1.000, test=0.756) total time= 0.0s
[CV 5/5] END C=1000, gamma=1, kernel=rbf, score=(train=1.000, test=0.780) total time= 0.0s
[CV 1/5] END C=1000, gamma=1, kernel=poly, score=(train=1.000, test=0.772) total time= 0.0s
[CV 2/5] END C=1000, gamma=1, kernel=poly, score=(train=1.000, test=0.787) total time= 0.0s
[CV 3/5] END C=1000, gamma=1, kernel=poly, score=(train=1.000, test=0.795) total time= 0.0s
[CV 4/5] END C=1000, gamma=1, kernel=poly, score=(train=1.000, test=0.795) total time= 0.0s
[CV 5/5] END C=1000, gamma=1, kernel=poly, score=(train=1.000, test=0.802) total time= 0.0s

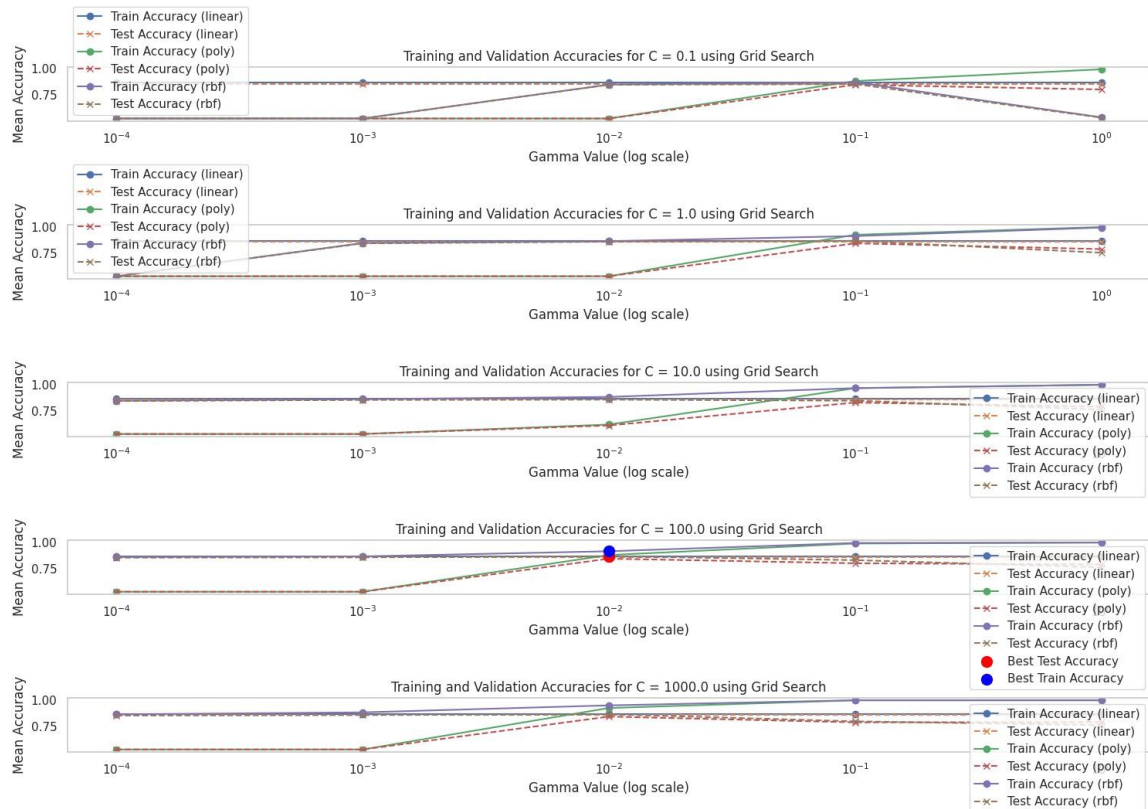
print("Best parameters found: ", grid.best_params_)
print("Best cross validation score: ", grid.best_score_)

Best parameters found: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
Best cross-validation score: 0.869116360454943
```

## Best parameters

**Best Parameters: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}**  
**Best Training Score: 0.9164049760052183**  
**Best Test Score: 0.869116360454943**

## Visualizing the results of hyperparameter tuning for an SVM classifier using GridSearchCV





## 5.2.2 Testing

Code:

Results:

```
# Evaluate on the validation set
print("Validation Metrics:")

# Recall, Precision, F1-Score for validation
SVM_recall_val = recall_score(y_val, y_val_pred_SVM, average='binary')
SVM_precision_val = precision_score(y_val, y_val_pred_SVM, average='binary')
SVM_f1_val = f1_score(y_val, y_val_pred_SVM, average='binary')

# Print validation metrics
print(f"Recall: {SVM_recall_val:.2f}")
print(f"Precision: {SVM_precision_val:.2f}")
print(f"F1-score: {SVM_f1_val:.2f}")

# Evaluate on the test set
print("\nTest Metrics:")

# Recall, Precision, F1-Score for test
SVM_recall_test = recall_score(y_test, y_test_pred_SVM, average='binary')
SVM_precision_test = precision_score(y_test, y_test_pred_SVM, average='binary')
SVM_f1_test = f1_score(y_test, y_test_pred_SVM, average='binary')

# Print test metrics
print(f"Recall: {SVM_recall_test:.2f}")
print(f"Precision: {SVM_precision_test:.2f}")
print(f"F1-score: {SVM_f1_test:.2f}")
```

```
Validation Metrics:
Recall: 0.87
Precision: 0.85
F1-score: 0.86

Test Metrics:
Recall: 0.91
Precision: 0.90
F1-score: 0.91
```

## 5.3 KNN

The K-Nearest Neighbors (KNN) classifier is a simple and intuitive supervised learning algorithm used for classification tasks. It works by identifying the K closest data points (neighbors) to a given input based on a distance metric, such as Euclidean distance, and assigning the majority class label among those neighbors to the input. KNN is a non-parametric and instance-based algorithm, meaning it does not assume any specific distribution for the data and makes predictions directly from the training data.

### 5.3.1 Hyperparameter Tuning Parameter Grid:

- `n_neighbors`: Number of neighbors to consider for making a prediction. Common choices are odd numbers (e.g., 3, 5, 7, 9, 11) to avoid ties.
- `weights`: How to assign weights to the neighbors:
  - `'uniform'`: All neighbors have equal weight.
  - `'distance'`: Closer neighbors have higher weight.
- `algorithm`: The algorithm used to compute nearest neighbors:
  - `'auto'`: Automatically chooses the best algorithm based on the dataset.
  - `'ball_tree'`, `'kd_tree'`, `'brute'`: Specific algorithms for different types of data.
- `leaf_size`: A hyperparameter that affects the speed and memory efficiency of tree-based algorithms (e.g., `'ball_tree'` or `'kd_tree'`).
- `p`: The power parameter for the Minkowski distance (used when `p=1` for Manhattan distance, `p=2` for Euclidean distance).



```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

KNN_Classifier = KNeighborsClassifier()

param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [20, 30, 40, 50],
    'p': [1, 2]
}

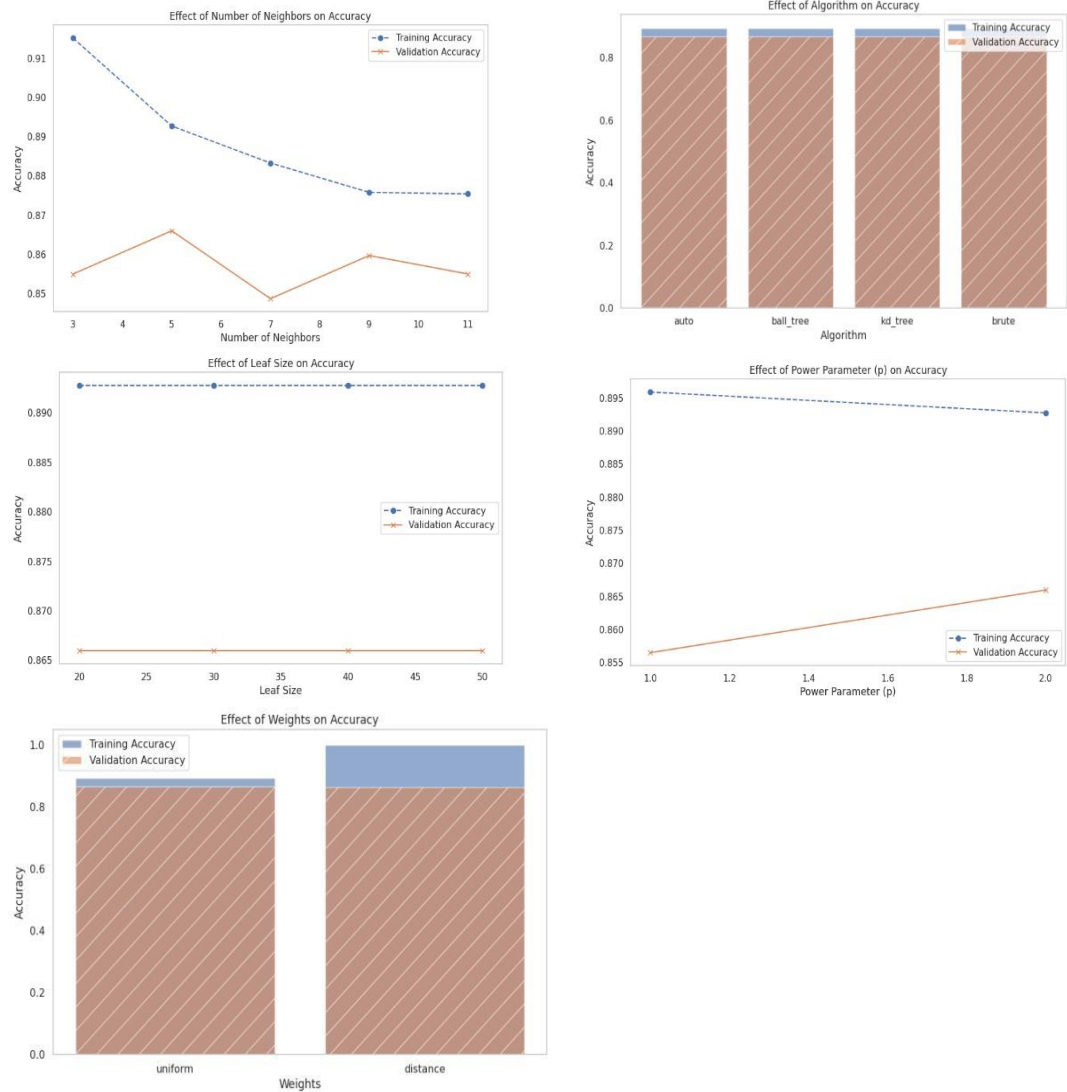
# Perform Grid Search
grid_search = GridSearchCV(KNN_Classifier, param_grid, cv=5, scoring='accuracy', return_train_score=True)
grid_search.fit(X_train, y_train)

# Extract results
results = grid_search.cv_results_
results_df = pd.DataFrame(results)
print(results_df)

# Extract the best parameters
best_params = grid_search.best_params_
print(f"Best hyperparameters: {best_params}")

```

**Visualizing the results of hyperparameter tuning for a KNN classifier using GridSearchCV**



## 5.2.1 Testing

Code:

```
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix

KNN_best_model = grid_search.best_estimator_
KNN_y_val_pred = KNN_best_model.predict(X_val)
KNN_y_test_pred = KNN_best_model.predict(X_test)

# Evaluate on the validation set
print("Validation Metrics:")

# Recall, Precision, F1-Score for validation
KNN_recall_val = recall_score(y_val, KNN_y_val_pred, average='binary')
KNN_precision_val = precision_score(y_val, KNN_y_val_pred, average='binary')
KNN_f1_val = f1_score(y_val, KNN_y_val_pred, average='binary')

# Print validation metrics
print(f"Recall: {KNN_recall_val:.2f}")
print(f"Precision: {KNN_precision_val:.2f}")
print(f"F1-score: {KNN_f1_val:.2f}")

# Evaluate on the test set
print("\nTest Metrics:")

# Recall, Precision, F1-Score for test
KNN_recall_test = recall_score(y_test, KNN_y_test_pred, average='binary')
KNN_precision_test = precision_score(y_test, KNN_y_test_pred, average='binary')
KNN_f1_test = f1_score(y_test, KNN_y_test_pred, average='binary')
```

Results:

**Validation Metrics:**  
Recall: 0.84  
Precision: 0.87  
F1-score: 0.86

**Test Metrics:**  
Recall: 0.93  
Precision: 0.91  
F1-score: 0.92

## 5.4 Decision Trees

A Decision Tree Classifier is a supervised machine learning algorithm used for classification tasks. It works by recursively splitting the dataset into subsets based on the feature that provides the best separation of the classes. Each node in the tree represents a feature, and each branch represents a decision rule based on that feature. The leaves of the tree represent the final class labels.

### 5.4.1 Hyperparameter Tuning Parameter Grid:

- **max\_depth:** [4, 5, 6, 7, 8]  
Specifies the maximum depth of the decision tree, i.e., the longest path from the root node to a leaf. The values range from 4 to 8, and deeper trees may capture more complex patterns but are more likely to overfit the training data.
- **min\_samples\_split:** [10, 15, 19, 25]  
Sets the minimum number of samples required to split an internal node. Larger values ensure that each split contains enough data, which can help prevent the model from becoming too complex and overfitting. For instance, if min\_samples\_split is set to 25, a node must have at least 25 samples to be split further.
- **min\_samples\_leaf:** [10, 20, 35]  
Defines the minimum number of samples that must be in a leaf node. Larger values ensure that the leaf nodes contain sufficient data, preventing the tree from creating overly specific splits that could lead to overfitting. Values like 10, 20, and 35 control how fine-grained the final leaf nodes can be.
- **max\_leaf\_nodes:** [5, 6, 7]  
Limits the number of leaf nodes in the tree. By setting this hyperparameter, the model is constrained to have a smaller, more manageable tree. This helps prevent overfitting by ensuring the tree doesn't grow too large. Values like 5, 6, and 7 limit the complexity of the tree.
- **max\_features:** ['log2', 'sqrt', None]  
Specifies the number of features to consider when looking for the best split:
  - 'log2': Uses the logarithm of the number of features ( $\log_2$ ), so only a small number of features are considered at each split.
  - 'sqrt': Uses the square root of the number of features, a common practice to introduce randomness and reduce overfitting in tree-based models.
  - None: Considers all available features when finding the best split, which can lead to a more overfitted model but may also improve accuracy if tuned well.

```

from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

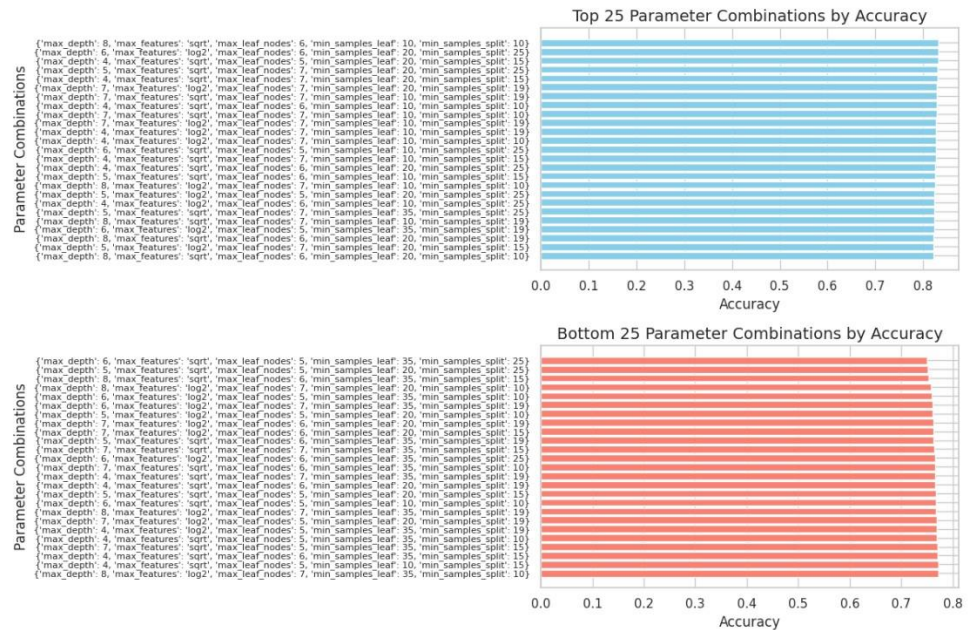
param_grid = {
    'max_depth': [4, 5, 6, 7, 8],
    'min_samples_split': [10, 15, 19, 25],
    'min_samples_leaf': [10, 20, 35],
    'max_leaf_nodes': [5, 6, 7],
    'max_features': ['log2', 'sqrt', None]
}

DT = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator=DT, param_grid=param_grid, cv=5, scoring='accuracy', verbose=1)
grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)

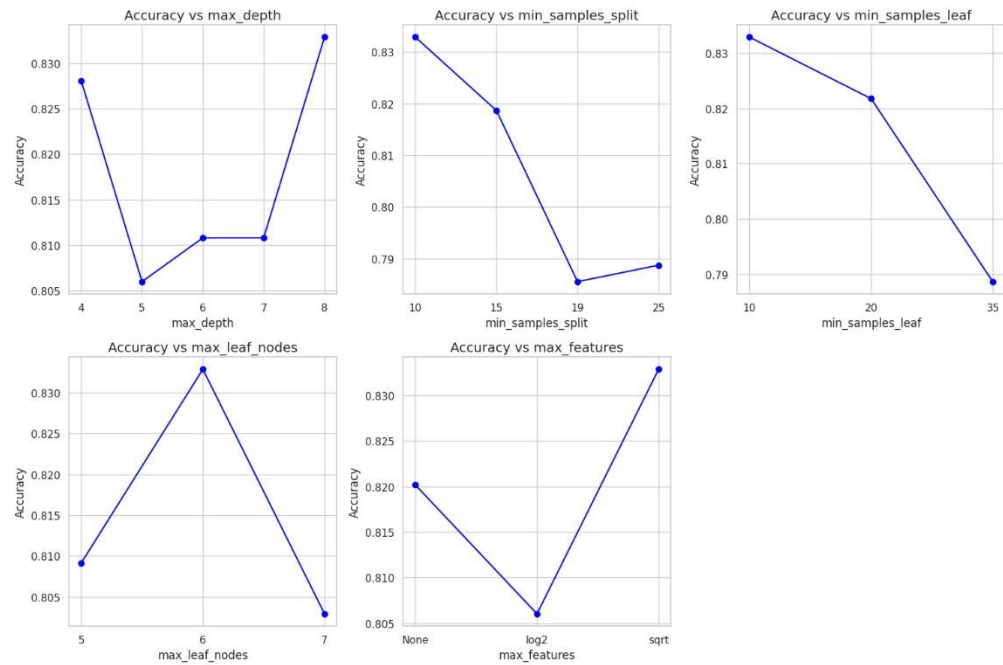
Fitting 5 folds for each of 540 candidates, totalling 2700 fits
Best Parameters: {'max_depth': 8, 'max_features': 'sqrt', 'max_leaf_nodes': 6, 'min_samples_leaf': 10, 'min_samples_split': 10}

```

Extraction of best and worst 25 combinations:



Plotting Accuracy variation for each parameter



## 5.4.2 Testing

Code:

```
DT_best_model = grid_search.best_estimator_
DT_y_val_pred = DT_best_model.predict(X_val)
DT_y_test_pred = DT_best_model.predict(X_test)

# Evaluate on the validation set
print("Validation Metrics:")

# Recall, Precision, F1-Score for validation
DT_recall_val = recall_score(y_val, DT_y_val_pred, average='binary')
DT_precision_val = precision_score(y_val, DT_y_val_pred, average='binary')
DT_f1_val = f1_score(y_val, DT_y_val_pred, average='binary')

# Print validation metrics
print(f"Recall: {DT_recall_val:.2f}")
print(f"Precision: {DT_precision_val:.2f}")
print(f"F1-score: {DT_f1_val:.2f}")

# Evaluate on the test set
print("\nTest Metrics:")

# Recall, Precision, F1-Score for test
DT_recall_test = recall_score(y_test, DT_y_test_pred, average='binary')
DT_precision_test = precision_score(y_test, DT_y_test_pred, average='binary')
DT_f1_test = f1_score(y_test, DT_y_test_pred, average='binary')

# Print test metrics
print(f"Recall: {DT_recall_test:.2f}")
print(f"Precision: {DT_precision_test:.2f}")
print(f"F1-score: {DT_f1_test:.2f}")
```

Results:

```
Validation Metrics:  
Recall: 0.66  
Precision: 0.95  
F1-score: 0.78  
  
Test Metrics:  
Recall: 0.74  
Precision: 0.92  
F1-score: 0.82
```

## 6. Histogram

This code compares the performance of four machine learning models (KNN, Naïve Bayes, SVM, and Decision Tree) by evaluating their precision, recall, and F1-score, and visualizing their confusion matrices. It first calculates the confusion matrices for each model using `confusion_matrix`, then converts them to percentages for easier interpretation with the `confusion_matrix_percentage` function. The confusion matrices are plotted as heatmaps with the percentage values for each model. Finally, the precision, recall, and F1-scores of each model are compared using a bar chart, showing how each model performs across these metrics.

### 6.1 Code:

```
def confusion_matrix_percentage(conf_matrix):  
    return conf_matrix / conf_matrix.sum(axis=1, keepdims=True) * 100  
  
# ---- Evaluate Metrics ---- #  
models = ["KNN", "Naïve Bayes", "SVM", "Decision Tree"]  
precision = [  
    KNN_precision_test,  
    nb_precision_test,  
    SVM_precision_test,  
    DT_precision_test  
]  
recall = [  
    KNN_recall_test,  
    nb_recall_test,  
    SVM_recall_test,  
    DT_recall_test  
]  
f1 = [  
    KNN_f1_test,  
    nb_f1_test,  
    SVM_f1_test,  
    DT_f1_test  
]  
  
# ---- Confusion Matrices ---- #  
knn_conf_matrix = confusion_matrix(y_test, KNN_y_test_pred)  
nb_conf_matrix = confusion_matrix(y_test, nb_y_test_pred)  
svm_conf_matrix = confusion_matrix(y_test, y_test_pred_SVM)  
dt_conf_matrix = confusion_matrix(y_test, DT_y_test_pred)
```



```

# ---- Convert Confusion Matrices to Percentages ---- #
knn_conf_matrix_percent = confusion_matrix_percentage(knn_conf_matrix)
nb_conf_matrix_percent = confusion_matrix_percentage(nb_conf_matrix)
svm_conf_matrix_percent = confusion_matrix_percentage(svm_conf_matrix)
dt_conf_matrix_percent = confusion_matrix_percentage(dt_conf_matrix)

# ---- Plot Confusion Matrices ---- #
plt.figure(figsize=(18, 6))

# KNN Confusion Matrix (Percentage)
plt.subplot(1, 4, 1)
sns.heatmap(knn_conf_matrix_percent, annot=True, fmt=".2f", cmap="Oranges")
plt.title("KNN Confusion Matrix (Percentage)")
plt.xlabel("Predicted")
plt.ylabel("Actual")

# Naïve Bayes Confusion Matrix (Percentage)
plt.subplot(1, 4, 2)
sns.heatmap(nb_conf_matrix_percent, annot=True, fmt=".2f", cmap="Blues")
plt.title("Naïve Bayes Confusion Matrix (Percentage)")
plt.xlabel("Predicted")
plt.ylabel("Actual")

# SVM Confusion Matrix (Percentage)
plt.subplot(1, 4, 3)
sns.heatmap(svm_conf_matrix_percent, annot=True, fmt=".2f", cmap="Greens")
plt.title("SVM Confusion Matrix (Percentage)")
plt.xlabel("Predicted")
plt.ylabel("Actual")

```

```

# Decision Tree Confusion Matrix (Percentage)
plt.subplot(1, 4, 4)
sns.heatmap(dt_conf_matrix_percent, annot=True, fmt=".2f", cmap="Purples")
plt.title("Decision Tree Confusion Matrix (Percentage)")
plt.xlabel("Predicted")
plt.ylabel("Actual")

plt.tight_layout()
plt.show()

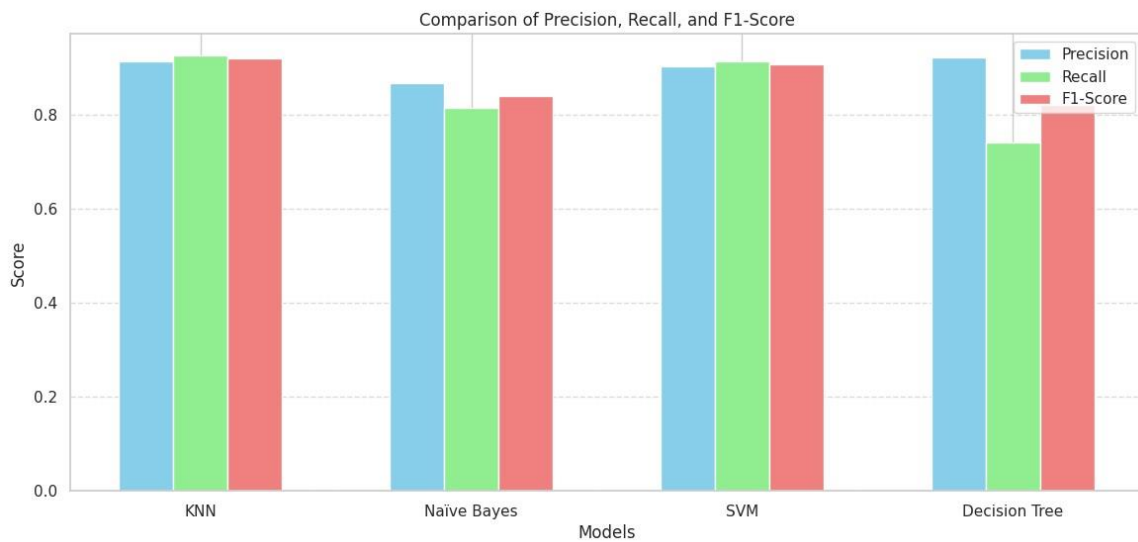
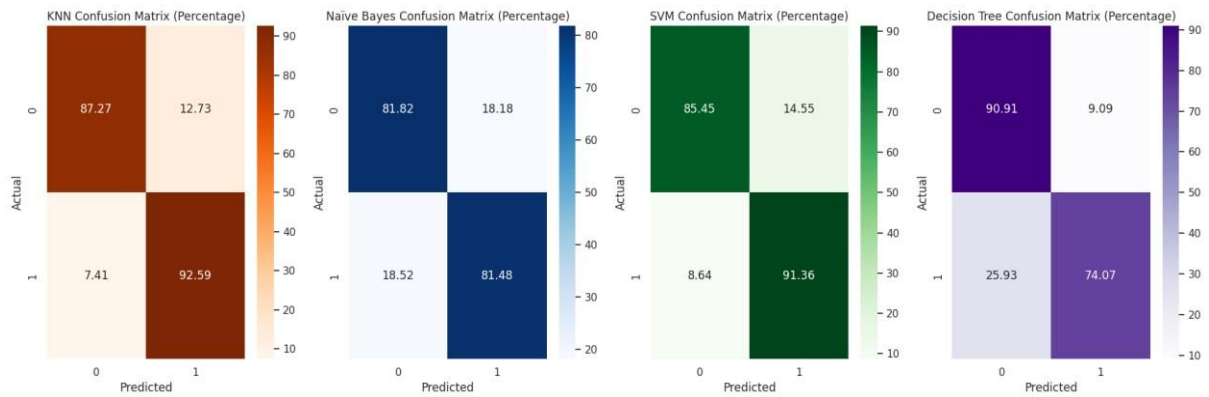
# ---- Compare Metrics in Bar Chart ---- #
x = np.arange(len(models)) # Model indices

plt.figure(figsize=(14, 6))
plt.bar(x - 0.2, precision, width=0.2, label="Precision", color="skyblue")
plt.bar(x, recall, width=0.2, label="Recall", color="lightgreen")
plt.bar(x + 0.2, f1, width=0.2, label="F1-Score", color="lightcoral")

plt.xticks(x, models)
plt.xlabel("Models")
plt.ylabel("Score")
plt.title("Comparison of Precision, Recall, and F1-Score")
plt.legend()
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()

```

## 6.2 Results:



## 7. Dendrogram

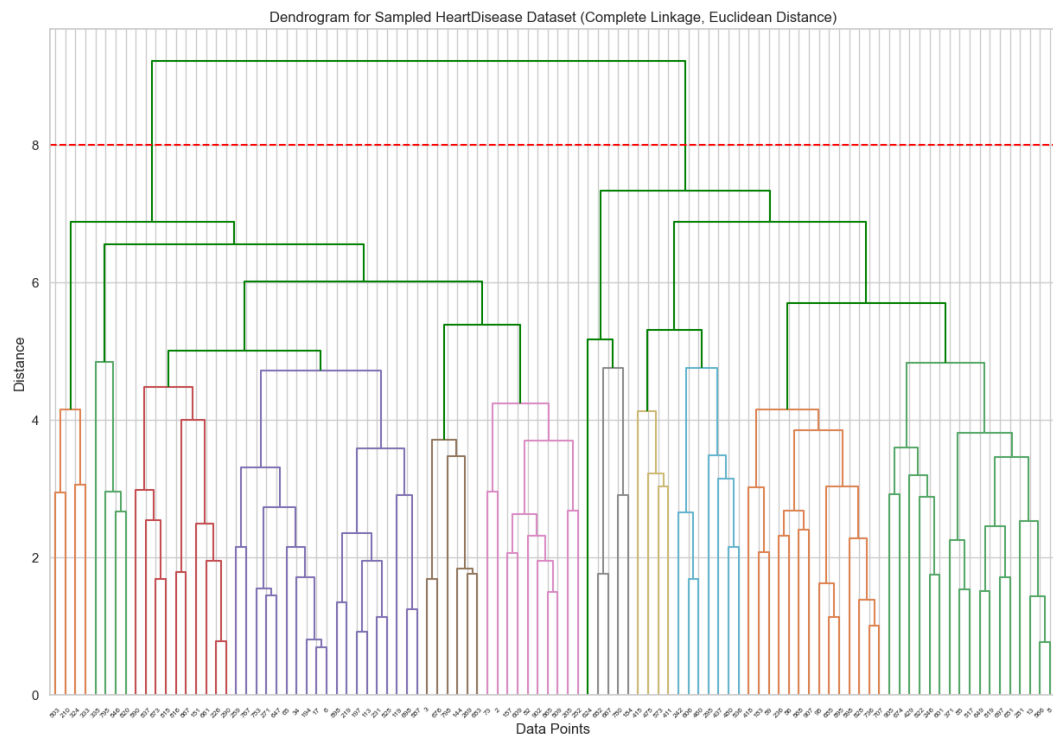
### 7.1 Analysis

We utilized hierarchical clustering to visualize the relationships among data points in the Heart Failure dataset through dendrograms.

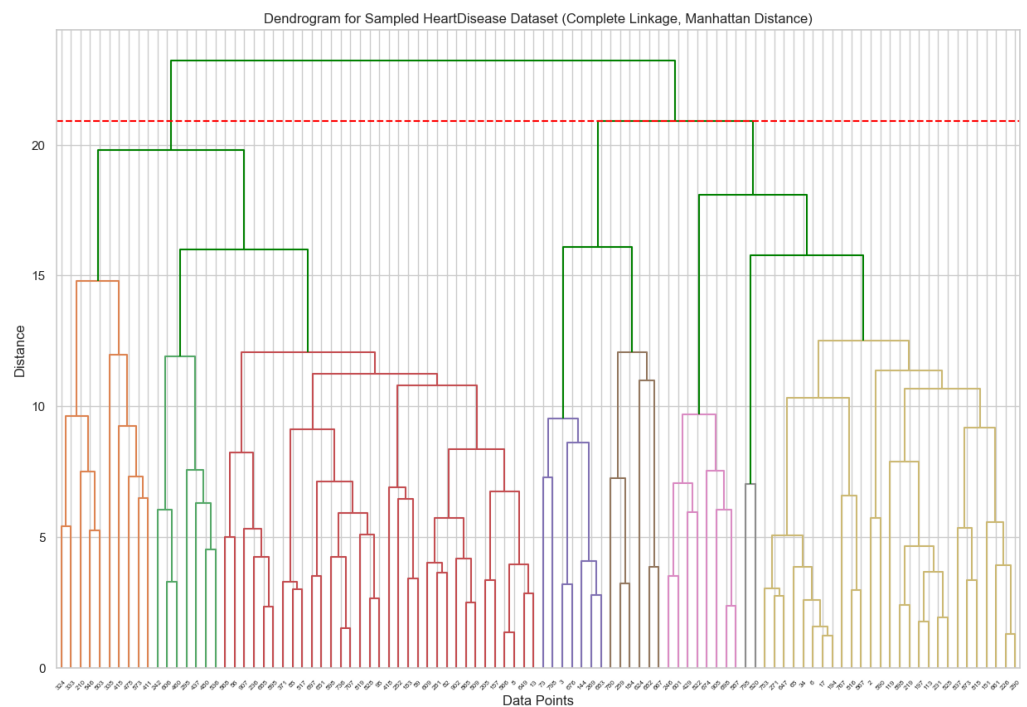
- A) **Sampling Data:** A random sample of 100 data points was selected from the dataset to facilitate analysis.
- B) **Complete Linkage with Euclidean Distance:** The first dendrogram was generated using complete linkage and the Euclidean distance metric. This



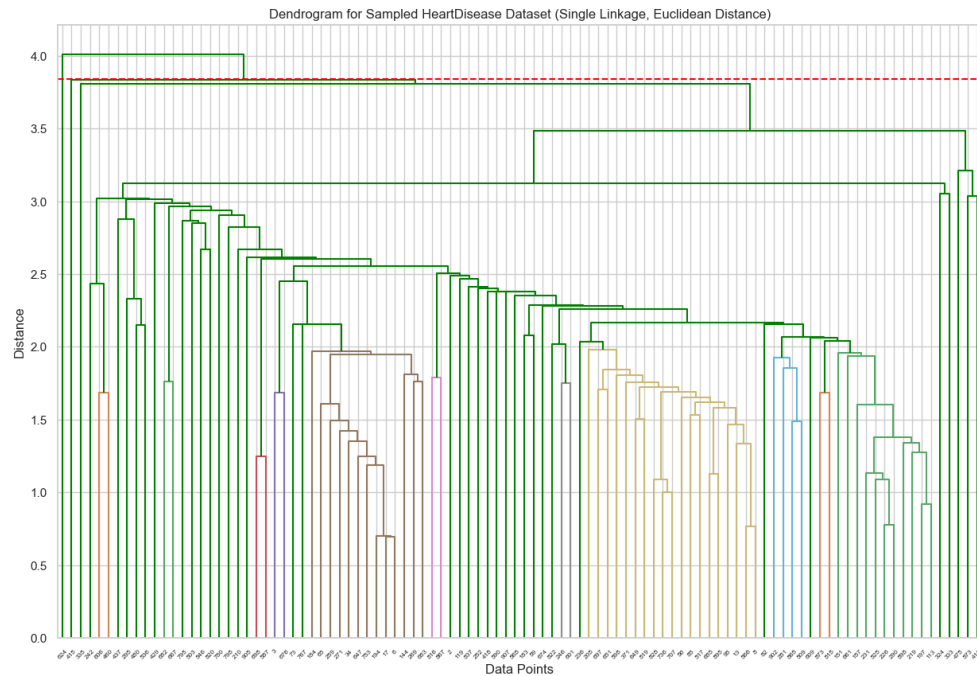
method helps in understanding the cluster formation based on the maximum distance between clusters.



- C) **Complete Linkage with Manhattan Distance:** The second dendrogram was created using complete linkage with the Manhattan distance (city block metric), providing an alternative view of the data clustering.



- D) **Single Linkage with Euclidean Distance:** A third dendrogram was plotted using single linkage with Euclidean distance. This method links clusters based on the closest points, offering a different perspective on the clustering structure.



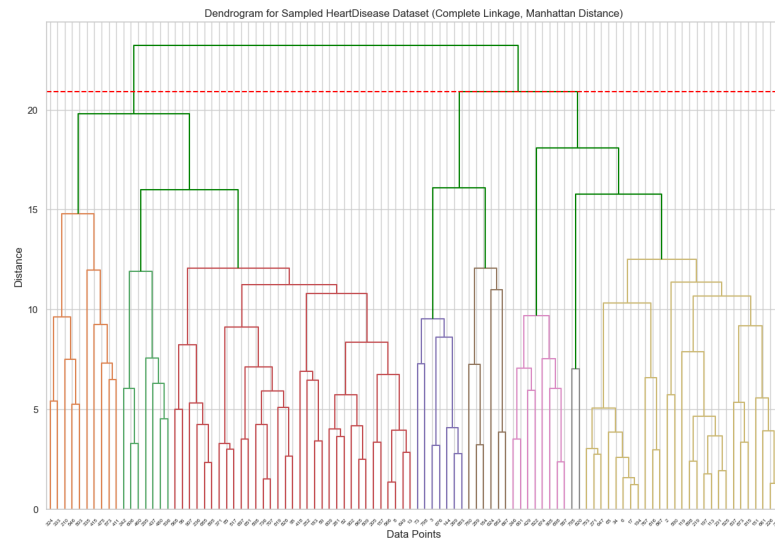
- E) **Single Linkage with Manhattan Distance:** Finally, a dendrogram was constructed using single linkage with the Manhattan distance metric to further explore the clustering relationships.



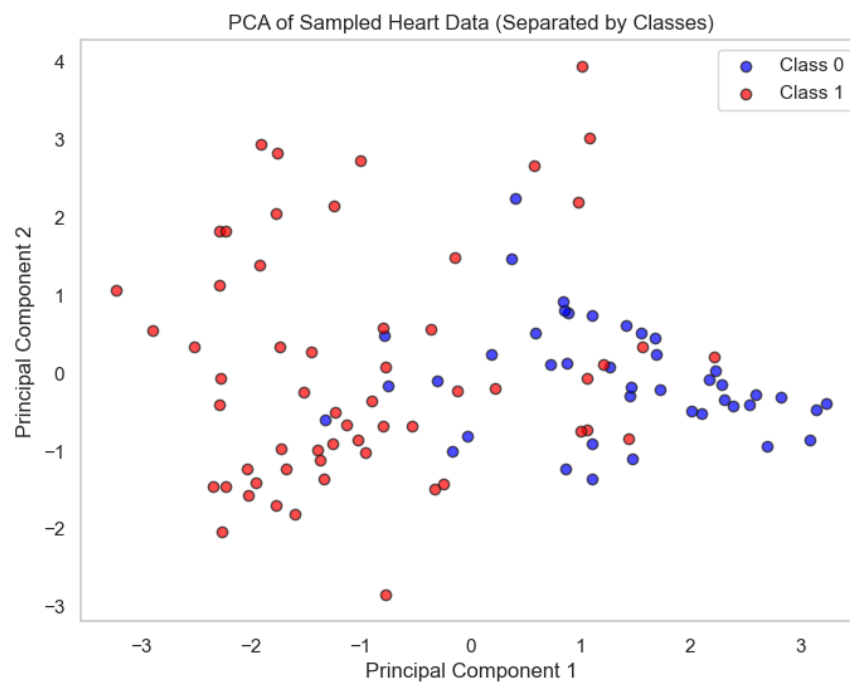
Each dendrogram provides insights into how data points are clustered, highlighting the distance between clusters and the potential grouping of similar observations.

## 7.2 Comparison

- **Hierarchical Clustering**



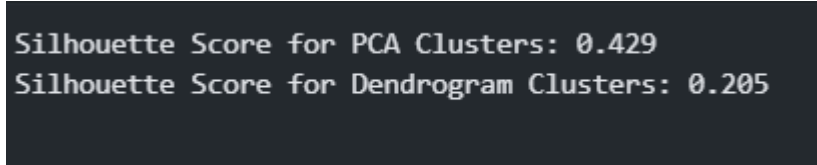
- **Principal Component Analysis (PCA)**



In this analysis, we compared the effectiveness of clustering methods applied to a heart disease dataset using two approaches: PCA (Principal Component Analysis) and hierarchical clustering via a dendrogram.

**Silhouette Score for PCA Clusters:** The Silhouette Score was **0.429**, indicating moderate separation between clusters. This suggests that PCA effectively captured significant variance in the dataset, allowing for well-defined clusters.

**Silhouette Score for Dendrogram Clusters:** The Silhouette Score for the dendrogram clusters was **0.205**, reflecting poor cluster separation. This indicates that the hierarchical clustering approach used, particularly with the single linkage and Manhattan distance metrics, resulted in overlapping clusters.



```
Silhouette Score for PCA Clusters: 0.429
Silhouette Score for Dendrogram Clusters: 0.205
```

The comparison of Silhouette Scores demonstrates that the PCA method provided better-defined clusters compared to the hierarchical clustering approach. This analysis highlights the importance of selecting appropriate clustering techniques and dimensionality reduction methods to accurately capture the underlying structure of the data.

## 8. Conclusion

In this report, we successfully analyzed the Heart Failure Prediction Dataset to develop predictive models that aid in identifying patients at risk for heart disease. By employing various machine learning classifiers, including Naïve Bayes, Support Vector Machines (SVM), K-Nearest Neighbors (KNN), and Decision Trees.

Our exploratory data analysis, complemented by visualizations such as PCA and outlier detection, provided valuable insights into the relationships between patient attributes and heart disease occurrence. The application of machine learning techniques allowed us to identify significant patterns, ultimately enhancing the predictive capabilities of our models.

Overall, this project highlights how data analysis can support better healthcare decisions and improve patient care.