Final Report

Team Members

Name	ID
Nancy Amro Hussein Mansour	2101421
Mennatalla Mohamed Samir Abdelatif Ezzelarab	2100497
Sara Mohamed Ashour Hussein	21P0337

Data Exploration and visualization

1. Libraries and Data Import

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA
import seaborn as sns
import numpy as np
from google.colab import files
```

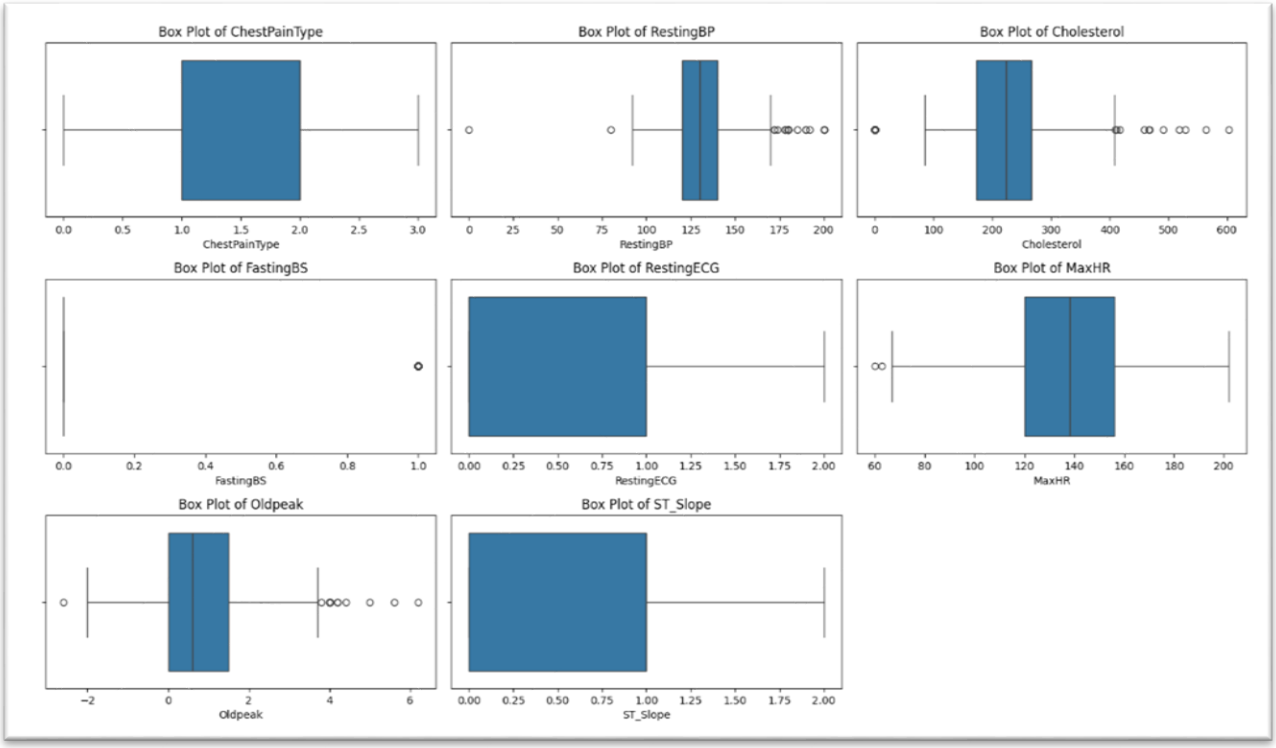
```
uploaded = files.upload()
df = pd.read_csv("heart.csv")  # Replace 'heart.c
print(df)
```

```
     Age Sex ChestPainType  RestingBP  Cholesterol  FastingBS RestingECG  \
0     40   M           ATA        140          289          0     Normal
1     49   F           NAP        160          180          0     Normal
2     37   M           ATA        130          283          0         ST
3     48   F           ASY        138          214          0     Normal
4     54   M           NAP        150          195          0     Normal
..   ...  ..           ...        ...          ...        ...        ...
913   45   M            TA        110          264          0     Normal
914   68   M           ASY        144          193          1     Normal
915   57   M           ASY        130          131          0     Normal
916   57   F           ATA        130          236          0        LVH
917   38   M           NAP        138          175          0     Normal

     MaxHR ExerciseAngina  Oldpeak ST_Slope  HeartDisease
0      172              N      0.0       Up             0
1      156              N      1.0     Flat             1
2       98              N      0.0       Up             0
3      108              Y      1.5     Flat             1
4      122              N      0.0       Up             0
..     ...            ...      ...      ...           ...
913    132              N      1.2     Flat             1
914    141              N      3.4     Flat             1
915    115              Y      1.2     Flat             1
916    174              N      0.0     Flat             1
917    173              N      0.0       Up             0

[918 rows x 12 columns]
```

2. Box Plot Visualization

As we can see, by using boxplots, we could visualize the outliers of each feature according to the dataset.

```
columns = [ 'ChestPainType', 'RestingBP', 'Cholesterol', 'FastingBS',
           'RestingECG', 'MaxHR', 'Oldpeak', 'ST_Slope']

# Create a box plot for each column
plt.figure(figsize=(16, 12))
for i, col in enumerate(columns, 1):
    plt.subplot(4, 3, i)  # 4 rows, 3 columns
    sns.boxplot(data=df, x=col)
    plt.title(f'Box Plot of {col}')
    plt.tight_layout()

plt.show()
```



3. Outlier Detection

This is a function that calculates the first and third quartiles based on the dataset. While using these quartiles we'll be able to calculate the interquartile range. Proceeding to calculating the lower and upper bound

As we can see below, the function of detecting the outliers detected that

```
def identify_outliers(df, columns):
    outliers = {}
    for col in columns:
        Q1 = df[col].quantile(0.25)  # First quartile
        Q3 = df[col].quantile(0.75)  # Third quartile
        IQR = Q3 - Q1  # Interquartile range
        lower_bound = Q1 - 1.5 * IQR  # Lower bound for outliers
        upper_bound = Q3 + 1.5 * IQR  # Upper bound for outliers

        # Identify outliers
        outliers[col] = df[(df[col] < lower_bound) | (df[col] > upper_bound)]

    return outliers

# Identify outliers for each feature
outliers = identify_outliers(df, columns)

# Print the outliers for each column
for col, outlier_data in outliers.items():
    print(f"Outliers in {col}:")
    print(outlier_data[[col]])  # Print only the outlier values for the column
    print("\n")
```

ChestPainType, RestingECG, and St_Slope have no outliers. While RestingBP, Cholestrol, Fasting BS, Max_HR, and OldPeak have several outliers.

	RestingBP
109	190
123	180
189	180
190	180
241	200
274	180
275	180
278	180
314	80
365	200
372	185
399	200
411	180
423	180
449	0
475	178
550	172
585	180
592	190
673	174
702	178
725	180
732	200
759	192
774	178
780	180
855	180
880	172

Outliers in Oldpeak:

	Oldpeak
68	4.0
166	5.0
324	-2.6
500	4.0
521	4.0
537	4.0
559	4.0
624	4.0
702	4.2
732	4.0
771	5.6
775	3.8
791	4.2
850	6.2
900	4.4
908	4.0

Outliers in ST_Slope:

Empty DataFrame

Columns: [ST_Slope]

Index: []

```
Outliers in FastingBS:
     FastingBS
36           1
38           1
52           1
84           1
86           1
..         ...
887          1
888          1
901          1
911          1
914          1

[214 rows x 1 columns]


Outliers in RestingECG:
Empty DataFrame
Columns: [RestingECG]
Index: []


Outliers in MaxHR:
     MaxHR
370     63
390     60
```

```
Outliers in Cholesterol:
     Cholesterol
28           468
30           518
69           412
76           529
103          466
..           ...
535            0
536            0
616          564
667          417
796          409

[183 rows x 1 columns]
```

```
Outliers in Cholesterol:
     Cholesterol
28           468
30           518
69           412
76           529
103          466
..           ...
535            0
536            0
616          564
667          417
796          409

[183 rows x 1 columns]
```

4. Data Preprocessing

We are encoding the categorical data which contains non-numeric values. Each feature will be processed in a different way according to the data being contained in each feature in the dataset.

We're checking that this was applied by viewing the first 10 rows of data.

Then we separated the target feature we are looking for, which is “HeartDisease”, from the rest of the data so that HeartDisease is the y feature, and the remaining data is the x feature. To clarify this, HeartDisease is the dependent variable (y) that will be determined by the remaining features, independent variable (x).

A scaler standardizes the feature data, meaning it transforms the features, so they have a mean of 0 and a standard deviation of 1. This is important because many machine learning algorithms perform better when the data is standardized, particularly those that rely on distances like KNN and SVM.

We used “fit” to calculate the mean and standard deviation and we used to transform to standardize the data to have a mean of 0 and scaled to have a standard deviation of 1.

```
df.Sex.replace({'F':0, 'M':1}, inplace =True)
df.ChestPainType.replace({'ATA':0, 'NAP':1, 'ASY':2, 'TA':3}, inplace =True)
df.RestingECG.replace({'Normal':0, 'ST':1, 'LVH':2}, inplace =True)
df.ExerciseAngina.replace({'N':0, 'Y':1}, inplace =True)
df.ST_Slope.replace({'Up':0, 'Flat':1, 'Down':2}, inplace =True)

df.head(10)


# Separate the target variable (HeartDisease) from the feature variables
X = df.drop(columns=['HeartDisease'])
y = df['HeartDisease']

# Standardize the feature data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

5. Principal Component Analysis (PCA)

The `n_components=2` argument specifies that we want to reduce the dataset to 2 principal components, which will be used for visualization in a 2D plot.

- fit_transform(X_scaled)

This method first "fits" the PCA model to the standardized data (`X_scaled`) by calculating the principal components, and then "transforms" the data into the new coordinate system defined by the principal components.

The result (`X_pca`) is a matrix where each row corresponds to an observation in the original dataset, but now represented by the two principal components (PC1 and PC2) instead of the original features.

Then we calculated the covariance.

The below code snippet shows the proportion of the total variance in the data that is explained by each of the principal components.

```
# Display explained variance ratio
explained_variance = pca.explained_variance_ratio_
print("Explained Variance Ratio:", explained_variance)
```

The whole code written to tackle the PCA calculation and visualization:

```
# Initialize PCA with 2 components (2D projection)
pca = PCA(n_components=2)


# Fit and transform the standardized data
X_pca = pca.fit_transform(X_scaled)
pca.get_covariance()

explained_variance=pca.explained_variance_ratio_
explained_variance
# Convert the result into a DataFrame for easy handling
pca_df = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])
pca_df['HeartDisease'] = y  # Adding the target variable for visualization

plt.figure(figsize=(8, 6))
sns.scatterplot(x='PC1', y='PC2', hue='HeartDisease', data=pca_df, palette='coolwarm', alpha=0.7)
plt.title('2D PCA of Heart Disease Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Heart Disease', loc='best')
plt.show()
```

Explained Variance:

```
# Print the explained variance ratio of the components
print("Explained Variance Ratio:", pca.explained_variance_ratio_)
```

Explained Variance Ratio: [0.2619465 0.13093813]

Let's break the above output to understand what we got. As we explained earlier, we knew that we use the explained_variance_ratio to determine the total variance captured but both PCAs. So. we can see that:

PCA1 has variance = 0. 2619465

PCA2 has variance = 0.13093813

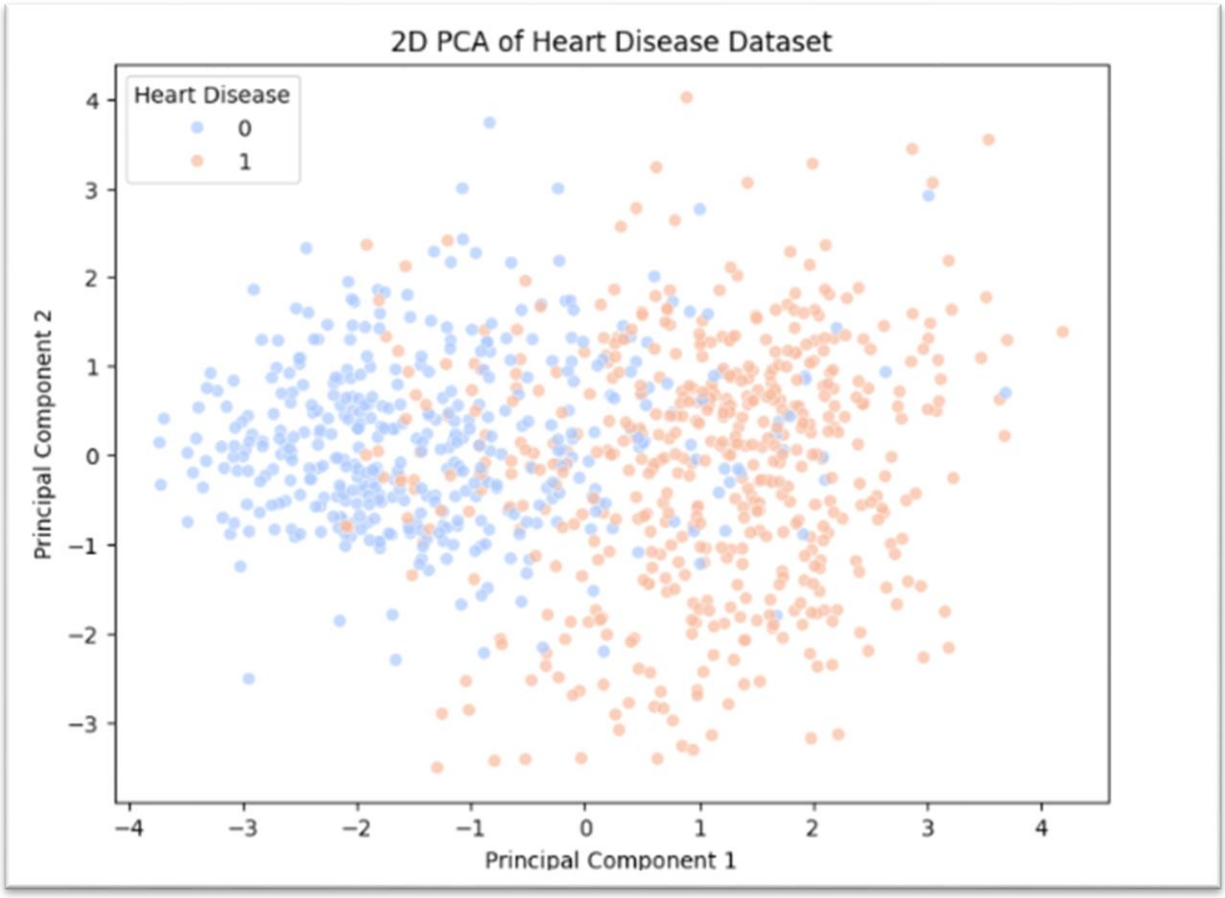
This can tell us that PCA1 captures **26.19%** of the variance in the original dataset. While PCA2 captures **13.0938%** of the variance. This indicates that PCA1 is the direction in the feature space that has the most variance or the most information about how the data points vary. PCA2 accounts for less variability compared to PCA1 but still adds some important information.

Overall Variance Explained:

- Together, **PC1 and PC2** explain about **40.28%** of the total variance in the dataset (26.19% + 13.09% = 40.28%).
- This means that the two principal components combined only capture about **40%** of the variability in the original data. While this is a substantial amount, it also indicates that more components (beyond just PC1 and PC2) may be necessary to capture the remaining 60% of the variance in the dataset.

Some proposed solutions to overcome this problem include either collecting more data or adding new features, like smoking habits, diet, or any genetic data, that would affect the overall variance of the dataset.

This is the graph plotting the PCA1 vs PCA2 (without removing the outliers yet)

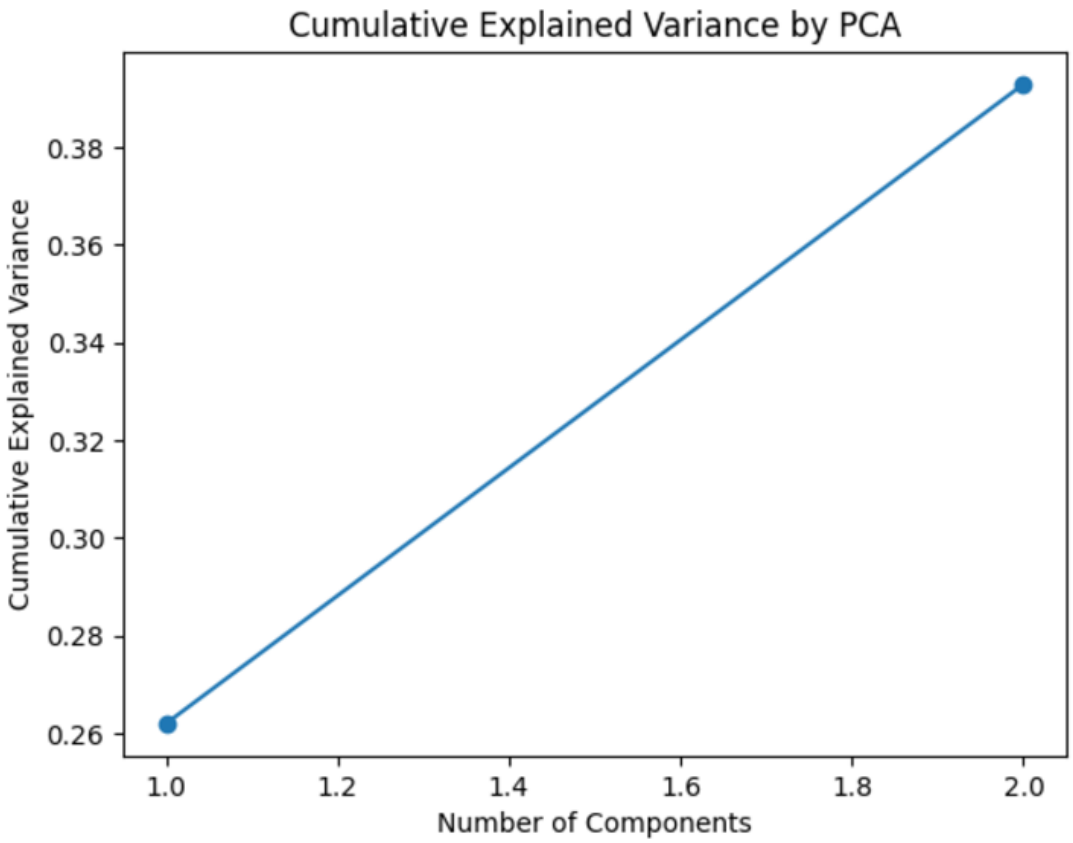


7.Cumulative Explained Variance

This shows the cumulative sum of both PCA1 and PCA2 variances, which tells us how much of the total variance is explained by the first 2 principal components. By plotting the cumulative variance, we can make an informed decision about how many components to retain, based on how much variance we want to capture

```
# Plot cumulative explained variance
explained_variance = np.cumsum(pca.explained_variance_ratio_)
plt.plot(range(1, len(explained_variance) + 1), explained_variance, marker='o')
plt.title('Cumulative Explained Variance by PCA')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.show()
```

```
Explained Variance Ratio: [0.2619465  0.13093813]
```



Data cleaning and processing

In the previous section, we determined how the non-numeric (categorical data) was processed so the model would be able to deal with them.

```
# Encode categorical features to numeric
df.Sex.replace({'F': 0, 'M': 1}, inplace=True)
df.ChestPainType.replace({'ATA': 0, 'NAP': 1, 'ASY': 2, 'TA': 3}, inplace=True)
df.RestingECG.replace({'Normal': 0, 'ST': 1, 'LVH': 2}, inplace=True)
df.ExerciseAngina.replace({'N': 0, 'Y': 1}, inplace=True)
df.ST_Slope.replace({'Up': 0, 'Flat': 1, 'Down': 2}, inplace=True)
```

In the code below, we will check for null values and data types.

```
# Check for null values
print(df.isnull().sum())

# Print data types of columns
print(df.dtypes)

# Dataset information and summary statistics
df.info()
print(df.describe())
```

```
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Age             918 non-null    int64
 1   Sex             918 non-null    object
 2   ChestPainType   918 non-null    object
 3   RestingBP       918 non-null    int64
 4   Cholesterol     918 non-null    int64
 5   FastingBS       918 non-null    int64
 6   RestingECG      918 non-null    object
 7   MaxHR           918 non-null    int64
 8   ExerciseAngina  918 non-null    object
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object
 11  HeartDisease    918 non-null    int64
```

Additionally, df.describe() shows the mean, standard deviation, minimum values, maximum values

```
memory usage: 86.2+ KB
              Age   RestingBP  Cholesterol   FastingBS       MaxHR  \
count  918.000000  918.000000   918.000000  918.000000  918.000000
mean    53.510893  132.396514   198.799564    0.233115  136.809368
std      9.432617   18.514154   109.384145    0.423046   25.460334
min     28.000000    0.000000     0.000000    0.000000   60.000000
25%     47.000000  120.000000   173.250000    0.000000  120.000000
50%     54.000000  130.000000   223.000000    0.000000  138.000000
75%     60.000000  140.000000   267.000000    0.000000  156.000000
max     77.000000  200.000000   603.000000    1.000000  202.000000

          Oldpeak  HeartDisease
count  918.000000    918.000000
mean     0.887364      0.553377
std      1.066570      0.497414
min     -2.600000      0.000000
25%      0.000000      0.000000
50%      0.600000      1.000000
75%      1.500000      1.000000
max      6.200000      1.000000
```

We also separated the target variable from the remaining variables

```
# Separate the target variable from the features
X = df.drop(columns=['HeartDisease'])
y = df['HeartDisease']
```

We standardized the data as well

```
# Standardize the feature data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

And lastly, dealing with outliers will be our concern in this section.

```python
# Function to identify outliers based on IQR
def identify_outliers(df, columns):
    outliers = {}
    for col in columns:
        if df[col].dtype in ['float64', 'int64']:  # Ensure the column is numeric
            Q1 = df[col].quantile(0.25)
            Q3 = df[col].quantile(0.75)
            IQR = Q3 - Q1
            lower_bound = Q1 - 1.5 * IQR
            upper_bound = Q3 + 1.5 * IQR
            outliers[col] = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
    return outliers

# Identify outliers for each feature
outliers = identify_outliers(df, columns)

# Print the outliers for each column
for col, outlier_data in outliers.items():
    print(f"Outliers in {col}:")
    print(outlier_data[[col]])  # Print only the outlier values for the column
    print("\n")
```

The above code was to identify and show all the outliers in each feature.

Now we need to deal with them either by removing the whole row of data, replacing the outlier with the median of the data or by keeping them as is.

Now, we'll look deeply to every feature and discuss how we dealt with outliers.

1- MaxHR

```
# Filter people aged 60 and 51
age_60 = df[df['Age'] == 60]  # Extract people aged 60
age_51 = df[df['Age'] == 51]  # Extract people aged 51

# Calculate the median MaxHR for each age group
median_maxhr_60 = age_60['MaxHR'].median()
median_maxhr_51 = age_51['MaxHR'].median()


# Define a function to replace outliers in MaxHR with the calculated median
def replace_outliers_with_median_for_age_group(df, age_group, median_maxhr):
    # Filter the data for the specific age group
    age_group_data = df[df['Age'] == age_group]

    # Calculate the IQR for MaxHR within this age group
    Q1 = age_group_data['MaxHR'].quantile(0.25)
    Q3 = age_group_data['MaxHR'].quantile(0.75)
    IQR = Q3 - Q1

    # Determine the lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Identify outliers within the age group
    outliers = age_group_data[(age_group_data['MaxHR'] < lower_bound) | (age_group_data['MaxHR'] > upper_bound)]

    # Replace the outliers with the median MaxHR of that age group
    df.loc[outliers.index, 'MaxHR'] = median_maxhr

    return df
```

```
# Replace outliers for age 60 with the median for age 60
df = replace_outliers_with_median_for_age_group(df, 60, median_maxhr_60)

# Replace outliers for age 51 with the median for age 51
df = replace_outliers_with_median_for_age_group(df, 51, median_maxhr_51)

# Verify changes by checking a subset of the data
df_subset = df[df['Age'].isin([60, 51])]  # Check people aged 60 and 51
print("\nSubset of data for people aged 60 and 51 after replacing outliers:")
print(df_subset[['Age', 'MaxHR']])


# Create a boxplot to visualize MaxHR for the two age groups
plt.figure(figsize=(8, 6))
sns.boxplot(x='Age', y='MaxHR', data=df_subset, palette='coolwarm')
plt.title('MaxHR Comparison for People Aged 60 vs. 51 (After Replacing Outliers)')
plt.xlabel('Age')
plt.ylabel('MaxHR')
plt.show()
```

We first extracted sub-groups of people aged 60 and 51. Then we calculated their medians to replace their outliers with these medians.

The outliers were 63 and 60 for people aged 60 and 51, respectively.

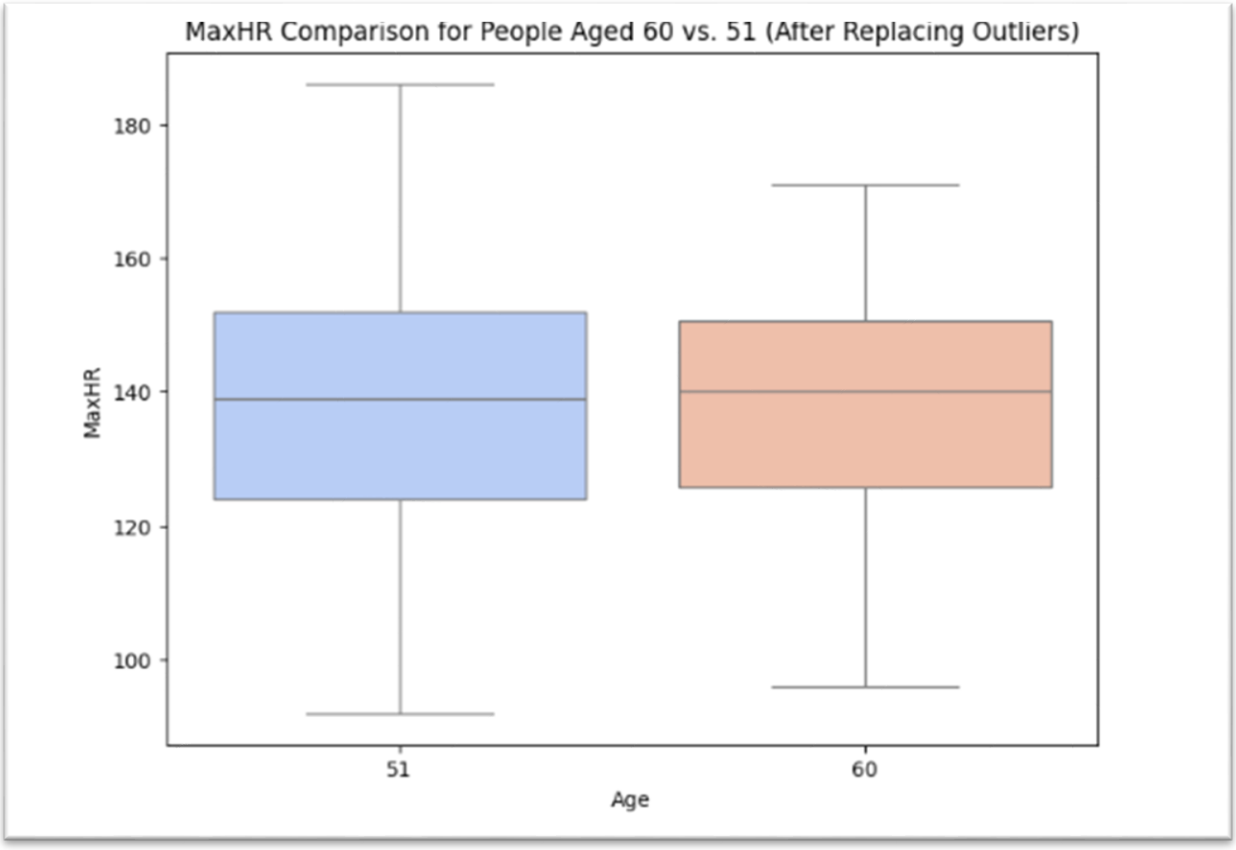
A general rule for estimating maximum heart rate is **220 - age**.

220 – 60 =160 bpm

220 – 51 = 169 bpm

So these are outliers.

We dealt with those **outliers by replacing them with the median** calculated for every sub-group



The above is a boxplot to show the median of every sub-group.

2- OldPeak

The **Oldpeak** feature in heart disease datasets typically represents the **depression** in the ST segment of an ECG during a stress test, and it's usually interpreted as a measure of heart strain. It can be positive or negative:

- **Positive values** typically indicate a **depressed ST segment**, suggesting possible ischemia (reduced blood flow to the heart).
- **Negative values** may suggest a **recovery phase** or a "downsloping" in the ST segment during the test.

Key Points:

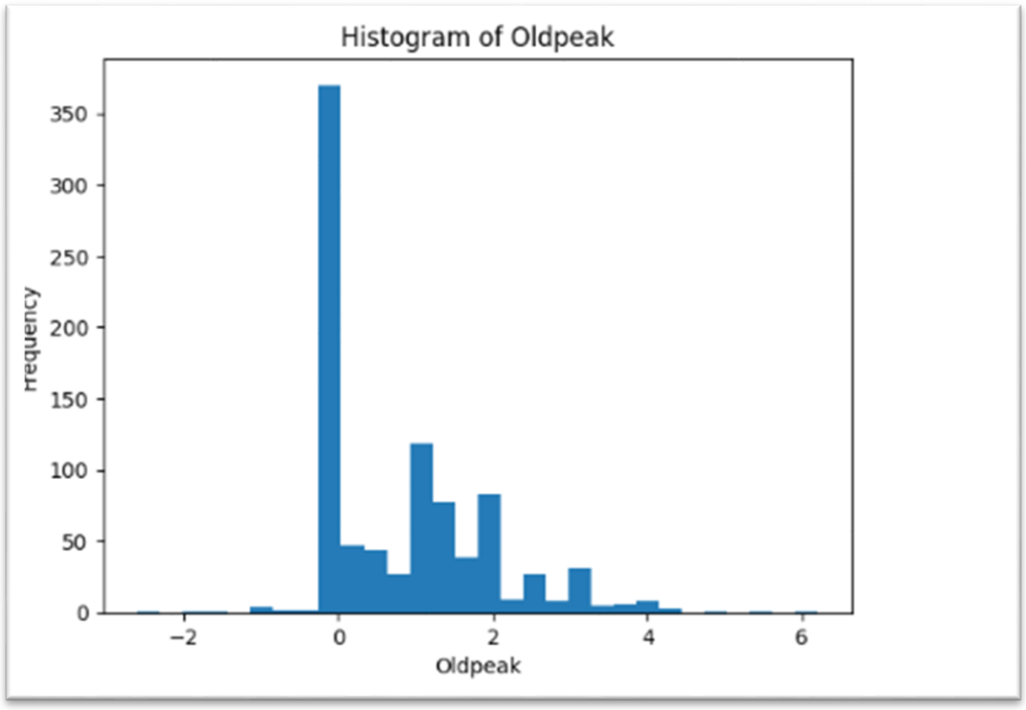
- **Oldpeak**: The **negative Oldpeak** of **-2.6** is notably **low**. In most heart disease datasets, **Oldpeak** is expected to be a positive value (indicating heart strain). Negative values are uncommon and might represent recovery phases or a specific anomaly in the data.

- **0 (Normal)**: No depression, no indication of ischemia.
- **Small Positive Values (1.0 to 2.0)**: Indicates possible **mild ischemia** or heart strain.
- **Larger Positive Values (3.0 or more)**: Indicates **severe ischemia** or other significant heart issues.
- **Negative Values**: Rare and usually suggest **anomalous results**, recovery phases, or specific types of heart behavior, but might need further investigation as outliers in the dataset.
  - **1.0 to 2.0** might indicate mild to moderate ischemia.
  - **3.0 and higher** could indicate more severe ischemia or heart issues.

Negative Values: While **Oldpeak** is mostly positive, negative values (e.g., **-2.6**) are less common and might suggest unusual results or recovery phases. Negative values should be examined carefully in the context of the patient's condition. In some cases, **downsloping ST segments** might be observed in normal or healthy individuals, though this is rare. Generally, negative values below **0** are seen as **outliers** and require careful interpretation.

This outlier caused record 326 to be removed from the dataset. Other outliers were kept as is, since they indicate different actual cases.

```
# Boxplot to visualize distribution of Oldpeak values
sns.boxplot(data=df['Oldpeak'])
plt.title('Box Plot of Oldpeak')
plt.show()

# Histogram to see distribution
plt.hist(df['Oldpeak'], bins=30)
plt.title('Histogram of Oldpeak')
plt.xlabel('Oldpeak')
plt.ylabel('Frequency')
plt.show()


df = df[df['Oldpeak'] != -2.6]

# Check if the outlier is removed
print(df[df['Oldpeak'] == -2.6])

df_subset = df.iloc[320:330]
print(df_subset)
```



A small subset to show that the record was removed

Note: that the record was removed, however the index 326 went to the next data record (as if there was a shift in the data).

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	\
320	59	1	1	125	0	1	0	
321	63	1	2	100	0	1	0	
322	38	0	2	105	0	1	0	
323	62	1	2	115	0	1	0	
325	42	1	2	105	0	1	0	
326	45	1	1	110	0	0	0	
327	59	1	2	125	0	1	0	
328	52	1	2	95	0	1	0	
329	60	1	2	130	0	1	1	
330	60	1	1	115	0	1	0	

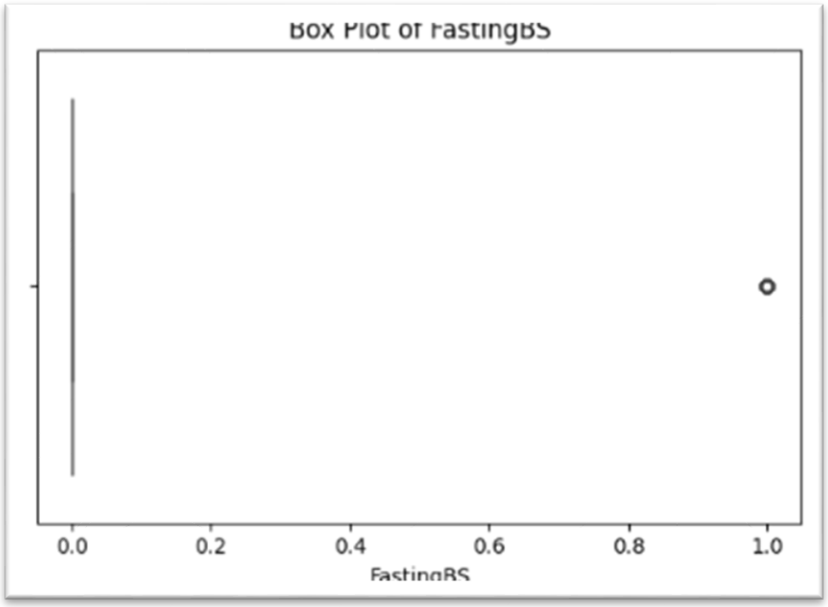
	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
320	175	0	2.6	1	1
321	109	0	-0.9	1	1
322	166	0	2.8	0	1
323	128	1	2.5	2	1
325	128	1	-1.5	2	1
326	138	0	-0.1	0	0
327	119	1	0.9	0	1
328	82	1	0.8	1	1
329	130	1	1.1	2	1
330	143	0	2.4	0	1

3- FastingBS

In heart disease datasets, **FastingBS** typically represents whether a person's **fasting blood sugar level** is above a certain threshold. It's a categorical variable where:

- **0**: Indicates that the person's fasting blood sugar level is **below 120 mg/dl** (normal range).
- **1**: Indicates that the person's fasting blood sugar level is **greater than 120 mg/dl** (indicative of high blood sugar, potentially diabetes).

Thus, no outliers



4- Cholestrol level

In medical terms, **cholesterol** refers to a fatty substance found in the blood. It's an essential building block for the body, but **too much cholesterol** can lead to **plaque buildup** in the arteries, which may increase the risk of **heart disease**. The cholesterol level is usually measured in **mg/dL (milligrams per deciliter)**. The typical components of cholesterol that are measured are:

1. **Total Cholesterol (TC)**: Includes both **good** and **bad** cholesterol.
2. **Low-Density Lipoprotein (LDL)**: Known as "bad" cholesterol. High levels of LDL can lead to plaque buildup in the arteries.
3. **High-Density Lipoprotein (HDL)**: Known as "good" cholesterol. It helps remove LDL cholesterol from the bloodstream.
4. **Triglycerides**: Another type of fat in the blood, but not specifically mentioned here.

Total Cholesterol:

- **Normal**: Less than 200 mg/dL
- **Borderline High**: 200-239 mg/dL
- **High**: 240 mg/dL or higher

LDL (Bad Cholesterol):

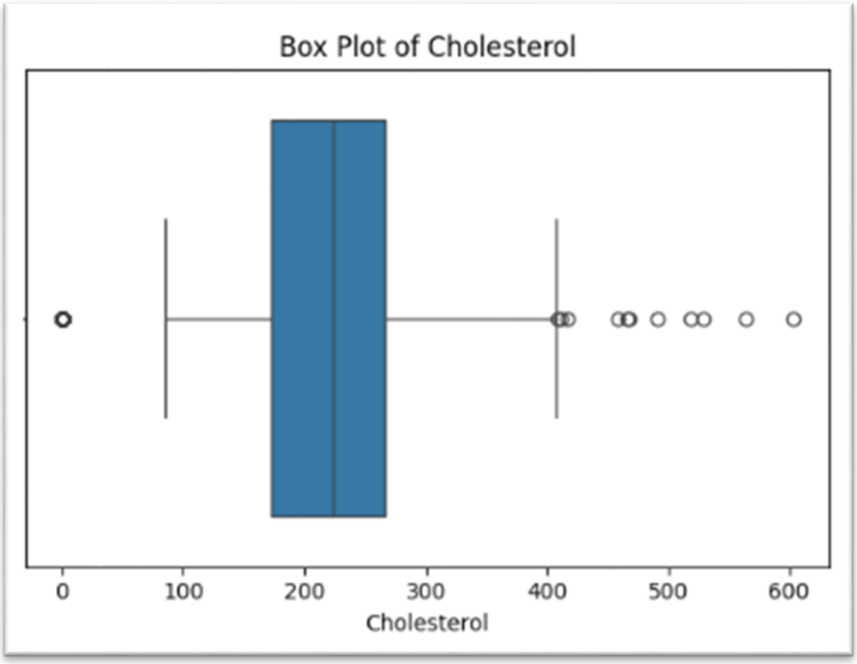
- **Optimal**: Less than 100 mg/dL
- **Near Optimal/Above Optimal**: 100-129 mg/dL
- **High**: 160 mg/dL or higher

HDL (Good Cholesterol):

- **Low**: Less than 40 mg/dL (for men) or less than 50 mg/dL (for women)
- **High**: 60 mg/dL or higher (considered protective)

There were several data containing 0 cholesterol level and cholesterol might not be specific to an age group; thus, we couldn't remove them or deal with it in the same way as we dealt with MaxHR.

The outliers were replaced by the median of all the cholesterol levels. All other data were kept as is.



5. RestingBP

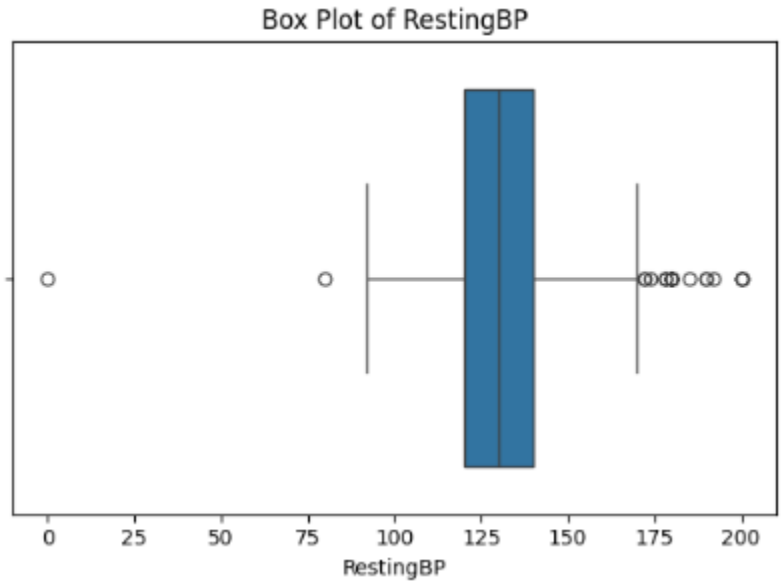
RestingBP represents the **resting blood pressure** of an individual, typically measured in **mmHg (millimeters of mercury)**. It is an important indicator of heart health and can provide insights into the risk of heart disease. The general categories for blood pressure are:

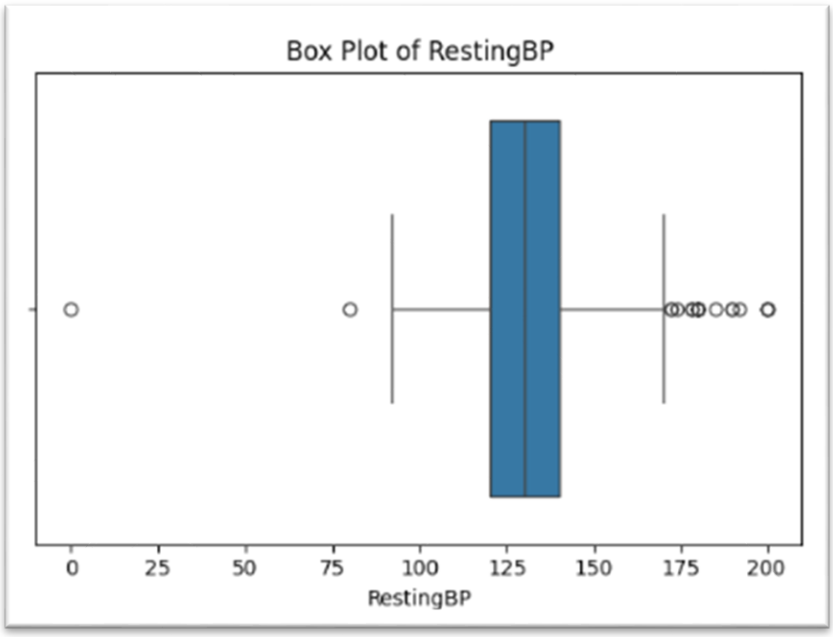
- **Normal**: Less than **120/80 mmHg**

- **Elevated**: Systolic (top number) between **120-129** and diastolic (bottom number) less than **80**
- **Hypertension Stage 1**: Systolic between **130-139** or diastolic between **80-89**
- **Hypertension Stage 2**: Systolic at least **140** or diastolic at least **90**
- **Hypertensive Crisis**: Systolic above **180** and/or diastolic above **120** (requires immediate medical attention)

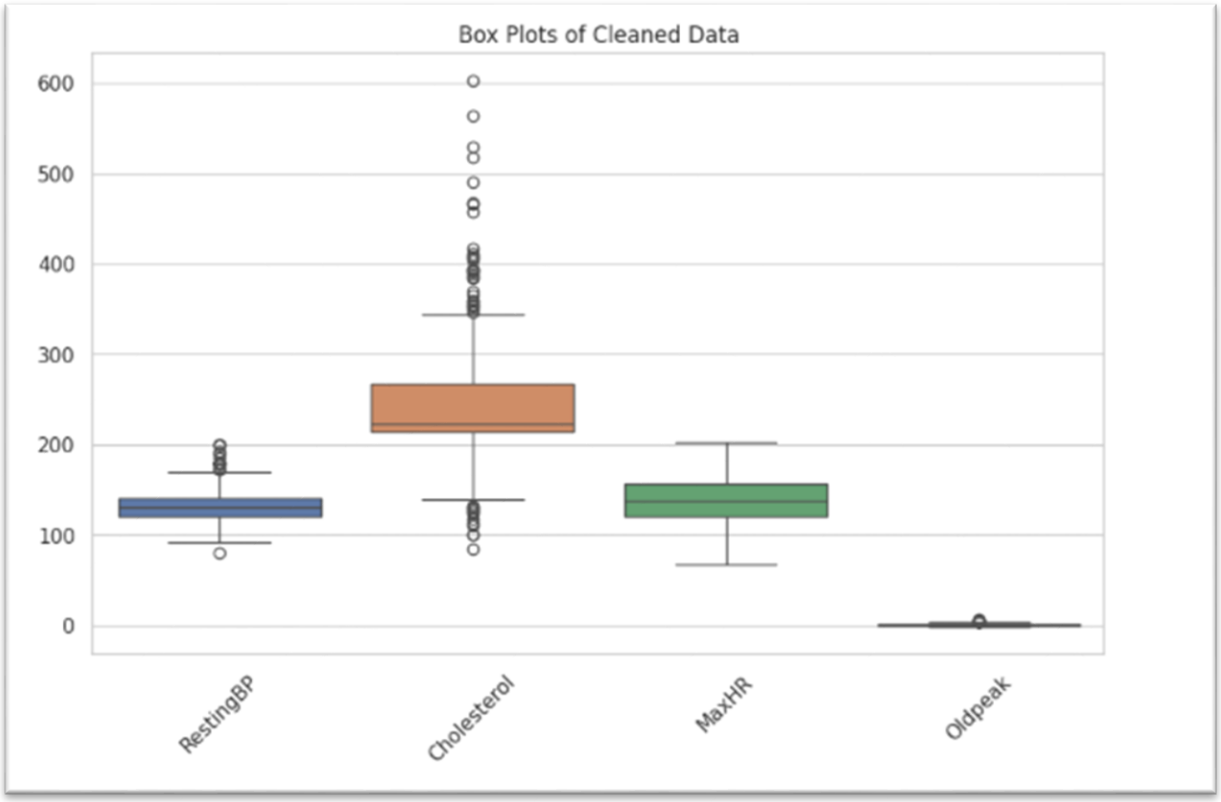
Only the zero values are removed; record 451.

All the rest are kept as is.





The box plots after the modifications.



Training Classifiers and Hyperparameters tuning

1- Data Splitting

- **Training Set Size:** 732 samples.
- **Testing Set Size:** 92 samples.
- **Validation Set Size:** 92 samples.
- The splits follow an 80-10-10 ratio as intended, ensuring the model is trained, tested, and validated on distinct subsets.

2- First, we will be talking about SVM and Naive Bayes.

- For Naive Bayes, we chose the hyperparameter to be "var_smoothing" which adds small value to the variance of each feature. This adjustment prevents division by zero or numerical instability when working with data where variance might be very close to zero.
  Purpose:
  - Controls how much smoothing is applied.

- Higher values of `var_smoothing` make the model more robust to noisy data by assuming slightly larger variance for features.
- Smaller values assume the data distribution closely matches the observed variance, which may lead to overfitting in noisy datasets.

- For SVM, we chose C as the regularization parameter that reduces the error on the training data.
  Purpose
  - Smaller values (e.g., 0.1) increase regularization, allowing for a simpler model by tolerating some misclassifications to improve generalization.
  - Larger values (e.g., 100) reduce regularization, aiming to classify the training data more accurately but potentially leading to overfitting.

  Also, we chose "Kernel" to be our second hyperparameter. Where the kernel type specifies the type of kernel function to use in mapping input features into a higher-dimensional space where they may be linearly separable.
- According to our case, we chose the smoothing parameters in naive bayes to be equal to 0.28 since this value corresponds to the highest validation accuracy
  Additionally, we chose the "C" regularization parameter to be equal 1 and the kernel to be RBF since this kernel is used with nonlinear data as in our case.

```
# 1. Hyperparameters for Naive Bayes
nb_param_grid = {
    'var_smoothing': np.logspace(0, -9, num=100)  # Range of smoothing parameter
}

# 2. Hyperparameters for SVM
svm_param_grid = {
    'C': [0.1, 1, 10, 100],  # Regularization parameter
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
}
```

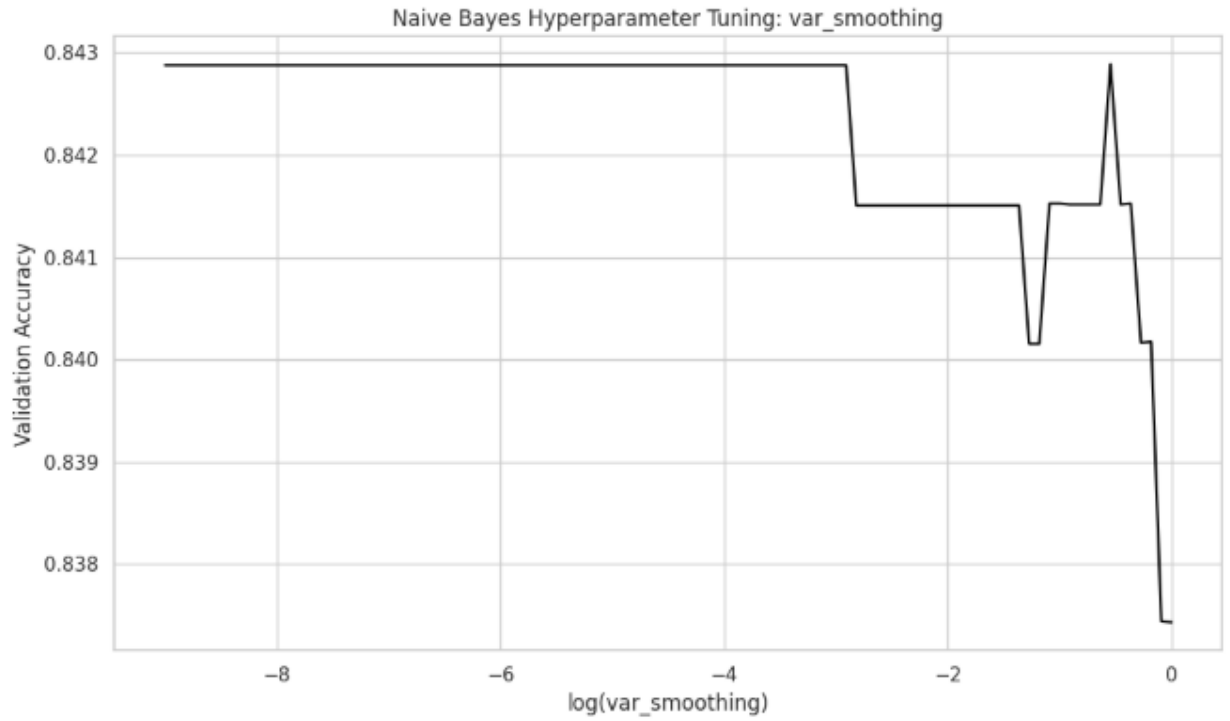
Note: **`np.logspace(0, -9, num=100)`** generates 100 values logarithmically spaced between 100^0=1 (minimal smoothing) and 10^−9=0.000000001 (very high smoothing).

```
# 3. KNN Hyperparameters
knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

# 4. Decision Tree Hyperparameters
dt_param_grid = {
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}
```

These are the different values for each hyperparameter in KNN and DT.

KNN as n neighbors and metric. While Decision tree has maximum depth of the tree itself, minimum number of samples required to split an internal node, minimum number of samples that a leaf node must have.

Now, let's see the reason behind choosing these values of the hyperparameters.

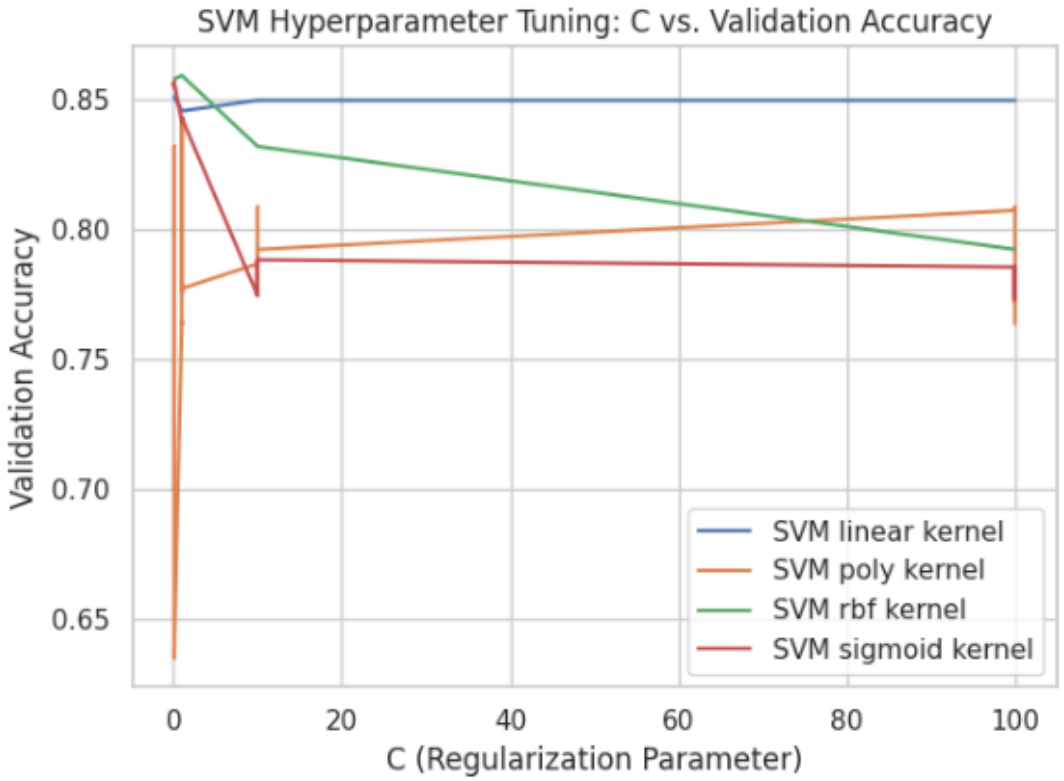


The above graph shows the different generated values of the smoothing parameters against the validation accuracy of each.

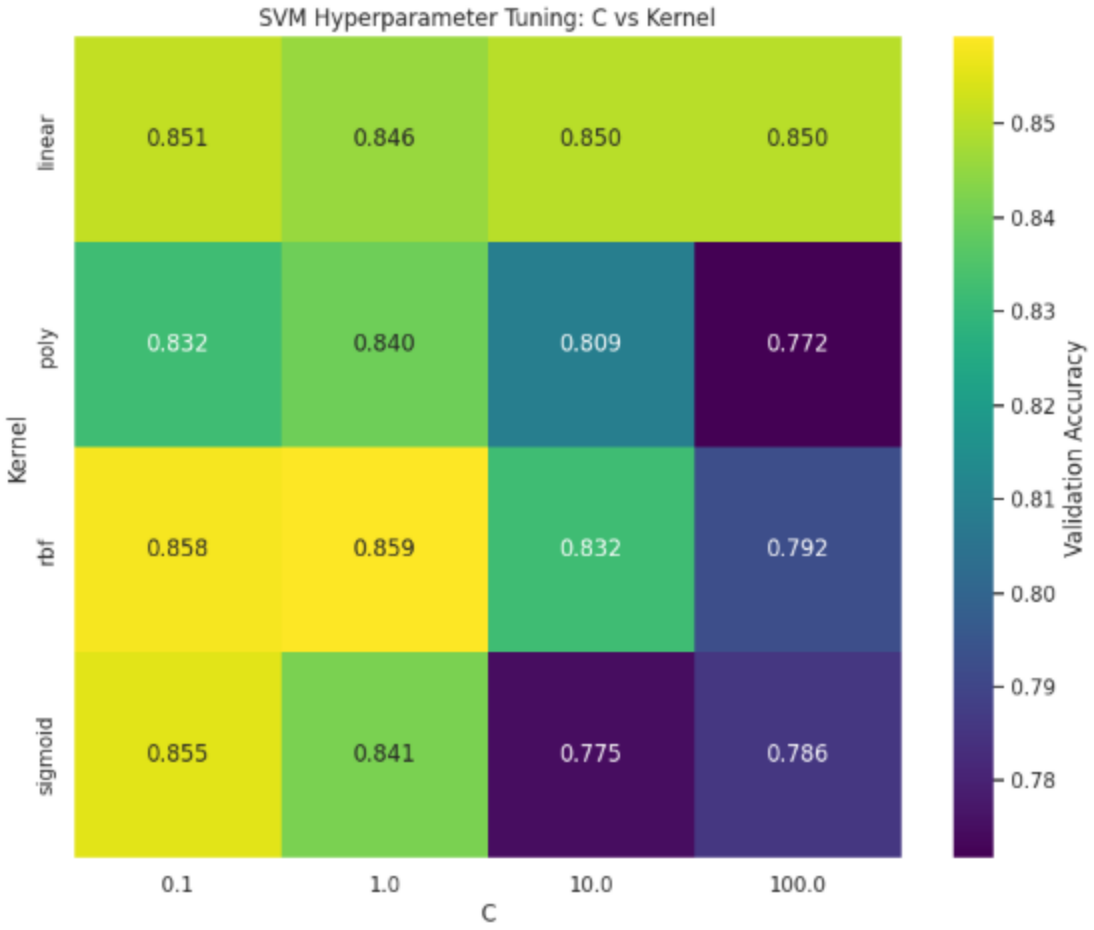
Here we come to an important question: why didn't we choose any of the negative var_smoothing values instead?

The answer is simply because a negative value will decrease the variance which may cause the model to be highly sensitive to small variations in data and being prone to numerical instability or division by zero errors if the variance becomes zero or negative.

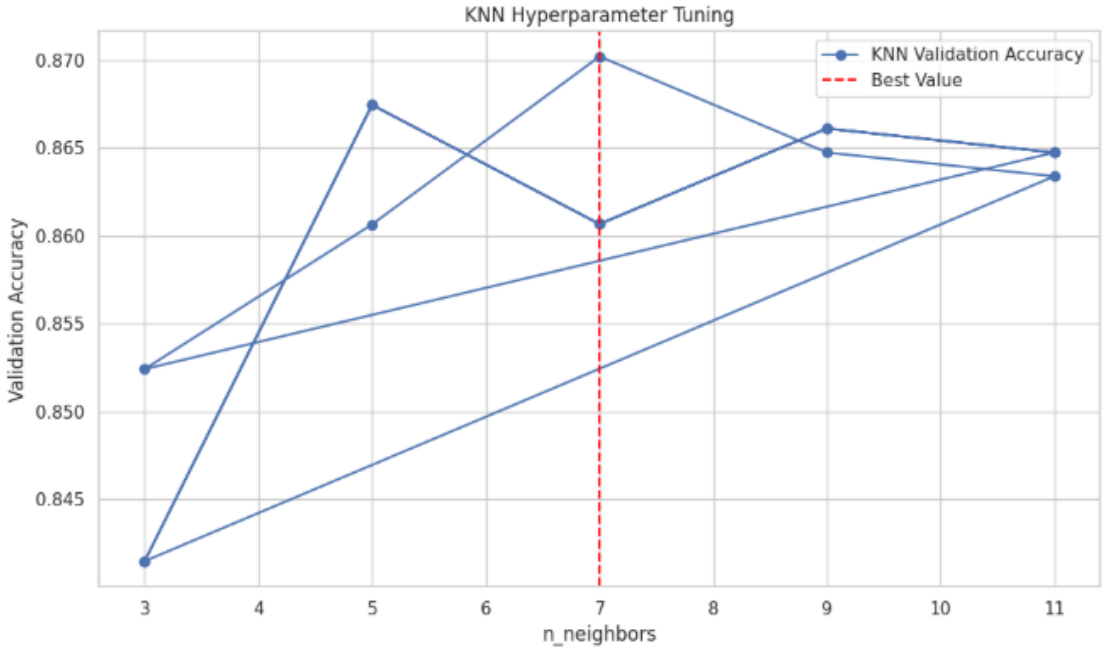
However, we need to increase the variance slightly to ensure robustness.



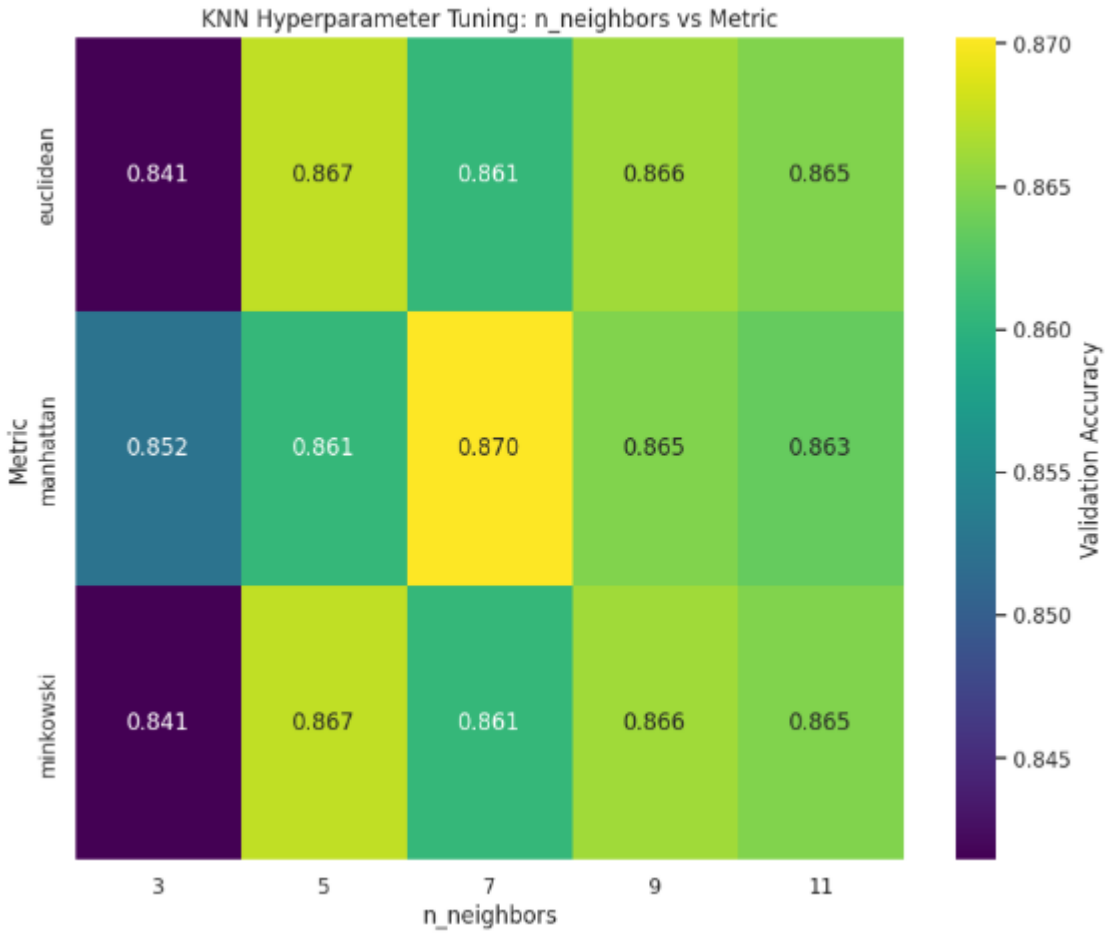

```
import pandas as pd

# Extract results from GridSearchCV for SVM
svm_results = grid_search_results["SVM"].cv_results_
df_svm = pd.DataFrame(svm_results)

# Filter relevant columns for heatmap
df_filtered = df_svm[['mean_test_score', 'param_C', 'param_kernel']]

# Pivot table to reshape data for heatmap
heatmap_data = df_filtered.pivot(index='param_kernel', columns='param_C', values='mean_test_score')

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(heatmap_data, annot=True, fmt=".3f", cmap="viridis", cbar_kws={'label': 'Validation Accuracy'})
plt.title("SVM Hyperparameter Tuning: C vs Kernel")
plt.xlabel("C")
plt.ylabel("Kernel")
plt.show()
```



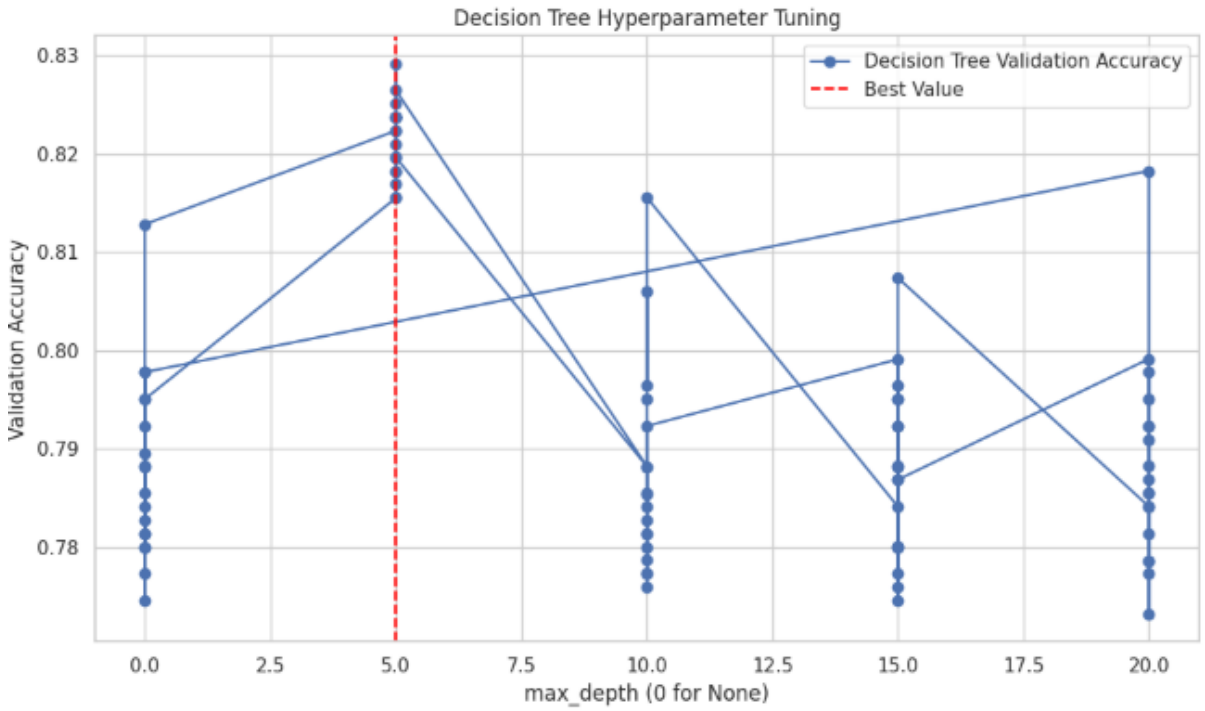
This diagram shows each kernel with a corresponding C parameter against the validation accuracy. As we can observe, rbf kernel and C value = 1 have the highest validation accuracy.




```
# Extract results from GridSearchCV for KNN
knn_results = grid_search_results["KNN"].cv_results_
df_knn = pd.DataFrame(knn_results)

# Filter relevant columns for heatmap
df_filtered = df_knn[['mean_test_score', 'param_n_neighbors', 'param_metric']]

# Pivot table to reshape data for heatmap
heatmap_data = df_filtered.pivot(index='param_metric', columns='param_n_neighbors', values='mean_test_score')

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(heatmap_data, annot=True, fmt=".3f", cmap="viridis", cbar_kws={'label': 'Validation Accuracy'})
plt.title("KNN Hyperparameter Tuning: n_neighbors vs Metric")
plt.xlabel("n_neighbors")
plt.ylabel("Metric")
plt.show()
```



The above graph shows that when k= 7, the validation accuracy is the highest. As we know, if K increases the decision boundary becomes smoother and simpler.



As we can see, the optimum maximum depth of the decision tree is 5 corresponding to 0.829 validation accuracy. With a depth of 5, the decision tree is relatively interpretable. It's not too deep to make it difficult for a human to follow the logic behind its decisions, which is one of the key advantages of decision trees. A tree of depth 5 strikes a good balance between accuracy and interpretability. A maximum depth of 5 indicates that the model has found a reasonable balance between underfitting and overfitting. The tree is deep enough to capture meaningful patterns in the data, but not so deep that it becomes overly complex and starts to model noise or outliers.

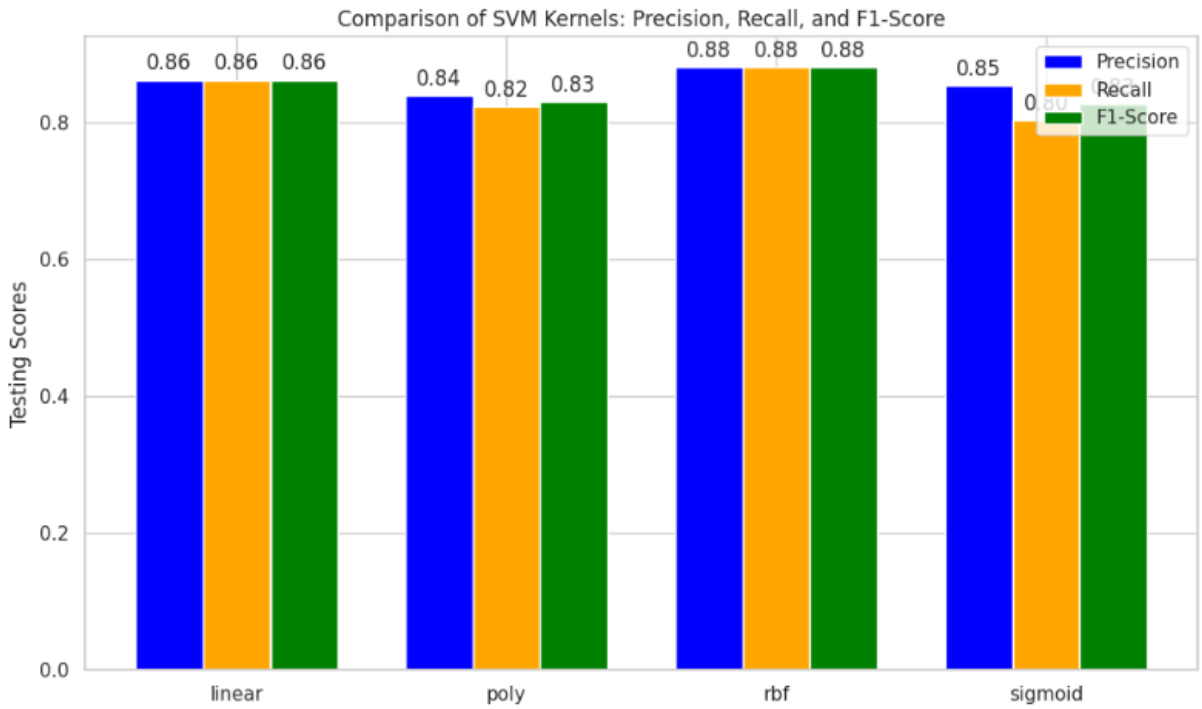
Those are the best hyperparameters we chose and their corresponding validation accuracy.

```
Best Parameters for Naïve Bayes: {'var_smoothing': 0.2848035868435802}
Best Cross-Validated Accuracy for Naïve Bayes: 0.8429
Running GridSearchCV for SVM
Best Parameters for SVM: {'C': 1, 'kernel': 'rbf'}
Best Cross-Validated Accuracy for SVM: 0.8592
Running GridSearchCV for KNN
Best Parameters for KNN: {'metric': 'manhattan', 'n_neighbors': 7}
Best Cross-Validated Accuracy for KNN: 0.8702
Running GridSearchCV for Decision Tree
Best Parameters for Decision Tree: {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 2}
Best Cross-Validated Accuracy for Decision Tree: 0.8292
```

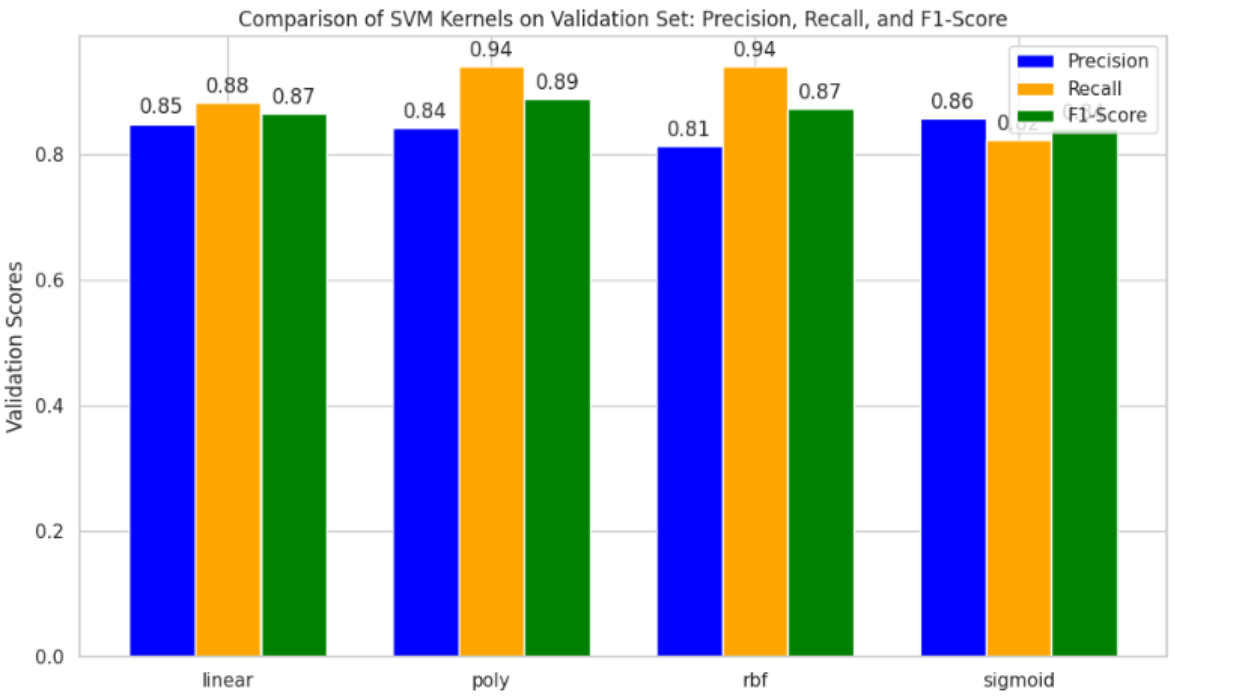
Now we will see different comparisons between both classifiers and within each of them.

```
Training set size: 732
Testing set size: 92
Validation set size: 92

Naïve Bayes Classifier Results:
Accuracy on Test Set: 0.7934782608695652
Confusion Matrix:
 [[33  8]
 [11 40]]
Classification Report:
              precision    recall  f1-score   support

           0       0.75      0.80      0.78        41
           1       0.83      0.78      0.81        51

    accuracy                           0.79        92
   macro avg       0.79      0.79      0.79        92
weighted avg       0.80      0.79      0.79        92


SVM Classifier Results:
Accuracy on Test Set: 0.8478260869565217
Confusion Matrix:
 [[34  7]
 [ 7 44]]
Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.83      0.83        41
           1       0.86      0.86      0.86        51

    accuracy                           0.85        92
   macro avg       0.85      0.85      0.85        92
weighted avg       0.85      0.85      0.85        92


Validation Set Evaluation:
Naïve Bayes Accuracy on Validation Set: 0.8804347826086957
SVM Accuracy on Validation Set: 0.8478260869565217
```

Note: we'll break this later in the section.



Every kernel against Testing Scores



Every kernel against Validation Scores

First, we need to discuss what each measure represents:

- Precision measures the proportion of correctly predicted positive cases out of all predicted positive cases.
- Recall measures the proportion of actual positive cases that the model correctly identifies.
- The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both.

Metric	Focus Area	Key Question Answered	Use When...
Precision	Correctness of positive predictions	"Of all predicted positives, how many are true?"	False positives are costly.
Recall	Coverage of actual positives	"Of all actual positives, how many are detected?"	False negatives are costly.
F1-Score	Balance of precision and recall	"What is the overall model quality in terms of precision and recall?"	Balance is needed.

Let's now investigate the difference between both classifiers according to precision, recall, and F1-score. As we can see, SVM looks higher than Naive Bayes by 0.02%.

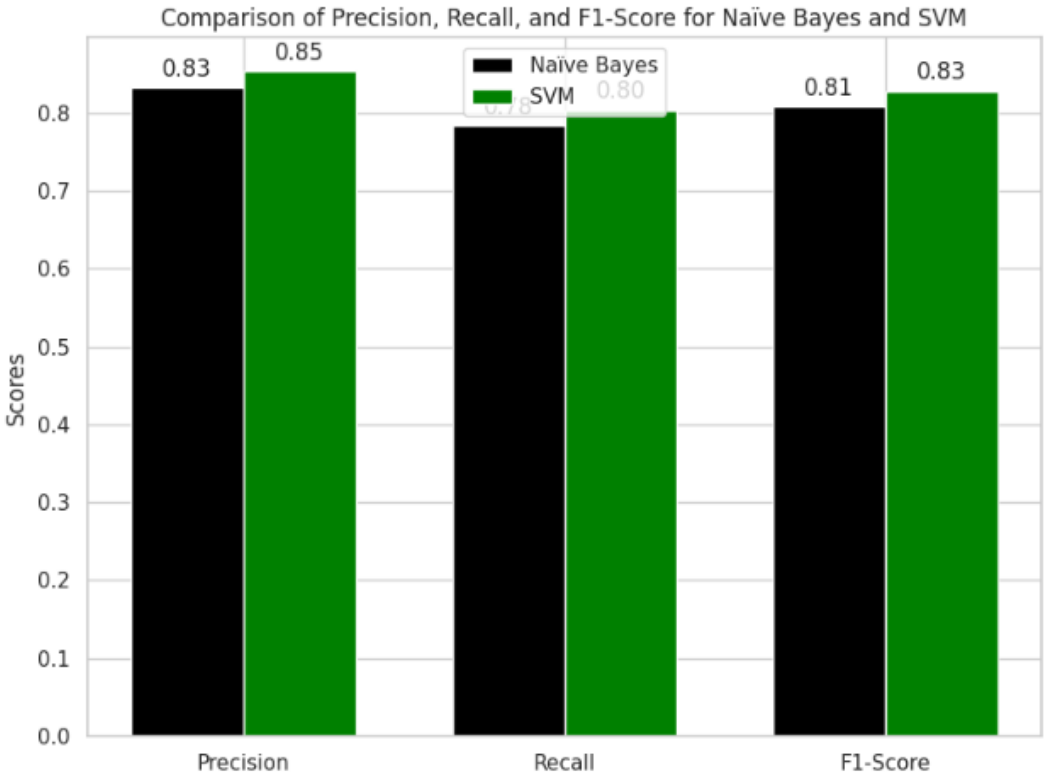
This might arise because of several factors including data distribution, feature scaling, model complexity and the response to noise as well as overfitting.

According to data distribution, Naive Bayes performs well with Gaussian or categorical data, depending on the implementation. SVM is more flexible in handling diverse data distributions and can work well with both linearly separable and non-linear data when using different kernels.

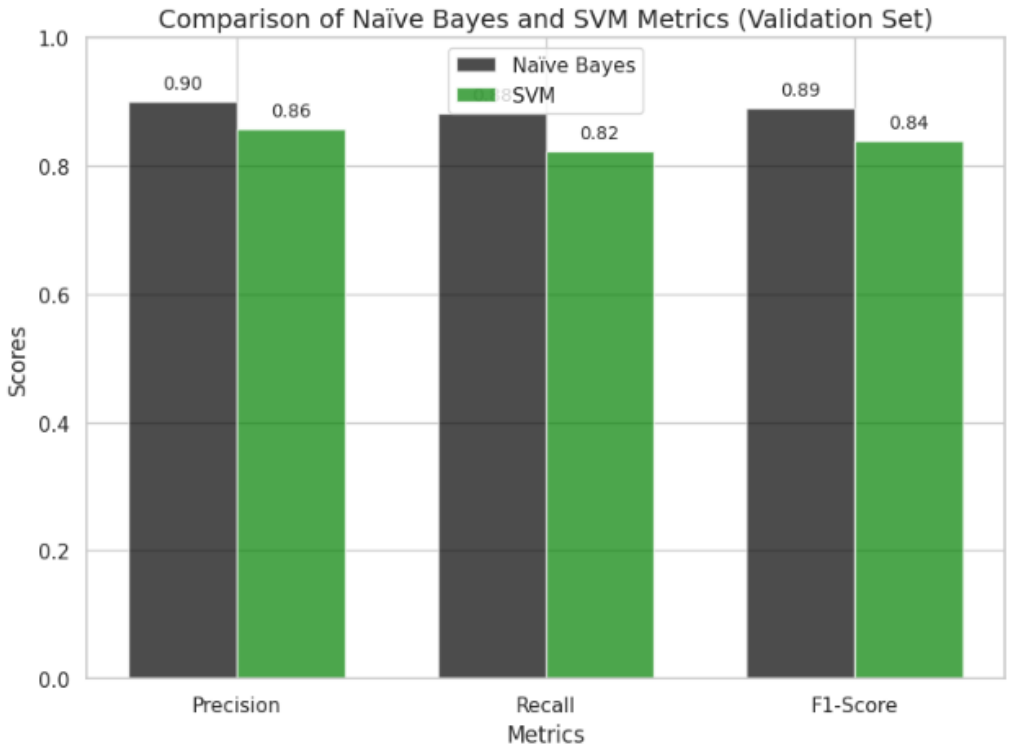
Going to feature scaling, SVM is sensitive to the scale of features, so standardization helps it perform optimally. Naive Bayes typically does not require feature scaling, but scaled features can affect how probabilities are computed.

Looking over model complexity, Naive Bayes is a simpler model, which may not capture complex decision boundaries well. SVM, especially with non-linear kernels, can model complex decision boundaries that Naive Bayes cannot.

Talking about noise and overfitting, Naive Bayes is less prone to overfitting because of its simplicity. SVM, especially with complex kernels, might slightly overfit the data, which could affect generalization performance.

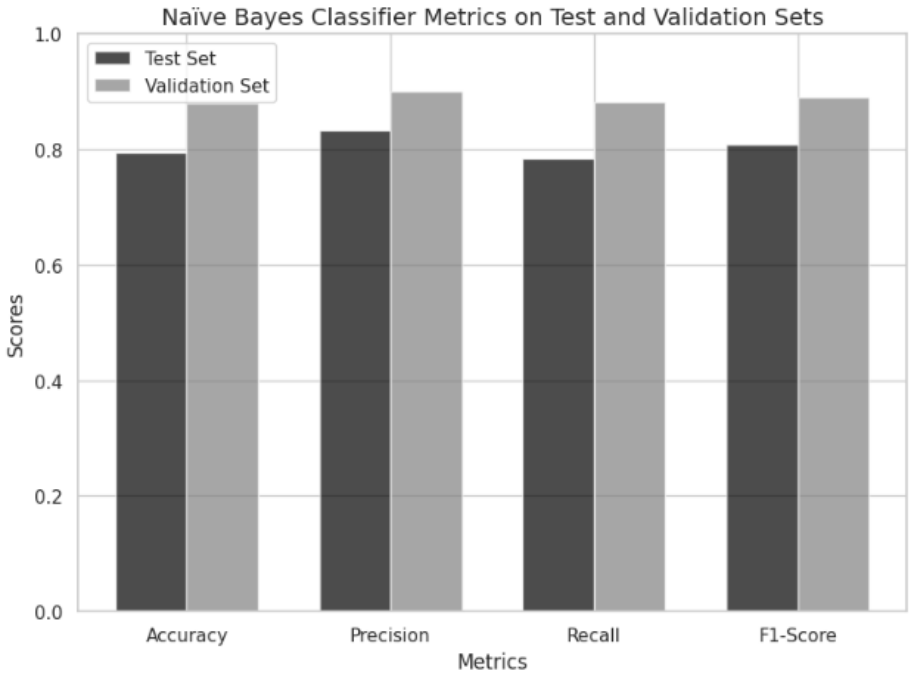


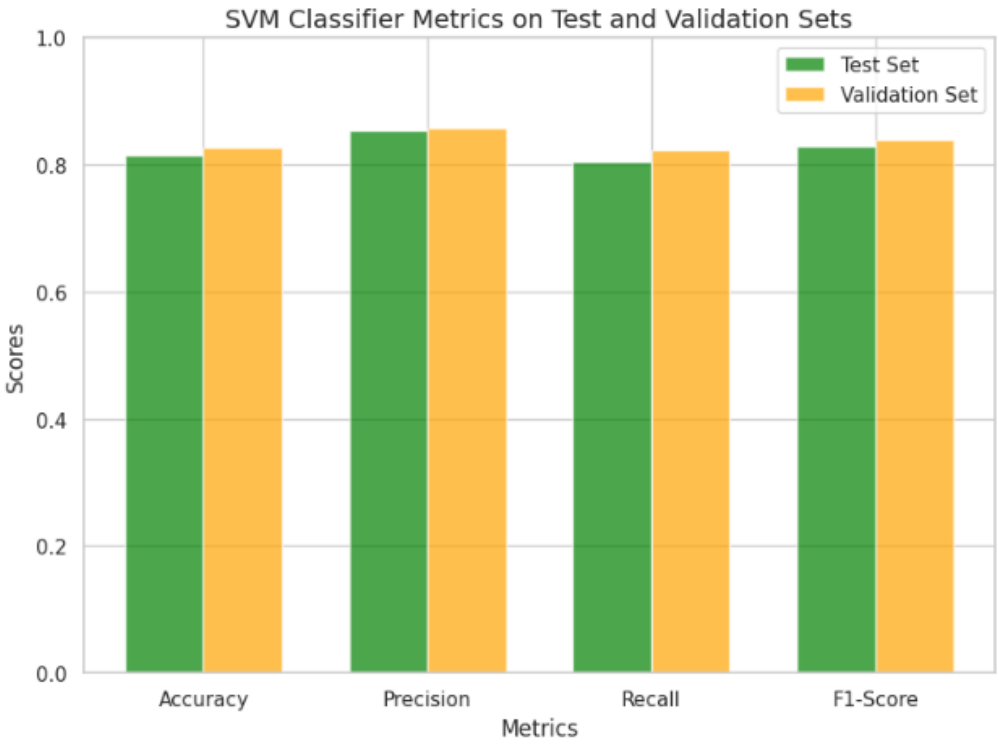
SVM and Naive Bayes **testing** set vs different metrics



SVM and Naive Bayes **validation** sets vs different metrics

Now, we will compare all metrics between validation and testing sets for each classifier.





As we can see, the model is performing better with validation sets than testing sets in both classifiers. However, the difference between validation and testing set scores with each metrics in Naive Bayes classifier is much higher than the difference in SVM. The larger difference between the validation and test performance for Naive Bayes compared to SVM indicates that Naive Bayes

might be more sensitive to the data splits or overfitting, whereas SVM, being more flexible, handles such discrepancies better. This difference in behavior highlights the strengths and weaknesses of the models in dealing with variations between different datasets.

Comparing the 4 classifiers

```
# 5. Train and Evaluate Naïve Bayes
nb_classifier = GaussianNB()
nb_classifier.fit(X_train_scaled, y_train)

# Predict on test and validation data
y_pred_nb = nb_classifier.predict(X_test_scaled)
y_pred_nb_val = nb_classifier.predict(X_val_scaled)

# Calculate metrics for Naïve Bayes
nb_accuracy = accuracy_score(y_test, y_pred_nb)
nb_precision = precision_score(y_test, y_pred_nb)
nb_recall = recall_score(y_test, y_pred_nb)
nb_f1 = f1_score(y_test, y_pred_nb)

# Calculate validation metrics for Naïve Bayes
nb_metrics_val = [
    accuracy_score(y_val, y_pred_nb_val),
    precision_score(y_val, y_pred_nb_val),
    recall_score(y_val, y_pred_nb_val),
    f1_score(y_val, y_pred_nb_val)
]
```

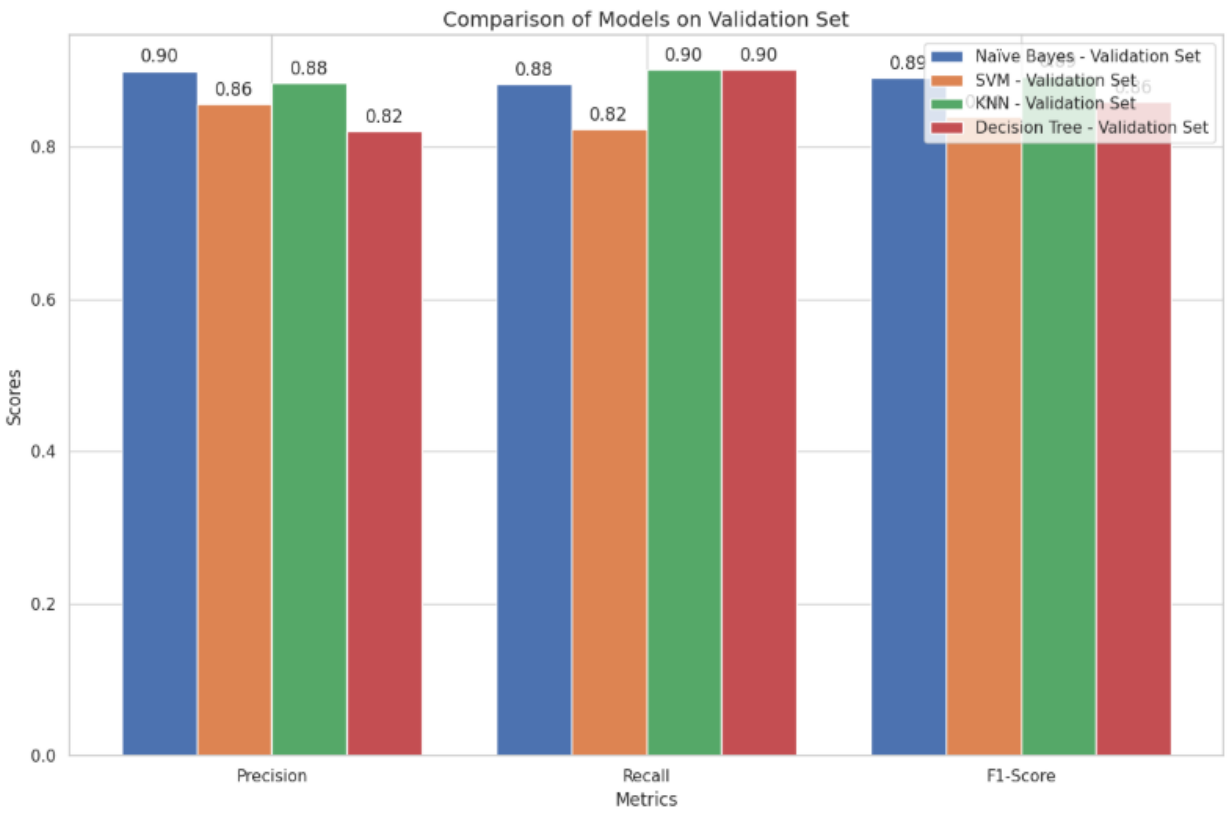
```
# 6. Train and Evaluate SVM
svm_classifier = SVC(kernel='rbf')
svm_classifier.fit(X_train_scaled, y_train)

# Predict on test and validation data
y_pred_svm = svm_classifier.predict(X_test_scaled)
y_pred_svm_val = svm_classifier.predict(X_val_scaled)

# Calculate metrics for SVM
svm_accuracy = accuracy_score(y_test, y_pred_svm)
svm_precision = precision_score(y_test, y_pred_svm)
svm_recall = recall_score(y_test, y_pred_svm)
svm_f1 = f1_score(y_test, y_pred_svm)

# Calculate validation metrics for SVM
svm_metrics_val = [
    accuracy_score(y_val, y_pred_svm_val),
    precision_score(y_val, y_pred_svm_val),
    recall_score(y_val, y_pred_svm_val),
    f1_score(y_val, y_pred_svm_val)
]
```

```
# 7. Train and Evaluate KNN
knn_classifier = KNeighborsClassifier(n_neighbors=7)
knn_classifier.fit(X_train_scaled, y_train)

# Predict on test and validation data
y_pred_knn = knn_classifier.predict(X_test_scaled)
y_pred_knn_val = knn_classifier.predict(X_val_scaled)

# Calculate metrics for KNN
knn_accuracy = accuracy_score(y_test, y_pred_knn)
knn_precision = precision_score(y_test, y_pred_knn)
knn_recall = recall_score(y_test, y_pred_knn)
knn_f1 = f1_score(y_test, y_pred_knn)

# Calculate validation metrics for KNN
knn_metrics_val = [
    accuracy_score(y_val, y_pred_knn_val),
    precision_score(y_val, y_pred_knn_val),
    recall_score(y_val, y_pred_knn_val),
    f1_score(y_val, y_pred_knn_val)
]
```

```
# 8. Train and Evaluate Decision Trees
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train_scaled, y_train)

# Predict on test and validation data
y_pred_dt = dt_classifier.predict(X_test_scaled)
y_pred_dt_val = dt_classifier.predict(X_val_scaled)

# Calculate metrics for Decision Trees
dt_accuracy = accuracy_score(y_test, y_pred_dt)
dt_precision = precision_score(y_test, y_pred_dt)
dt_recall = recall_score(y_test, y_pred_dt)
dt_f1 = f1_score(y_test, y_pred_dt)

# Calculate validation metrics for Decision Trees
dt_metrics_val = [
    accuracy_score(y_val, y_pred_dt_val),
    precision_score(y_val, y_pred_dt_val),
    recall_score(y_val, y_pred_dt_val),
    f1_score(y_val, y_pred_dt_val)
]
```

As we can see above, we have used all the hyperparameters that would provide the maximum validation accuracy while training our model.

```
# Prepare Data for Combined Plot
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
models = ['Naïve Bayes', 'SVM', 'KNN', 'Decision Tree']

# Collect results for each classifier for the test set
test_scores = {
    'Naïve Bayes': [nb_accuracy, nb_precision, nb_recall, nb_f1],
    'SVM': [svm_accuracy, svm_precision, svm_recall, svm_f1],
    'KNN': [knn_accuracy, knn_precision, knn_recall, knn_f1],
    'Decision Tree': [dt_accuracy, dt_precision, dt_recall, dt_f1]
}

# Collect results for each classifier for the validation set
val_scores = {
    'Naïve Bayes': nb_metrics_val,
    'SVM': svm_metrics_val,
    'KNN': knn_metrics_val,
    'Decision Tree': dt_metrics_val
}
```



Precision

By inspecting each metric separately, the high precision of Naïve Bayes (0.90) suggests that the model is effectively discriminating between the classes, especially in terms of minimizing false positives. This could be due to the dataset's characteristics, such as feature independence or class distribution, which play to the strengths of Naïve Bayes. Meanwhile, KNN's slightly lower score (0.88) might indicate sensitivity to noise or suboptimal parameters, and the SVM and Decision Tree models might be facing issues like overfitting or suboptimal tuning, causing them to score the least.

Recall

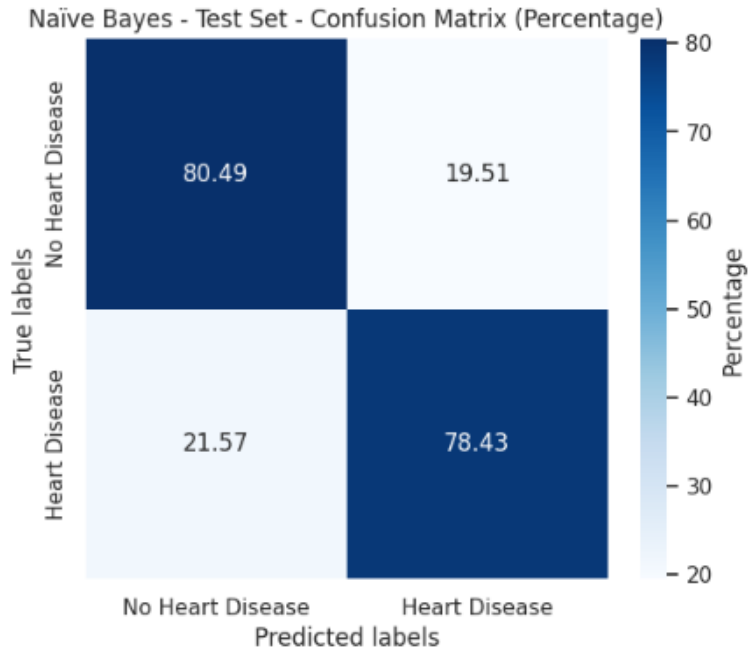
Looking at the recall metric, Naive Bayes again outstands all other classifiers by scoring 0.88. While DT and KNN are the next, SVM is still the least, scoring 0.82. SVM's performance can be sensitive to hyperparameter settings (such as the C value and kernel choice), and suboptimal parameters may reduce recall. Since our distribution is more imbalanced, Naive Bayes is more likely to predict the positive class more often, resulting in higher recall.

F1-score

While for F1-score, KNN and Naive bayes score the highest (0.89) and DT scores (0.86). Moreover, SVM is the least scoring. This might happen due to the several reasons mentioned above.

Testing Classifiers

Now we will compare the confusion matrix of validation and testing for both classifiers.



The performance of the Naive Bayes classifier in classifying heart disease can be analyzed as follows:

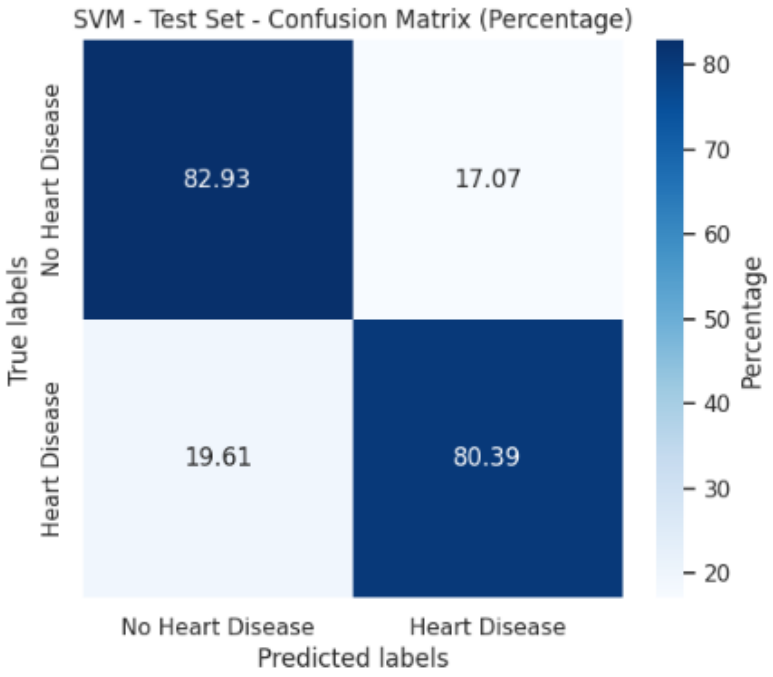
- The classifier correctly identifies individuals with heart disease with an accuracy of **78.43%** (True Positive Rate or Sensitivity).
- It accurately classifies individuals without heart disease as not having heart disease with an accuracy of **80.49%** (True Negative Rate or Specificity).

However, the classifier exhibits limitations in its predictive capabilities. Specifically:

- It **misclassifies individuals without heart disease as having heart disease** (False Positive Rate).
- It **misclassifies individuals with heart disease as not having heart disease** (False Negative Rate).

Among these errors, the **False Negative Rate** (failing to identify heart disease in a person who actually has it) is particularly critical due to its severe implications for patient health. The False Negative Rate is approximately **2% higher** than the False Positive Rate, indicating a greater prevalence of this

type of misclassification. This discrepancy highlights a key area for model improvement, as undetected cases of heart disease could lead to delayed or missed treatment, posing significant risks to affected individuals.



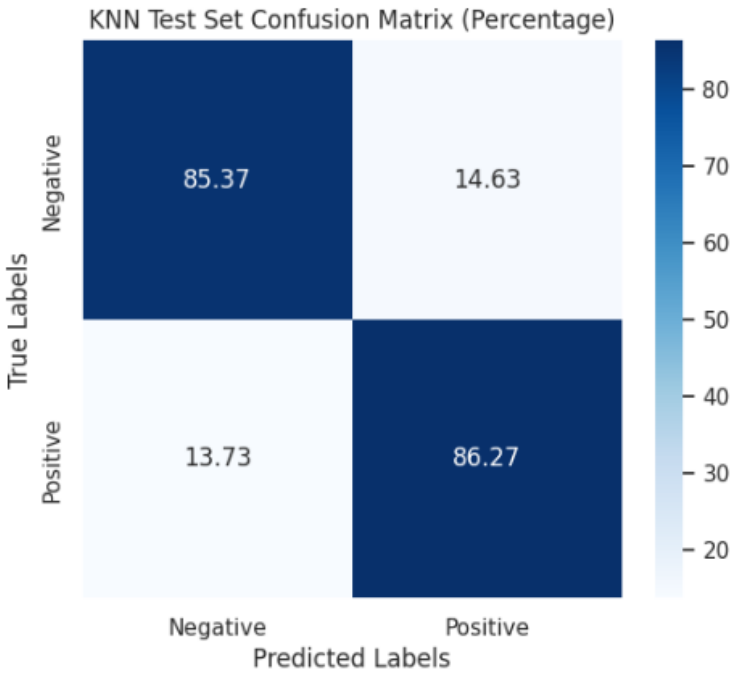
The performance of the SVM classifier in classifying heart disease can be analyzed as follows:

- The classifier correctly identifies individuals with heart disease with an accuracy of **80.39%** (True Positive Rate or Sensitivity).
- It accurately classifies individuals without heart disease as not having heart disease with an accuracy of **82.93%** (True Negative Rate or Specificity).

However, the classifier exhibits limitations in its predictive capabilities. Specifically:

- It **misclassifies individuals without heart disease as having heart disease** (False Positive Rate).
- It **misclassifies individuals with heart disease as not having heart disease** (False Negative Rate).

Among these errors, the **False Negative Rate** (failing to identify heart disease in a person who has it) is particularly critical due to its severe implications for patient health. The False Negative Rate is approximately **2.5% higher** than the False Positive Rate, indicating a greater prevalence of this type of misclassification. This discrepancy highlights a key area for model improvement, as undetected cases of heart disease could lead to delayed or missed treatment, posing significant risks to affected individuals.



The performance of the k-Nearest Neighbors (k-NN) classifier in predicting heart disease can be evaluated as follows:

- The model correctly identifies individuals with heart disease with a **True Positive Rate (Sensitivity)** of **86.27%**, indicating strong performance in detecting positive cases.
- It also classifies individuals without heart disease as negative with a **True Negative Rate (Specificity)** of **85.37%**, reflecting a comparable ability to recognize negative cases accurately.

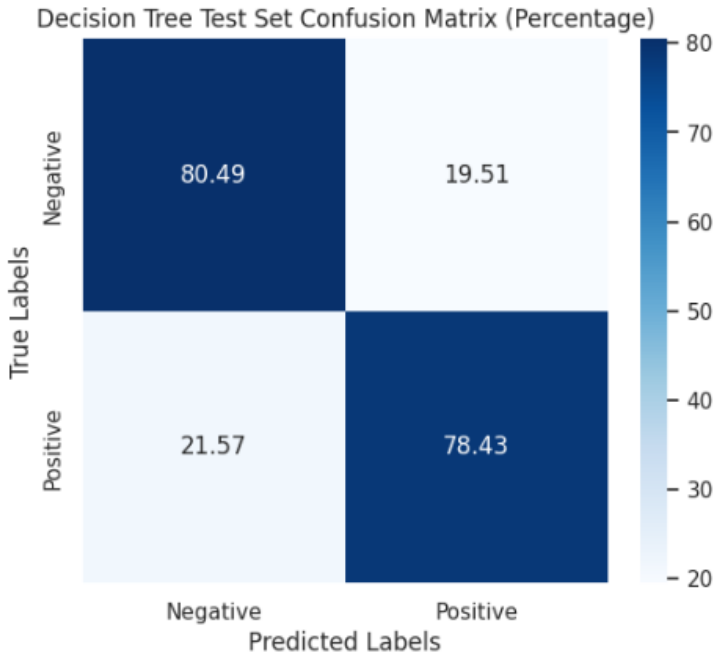
However, the classifier demonstrates some limitations in misclassification:

- **13.73%** of individuals with heart disease are misclassified as not having heart disease (False Negatives).

- **14.63%** of individuals without heart disease are misclassified as having heart disease (False Positives).

While the overall accuracy of the k-NN classifier is notable, the **False Negative Rate** (13.73%) is a critical concern, as it represents cases where individuals with heart disease go undetected. This misclassification can delay diagnosis and treatment, potentially leading to severe health outcomes. The False Negative Rate is slightly lower than the False Positive Rate (14.63%), but both errors are significant and warrant attention for model improvement.

To enhance the model's performance, strategies such as optimizing the choice of k, fine-tuning the distance metric, incorporating feature scaling, or selecting relevant features could be explored. Additionally, evaluating the model under different decision thresholds or employing hybrid approaches may help strike a better balance between sensitivity and specificity.



The performance of the Decision Tree (DT) classifier in detecting heart disease is as follows:

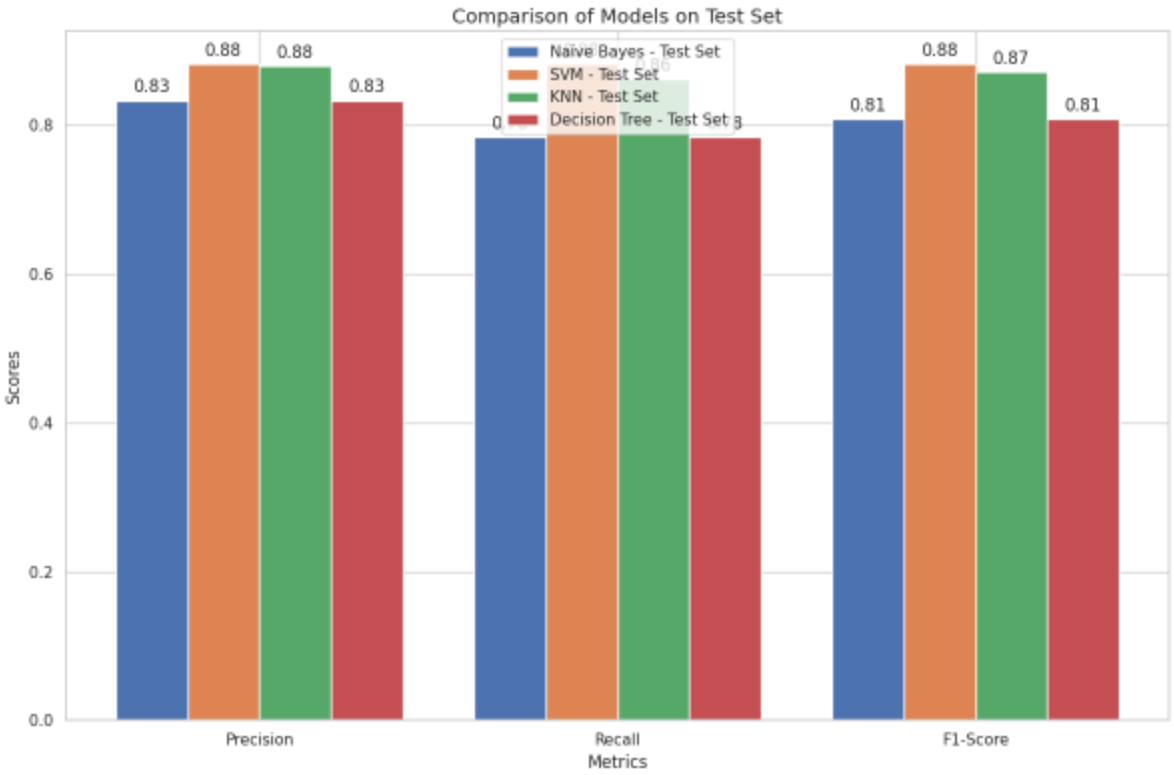
- The classifier achieves a **True Positive Rate (Sensitivity)** of **78.43%**, meaning it correctly identifies individuals with heart disease in approximately four out of five cases.
- It also demonstrates a **True Negative Rate (Specificity)** of **80.49%**, indicating its ability to correctly classify individuals without heart disease.

However, the model exhibits some misclassification issues:

- **21.57%** of individuals with heart disease are misclassified as not having heart disease (False Negatives).
- **19.51%** of individuals without heart disease are misclassified as having heart disease (False Positives).

Among these errors, the **False Negative Rate** (21.57%) is particularly concerning due to the significant health implications of failing to detect heart disease. Although this rate is higher than the False Positive Rate (19.51%), both types of misclassification are non-negligible and highlight areas for improvement in the model.

To enhance the Decision Tree classifier's performance, techniques such as pruning, adjusting the splitting criteria (e.g., Gini impurity or entropy), or exploring ensemble methods like Random Forests or Gradient Boosting could be considered. These approaches may help reduce the misclassification rates and improve the overall reliability of the model.



Precision Comparison

- **Naive Bayes** and **Decision Tree** have a precision score of **0.83**, indicating a similar ability to minimize false positives.
- **SVM and KNN**, on the other hand, achieve the highest precision score of **0.88**. This suggests that SVM and KNN are more accurate in predicting positive instances, as they make fewer false positive errors compared to the other classifiers.

Recall Comparison

- Both **Naive Bayes** and **Decision Tree achieve a recall score of 0.78**, while **KNN reaches 0.86**. This indicates that this model can correctly identify 86% of the actual positive instances.
- **SVM** stands out again, with the highest recall score of **0.88**, demonstrating its effectiveness in identifying more of the actual

positives. This suggests that SVM makes fewer false negative errors compared to the other models.

F1-Score Comparison

- **Naive Bayes** and **Decision Tree achieve an F1-Score of 0.81**, reflecting their equal balance between precision and recall.
- **KNN performs better, with an F1-Score of 0.87**, indicating that it strikes a good balance between correctly identifying positive instances and avoiding false positives.
- **SVM** leads with an F1-Score of **0.88**, highlighting its overall effectiveness in both precision and recall. It suggests that SVM is not only accurate in making predictions but also maintains a balance in minimizing false positives and negatives.

The results clearly show that **SVM** outperforms the other classifiers across all three metrics, making it the most reliable model for this test set. It has the highest precision, meaning it predicts positive classes with greater accuracy, and the highest recall, meaning it identifies more actual positive instances than the other models. Its high F1-Score demonstrates a strong balance between precision and recall.

Dendrogram analysis

Description

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

# Linkage method determines how distances between clusters are calculated
# Use Ward's method (minimizes variance within clusters)
linkage_matrix0 = linkage(X_train_scaled, method='ward', metric='euclidean')

# Create the dendrogram
plt.figure(figsize=(100, 8))
dendrogram(linkage_matrix0)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Points (or Clusters)')
plt.ylabel('Euclidean Distance (Proximity Measure)')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Purpose of the Code

1. **Hierarchical Clustering**:
    a. Uses the **Ward's method** for clustering, which minimizes the variance within clusters.
    b. Computes a linkage matrix, which encodes the hierarchical relationships between data points or clusters.
2. **Dendrogram Visualization**:
    a. Displays how individual data points are merged into clusters based on their distances.

1. Perform Hierarchical Clustering

- `linkage_matrix0`:
    - Encodes the hierarchical structure of the dataset.
    - Each row represents a merge operation between two clusters or points, with:
        - Columns 0 and 1: Indices of the merged clusters.
        - Column 2: The distance between them (Euclidean distance here).

- Column 3: The number of data points in the merged cluster.

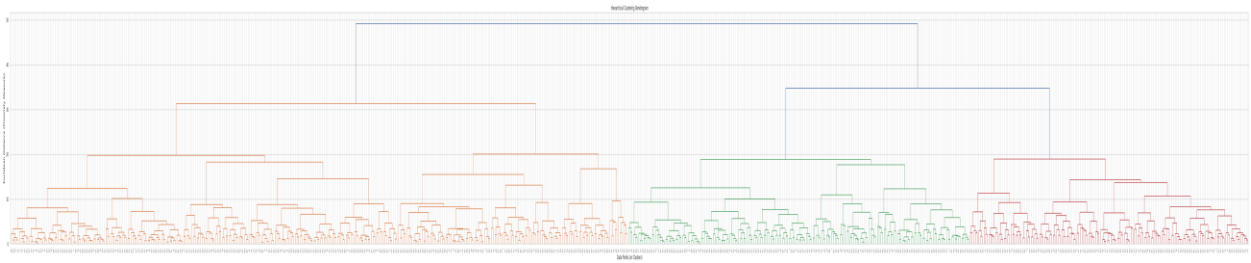
2. Plot the Dendrogram

- **Dendrogram Visualization**:
  - The x-axis represents individual data points or smaller clusters.
  - The y-axis represents the distance (proximity measure) at which clusters are merged.
- **Features**:
  - Vertical lines: Merging steps in hierarchical clustering.
  - Horizontal lines: Represent clusters at the corresponding merge distance.

3. Customize Plot

- **Grid**: Added for better readability of distance values.
- **Tight Layout**: Ensures the labels and titles are properly displayed without overlap.

Output



Random sample from training data

```
num_samples = 30  # Define the number of samples you want
random_indices = np.random.choice(X_train_scaled.shape[0], size=num_samples, replace=False)
X_sampled = X_train_scaled[random_indices]

# Perform hierarchical clustering
linked = linkage(X_sampled, method='ward', metric='euclidean')
plt.figure(figsize=(15, 10))
dendrogram(linked, leaf_rotation=90., leaf_font_size=10.)
plt.title('Dendrogram of Heart Failure Dataset')
plt.xlabel('Sample Index')
plt.ylabel('Euclidean Distance')
plt.show()
```

1. Sampling the Dataset

```
num_samples = 30  # Define the number of samples you
want
random_indices =
np.random.choice(X_train_scaled.shape[0],
size=num_samples, replace=False)
X_sampled = X_train_scaled[random_indices]
```

- **Sampling**:
  - `num_samples = 30` specifies that 30 random samples will be selected from the original dataset `X_train_scaled`.
  - `np.random.choice` randomly picks 30 unique indices without replacement.
  - `X_sampled` stores the selected subset for clustering.

2. Perform Hierarchical Clustering

```
linked = linkage(X_sampled, method='ward',
metric='euclidean')
```

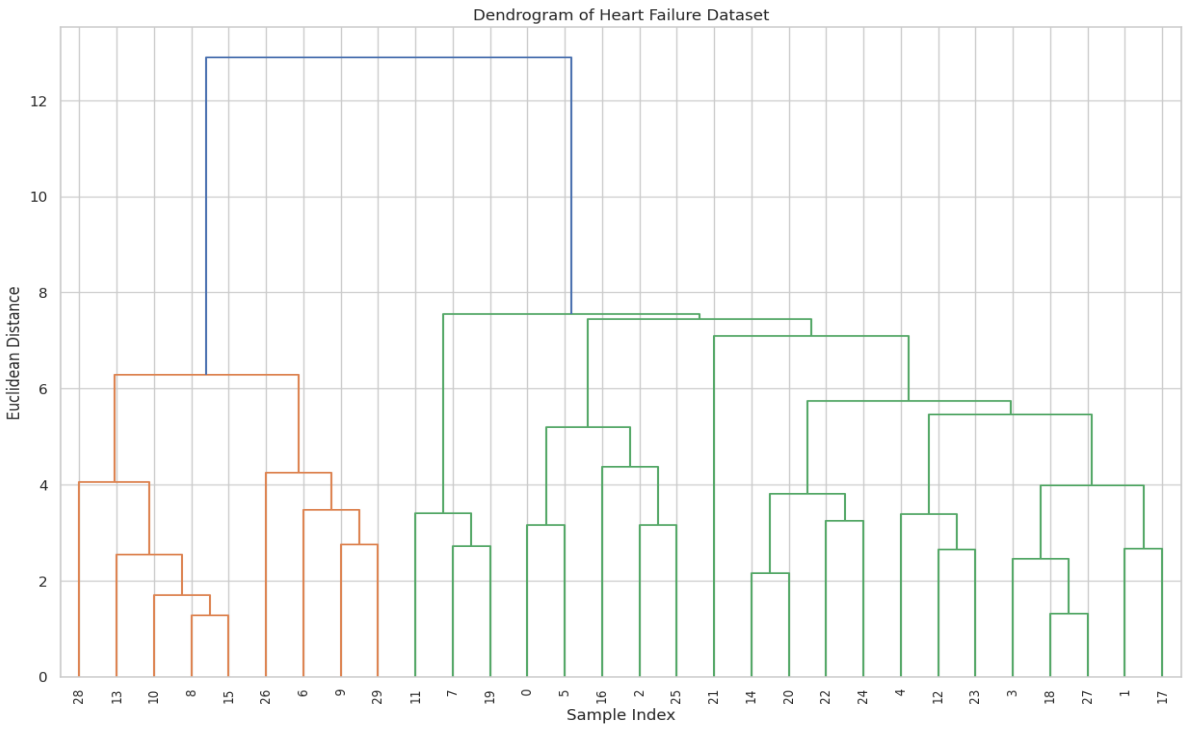
- **Linkage**:
  - `linkage` performs hierarchical clustering on the sampled data.
  - **Ward's Method** is used to minimize variance within clusters, and **Euclidean distance** is the measure of proximity between points.
  - The result, `linked`, is a linkage matrix that encodes the hierarchical clustering information.

3. Dendrogram Visualization

```
plt.figure(figsize=(15, 10))
dendrogram(linked, leaf_rotation=90., leaf_font_size=10.)
plt.title('Dendrogram of Heart Failure Dataset')
plt.xlabel('Sample Index')
plt.ylabel('Euclidean Distance')
plt.show()
```

- **Dendrogram Plot**:
  - The `dendrogram` function visualizes hierarchical clustering.
  - `leaf_rotation=90.` rotates the sample labels for better readability.
  - `leaf_font_size=10.` adjusts the font size for the sample labels.
  - The title and labels are added for clarity, with the x-axis representing sample indices and the y-axis representing the Euclidean distance between merged clusters.

Output



Calculating the number of clusters created at a cut

```python
# Step 4: Cut the Dendrogram to Form Clusters
max_d = 10 # Threshold distance for cutting the dendrogram
clusters = fcluster(linked, max_d, criterion='distance')

# Step 5: Print Number of Clusters
num_clusters = len(set(clusters))
print(f"Number of clusters: {num_clusters}")

# Step 6: Analyze Cluster Assignments
clusters_dict = {i: [] for i in range(1, num_clusters + 1)}
for sample_idx, cluster_label in enumerate(clusters):
    clusters_dict[cluster_label].append(sample_idx)

print("\nCluster Assignments:")
for cluster, points in clusters_dict.items():
    print(f"Cluster {cluster}: {points}")

# Step 7: Visualize Clusters on Dendrogram
plt.figure(figsize=(15, 10))
dendrogram(linked, leaf_rotation=90., leaf_font_size=10.,
           color_threshold=max_d)
plt.axhline(y=max_d, color='r', linestyle='--', label='Cut Threshold')
plt.title('Dendrogram with Clusters Highlighted')
plt.xlabel('Sample Index')
plt.ylabel('Euclidean Distance')
plt.legend()
plt.tight_layout()
plt.show()
```

- ### Cut the Dendrogram to Form Clusters
    - **What happens**:
        - The dendrogram (a tree diagram representing data point connections) is "cut" at a specific vertical level, known as the distance threshold (`max_d = 10`).
        - Cutting at this height groups data points into clusters where the merging distance (on the y-axis) is less than `max_d`.
    - **Outcome**:
        - Each data point is assigned a cluster label based on proximity to other points.

- ## Print Number of Clusters
    - **What happens**:
        - After cutting the dendrogram, the unique cluster labels are counted.
        - This gives the **total number of distinct clusters** formed by cutting at the chosen threshold.
    - **Outcome**:
        - For example, cutting at `max_d = 10` might result in 2 clusters if the data forms 2 distinct groups at that height.

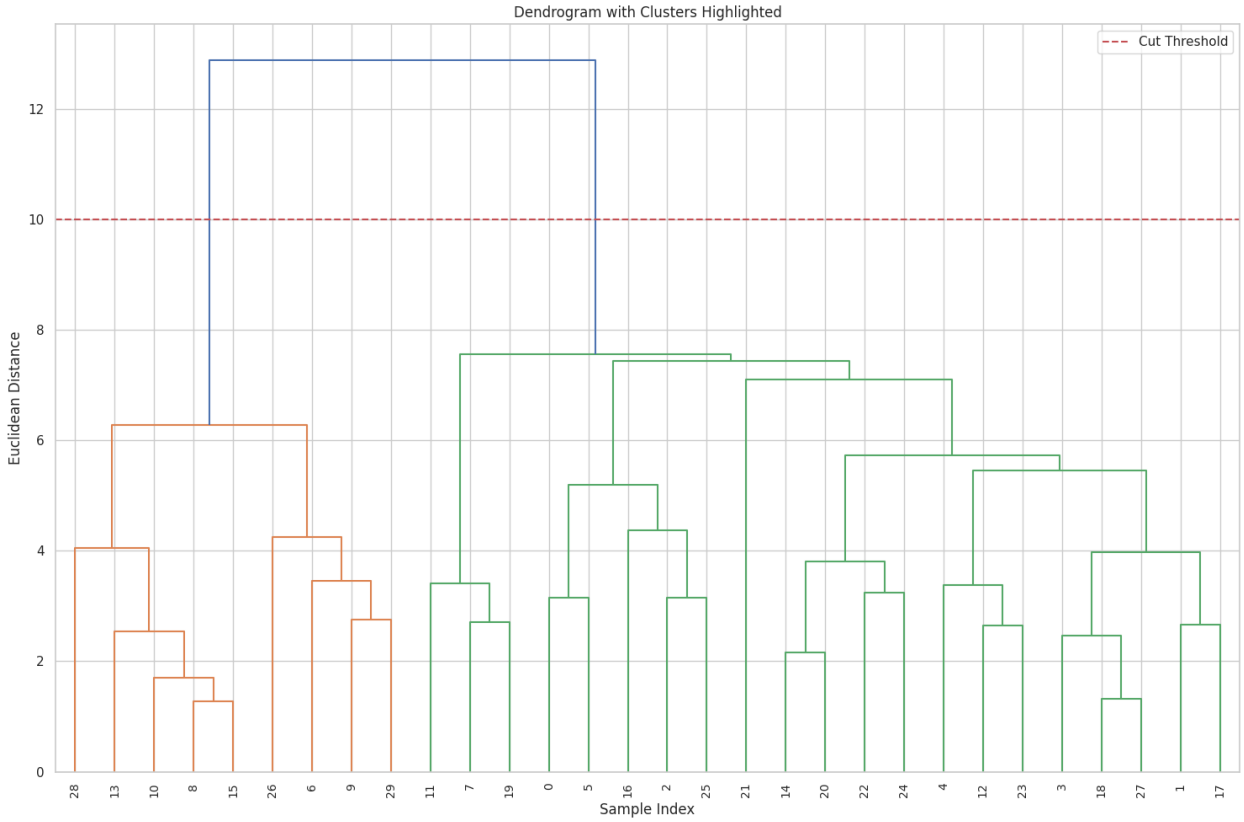
- ## Analyze Cluster Assignments
    - **What happens**:
        - Each data point (or sampled index) is grouped into its respective cluster based on the dendrogram cut.
        - A dictionary or list structure is used to organize the points by their cluster label.
    - **Outcome**:
        - You get a clear understanding of which data points belong to which clusters.

- ## Visualize Clusters on Dendrogram
    - **What happens**:
        - The dendrogram is displayed, with clusters highlighted in distinct colors based on the cut (`color_threshold=max_d`).
        - A horizontal line (`max_d = 10`) is drawn to show the cut level visually.
    - **Outcome**:
        - Clusters appear as groups of points below the red dashed line.

- Points connected above the line belong to different clusters.
- Example:
    - Below the red line: All points within a vertical branch are in the same cluster.
    - Above the red line: These branches are distinct clusters.

Output

Number of clusters: 2
Cluster Assignments:
Cluster 1: [6, 8, 9, 10, 13, 15, 26, 28, 29]
Cluster 2: [0, 1, 2, 3, 4, 5, 7, 11, 12, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27]



Interpretation of the Clusters and Proximity:

- **Low Proximity (High Similarity within Clusters)**:
  - If the clusters are tightly bound (i.e., they merge only at small distances), this suggests that the data points within the clusters share strong similarities, potentially indicating clearly distinct subgroups of your data.
  - For instance, if **heart failure patients** form clusters that are close together (low Euclidean distance), it suggests that their characteristics (e.g., age, medical history, lab results) are very similar within each cluster.
- **High Proximity (Low Similarity within Clusters)**:
  - If the clusters are more spread out (i.e., they merge at higher distances), this could indicate that the data points within a cluster may be more diverse, or the data does not naturally group well based on the chosen proximity measure.
  - For instance, in heart failure prediction, this could mean that some clusters may represent a mixture of different severity levels of the condition or that the features used do not provide clear distinction between different types of patients.

Finally:

- The **number of clusters** and their **proximity** are both influenced by the **cut-off threshold** you choose in the dendrogram and the underlying characteristics of your data.
- Clusters with **smaller proximity** are likely more homogeneous and represent tight-knit groups, while clusters with **larger proximity** are more diverse or less strongly defined.
- By adjusting the cut-off threshold, you can control the **granularity of your clusters** and tailor your analysis to better understand the dataset's inherent structure.

2D PCA with 2 clusters

```python
from sklearn.decomposition import PCA

# PCA transformation (assuming PCA has already been done)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_sampled)

# Plot the PCA results with clusters
plt.figure(figsize=(10, 8))
for cluster in set(clusters):
    cluster_points = X_pca[clusters == cluster]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster {cluster}')

plt.title('PCA Visualization of Hierarchical Clusters')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Purpose of the PCA-Based Visualization

1. **Dimensionality Reduction**:
    a. PCA (Principal Component Analysis) reduces the high-dimensional dataset into a 2D space (two principal components).
    b. This makes it easier to visually understand the clustering structure.
2. **Cluster Visualization**:
    a. Each cluster identified through hierarchical clustering is plotted in the 2D PCA space.
    b. Data points in the same cluster are represented by the same color, showing how well-separated the clusters are.

1. Perform PCA Transformation

- PCA reduces the dataset's dimensions to 2 components while retaining most of the variance.
- For hierarchical clustering, this step simplifies the visualization of clusters formed in multi-dimensional space.

2. Plot Clusters in PCA Space

- Each cluster is plotted in the 2D PCA-transformed space.
- Points from the same cluster are grouped and assigned the same color.
- Axes represent the two most significant principal components.

Output