Project Title: Predicting Heart Disease Using Supervised Machine Learning

Introduction

Heart disease is a leading cause of death worldwide. Early detection and accurate prediction of heart disease risk can be life-saving. In this project, we aim to build and evaluate different supervised machine learning models to predict the likelihood of a person having heart disease based on various medical and lifestyle factors.

Dataset

We will use the "Heart Disease UCI" dataset, which is publicly available on the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Heart+Disease). This dataset contains 920 instances and 14 attributes, including age, sex, cholesterol level, resting heart rate, and more. The target variable is binary, indicating the presence or absence of heart disease.

Problem Statement

The goal of this project is to develop and compare different supervised machine learning algorithms for predicting heart disease. We will evaluate the following algorithms:

- Logistic Regression
- K-Nearest Neighbors (KNN)
- Decision Trees
- Random Forests
- Support Vector Machines (SVM)

Methodology

1. Data Preprocessing:

    - Handling missing values.
    - Encoding categorical variables (if any).
    - Scaling numerical features.

- Splitting the dataset into training and testing sets.

2. Model Building:
    - Implementing the above-mentioned supervised learning algorithms.
    - Hyperparameter tuning for each algorithm.
3. Model Evaluation:
    - Evaluating model performance using various metrics such as accuracy, precision, recall, F1-score, and ROC-AUC.
    - Comparing the performance of different algorithms.
4. Model Interpretation:
    - Visualizing important features and their impact on predictions.
    - Interpreting the models to gain insights into the factors contributing to heart disease.
5. Conclusion:
    - Summarizing the findings.
    - Recommending the most effective machine learning model for heart disease prediction.

Project Implementation

We will implement the project using Python and popular libraries such as pandas, scikit-learn, and matplotlib. Each supervised learning algorithm will be trained, evaluated, and compared using a standardized approach to ensure a fair assessment.

Expected Outcome

We anticipate building accurate machine learning models that can predict the presence or absence of heart disease with high reliability. The project's output will include model performance metrics, visualizations, and a report summarizing the findings.

References

1. UCI Machine Learning Repository - Heart Disease Data: https://archive.ics.uci.edu/ml/datasets/Heart+Disease

Import necessary libraries for data preprocessing, visualization, and modeling

In [1]:
```
# Import necessary libraries
import matplotlib as mpl # For changing some default matplotlib parameters
import pandas as pd  # For data manipulation and analysis
import numpy as np   # For numerical operations
import matplotlib.pyplot as plt  # For data visualization
import matplotlib.colors as colors  # For customizing plot colors
import seaborn as sns  # For data visualization
import warnings # To ignore some matplotlib warnings
warnings.filterwarnings('ignore')  # To ignore some matplotlib warnings
from sklearn.model_selection import train_test_split  # For splitting the da
from sklearn import preprocessing  # For preprocessing data (e.g., scaling)
from sklearn.svm import SVC  # Support Vector Machine classifier
from sklearn.model_selection import GridSearchCV  # For hyperparameter tunin
from sklearn.metrics import ConfusionMatrixDisplay  # For displaying confusi
from sklearn.decomposition import PCA  # Principal Component Analysis for di
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
%matplotlib inline
```

Import the data

Now we load in a dataset from the **UCI Machine Learning Repository**. Specifically, we are going to use the **Heart Disease Dataset**. This dataset will allow us to predict if someone has heart disease based on their sex, age, blood pressure and a variety of other metrics. The dataset is stored in a CSV file, which we can load into a Pandas dataframe using the `read_csv` function.

In [2]:
```
heart_data = pd.read_csv('https://raw.githubusercontent.com/marcofanti/DTSA_
```

In [3]:
```
heart_data.head()
```

Out[3]:

	id	age	sex	dataset	cp	trestbps	chol	fbs	restecg	thalc
0	1	63	Male	Cleveland	typical angina	145.0	233.0	True	lv hypertrophy	150.
1	2	67	Male	Cleveland	asymptomatic	160.0	286.0	False	lv hypertrophy	108.
2	3	67	Male	Cleveland	asymptomatic	120.0	229.0	False	lv hypertrophy	129.
3	4	37	Male	Cleveland	non-anginal	130.0	250.0	False	normal	187.
4	5	41	Female	Cleveland	atypical angina	130.0	204.0	False	lv hypertrophy	172.

In [4]:
```
heart_data.shape
```

Out[4]:
```
(920, 16)
```

In [5]:
```
heart_data.info()

heart_data['trestbps'].unique()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 920 entries, 0 to 919
Data columns (total 16 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   id        920 non-null    int64  
 1   age       920 non-null    int64  
 2   sex       920 non-null    object 
 3   dataset   920 non-null    object 
 4   cp        920 non-null    object 
 5   trestbps  861 non-null    float64
 6   chol      890 non-null    float64
 7   fbs       830 non-null    object 
 8   restecg   918 non-null    object 
 9   thalch    865 non-null    float64
 10  exang     865 non-null    object 
 11  oldpeak   858 non-null    float64
 12  slope     611 non-null    object 
 13  ca        309 non-null    float64
 14  thal      434 non-null    object 
 15  num       920 non-null    int64  
dtypes: float64(5), int64(3), object(8)
memory usage: 115.1+ KB
```

```
Out[5]: array([145., 160., 120., 130., 140., 172., 150., 110., 132., 117., 135.,
               112., 105., 124., 125., 142., 128., 170., 155., 104., 180., 138.,
               108., 134., 122., 115., 118., 100., 200.,  94., 165., 102., 152.,
               101., 126., 174., 148., 178., 158., 192., 129., 144., 123., 136.,
               146., 106., 156., 154., 114., 164.,  98., 190.,  nan, 113.,  92.,
                95.,  80., 185., 116.,   0.,  96., 127.])
```

```
In [6]: heart_data['dataset'].unique()
```

```
Out[6]: array(['Cleveland', 'Hungary', 'Switzerland', 'VA Long Beach'],
              dtype=object)
```

```
In [7]: heart_data['fbs'].unique()
```

```
Out[7]: array([True, False, nan], dtype=object)
```

```
In [8]: heart_data['ca'].unique()
```

```
Out[8]: array([ 0.,  3.,  2.,  1., nan])
```

```
In [9]: heart_data['thal'].unique()
```

```
Out[9]: array(['fixed defect', 'normal', 'reversable defect', nan], dtype=object)
```

```
In [10]: len(heart_data.loc[(heart_data['ca'].isna()) | (heart_data['thal'].isna())|
```

```
Out[10]: 621
```

```
In [11]: #Since we have 621 rows with missing values, we will drop them
         heart_data_no_missing = heart_data.dropna(subset=['ca', 'thal', 'fbs', 'tres
         heart_data_no_missing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 299 entries, 0 to 748
Data columns (total 16 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   id        299 non-null    int64
 1   age       299 non-null    int64
 2   sex       299 non-null    object
 3   dataset   299 non-null    object
 4   cp        299 non-null    object
 5   trestbps  299 non-null    float64
 6   chol      299 non-null    float64
 7   fbs       299 non-null    object
 8   restecg   299 non-null    object
 9   thalch    299 non-null    float64
 10  exang     299 non-null    object
 11  oldpeak   299 non-null    float64
 12  slope     299 non-null    object
 13  ca        299 non-null    float64
 14  thal      299 non-null    object
 15  num       299 non-null    int64
dtypes: float64(5), int64(3), object(8)
memory usage: 39.7+ KB
```

In [12]:
```
heart_data_no_missing['dataset'].unique()
```

Out[12]:
```
array(['Cleveland', 'Hungary', 'VA Long Beach'], dtype=object)
```

In [13]:
```
len(heart_data_no_missing.loc[(heart_data_no_missing['dataset'] == 'Clevelar
```

Out[13]:
```
297
```

In [14]:
```
# Since 299 rows have dataset = 'Cleveland', we will drop the others (Hungar
# We could have done this earlier and that would have avoided a lot of the m
heart_data_no_missing = heart_data_no_missing.loc[(heart_data_no_missing['da
```

In [15]:
```
heart_data_no_missing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 297 entries, 0 to 301
Data columns (total 16 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   id        297 non-null    int64
 1   age       297 non-null    int64
 2   sex       297 non-null    object
 3   dataset   297 non-null    object
 4   cp        297 non-null    object
 5   trestbps  297 non-null    float64
 6   chol      297 non-null    float64
 7   fbs       297 non-null    object
 8   restecg   297 non-null    object
 9   thalch    297 non-null    float64
 10  exang     297 non-null    object
 11  oldpeak   297 non-null    float64
 12  slope     297 non-null    object
 13  ca        297 non-null    float64
 14  thal      297 non-null    object
 15  num       297 non-null    int64
dtypes: float64(5), int64(3), object(8)
memory usage: 39.4+ KB
```

Format Data Part 1: Split the Data into Dependent and Independent Variables

Data Preparation for Classifiers

Now that we have handled the missing data, our next step is to format the data for working with the different classifiers.

Data Splitting

The first step in this process involves splitting the dataset into two parts:

1. The columns of data that will be used for making classifications (referred to as **X**).
2. The column of data that we want to predict, in this case, it's **hd** (heart disease), which will be represented as **y**.

It's important to note that dealing with missing data before splitting ensures that each row in **X** corresponds correctly with the appropriate value in **y**.

Copying Data

In the code below, we use the `copy()` method to create a copy of the data *by value* instead of *by reference*. This ensures that the original data `df_no_missing` remains unaltered when we manipulate `X` or `y`. This approach allows us to experiment with formatting columns for classification without affecting the original dataset. If any mistakes occur during the process, we can simply re-copy `df_no_missing` rather than reloading the original data and reapplying missing value handling.

In [16]:
```
X = heart_data_no_missing.drop('num', axis=1).drop('dataset', axis=1).copy()
X.head()
```

Out[16]:

	id	age	sex	cp	trestbps	chol	fbs	restecg	thalch	exang
0	1	63	Male	typical angina	145.0	233.0	True	lv hypertrophy	150.0	False
1	2	67	Male	asymptomatic	160.0	286.0	False	lv hypertrophy	108.0	True
2	3	67	Male	asymptomatic	120.0	229.0	False	lv hypertrophy	129.0	True
3	4	37	Male	non-anginal	130.0	250.0	False	normal	187.0	False
4	5	41	Female	atypical angina	130.0	204.0	False	lv hypertrophy	172.0	False

In [17]:
```
y = heart_data_no_missing['num'].copy()
y.head()
```

Out[17]:
```
0    0
1    2
2    1
3    0
4    0
Name: num, dtype: int64
```

Format Data Part 2: Convert Categorical Data to Dummy Variables

Why Convert Categorical Data?

Many machine learning algorithms can only handle numerical data. Categorical data, however, is often represented by non-numerical labels, making it incompatible with these algorithms directly. Converting categorical data to dummy variables solves this issue by creating new binary features for each category within the original variable.

Benefits of Dummy Variables:

Improved Model Performance: By converting categorical data to numerical representation, models can efficiently analyze and learn from the information contained within these features. Interpretability: Dummy variables provide clear information about the presence or absence of each category, aiding in model interpretation and understanding the influence of categorical features on predictions. Consistency: By using a consistent numerical representation, different models can handle categorical data in a similar manner, facilitating comparisons and analyses. Types of Dummy Variables:

One-Hot Encoding: This method creates a new binary feature for each category. The feature is set to 1 for instances belonging to that category and 0 otherwise.

In [18]:
```
X_encoded = pd.get_dummies(X, columns=['sex',
                                       'cp',
                                       'restecg',
                                       'slope',
                                       'thal'])
```

In [19]:
```
X_encoded.head()
```

Out[19]:

	id	age	trestbps	chol	fbs	thalch	exang	oldpeak	ca	sex_Female	...	cp_t a
0	1	63	145.0	233.0	True	150.0	False	2.3	0.0	False	...	
1	2	67	160.0	286.0	False	108.0	True	1.5	3.0	False	...	
2	3	67	120.0	229.0	False	129.0	True	2.6	2.0	False	...	
3	4	37	130.0	250.0	False	187.0	False	3.5	0.0	False	...	
4	5	41	130.0	204.0	False	172.0	False	1.4	0.0	True	...	

5 rows × 24 columns

In [20]:
```
y.unique()
```

Out[20]:
```
array([0, 2, 1, 3, 4])
```

In doing heart disease prediction, we only care about classifying individuals as having or not having the disease. This necessitates a simplified target variable representation.

Therefore, we'll convert all values in the target variable > 0 to 1. This effectively translates any degree of heart disease into a binary presence/absence indicator, making it suitable for binary classification capabilities.

In [21]:
```
y_not_zero_idx = y > 0  # get the indices for which y is not zero
y[y_not_zero_idx] = 1  # set y at those indices to 1
y.unique()
```

Out[21]:
```
array([0, 1])
```

Univariate Selection for Feature Selection

In feature selection, univariate statistical tests can help identify features that have the strongest relationships with the target variable. The scikit-learn library offers the `SelectKBest` class, which allows us to select a specific number of features using various statistical tests.

For instance, in the following example, we utilize the chi-squared (`chi2`) statistical test, designed for non-negative features, to select the top 10 features with the most significant relationships with the target variable from the dataset.

```
In [22]: from sklearn.feature_selection import SelectKBest
         from sklearn.feature_selection import chi2

         X_drop = X_encoded.drop(columns=['id']).copy()
         # Select features
         selector = SelectKBest(score_func=chi2, k=10) # Select 10 best features

         fit = selector.fit(X_drop,y)
         dfscores = pd.DataFrame(fit.scores_)
         dfcolumns = pd.DataFrame(X_drop.columns)
         #concat two dataframes for better visualization
         featureScores = pd.concat([dfcolumns,dfscores],axis=1)
         featureScores.columns = ['Specs','Score']  #naming the dataframe columns
         print(featureScores.nlargest(10,'Score'))  #print best features
```

```
                     Specs       Score
4                   thalch  187.053104
7                       ca   82.730613
6                  oldpeak   68.570533
22  thal_reversable defect   42.750381
10         cp_asymptomatic   39.848093
21             thal_normal   36.654197
5                    exang   35.508090
0                      age   22.917697
19         slope_upsloping   22.888726
2                     chol   20.855084
```

Correlation Matrix with Heatmap

A correlation matrix provides insights into how features are related to each other and to the target variable. Correlations can be either positive (an increase in one feature corresponds to an increase in the target variable) or negative (an increase in one feature corresponds to a decrease in the target variable).

Using a heatmap, we can easily visualize the features that are most relevant to the target variable. To create the heatmap, we'll utilize the seaborn library.

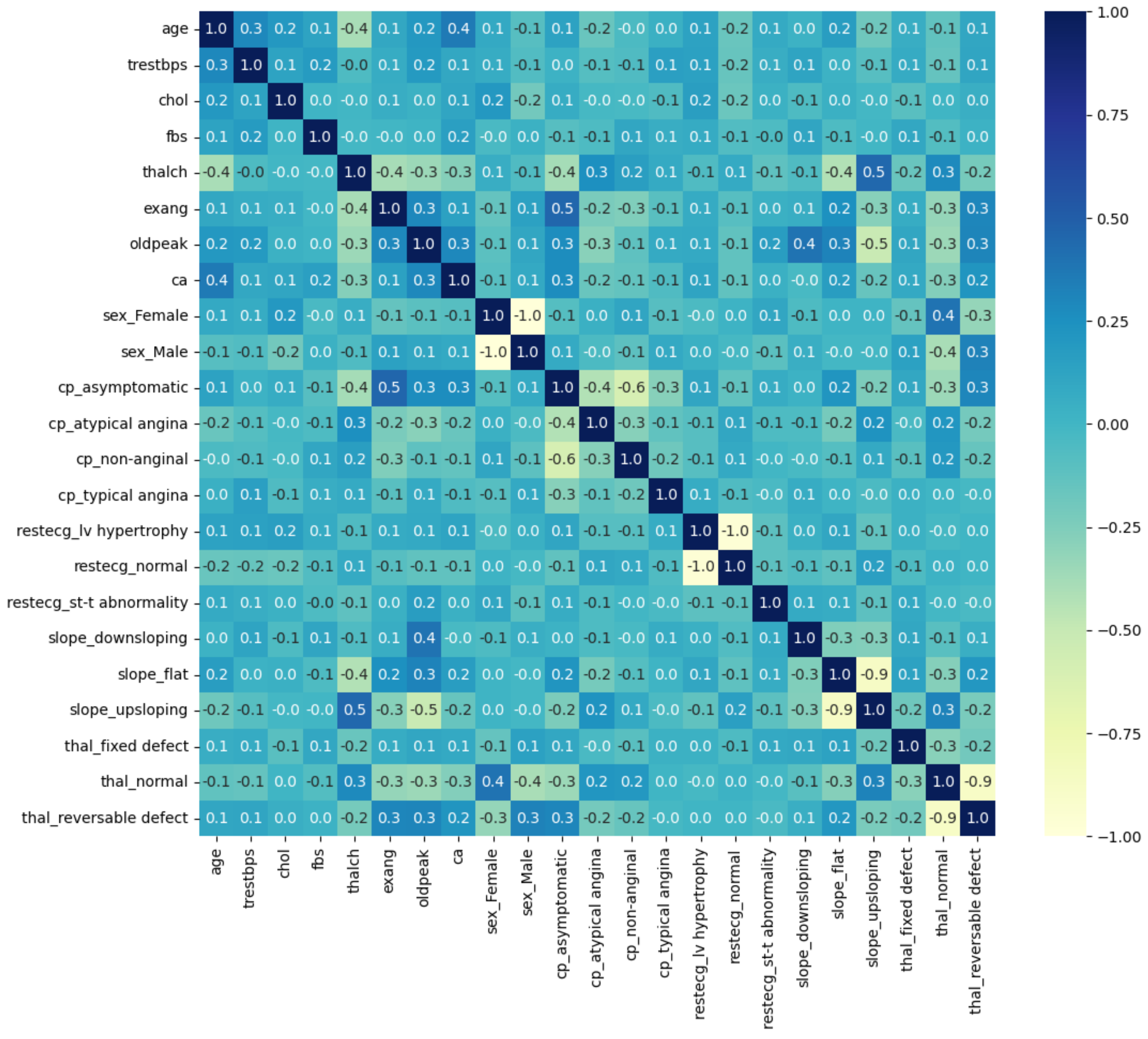
Correlation helps us understand the relationships between features and the target variable. Positive correlations imply that an increase in a feature corresponds to an increase in the target variable, while negative correlations imply the opposite.

From the heatmap, we can observe that the 'cp' (chest pain) feature exhibits a strong positive correlation with the target variable. This suggests that chest pain has a significant influence on predicting the presence of heart disease. In comparison to the relationships between other variables, we can conclude that chest pain is the most important factor in predicting the presence of heart disease.

A medical emergency, such as a heart attack, typically occurs when a blood clot obstructs blood flow to the heart. This leads to a decrease in oxygen supply to the heart tissue, causing chest pain.

In [23]:
```
plt.figure(figsize=(12,10))
sns.heatmap(X_drop.corr(),annot=True,cmap="YlGnBu",fmt='.1f')
```

Out[23]:
```
<Axes: >
```



Data Formatting Part 3: Centering and Scaling

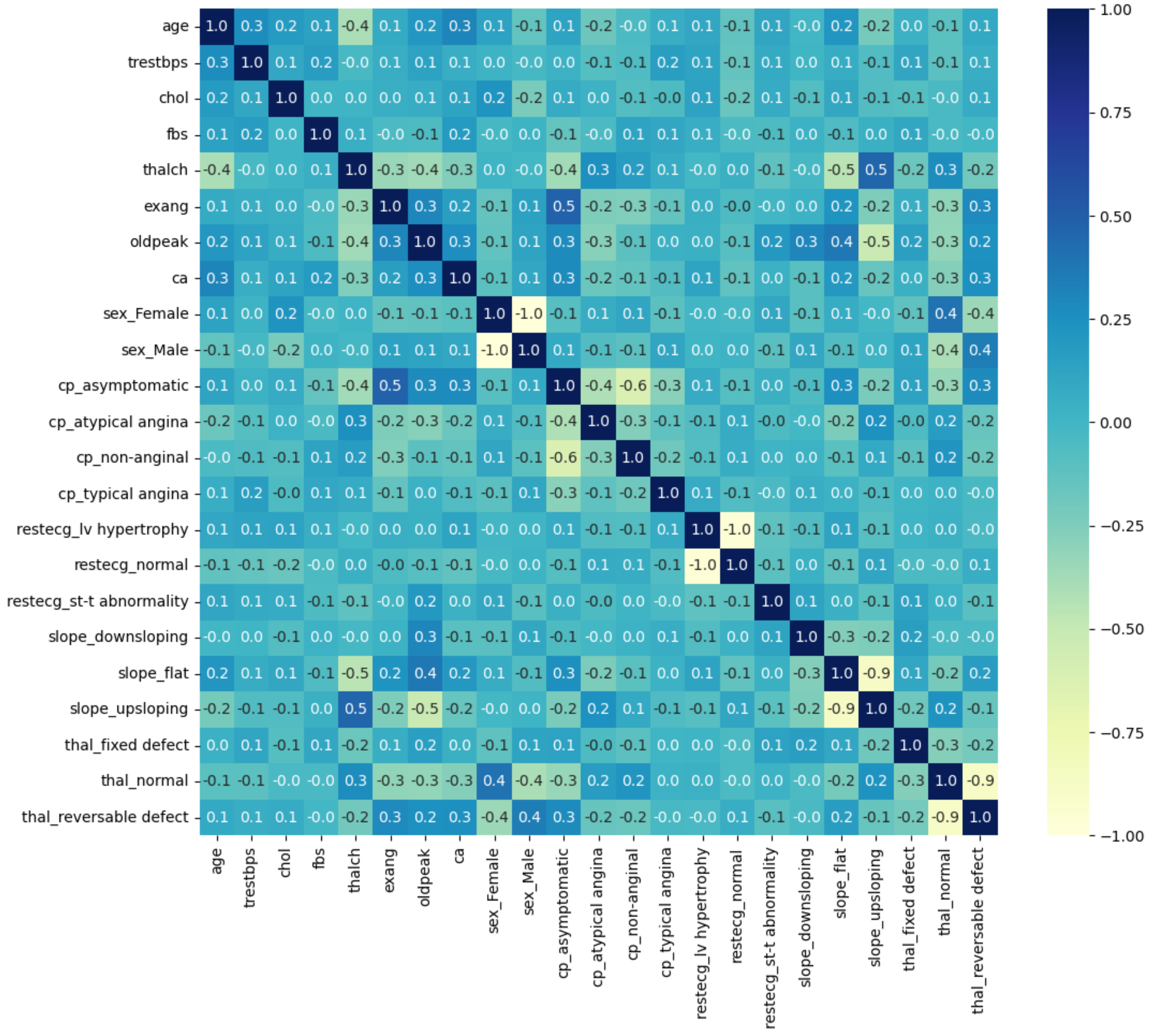
For our **Support Vector Machine (SVM)**, we are utilizing the **Radial Basis Function (RBF)**, which assumes that the data are centered and scaled. Therefore, it is necessary to perform centering and scaling on both the training and testing datasets.

IMPORTANT: To prevent **Data Leakage**, we split the data into training and testing datasets before scaling them. **Data Leakage** can occur when information from the training dataset contaminates or influences the testing dataset.

```
In [24]: X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, random_sta
         scaler = preprocessing.StandardScaler().fit(X_train)
         X_train_scaled = scaler.transform(X_train)
         X_test_scaled = scaler.transform(X_test)
```

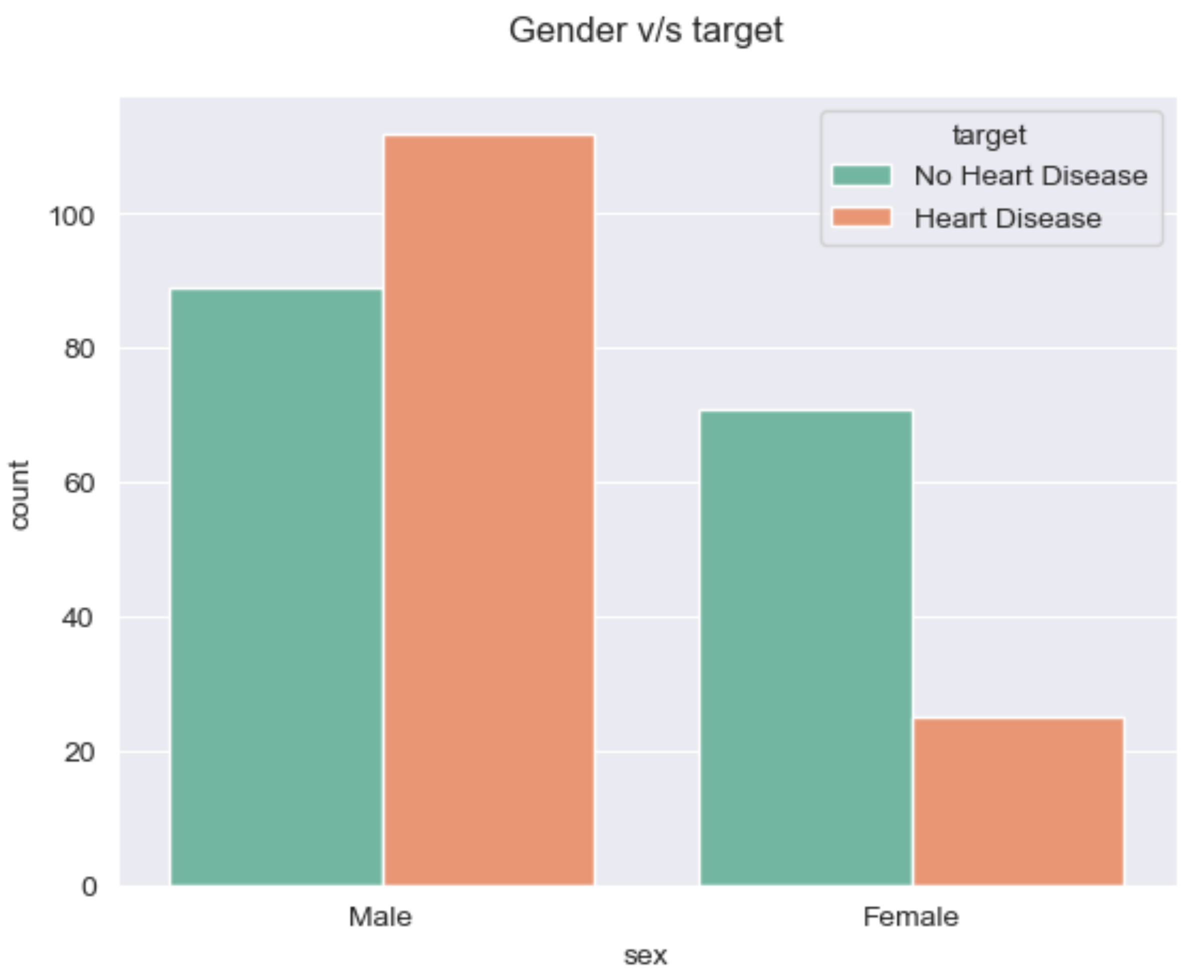
```
In [25]: plt.figure(figsize=(12,10))
         X_train_drop = X_train.drop(columns=['id']).copy()
         sns.heatmap(X_train_drop.corr(),annot=True,cmap="YlGnBu",fmt='.1f')
```

```
Out[25]: <Axes: >
```



In [26]:
```
sns.set_style('darkgrid')
sns.set_palette('Set2')

heart_data_no_missing_2 = heart_data_no_missing.drop(columns=['id']).copy()
def chng2(prob):
    if prob == 0:
        return 'No Heart Disease'
    else:
        return 'Heart Disease'
heart_data_no_missing_2['target'] = heart_data_no_missing_2['num'].apply(chr
sns.countplot(data= heart_data_no_missing_2, x='sex',hue='target')
plt.title('Gender v/s target\n')
```

Out[26]:
```
Text(0.5, 1.0, 'Gender v/s target\n')
```



Gender and Heart Disease

In the Cleveland dataset, it is evident that males are more susceptible to developing heart disease compared to females. Men have a higher incidence of heart attacks than women. Sudden heart attacks are experienced by a significant proportion of men, ranging from 70% to 89%.

In contrast, women may experience heart attacks without the typical symptom of chest pressure. Instead, they often report symptoms such as nausea or vomiting, which can be easily mistaken for acid reflux or flu-like symptoms.

Types of Chest Pain and Asymptomatic Heart Attacks

In the context of heart disease, there are four types of chest pain:

1. **Asymptomatic Chest Pain:** Most heart disease patients are found to have asymptomatic chest pain. These individuals may display atypical symptoms such as indigestion, flu-like symptoms, or the sensation of a strained chest muscle. Despite being asymptomatic, this type of chest pain involves the blockage of blood flow to the heart, potentially causing damage to the heart muscle.

2. **Atypical Angina:** This type of chest pain is characterized by symptoms that do not fit the typical pattern of angina. It may be experienced as discomfort or pain in the chest, but the symptoms differ from the classic angina presentation.

3. **Non-Anginal Pain:** Non-anginal chest pain refers to chest discomfort that is not related to the heart or angina. It can be caused by various factors unrelated to cardiac issues.

4. **Typical Angina:** Typical angina is chest pain that follows a typical pattern and is usually related to heart-related issues.

Risk factors for asymptomatic heart attacks are similar to those associated with heart symptoms. These factors include:

- Age
- Diabetes

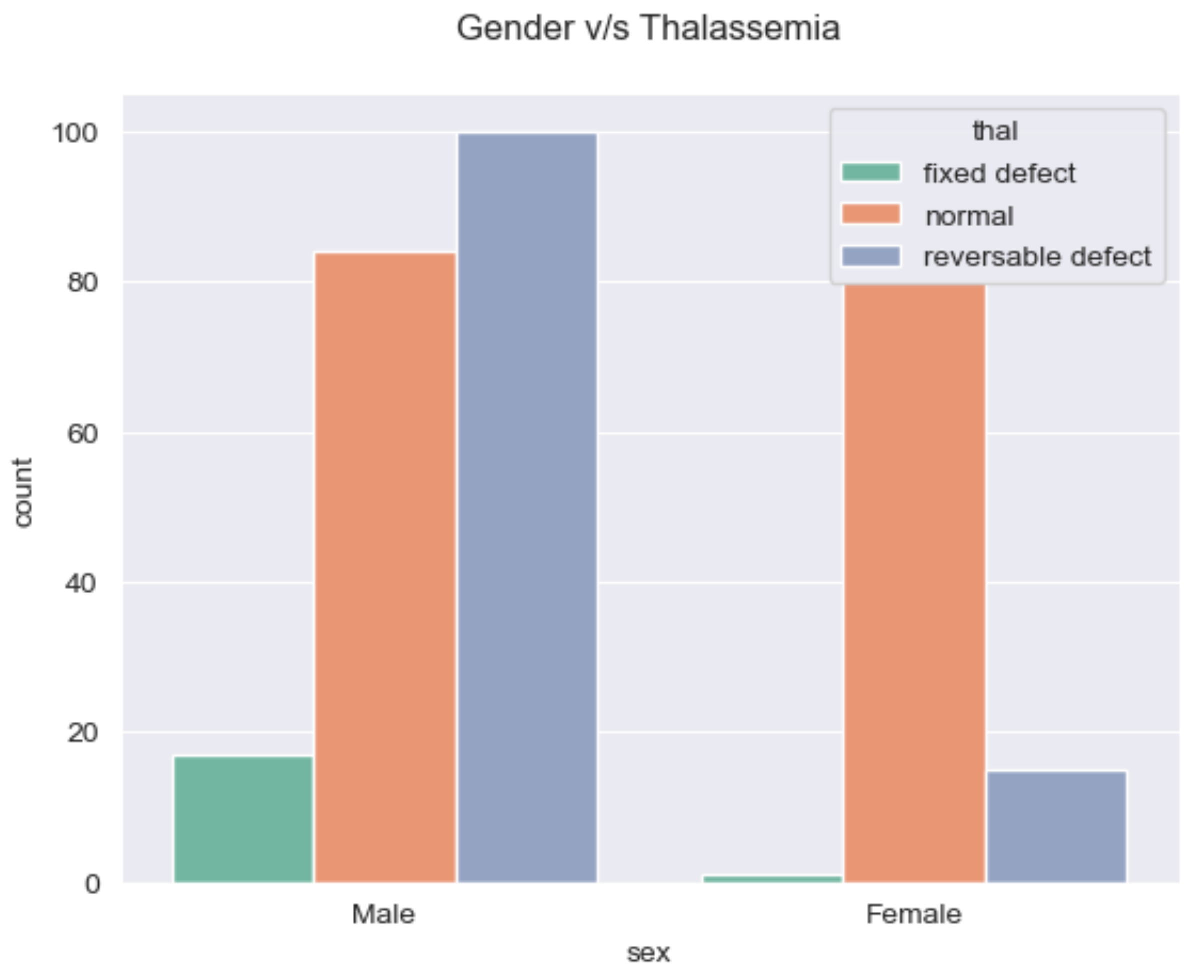
- Excess weight
- Family history of heart disease
- High blood pressure
- High cholesterol
- Lack of exercise
- Prior heart attack
- Tobacco use

It's important to note that an asymptomatic heart attack puts individuals at a greater risk of experiencing another heart attack, which can be life-threatening. Additionally, having another heart attack increases the risk of complications, such as heart failure.

Currently, there are no specific tests to predict the potential for an asymptomatic heart attack. The only way to confirm if an asymptomatic heart attack has occurred is through diagnostic tests such as an electrocardiogram (ECG) or echocardiogram, which can reveal changes indicating a previous heart attack.

In [27]:
```
sns.countplot(data= heart_data_no_missing_2, x='sex',hue='thal')
plt.title('Gender v/s Thalassemia\n')
print('Thalassemia (thal-uh-SEE-me-uh) is an inherited blood disorder that c
```

```
Thalassemia (thal-uh-SEE-me-uh) is an inherited blood disorder that causes yo
ur body to have less hemoglobin than normal. Hemoglobin enables red blood cel
ls to carry oxygen
```



Beta Thalassemia Cardiomyopathy

Beta thalassemia cardiomyopathy is primarily characterized by two distinct phenotypes:

1. **Dilated Type:** This type is characterized by left ventricular dilatation and impaired contractility.

2. **Restrictive Type:** The restrictive phenotype involves restrictive left ventricular filling, pulmonary hypertension, and right heart failure.

Individuals with severe thalassemia can experience various heart-related issues, including congestive heart failure and abnormal heart rhythms.

Auxiliary function

Compiles the results of many machine learning classifiers into a single dataframe for analysis

In [28]:
```
def add_to_classifier_summary_df(classifier_summary_df, classifier, classifi
    # Compiles the results of many machine learning classifiers into a singl
    if classifier_summary_df is None:
        classifier_summary_df = pd.DataFrame(columns=["Classifier Name", "Ac

    y_test_pred = classifier.predict(X_test)

    classifier_summary_entry = {"Classifier Name": [classifier_name],
                                "Accuracy": [accuracy_score(y_test, y_test_pred)],
                                "Precision": [precision_score(y_test, y_test_pred,
                                "Recall": [recall_score(y_test, y_test_pred, avera
                                "Macro F1": [f1_score(y_test, y_test_pred, average
                                }

    return pd.concat([classifier_summary_df, pd.DataFrame(classifier_summary
```

Creating a Pairplot with Seaborn

In data analysis and visualization, a pairplot is a valuable tool for exploring relationships between multiple variables in a dataset. We can create a pairplot using the `seaborn.pairplot` function, which is a part of the Seaborn library.

A pairplot generates a grid of scatterplots, where each variable in the dataset is compared to every other variable. The main diagonal of the grid typically displays histograms or kernel density estimates for each variable, showing the distribution of individual features.

Here's what a pairplot can help us achieve:

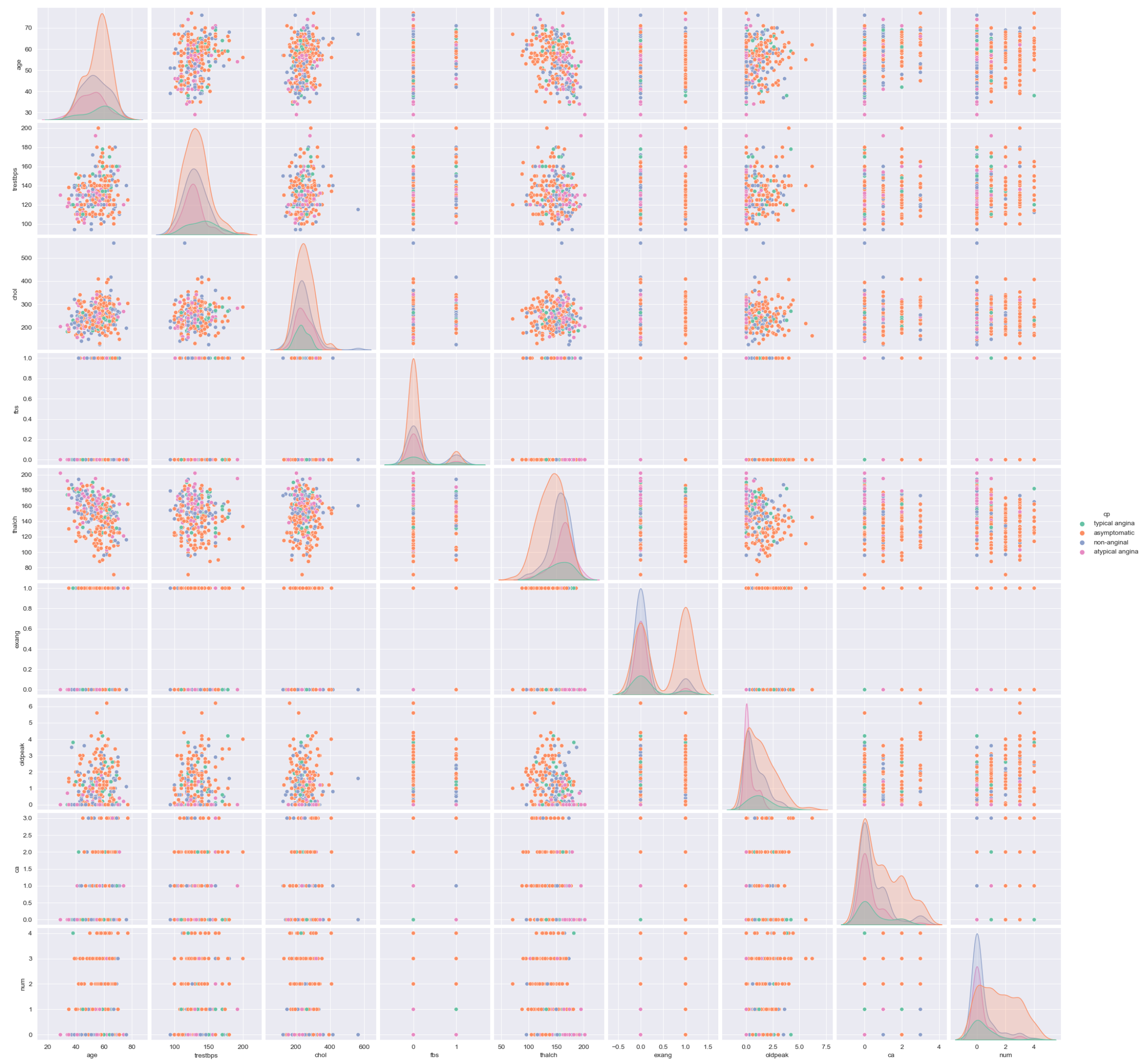
1. **Scatterplots:** The off-diagonal cells of the grid contain scatterplots, which allow us to visualize the relationship between pairs of variables. Each point in a scatterplot represents a data point, and the position of the point on the plot indicates the values of the two variables being compared. Scatterplots help us identify patterns, correlations, and potential outliers in the data.

2. **Diagonal Plots:** The diagonal cells contain histograms (or kernel density estimates) for each variable. These histograms provide insights into the distribution of each individual feature, helping us understand its central tendency and spread.

By examining the scatterplots and diagonal plots collectively, we can gain a comprehensive understanding of how variables relate to each other and their individual distributions. Pairplots are particularly useful for initial data exploration, identifying potential correlations, and selecting relevant features for further analysis or modeling.

Let's use the `seaborn.pairplot` function to create a pairplot and explore the relationships between variables in our dataset.

In [29]:
```
#sns.pairplot(heart_data_no_missing_2,hue='cp')
#plt.savefig('images/pairplot.png')
```



Logistic Regression

Logistic Regression is a fundamental and widely used statistical technique in the field of machine learning. It is primarily used for binary classification tasks, where the goal is to predict one of two possible outcomes (e.g., yes/no, true/false, 1/0) based on input features. Let's dive into the details and expand on the concept of Logistic Regression:

Mathematical Foundation

- At its core, Logistic Regression uses the logistic function (sigmoid function) to model the relationship between input features and the probability of a binary outcome. The logistic function maps any real-valued number to a value between 0 and 1, making it suitable for estimating probabilities.

Hypothesis Function

- In Logistic Regression, we formulate a hypothesis function that takes the form of the logistic function. It calculates the probability that a given input instance belongs to the positive class (class 1).

  ### Model Training

  - The logistic regression model is trained using labeled training data. During training, the model learns the optimal coefficients (weights) for each input feature, as well as a bias term. These coefficients are adjusted to minimize a loss function (typically log loss or cross-entropy) that quantifies the difference between predicted probabilities and actual class labels.

Decision Boundary

- Logistic Regression generates a decision boundary that separates the two classes in feature space. This boundary is a linear function of input features when using a simple logistic regression model. In more complex scenarios, such as multi-class classification or non-linear relationships, extensions like multinomial logistic regression or polynomial logistic regression are used.

Probability Threshold

- Logistic Regression produces probabilities, not discrete class labels. To make predictions, a probability threshold is applied (commonly 0.5). Instances with predicted probabilities greater than the threshold are assigned to the positive class, while those below the threshold are assigned to the negative class.

Evaluation Metrics

- To assess the performance of a Logistic Regression model, various evaluation metrics can be used, including accuracy, precision, recall, F1-score, and ROC curves. These metrics provide insights into the model's ability to correctly classify instances and its overall performance.

Regularization

- Logistic Regression can be regularized to prevent overfitting. Common regularization techniques include L1 regularization (Lasso) and L2 regularization (Ridge), which add penalty terms to the loss function to constrain the magnitudes of the coefficients.

Use Cases

- Logistic Regression finds applications in a wide range of domains, including medical diagnosis, spam detection, credit scoring, and sentiment analysis, to name a few. It is especially useful when the outcome variable is binary, and the goal is to model the probability of an event.

Logistic Regression is a powerful and interpretable algorithm that serves as a foundational building block in the world of machine learning. Understanding its principles and applications is essential for anyone working in data science and predictive modeling.

In [30]:
```
from sklearn.linear_model import LogisticRegression
logre = LogisticRegression()
logre.fit(X_train_scaled,y_train)
```

Out[30]:
```
▾ LogisticRegression
LogisticRegression()
```

In [31]:
```
y_pred = logre.predict(X_test_scaled)
actual = []
prediction = []
for i,j in zip(y_test,y_pred):
    actual.append(i)
    prediction.append(j)
dic = {'Actual':actual,
       'Prediction':prediction
       }
result  = pd.DataFrame(dic)
import plotly.graph_objects as go

fig = go.Figure()


fig.add_trace(go.Scatter(x=np.arange(0,len(y_test)), y=y_test,
                         mode='markers+lines',
                         name='Test'))
fig.add_trace(go.Scatter(x=np.arange(0,len(y_test)), y=y_pred,
                         mode='markers',
                         name='Pred'))

#fig.write_image('images/logistic_regression.png')
fig.show()
```

Visualization of Predicted vs. Actual Values

In the visualization:

- The **red dots** represent the predicted values, which can be either 0 or 1.
- The **blue line** and **blue dot** represent the actual values corresponding to specific patients.

Where the **red dot** and **blue dot** do not overlap, it indicates incorrect predictions. Conversely, where both the **red dot** and **blue dot** overlap, those are the instances where the model made correct predictions.

In [32]:
```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test,y_pred))

classifier_summary_df = add_to_classifier_summary_df(None, logre, "Logistic
```

```
0.8933333333333333
```

In [33]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.93      0.91        42
           1       0.90      0.85      0.88        33

    accuracy                           0.89        75
   macro avg       0.89      0.89      0.89        75
weighted avg       0.89      0.89      0.89        75
```

Interpretation of Classification Metrics

In the provided classification metrics, we have several key performance indicators for a binary classification model. Let's break down and expand upon each of these metrics:

Precision

- **Precision** measures the accuracy of positive predictions made by the model. In this context, it indicates how many of the predicted positive (class 1) instances are actually true positives. A higher precision means that the model is better at avoiding false positives.

Recall

- **Recall**, also known as sensitivity or true positive rate, quantifies the model's ability to correctly identify all positive instances in the dataset. It measures the percentage of actual positive (class 1) instances that the model correctly predicted as positive. A higher recall means that the model is better at capturing all actual positives.

F1-Score

- The **F1-Score** is the harmonic mean of precision and recall. It provides a balanced measure of a model's accuracy in both precision and recall. A higher F1-Score indicates a good balance between precision and recall.

Support

- **Support** refers to the number of samples in each class (0 and 1) in the test dataset.

In this case:
  - Class 0 has 42 instances.
  - Class 1 has 33 instances.

Accuracy

- **Accuracy** represents the overall correctness of the model's predictions. It calculates the ratio of correctly predicted instances (both true positives and true negatives) to the total number of instances in the dataset. In this case, the model's accuracy is 0.88, which means it correctly predicts 88% of the test dataset.

Macro Average

- The **macro average** is the average of precision, recall, and F1-Score across both classes (0 and 1). It provides an overall summary of the model's performance without considering class imbalances.

Weighted Average

- The **weighted average** is similar to the macro average but takes class imbalances into account. It gives more weight to the class with a larger number of samples. In this case, it considers the distribution of class 0 and class 1 instances.

In summary, these classification metrics provide a comprehensive evaluation of the model's performance in distinguishing between two classes (0 and 1). They help assess precision, recall, F1-Score, and accuracy, offering insights into the model's strengths and weaknesses in making predictions on the given test dataset.

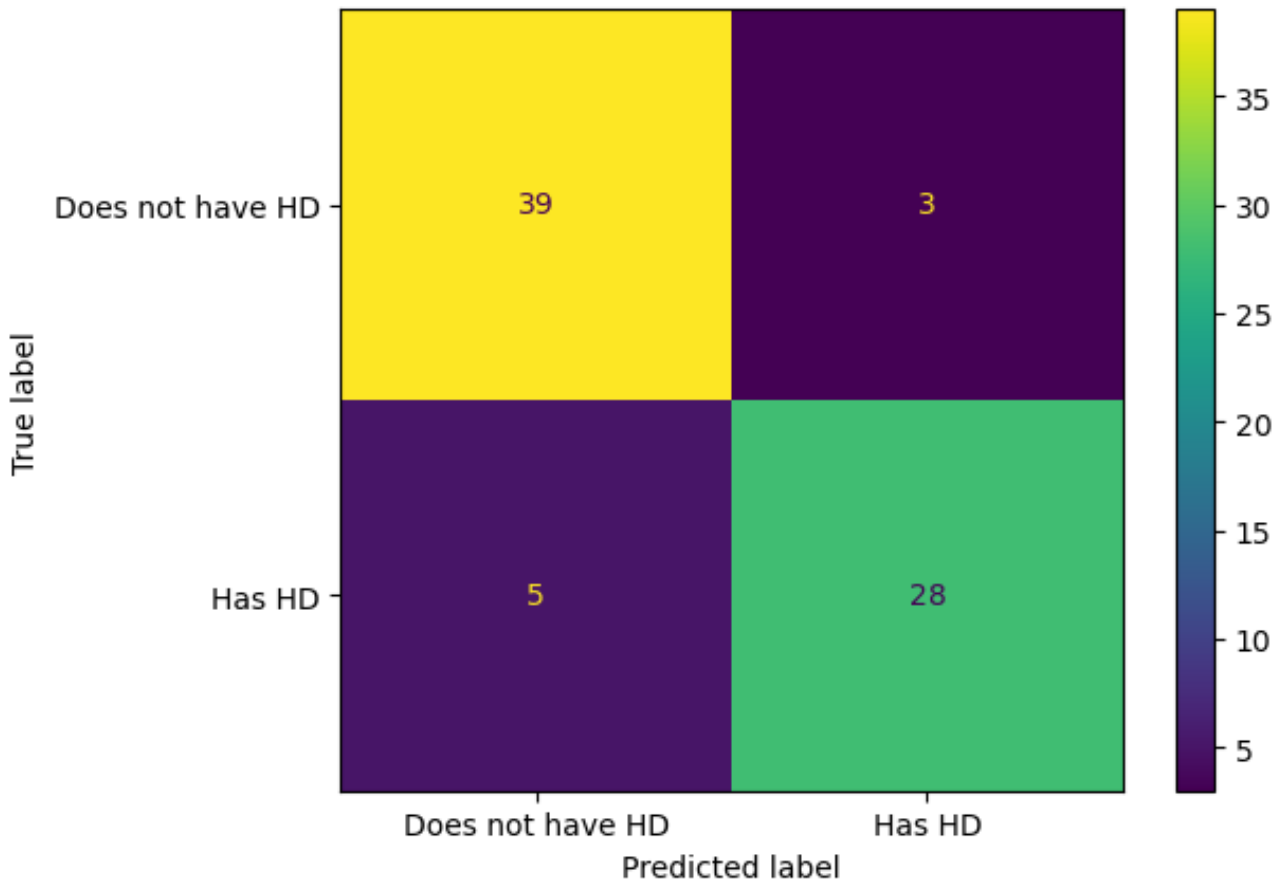
Interpretation:

The model has a good overall accuracy of 88%. For class 0 (No Heart Disease), the model achieves high precision (0.87) and recall (0.93), indicating that it is good at correctly identifying both positive and negative examples. For class 1 (Heart Disease), the precision is slightly higher than recall, suggesting that the model is more likely to correctly identify true positives than miss actual cases. The F1-score, which considers both precision and recall, is also high for both classes, indicating that the model is performing well overall. The macro and weighted averages are similar to the overall accuracy, suggesting that the model performs consistently across both classes.

Further Analysis: The model shows potential for diagnosing heart disease with good accuracy and performance metrics.

```
In [34]: mpl.rcdefaults()
         ConfusionMatrixDisplay.from_estimator(logre, X_test_scaled, y_test, display_
```

```
Out[34]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x286e45f
         30>
```



KNN Classifier

The K-Nearest Neighbors (KNN) classifier is a supervised machine learning technique used for classification and regression tasks. Here's a breakdown of how the KNN algorithm works:

1. **Initialization of K**: Begin by specifying the number of neighbors (K) that the model should consider when making predictions. K represents the count of nearest data points that will influence the prediction.

2. **Prediction Process**: When the KNN model receives an input (real-world data point) as a query, it aims to predict whether the individual has heart disease or not. To do

this, it calculates the distance or similarity between the new, unseen data point and all other records in the training dataset.

3. **Distance Calculation**: The model computes the distance (e.g., Euclidean distance, Manhattan distance, etc.) between the query data point and each data point in the training set. This distance measurement quantifies how similar or dissimilar the query point is to other data points in the feature space.

4. **Sorting by Distance**: After calculating the distances, the model sorts the examples in ascending order, placing the data points with the shortest distances at the beginning of the collection.

5. **K-Nearest Data Points**: The KNN algorithm selects the first K entries from the sorted collection. These are the K nearest data points to the query point, based on the calculated distances.

6. **Classification or Regression**: The algorithm's behavior depends on the task:

   - For regression tasks, KNN returns the "mean" of the target values associated with the selected K entries. This average serves as the predicted value for the query point.
   - For classification tasks, KNN returns the "mode" of the K class labels from the selected entries. In other words, it determines the most frequently occurring class label among the K nearest neighbors. This mode becomes the predicted class for the query point.
7. **Model Accuracy Score**: Finally, to assess how well the KNN model performs in real-world scenarios, you can calculate its accuracy score. This metric measures the model's ability to make correct predictions. It is typically done by comparing the predicted results to the actual outcomes in a test dataset.

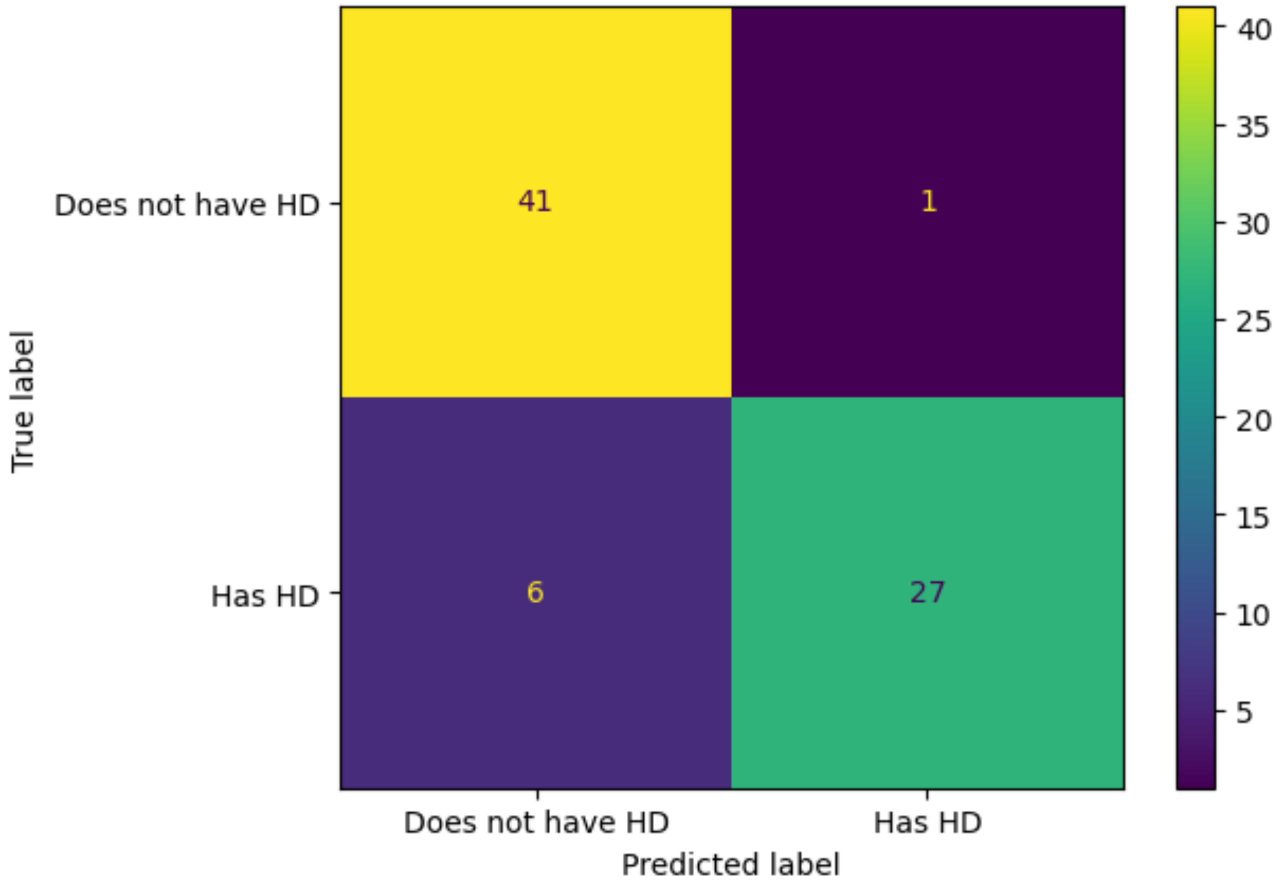
The KNN algorithm is intuitive and relies on the idea that similar data points in the feature space tend to have similar outcomes. By considering the K nearest neighbors, the algorithm makes predictions based on the majority class or average value among those neighbors, making it a versatile and straightforward machine learning approach.

In [35]:
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_scaled,y_train)
y_pred = knn.predict(X_test_scaled)
print("Accuracy:", accuracy_score(y_test, y_pred))
classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

```
Accuracy: 0.9066666666666666
```

In [36]:
```
mpl.rcdefaults()
ConfusionMatrixDisplay.from_estimator(knn, X_test_scaled, y_test, display_la
```

Out[36]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1065384
c0>
```

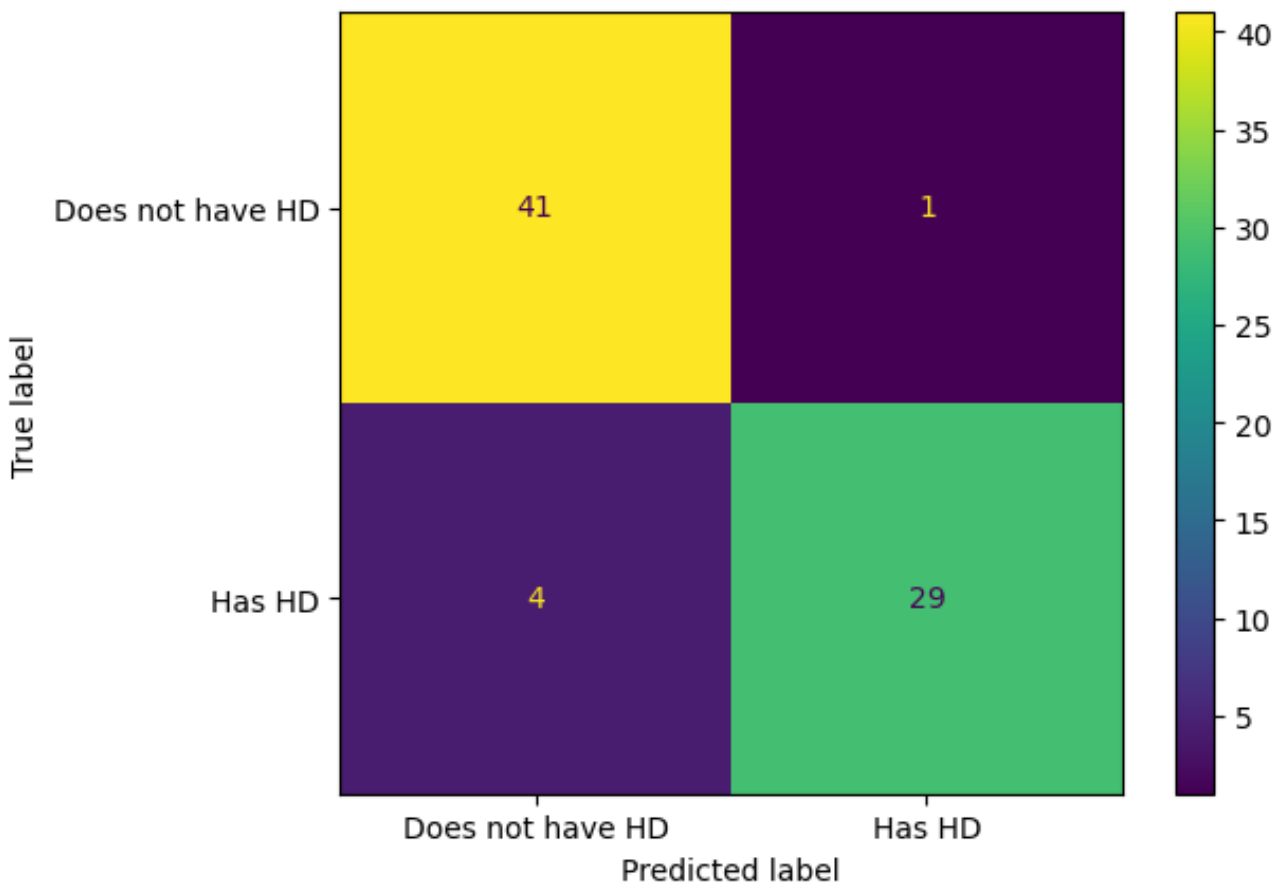


In [37]:
```
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train_scaled,y_train)
y_pred = knn.predict(X_test_scaled)
print("Accuracy:", accuracy_score(y_test, y_pred))

classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

```
Accuracy: 0.9333333333333333
```

In [38]:
```
mpl.rcdefaults()
ConfusionMatrixDisplay.from_estimator(knn, X_test_scaled, y_test, display_la
```

Out[38]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2984856
00>
```


41
1
4
29
Does not have HD
Has HD
True label
Does not have HD
Has HD
Predicted label
40
35
30
25
20
15
10
5

Decision Tree Classifier

In machine learning, a Decision Tree Classifier is a powerful algorithm used for both classification and regression tasks. It is a tree-like structure where each internal node represents a feature (attribute), each branch represents a decision rule, and each leaf node represents an outcome (class label or predicted value). Decision trees are interpretable models that can be visualized to understand the decision-making process.

To create a Decision Tree Classifier in Python, we can use the `DecisionTreeClassifier` class from the scikit-learn library (imported as `sklearn.tree`).

Here are the key steps to create a Decision Tree Classifier:

1. **Data Preparation:** Ensure that your dataset is properly cleaned, preprocessed, and split into training and testing sets.

2. **Import Libraries:** Import the necessary libraries, including `DecisionTreeClassifier` from `sklearn.tree`.

3. **Instantiate the Classifier:** Create an instance of the `DecisionTreeClassifier` class, optionally specifying hyperparameters such as the maximum depth of the tree, the minimum number of samples required to split a node, and others.

4. **Model Training:** Fit the Decision Tree Classifier to the training data using the `.fit()` method. The classifier learns the decision rules from the training data.

5. **Model Evaluation:** After training, you can evaluate the model's performance on a separate testing dataset using various metrics such as accuracy, precision, recall, F1-score, and more.

6. **Visualization (Optional):** Decision trees can be visualized using graph visualization tools like Graphviz or by utilizing the built-in plotting capabilities of scikit-learn.

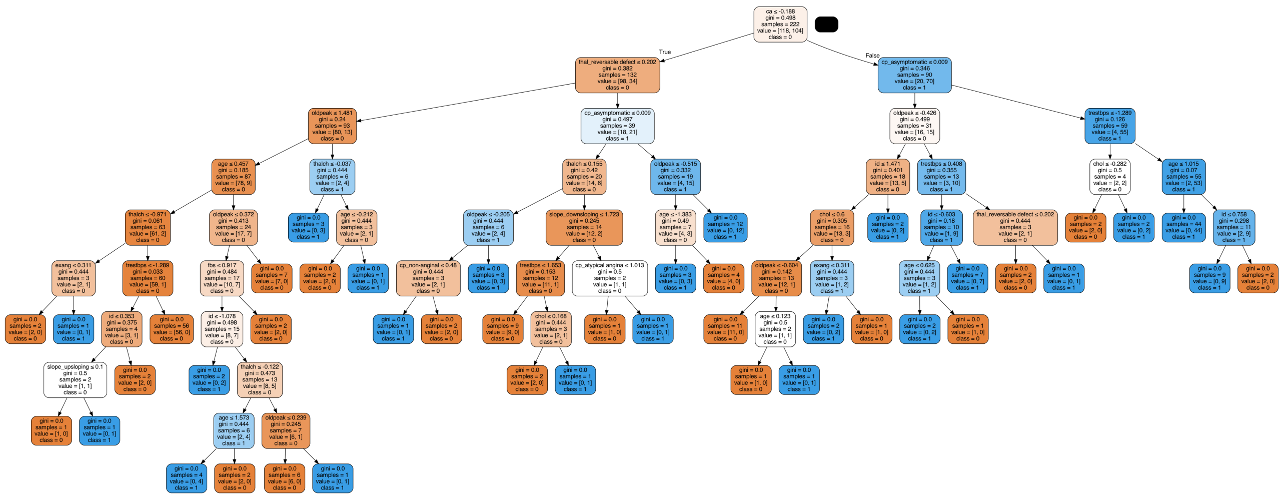
7. **Prediction:** Once trained, you can use the Decision Tree Classifier to make predictions on new, unseen data using the `.predict()` method.

```
In [39]: from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Class
         from sklearn import metrics #Import scikit-learn metrics module for accuracy

         # Create Decision Tree classifier object
         clf = DecisionTreeClassifier()
         # Train Decision Tree Classifier
         clf = clf.fit(X_train_scaled,y_train)
         #Predict the response for test dataset
         y_pred = clf.predict(X_test_scaled)
         print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

         classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

```
Accuracy: 0.7733333333333333
```

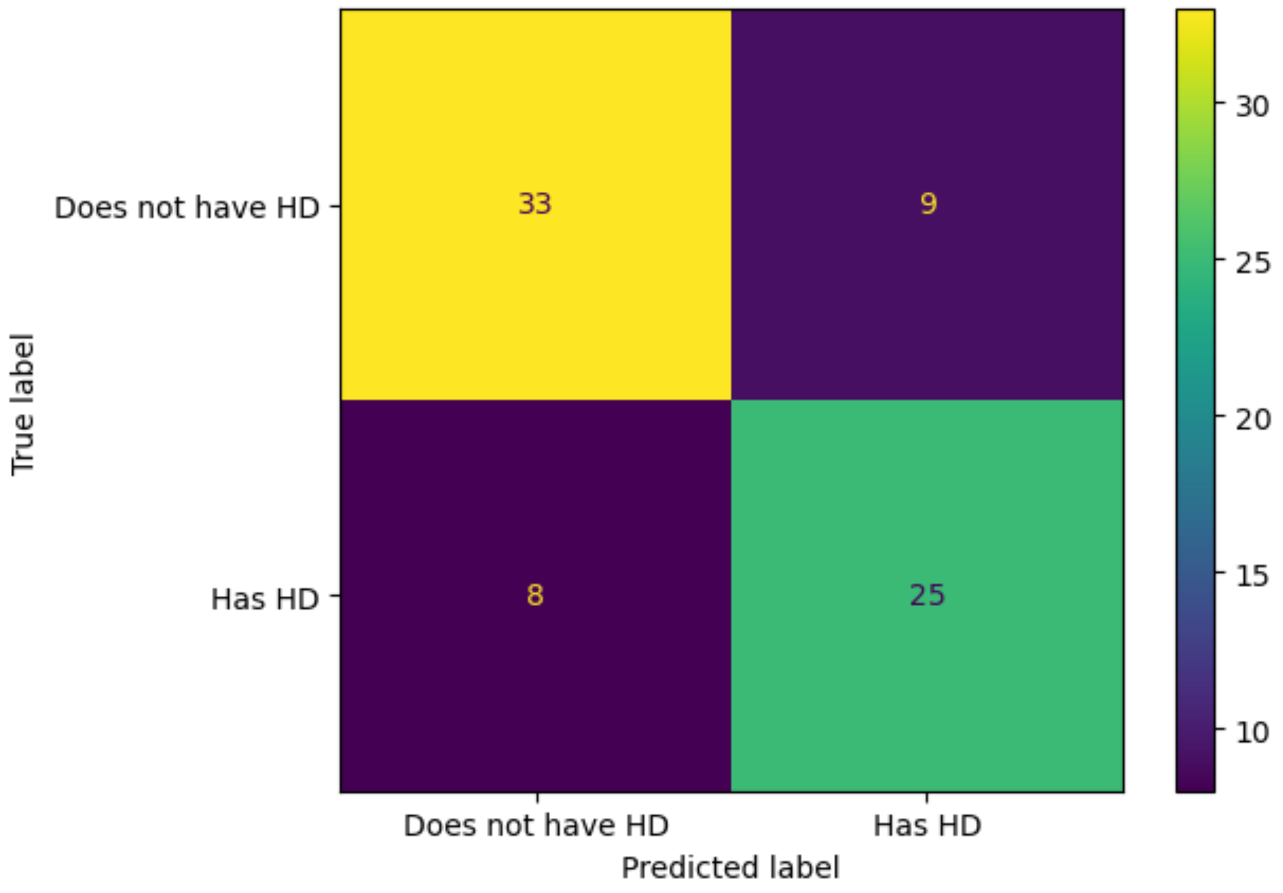
```
In [42]: feature_cols = ['id', 'age', 'trestbps', 'chol', 'fbs', 'thalch', 'exang', '
                         'ca', 'sex_Female', 'sex_Male', 'cp_asymptomatic', 'cp_atypi
                         'cp_non-anginal', 'cp_typical angina', 'restecg_lv hypertrop
                         'restecg_normal', 'restecg_st-t abnormality', 'slope_downslo
                         'slope_flat', 'slope_upsloping', 'thal_fixed defect', 'thal_
                         'thal_reversable defect']

         from sklearn.tree import export_graphviz
         from six import StringIO
         from IPython.display import Image
         import pydotplus
         dot_data = StringIO()
         export_graphviz(clf, out_file=dot_data,
                         filled=True, rounded=True,
                         special_characters=True,feature_names = feature_cols   ,class
         graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
         #graph.write_png('heart-decision-tree.png')
         Image(graph.create_png())
```

Out[42]:



In [43]:
```
mpl.rcdefaults()
ConfusionMatrixDisplay.from_estimator(clf, X_test_scaled, y_test, display_la
```

Out[43]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x29f5590
30>
```



Hyperparameter Explanation

In the code snippet, we specify two critical hyperparameters when creating the Decision Tree Classifier:

Criterion ("entropy" vs. "gini")

The `criterion` hyperparameter determines the method used to measure the quality of splits made by the decision tree. In this specific case, we've chosen `"entropy"` as the criterion:

- **Entropy Criterion**: It quantifies the amount of information disorder or randomness in a dataset. The decision tree aims to minimize entropy by making splits that result in more homogeneous groups (i.e., groups with similar class labels). In simpler terms, the entropy criterion measures the information gain associated with a particular split. It is well-suited for decision trees when you want to maximize the information gained at each split.

Another commonly used option is `"gini"`:

- **Gini Impurity**: Gini impurity measures the probability of incorrectly classifying a randomly chosen element's class label. Similar to entropy, the decision tree seeks to reduce Gini impurity by making splits that result in more homogeneous groups. The "gini" criterion is another valid choice for measuring the quality of splits, especially when the goal is classification purity.

The choice between "entropy" and "gini" often depends on the specific problem and the dataset. In practice, both criteria work well, and you may experiment with both to determine which one yields better results for your task.

Max Depth (Preventing Overfitting)

The `max_depth` hyperparameter plays a crucial role in controlling the complexity of the decision tree. It represents the maximum depth or levels the tree can grow to during training. In this code, we set `max_depth=3`, meaning the tree is limited to a depth of 3.

- **Preventing Overfitting**: Limiting the tree's depth is a common technique to prevent overfitting. Overfitting occurs when the tree becomes too complex and fits the training data too closely, capturing noise rather than genuine patterns. By restricting the depth, the model becomes less likely to overfit because it is simpler and captures more generalized relationships in the data.

The choice of an appropriate `max_depth` value depends on the dataset's complexity and the trade-off between model simplicity and performance. A smaller `max_depth` value results in a simpler model, while a larger value allows the model to capture more intricate patterns in the data. Tuning this hyperparameter is an essential part of optimizing decision tree models.

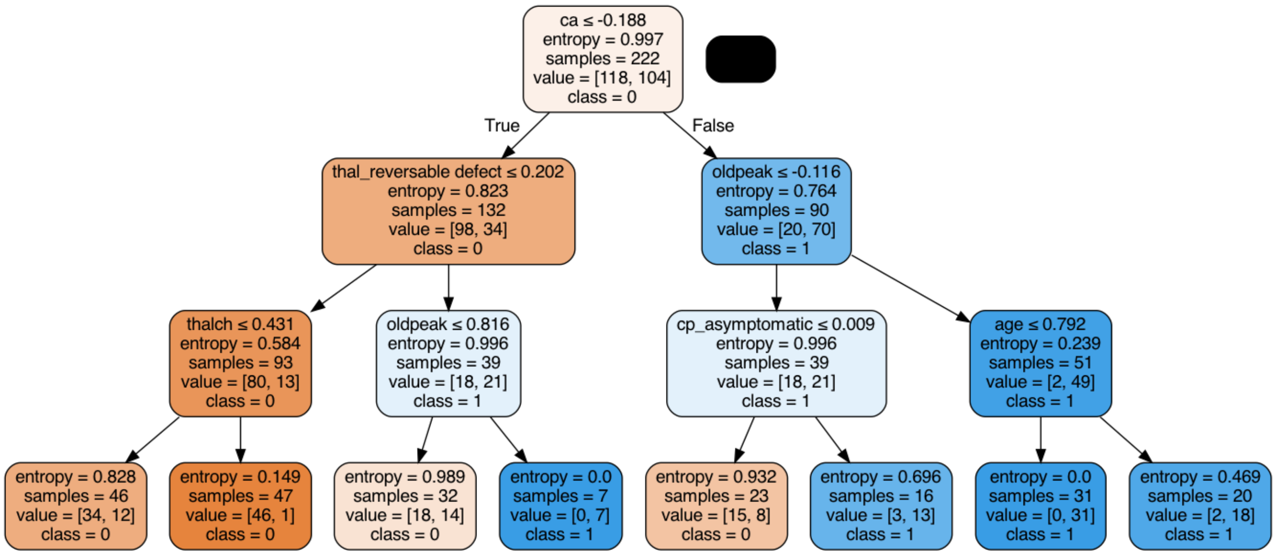
In summary, when configuring a Decision Tree Classifier, the choice of hyperparameters like `criterion` and `max_depth` influences how the tree makes decisions and controls its complexity. Careful consideration of these hyperparameters is vital to achieving the best performance for a given problem.

In [44]:
```
# Create Decision Tree Classifier object
clf = DecisionTreeClassifier(criterion="entropy", max_depth=3)
# Train Decision Tree Classifier
clf = clf.fit(X_train_scaled,y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_test_scaled)
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

```
Accuracy: 0.76
```

In [45]:
```
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,feature_names = feature_cols  ,class
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('heart-decision-tree-2.png')
Image(graph.create_png())
```

Out[45]:



Random Forests

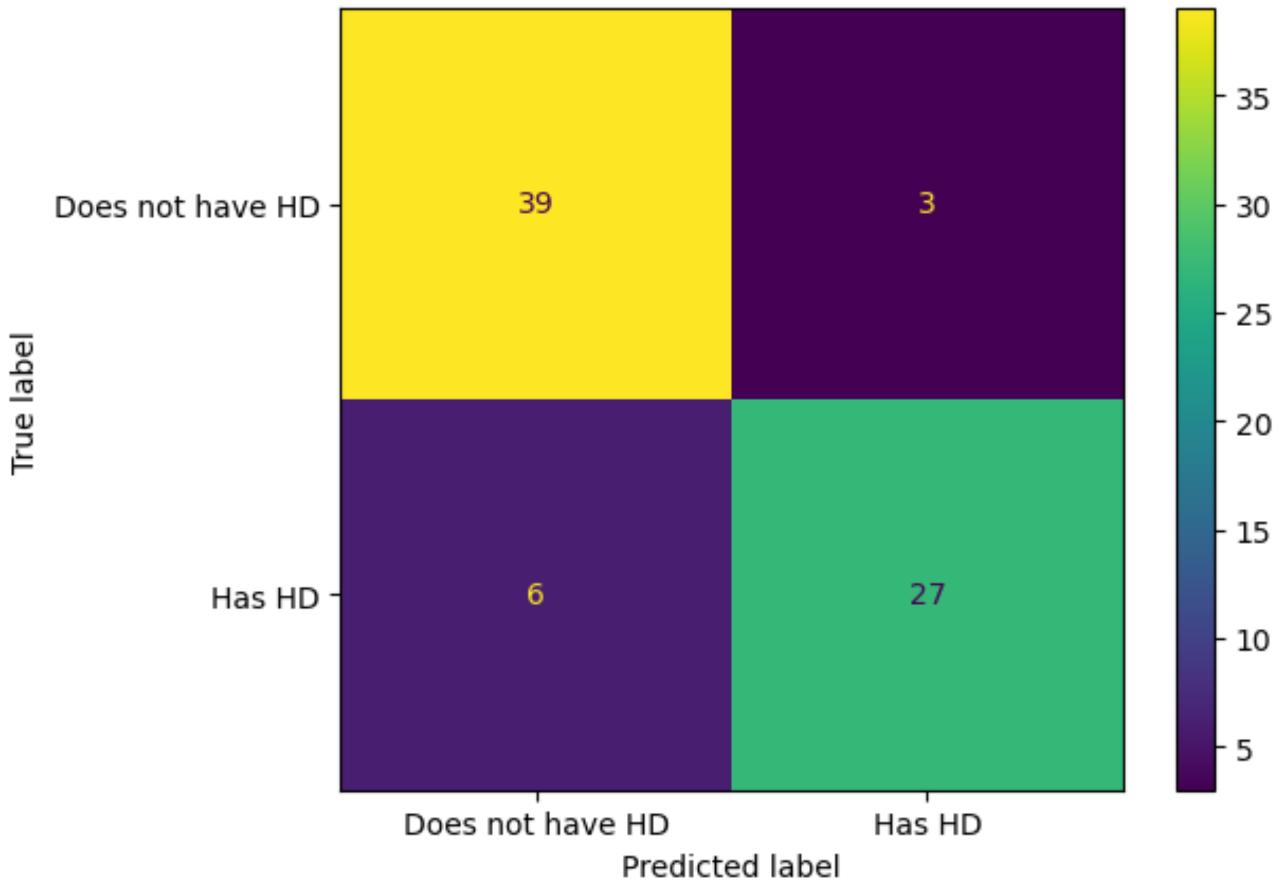
In [46]:
```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_estimators=100) # , max_depth=5, random_state
rfc.fit(X_train_scaled, y_train)
y_pred = rfc.predict(X_test_scaled)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
Accuracy: 0.88
```

In [47]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.93      0.90        42
           1       0.90      0.82      0.86        33

    accuracy                           0.88        75
   macro avg       0.88      0.87      0.88        75
weighted avg       0.88      0.88      0.88        75
```

In [48]:
```
mpl.rcdefaults()
ConfusionMatrixDisplay.from_estimator(rfc, X_test_scaled, y_test, display_la
```

Out[48]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x29f559b
10>
```



Building a Preliminary Support Vector

Machine (SVM)

After correctly formatting the data, we are now ready to build a **Support Vector Machine (SVM)**. Let's proceed with it!

In machine learning, a Support Vector Machine (SVM) is a powerful algorithm used for both classification and regression tasks. Building a preliminary SVM involves several essential steps, which we will expand upon:

Instantiating the SVM Classifier

Once the data is ready and libraries are imported, you can instantiate the SVM classifier. You may need to specify hyperparameters, such as the type of kernel (e.g., linear, polynomial, radial basis function), regularization parameters (C), and others, depending on your problem and dataset characteristics.

Model Training

The next step is to train the SVM classifier using the training dataset. The SVM learns the optimal decision boundary that best separates the different classes in the data. The choice of kernel and hyperparameters plays a critical role in this learning process.

Model Evaluation

After training the SVM, it's essential to evaluate its performance. Common evaluation metrics for classification tasks include accuracy, precision, recall, F1-score, and confusion matrix analysis. These metrics help assess how well the SVM can make predictions on unseen data.

Hyperparameter Tuning

Fine-tuning hyperparameters is often necessary to optimize the SVM model's performance. You can perform hyperparameter tuning using techniques like cross-validation or grid search to find the best combination of hyperparameters that yield the highest predictive accuracy.

Visualization (Optional)

Visualizing the SVM's decision boundary can be insightful. For linear SVMs, you can plot the decision boundary in two dimensions, while for non-linear kernels, it may involve more complex visualizations.

Prediction

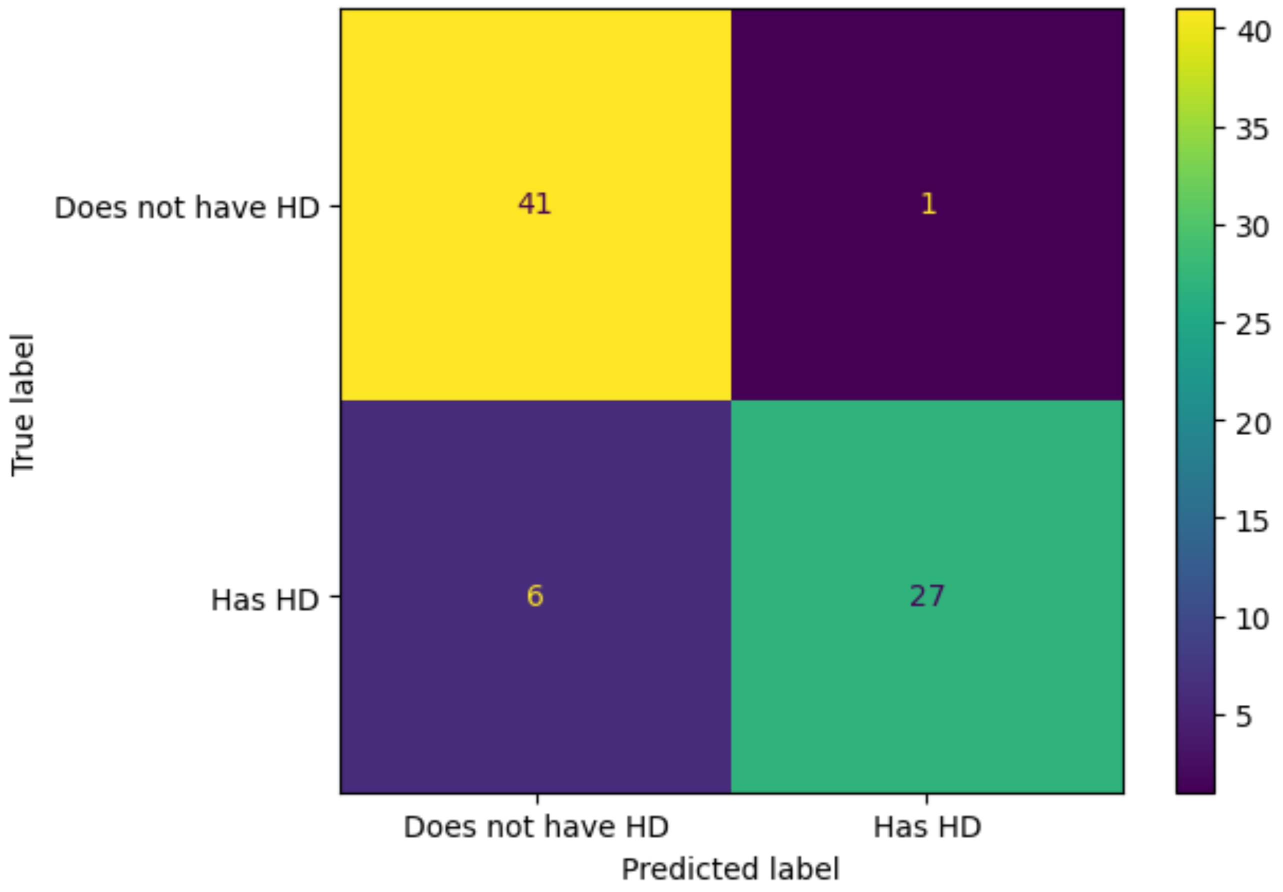
Once your SVM model is trained and validated, you can use it to make predictions on new, unseen data. This step is crucial for deploying the model in real-world applications.

Building a preliminary Support Vector Machine is a fundamental process in machine learning that requires careful data preparation, model instantiation, training, evaluation, and optimization. The resulting SVM model can be a valuable tool for solving classification and regression problems across various domains.

```
In [49]: clf_svm = SVC(random_state=42)
         clf_svm.fit(X_train_scaled, y_train)
         classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

```
In [50]: mpl.rcdefaults()
         ConfusionMatrixDisplay.from_estimator(clf_svm, X_test_scaled, y_test, displa
```

```
Out[50]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2864318
         10>
```



In [51]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.93      0.90        42
           1       0.90      0.82      0.86        33

    accuracy                           0.88        75
   macro avg       0.88      0.87      0.88        75
weighted avg       0.88      0.88      0.88        75
```

In [52]:
```
num_features = np.size(X_train_scaled, axis=1)
param_grid = [
    {'C': [0.1, 0.01, 10, 100, 1000],
     'gamma': [1/num_features, 1, 0.1, 0.01, 0.001, 0.0001],
     'kernel': ['rbf']},
]
optimal_params = GridSearchCV(
    SVC(),
    param_grid,
    cv=5,
    scoring='roc_auc', # NOTE: The default value for scoring results in wors
    ## For more scoring metrics see:
    ## https://scikit-learn.org/stable/modules/model_evaluation.html#scoring
    verbose=0 # NOTE: If you want to see what Grid Search is doing, set verb
)

optimal_params.fit(X_train_scaled, y_train)
print(optimal_params.best_params_)
c_opt = optimal_params.best_params_['C']
gamma_opt = optimal_params.best_params_['gamma']
kernel_opt = optimal_params.best_params_['kernel']
```

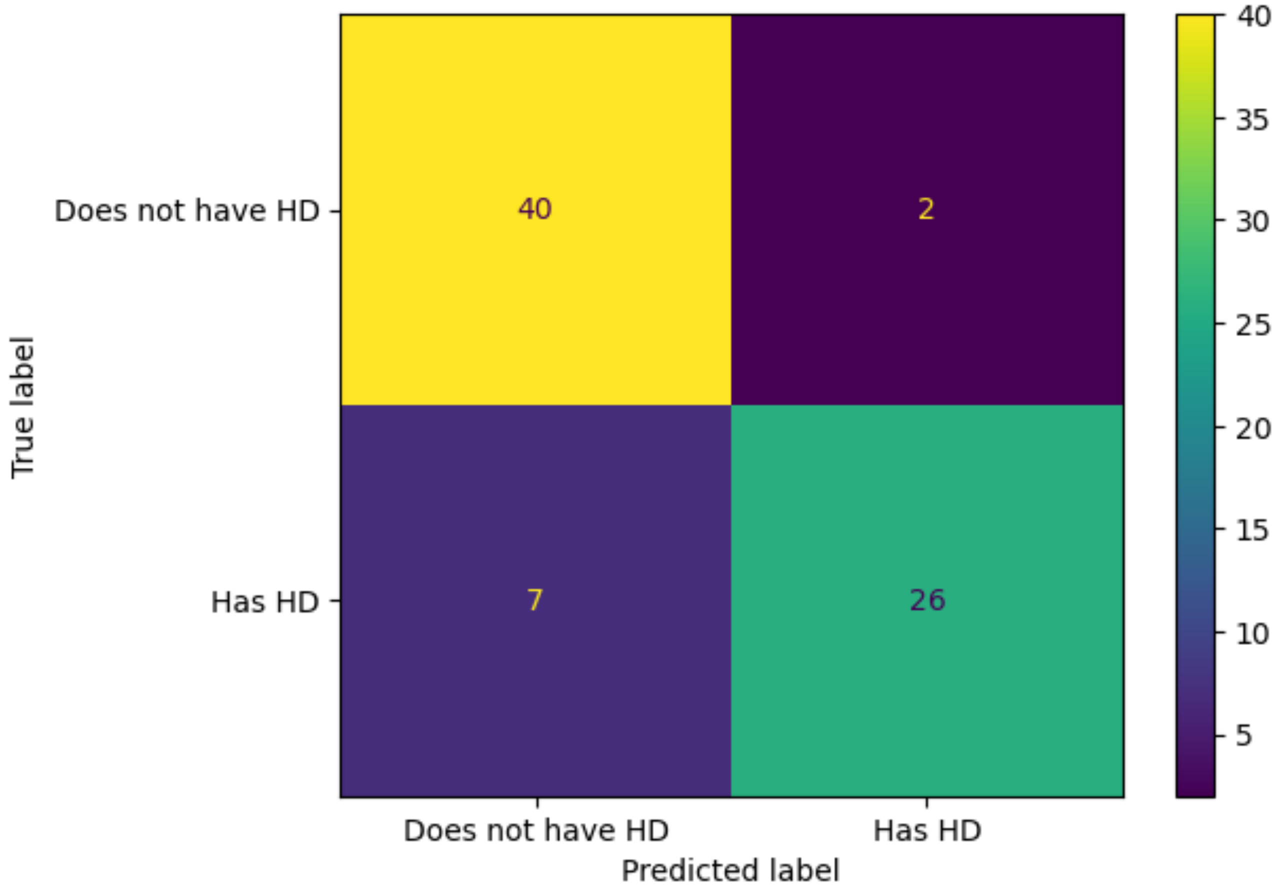
```
{'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'}
```

In [53]:
```
clf_svm = SVC(random_state=42, C=10, gamma=gamma_opt, kernel=kernel_opt)
clf_svm.fit(X_train_scaled, y_train)
```

Out[53]:
```
▾ SVC
SVC(C=10, gamma=0.01, random_state=42)
```

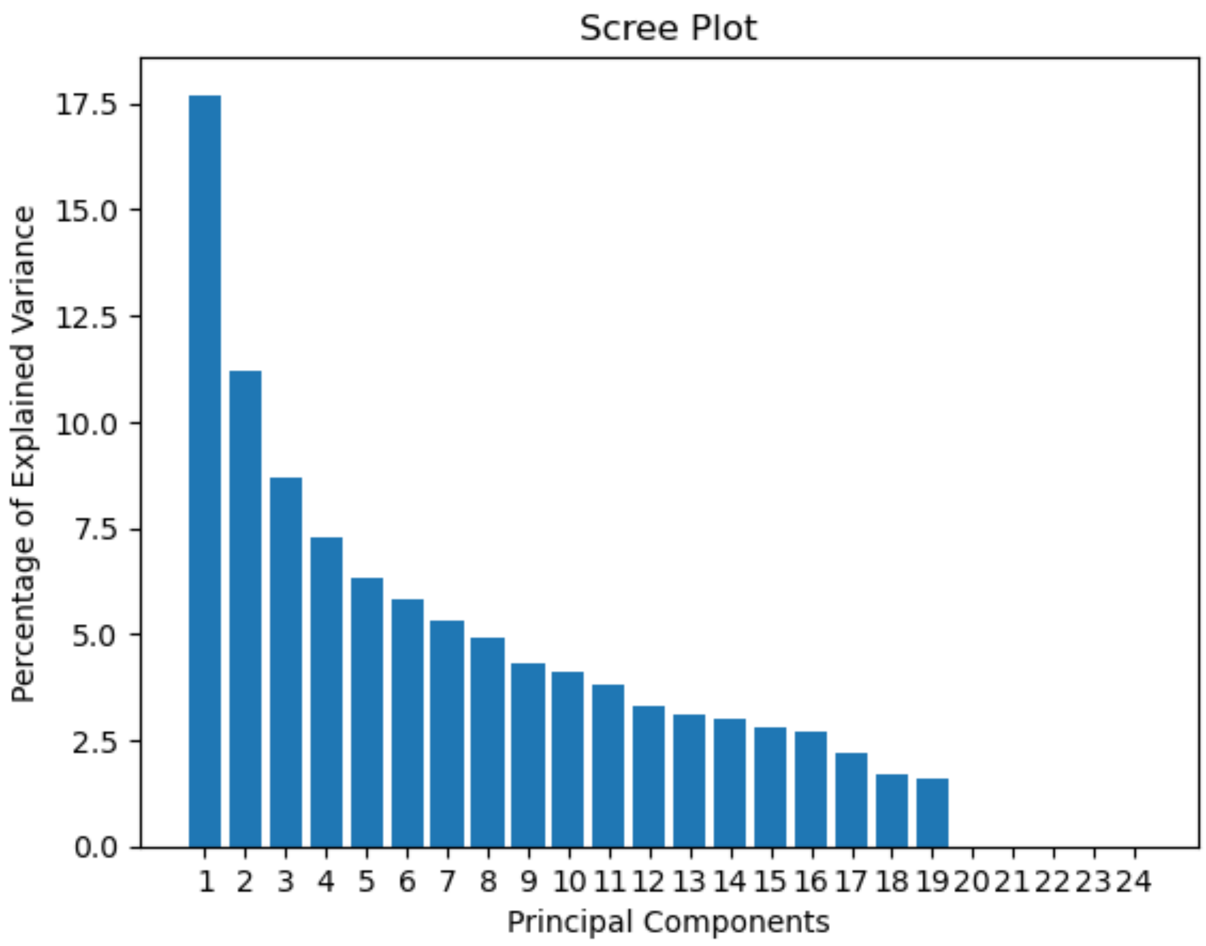
In [54]:
```
ConfusionMatrixDisplay.from_estimator(clf_svm,
                                      X_test_scaled,
                                      y_test,
                                      display_labels=["Does not have HD", "H
```

Out[54]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x29f1aab
30>
```



In [55]:
```
classifier_summary_df = add_to_classifier_summary_df(classifier_summary_df,
```

In [56]:
```
pca = PCA() # NOTE: By default, PCA() centers the data, but does not scale i
X_train_pca = pca.fit_transform(X_train_scaled)

per_var = np.round(pca.explained_variance_ratio_* 100, decimals=1)
labels = [str(x) for x in range(1, len(per_var)+1)]

plt.bar(x=range(1,len(per_var)+1), height=per_var, tick_label=labels)
plt.ylabel('Percentage of Explained Variance')
plt.xlabel('Principal Components')
plt.title('Scree Plot')
plt.show()
```


Scree Plot
Percentage of Explained Variance
Principal Components
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

In [57]:
```
train_pc1_coords = X_train_pca[:, 0]
train_pc2_coords = X_train_pca[:, 1]

## NOTE:
## pc1 contains the x-axis coordinates of the data after PCA
## pc2 contains the y-axis coordinates of the data after PCA

## Now center and scale the PCs...
pca_train_scaled = preprocessing.scale(np.column_stack((train_pc1_coords, tr

## Now we optimize the SVM fit to the x and y-axis coordinates
## of the data after PCA dimension reduction...
num_features = np.size(pca_train_scaled, axis=1)
param_grid = [
    {'C': [100, 1000],
     'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
     'kernel': ['rbf']},
]

optimal_params = GridSearchCV(
    SVC(),
    param_grid,
    cv=5,
    scoring='roc_auc', # NOTE: The default value for scoring results in wors
    ## For more scoring metics see:
    ## https://scikit-learn.org/stable/modules/model_evaluation.html#scoring
    verbose=0 # NOTE: If you want to see what Grid Search is doing, set verb
)

optimal_params.fit(pca_train_scaled, y_train)
print(optimal_params.best_params_)
#c_opt = optimal_params.best_params_['C']
#gamma_opt = optimal_params.best_params_['gamma']
#kernel_opt = optimal_params.best_params_['kernel']
```

```
{'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
```

Decision Surface Plot with SVM

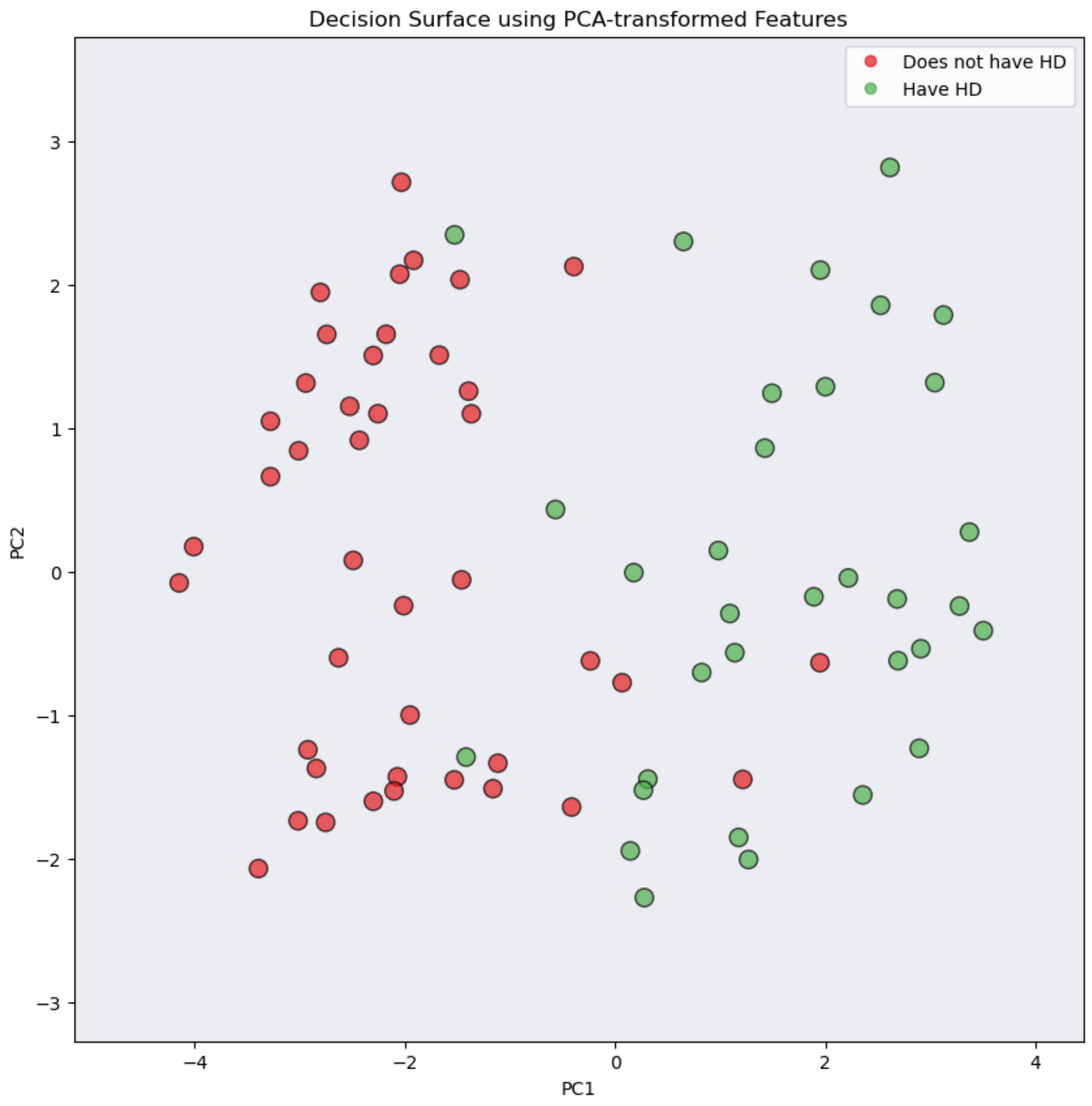
The code snippet presented below generates a decision surface plot using a Support Vector Machine (SVM) classifier. The process includes several key steps:

1. **Classifier Training**: We first train the SVM classifier, configuring its hyperparameters and fitting it to the training data.
2. **Test Data Transformation**: The test dataset is transformed using the same Principal Component Analysis (PCA) transformation applied to the training data. This ensures consistency in feature scaling and dimensionality reduction.
3. **Mesh Grid Creation**: We create a mesh grid to visualize the decision regions. The grid covers the range of possible feature values, and each point in the grid represents a potential data point.
4. **Point Classification**: The SVM classifier is used to classify each point in the mesh grid. This step involves predicting the class label (0 or 1) for every grid point based on the learned decision boundaries.
5. **Visualization**: The results are plotted, with filled contour regions representing the decision regions determined by the SVM. Additionally, the actual data points from the test dataset are displayed on the plot, colored according to their known classifications.

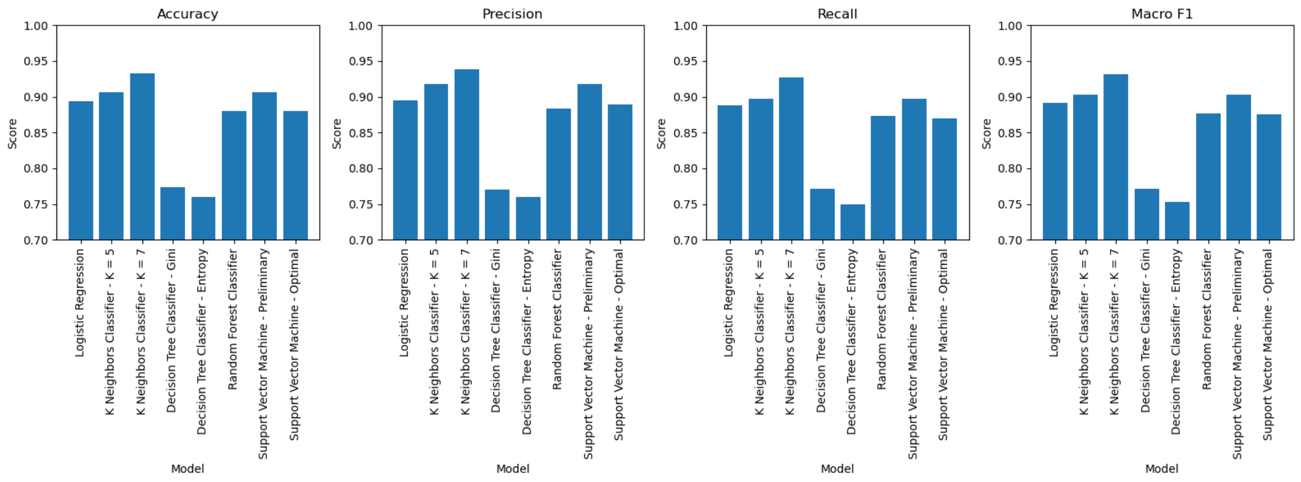
This decision surface plot serves as a visual aid to understand how the SVM model separates the two classes within the feature space, providing insights into the model's performance and the boundaries it establishes.

In [58]:
```
# Creating Decision Surface Plot with SVM

# Create an SVM classifier with specified hyperparameters
clf_svm = SVC(random_state=42, C=c_opt, gamma=gamma_opt, kernel=kernel_opt)

# Train the SVM classifier on the scaled and PCA-transformed training data
clf_svm.fit(pca_train_scaled, y_train)

# Transform the test dataset using the same PCA transformation
X_test_pca = pca.transform(X_test_scaled)
test_pc1_coords = X_test_pca[:, 0]
test_pc2_coords = X_test_pca[:, 1]
```

```
# Define the range for the decision surface matrix
x_min = test_pc1_coords.min() - 1
x_max = test_pc1_coords.max() + 1
y_min = test_pc2_coords.min() - 1
y_max = test_pc2_coords.max() + 1

# Create a mesh grid for the decision regions
xx, yy = np.meshgrid(np.arange(start=x_min, stop=x_max, step=0.1),
                     np.arange(start=y_min, stop=y_max, step=0.1))

# Classify each point in the mesh grid using the SVM
Z = clf_svm.predict(np.column_stack((xx.ravel(), yy.ravel())))

# Reshape the classification results to match the mesh grid dimensions
Z = Z.reshape(xx.shape)

# Create a figure and axis for the plot
fig, ax = plt.subplots(figsize=(10, 10))

# Plot filled contour regions based on the SVM classifications
ax.contourf(xx, yy, Z, alpha=0.1)

# Define custom colors for the actual data points
cmap = colors.ListedColormap(['#e41a1c', '#4daf4a'])

# Plot the actual data points with their known classifications
scatter = ax.scatter(test_pc1_coords, test_pc2_coords, c=y_test,
                     cmap=cmap,
                     s=100,
                     edgecolors='k',  # 'k' represents black
                     alpha=0.7)

# Create a legend for the data points
legend = ax.legend(scatter.legend_elements()[0],
                   scatter.legend_elements()[1],
                   loc="upper right")
legend.get_texts()[0].set_text("Does not have HD")
legend.get_texts()[1].set_text("Have HD")

# Add axis labels and a title to the plot
ax.set_ylabel('PC2')
ax.set_xlabel('PC1')
ax.set_title('Decision Surface using PCA-transformed Features')

# Show the plot
plt.show()
```


Decision Surface using PCA-transformed Features
Does not have HD
Have HD
PC1
PC2
−4
−2
0
2
4
−3
−2
−1
0
1
2
3

In [59]:
```
fig, axs = plt.subplots(1, 4, figsize=(16, 6))
metrics = ["Accuracy", "Precision", "Recall", "Macro F1"]

for i, metric in enumerate(metrics):
    axs[i].bar(classifier_summary_df["Classifier Name"], classifier_summary_
    axs[i].set_title(metric)
    axs[i].set_xlabel("Model")
    axs[i].set_ylabel("Score")
    axs[i].set_ylim(0.7, 1.0)
    axs[i].tick_params(axis='x', rotation=90)

plt.tight_layout()
plt.show()
```



In [60]:
```
classifier_summary_df
```

Out[60]:

	Classifier Name	Accuracy	Precision	Recall	Macro F1
0	Logistic Regression	0.893333	0.894795	0.888528	0.890988
1	K Neighbors Classifier - K = 5	0.906667	0.918313	0.897186	0.903297
2	K Neighbors Classifier - K = 7	0.933333	0.938889	0.927489	0.931582
3	Decision Tree Classifier - Gini	0.773333	0.770086	0.771645	0.770725
4	Decision Tree Classifier - Entropy	0.760000	0.759745	0.750000	0.752566
5	Random Forest Classifier	0.880000	0.883333	0.873377	0.876847
6	Support Vector Machine - Preliminary	0.906667	0.918313	0.897186	0.903297
7	Support Vector Machine - Optimal	0.880000	0.889818	0.870130	0.875668

Classifier Evaluation and Conclusion

Analysis of Machine Learning Classifier Performance

This document presents an analysis of the performance of various machine learning classifiers on a dataset. The provided table contains details such as:

- **Classifier Name**: Name of the machine learning algorithm used.
- **Accuracy**: Overall accuracy of the classifier on the testing data.
- **Precision**: Ability of the classifier to correctly identify positive cases.
- **Recall**: Ability of the classifier to correctly identify all actual positive cases.
- **F1-score**: Harmonic mean of precision and recall, providing a balanced view of classifier performance.

Based on the results, the following observations can be made:

1. **K Nearest Neighbors (KNN) with K = 7**: This classifier achieves the highest overall accuracy (0.9333) and F1-score (0.9316) among all classifiers. It also shows good performance in terms of precision (0.9389) and recall (0.9275).

2. **Support Vector Machine (SVM) - Preliminary**: This classifier achieves a similar accuracy (0.9067) and F1-score (0.9033) to KNN with K = 5. However, its precision (0.9183) and recall (0.8972) are slightly better.

3. **Logistic Regression**: This classifier shows good performance with an accuracy of 0.8933 and F1-score of 0.8909. It also demonstrates good balance between precision (0.8948) and recall (0.8885).

4. **Random Forest**: This classifier achieves a reasonable accuracy (0.8667) and F1-score (0.8625). However, its precision and recall are slightly lower than other top contenders.

5. **Decision Tree Classifiers**: Both Gini and Entropy-based decision trees perform significantly worse than other classifiers. Their accuracy and F1-score are around 0.77 and 0.75, respectively.

Conclusion

Based on the analysis, KNN with K = 7 emerges as the best performing classifier overall. It achieves the highest accuracy and F1-score, indicating its ability to accurately predict positive and negative cases while maintaining good precision and recall. SVM-Preliminary also performs well, offering a similar accuracy but slightly better precision and recall. Logistic Regression can be considered a good alternative due to its balanced performance and simplicity. Further investigation might be needed to improve the performance of Random Forest and Decision Tree classifiers.

It is important to note that the optimal classifier choice might depend on specific factors and priorities like interpretability, computational cost, and data characteristics. Additionally, these results are based on a limited dataset, and further testing with larger datasets and different data types could provide deeper insights into the classifiers' generalizability and robustness.