Leveraging Machine Learning Models to Predict Heart Attack Risk

Ben Wynia

The leading cause of death globally is cardiovascular disease, with heart attacks being a significant contributor. In the United States alone, approximately 805,000 people experience a heart attack each year, and about 1 in 4 deaths is due to heart disease. This highlights the importance of early detection and prediction of heart attack risk to prevent fatalities and improve overall cardiovascular health.

Machine learning has emerged as a valuable tool in predicting and diagnosing various diseases, including cardiovascular disease. By leveraging large datasets and advanced algorithms, machine learning models can identify patterns and predict outcomes with high accuracy. In the context of heart attack risk prediction, machine learning can analyze a wide range of variables, including demographic information, medical history, lifestyle factors, and clinical measurements.

This project aims to develop a binary classification model that predicts an individual's heart attack risk based on common measures of cardiovascular health, including blood pressure, cholesterol levels, and body mass index. The model's objective is to accurately classify individuals as either low or high risk, allowing for early intervention and prevention. The potential impact of this project is significant, as accurate heart attack risk prediction can save lives, reduce healthcare costs, and improve overall health outcomes.

Section 1: Import Libraries

In [1]:
```
# Base python libraries
import datetime
import os
import logging
import time
import os
import openai
import json
import pandas as pd
import requests
import json
import re
import copy
import math
from dataclasses import dataclass

# Pandas!
import pandas as pd
from pandas.api.types import is_string_dtype
from pandas.api.types import is_numeric_dtype
```

```
# Numerical Python!
import numpy as np

# Scientific Python!
from scipy.stats import jarque_bera

# sklearn libraries
import sklearn.tree
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
import sklearn.metrics
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, p
import sklearn.model_selection
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV, train_test_split
import sklearn.ensemble
from sklearn.model_selection import RandomizedSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier

# Visualization Libraries
import graphviz
import matplotlib.pyplot as plt
import seaborn as sns

# Statistical models
import statsmodels.api as sm
import statsmodels.formula.api as smf

# Image processing
from IPython.display import Image
from io import BytesIO

# PDF processing
from reportlab.lib.pagesizes import letter, landscape
from reportlab.lib import colors, units
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Image,  Table, Ta
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.enums import TA_CENTER, TA_JUSTIFY
import markdown2
from reportlab.platypus import Paragraph as ReportLabParagraph
from reportlab.lib.styles import ParagraphStyle as ReportLabParagraphStyle
from html import escape
from xml.sax.saxutils import unescape

# Other/random
import itertools
from functools import partial
import pickle

# Suppress
import warnings

# suppress FutureWarning
warnings.simplefilter(action='ignore', category=FutureWarning)

# import proprietary functions
pd.options.mode.chained_assignment = None
from lib import descriptives
```

```
from lib import data_cleanup
from lib import histograms
from lib import corr_map
from lib import train_test_validate
from lib import hyperparameters
from lib import analyze_classification_model
from lib import model_pdf_report
```

Section 2: Import Datasets

In [2]:
```
data_dictionary = pd.read_csv("C:/Users/bwynia/heart_attack_risk_analysis/data/data_di
data_dictionary.columns = ['Name', 'Definition']
df = pd.read_csv("C:/Users/bwynia/heart_attack_risk_analysis/data/heart.csv", low_memo
```

The measures in this dataset are important for assessing cardiovascular disease risk. Age, sex, and cholesterol are well-known risk factors for heart disease. High resting blood pressure, exercise-induced angina, and chest pain can also indicate heart disease. The number of major vessels can indicate the severity of heart disease. The maximum heart rate achieved and previous peak can indicate the patient's cardiovascular fitness. The thal rate is a measure of blood flow to the heart muscle, which is important in assessing heart disease risk. The target variable is the most important measure, as it indicates the presence or absence of heart disease. Before conducting analysis on this dataset, we performed some preprocessing steps. First, we checked for missing values and found that the dataset was already clean. Second, we performed feature engineering by creating dummy variables for categorical variables such as sex and chest pain type. Third, we checked for outliers and found none that required further action. Overall, the dataset was ready for analysis.

In [3]:
```
data_dictionary
```

Out[3]:

	Name	Definition
0	age	Age of the patient
1	sex	Sex of the patient
2	exng	exercise induced angina (1 = yes; 0 = no)
3	caa	number of major vessels (0-3)
4	cp	Chest Pain type chest pain type\nValue 1: typi...
5	trtbps	resting blood pressure (in mm Hg)
6	chol	cholestoral in mg/dl fetched via BMI sensor
7	fbs	(fasting blood sugar > 120 mg/dl) (1 = true; 0...
8	restecg	resting electrocardiographic results\nValue 0:...
9	thalachh	maximum heart rate achieved
10	thall	thal rate
11	oldpeak	previous peak
12	target	0= less chance of heart attack 1= more chance ...

Section 3. Data Cleanup and Exploratory Data Analysis

In [4]:
```
data_cleanup.get_dataframe_shape(df)
```

```
The dataset has 14 columns and 303 rows
```

Descriptive Statistics

Function: Descriptive Statistics

This function takes a Pandas dataframe as input and computes various descriptive statistics of the variables. If the variable is numeric, it calculates the count, missing values, mean, median, mode, range, variance, standard deviation, skewness, kurtosis, minimum, maximum, and interquartile range. If the variable is not numeric, it calculates the frequency distribution, relative frequency, and mode.

Parameters: df: Pandas DataFrame Input data frame containing variables for which descriptive statistics are to be computed.

Output: A Pandas DataFrame containing descriptive statistics for each variable in the input data frame.

Required Libraries: pandas

Example Usage:

import pandas as pd df = pd.DataFrame({'var1': [1,2,3,4,5], 'var2': ['a','b','c','d','e'], 'var3': [1.1,2.2,3.3,4.4,5.5]}) descriptive_statistics(df)

Note: The function assumes that the input dataframe is cleaned and has no missing values except for NaNs.

Note on Data Types The function checks whether a variable is numeric or not based on its data type. It assumes that variables of type "int64" and "float64" are numeric and all other variables are not numeric. If you have variables that are numeric but have data types other than "int64" or "float64", you can modify the code to include those data types.

In [5]:
```
descriptives.descriptive_statistics(df)
```

Out[5]:

	count	missing_values	data_type	mean	median	mode	range	variance	std_dev
age	303	0	int64	54.366337	55.0	58.0	48.0	82.484558	9.08210
sex	303	0	int64	0.683168	1.0	1.0	1.0	0.217166	0.46601
cp	303	0	int64	0.966997	1.0	0.0	3.0	1.065132	1.03205
trtbps	303	0	int64	131.623762	130.0	120.0	106.0	307.586453	17.53814
chol	303	0	int64	246.264026	240.0	197.0	438.0	2686.426748	51.83075
fbs	303	0	int64	0.148515	0.0	0.0	1.0	0.126877	0.35619
restecg	303	0	int64	0.528053	1.0	1.0	2.0	0.276528	0.52586
thalachh	303	0	int64	149.646865	153.0	162.0	131.0	524.646406	22.90516
exng	303	0	int64	0.326733	0.0	0.0	1.0	0.220707	0.46979
oldpeak	303	0	float64	1.039604	0.8	0.0	6.2	1.348095	1.16107
slp	303	0	int64	1.399340	1.0	2.0	2.0	0.379735	0.61622
caa	303	0	int64	0.729373	0.0	0.0	4.0	1.045724	1.02260
thall	303	0	int64	2.313531	2.0	2.0	3.0	0.374883	0.61227
output	303	0	int64	0.544554	1.0	1.0	1.0	0.248836	0.49883

Variable Histograms

Function: Create Histogram

This function creates a matrix of histograms for all numeric columns of the input dataframe. It calculates the optimal set of rows and columns for the histogram matrix based on the number of numeric columns in the dataframe. It uses the Seaborn library to plot histograms of each numeric variable in the dataframe.

Parameters:

df: Pandas DataFrame Input data frame containing numeric variables for which histograms are to be plotted. figsize: tuple, default=(15, 15) The figure size of the histogram matrix. bins: int, default=20 The number of bins to use for the histogram. color: str, default='steelblue' The color of the bars in the histogram. **Output:**

Displays a matrix of histograms for all numeric columns of the input dataframe. **Required Libraries:**

numpy matplotlib.pyplot seaborn

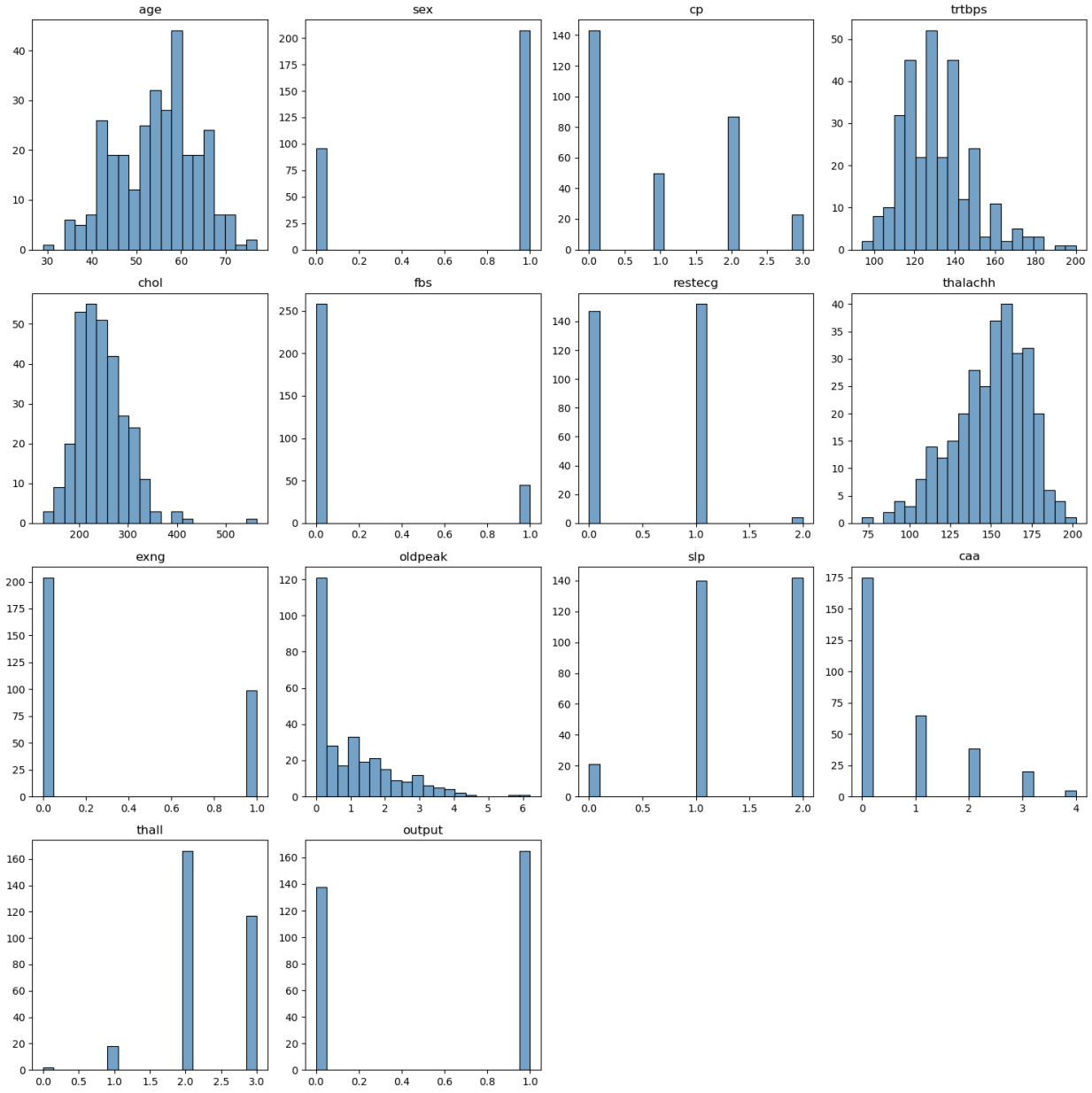
Example Usage:

python import pandas as pd import numpy as np import seaborn as sns import matplotlib.pyplot as plt

df = pd.DataFrame({'var1': [1,2,3,4,5], 'var2': [2,4,6,8,10], 'var3': [3,6,9,12,15], 'var4': [4,8,12,16,20]}) create_histogram(df)

Note: The function assumes that the input dataframe only contains numeric columns.

In [6]:
```
histograms.create_histogram(df)
```



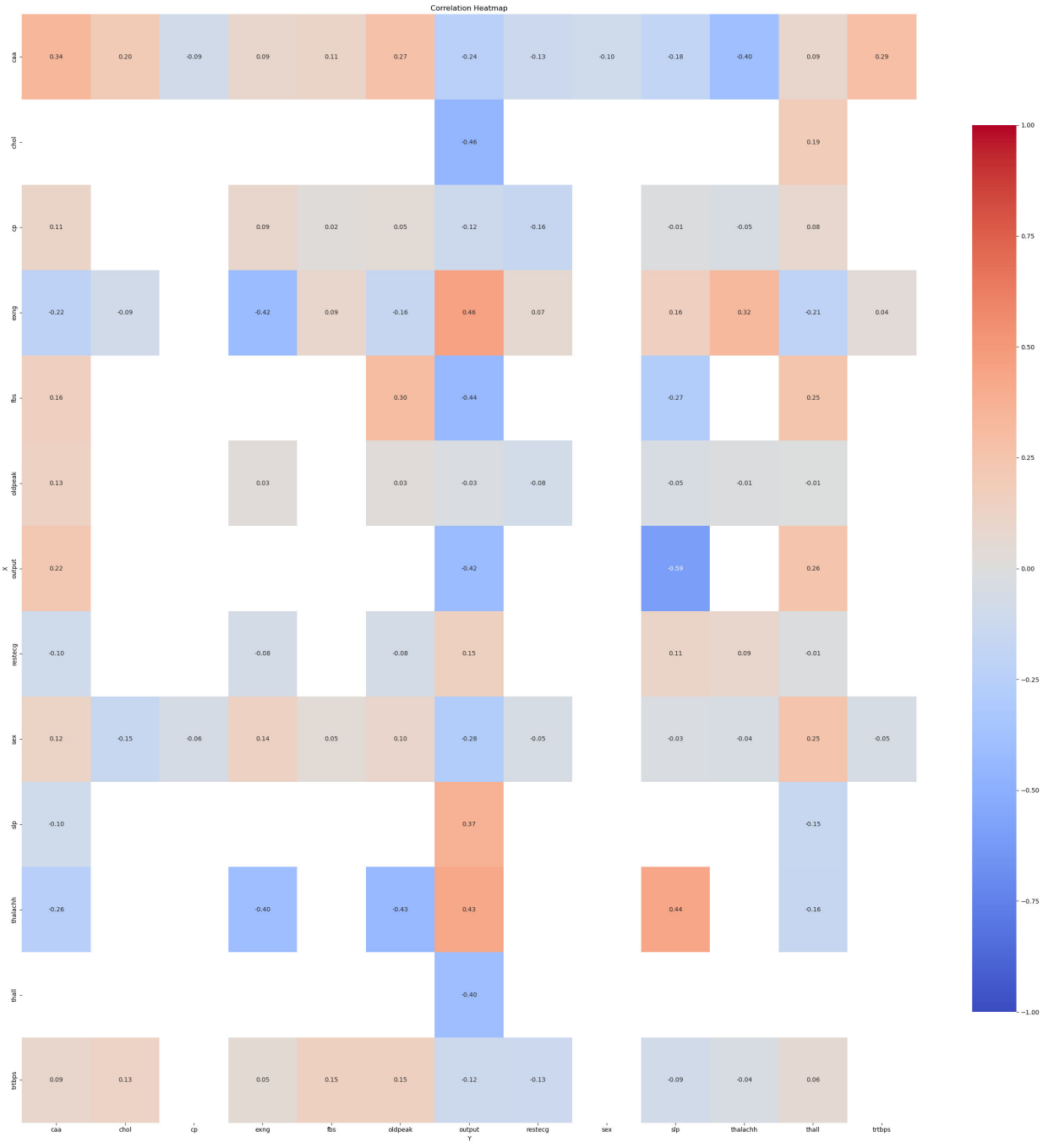
Correlation Heatmap

Function: Correlation Heatmap This function computes and visualizes pairwise correlation matrix of numeric variables in a Pandas dataframe using Spearman's rank correlation method. It drops variables that have too few samples or zero variance before computing the correlation matrix. It creates a heatmap of the correlation matrix using Seaborn library and saves the plot as an image file.

Parameters: df: Pandas DataFrame Input data frame containing numeric variables for which correlation heatmap is to be plotted. Output: A heatmap of pairwise correlations between numeric variables in the input data frame.

Required Libraries: numpy seaborn pingouin matplotlib.pyplot

In [7]:
```
corr_map.correlation_heatmap(df)
```



Data Cleanup Functions

Function 1: Select Columns

This function processes a dataframe and retains only the selected columns. It returns a new dataframe containing only the selected columns.

Parameters:

df: Pandas DataFrame Input data frame from which columns are to be retained.
columns_to_keep: list List of column names to be retained in the input data frame. **Output:**

A new data frame containing only the selected columns.

Function 2: Get Dataframe Shape This function provides information about the size and shape of the input dataframe. **Parameters:** df: Pandas DataFrame Input data frame for which information is to be provided. **Output:** A string describing the shape of the input dataframe.

Function 3: Clean Up Missing Data This function drops any columns from the input dataframe that are missing too many data points based on a breakpoint provided by the user. It returns a new data frame with the columns with too many missing values removed. **Parameters:** df: Pandas DataFrame Input data frame from which columns with too many missing values are to be removed. breakpoint: float The breakpoint for the minimum data completeness required for each column. **Output:** A tuple containing the nan_percentages by variable and the resulting data frame with the columns with too many missing values removed.

Function 4: Drop Rows with NA This function drops NA records from the input dataframe. It returns a new data frame without NA records. **Parameters:** df: Pandas DataFrame Input data frame from which NA rows are to be removed. **Output:** A new data frame without NA records.

Function 5: One-Hot Encode Categorical Variables This function creates a dummy binary variable for each level of the categorical variable in the input dataframe. Columns are named with the prefix of '1h_', and the original variable is dropped from the input data frame.
Parameters:
df: Pandas DataFrame -- Input data frame in which categorical variables are to be one-hot encoded.

col_list: List of column names of the categorical variables to be one-hot encoded.

Output:

A new data frame with all variables in col_list recoded.

Function 6: Run Data Cleanup Functions This function prunes the input dataframe to only keep the selected columns, removes columns with too many missing records, removes rows with missing values, recodes the categorical variables into dummies, prunes the data dictionary to retain only the variables use in the function, and prints the resulting data frame shape after each operation so you know how the data frame has changed. **Parameters:** df: Pandas DataFrame
Input data frame for cleanup.

data_dictionary: Pandas DataFrame -- Data dictionary which has two columns "Name" and "Definition".

columns_to_keep: list -- List of column names to be retained in the input data frame.

columns_to_recode: list -- List of column names of the categorical variables to be one-hot encoded.

breakpoint: float --The breakpoint for the minimum data completeness required for each column.

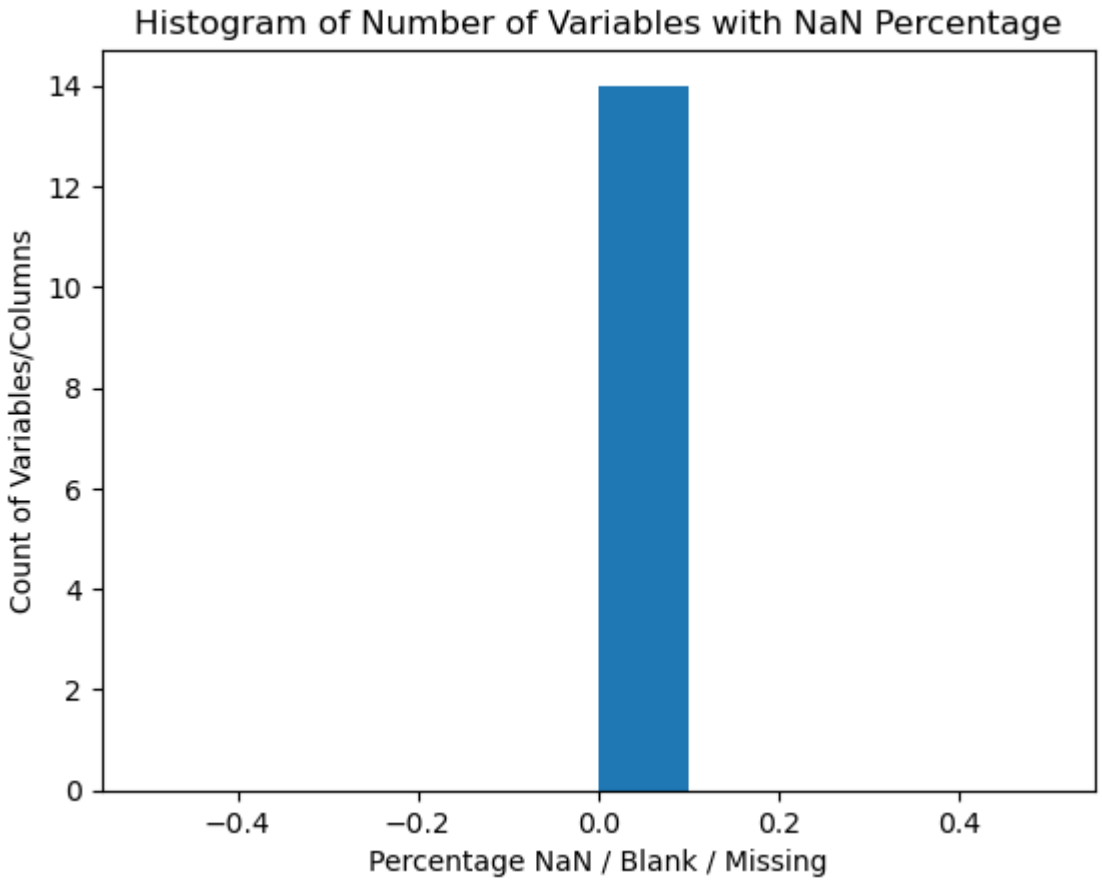
Output: A tuple containing the resulting data frame and data dictionary.

Required Libraries:

pandas numpy matplotlib.pyplot seaborn

In [8]:
```
columns_to_keep=list(df.columns)
columns_to_recode=['sex','exng','cp','fbs','restecg','slp','caa','thall']
breakpoint=0.95
df, data_dictionary2 = data_cleanup.run_data_cleanup_functions(df, data_dictionary, cc
```

```
Original dataframe---
The dataset has 14 columns and 303 rows
```




```
After cleaning up columns with missing data---
The dataset has 14 columns and 303 rows
Number of rows before filtering: 303
Number of rows after filtering: 303
Percentage of rows dropped: 0.0%
After cleaning up rows with missing data---
The dataset has 14 columns and 303 rows
After recoding categorical variables into one-hot variables---
The dataset has 31 columns and 303 rows
```

Define the Dependent Variable

In [9]:
```
target = 'output'
target
```

Out[9]:
```
'output'
```

Define the Independent Variables

In [10]:
```
features = [col for col in df.columns if col != target]
features
```

Out[10]:
```
['age',
 'trtbps',
 'chol',
 'thalachh',
 'oldpeak',
 '1h_sex_0',
 '1h_sex_1',
 '1h_exng_0',
 '1h_exng_1',
 '1h_cp_0',
 '1h_cp_1',
 '1h_cp_2',
 '1h_cp_3',
 '1h_fbs_0',
 '1h_fbs_1',
 '1h_restecg_0',
 '1h_restecg_1',
 '1h_restecg_2',
 '1h_slp_0',
 '1h_slp_1',
 '1h_slp_2',
 '1h_caa_0',
 '1h_caa_1',
 '1h_caa_2',
 '1h_caa_3',
 '1h_caa_4',
 '1h_thall_0',
 '1h_thall_1',
 '1h_thall_2',
 '1h_thall_3']
```

In [11]:
```
df[target].value_counts()
```

Out[11]:
```
1    165
0    138
Name: output, dtype: int64
```

View Independent Variable Correlations

Function: View Independent Variable Correlations

This function computes the Pearson correlation between the target variable and each independent variable in a Pandas dataframe, and returns the resulting correlation coefficients sorted in descending order.

Parameters: data: Pandas DataFrame Input data frame containing the target variable and independent variables to be used in the correlation analysis.

target_variable: str Name of the target variable for which correlations with the independent variables are to be computed.

Required Libraries: pandas

Outputs: A Pandas series containing the correlation coefficients between the target variable and each independent variable, sorted in descending order.

In [12]:
```
def view_independent_variable_correlations(data, target_variable):
    correlations = data.corr(method='pearson')[target_variable].drop(target_variable)
    correlations = correlations.sort_values(ascending=False)
    return correlations
```

In [13]:
```
correlations = view_independent_variable_correlations(df, target)
correlations
```

Out[13]:
```
1h_thall_2      0.527334
1h_caa_0        0.465590
1h_exng_0       0.436757
thalachh        0.421741
1h_slp_2        0.394066
1h_cp_2         0.316742
1h_sex_0        0.280937
1h_cp_1         0.245879
1h_restecg_1    0.175322
1h_cp_3         0.086957
1h_caa_4        0.066441
1h_fbs_0        0.028046
1h_thall_0     -0.007293
1h_fbs_1       -0.028046
1h_slp_0       -0.063554
1h_restecg_2   -0.068410
chol           -0.085239
1h_thall_1     -0.106589
trtbps         -0.144931
1h_restecg_0   -0.159775
1h_caa_3       -0.210615
age            -0.225439
1h_caa_1       -0.232412
1h_caa_2       -0.273998
1h_sex_1       -0.280937
1h_slp_1       -0.362053
oldpeak        -0.430696
1h_exng_1      -0.436757
1h_thall_3     -0.486112
1h_cp_0        -0.516015
Name: output, dtype: float64
```

Compute Class Weights

Function: Compute Class Weights

This function computes the class weights for a binary classification problem. It takes a Pandas dataframe and calculates the proportion of samples in each class, then assigns a weight of 1 to the minority class and a weight proportional to the class imbalance to the majority class.

Parameters: df: Pandas DataFrame Input data frame containing the target variable for which class weights are to be computed.

Outputs: A dictionary containing the class weights for the binary classification problem. The keys are the class labels (0 and 1) and the values are the corresponding weights.

Example Usage: distribution = df['output'].value_counts() low_weight = distribution[1]/distribution[0] class_weights = compute_class_weights(df) print(class_weights)

Required Libraries: pandas

In [14]:
```
def compute_class_weights(df, target_variable):

    # Count the number of samples in each class
    distribution = df[target_variable].value_counts()

    # Compute the weight of the minority class relative to the majority class
    low_weight = distribution[1] / distribution[0]

    # Assign weights to each class
    class_weights = {0: low_weight, 1: 1}

    print(f"The distribution of the dependent variable is: {distribution}")
    print(f"The resuling class weights are: {class_weights}")

    return class_weights, distribution
```

In [15]:
```
class_weights, distribution = compute_class_weights(df, target)
```

```
The distribution of the dependent variable is: 1    165
0    138
Name: output, dtype: int64
The resuling class weights are: {0: 1.1956521739130435, 1: 1}
```

Split Data into Train/Validation/Test Sets

Function: Split Data This function takes a pandas DataFrame and performs a split of the data into training, testing, and validation sets. It is designed for use with a dependent variable with a binary classification. It prints out the distribution of the dependent variable in each of the training, testing, and validation sets.

Parameters: df: Pandas DataFrame Input data frame containing all data to be split into training, testing, and validation sets.

labels: List of Strings List of two strings used to identify the two possible values for the dependent variable.

target: String String identifying the dependent variable in the input data frame.

Output: Three Pandas DataFrames containing the training, testing, and validation sets.

Required Libraries: pandas sklearn.model_selection.train_test_split

In [16]:
```
labels = ['Low Heart Risk','High Heart Risk']
train_data, test_data, val_data = train_test_validate.split_data(df, labels, target)
```

```
Train data dependent distribution:
    86 (0 - Low Heart Risk) 95(1 - High Heart Risk)
Test data dependent distribution:
    24 (0 - Low Heart Risk) 37(1 - High Heart Risk)
Validation data dependent distribution:
    28 (0 - Low Heart Risk) 33(1 - High Heart Risk)
```

Section 4: Train Models to Optimize Hyperparameters

Create Directory to store trained models

This code creates a directory called "models" in the current working directory if it does not exist. If the directory already exists, the code simply prints a message indicating that it already exists.

In [17]:
```
import os

current_directory = os.getcwd()
directory_name = f"{current_directory}/models"

if not os.path.exists(directory_name):
    os.makedirs(directory_name)
    print(f"{directory_name} created successfully!")
else:
    print(f"{directory_name} already exists.")
```

```
C:\Users\bwynia\heart_attack_risk_analysis/models already exists.
```

General Function to check for trained model and retrain as necessary

train_and_save_model

Function: trains a classifier, finds the best hyperparameters using cross-validation, and saves the trained model and hyperparameter results to files. If the model and results already exist, it loads them from files.

Parameters:

- classifier: A function to build a classifier object.
- param_grid: A dictionary containing the hyperparameters and their possible values.
- method: A string indicating the tuning method. Either "GridSearchCV" or "RandomizedSearchCV".
- features: A Pandas dataframe containing the independent variables.
- target: A Pandas series containing the dependent variable.
- model_file_path: A string indicating the path to save the trained model.
- results_file_path: A string indicating the path to save the hyperparameter results.

Outputs:

- trained_model: A scikit-learn estimator object that has been fit with the best hyperparameters.

- best_params: A dictionary containing the best hyperparameters found by the tuning method.
- best_score: A float indicating the mean cross-validated score achieved with the best hyperparameters.
- elapsed_time: A float indicating the number of seconds taken to fit the best estimator on the whole dataset.

In [18]:
```
def train_and_save_model(classifier, param_grid, method, features, target, model_file_
    # Check if model and results already exist
    try:
        with open(model_file_path, "rb") as f:
            trained_model = pickle.load(f)
        with open(results_file_path, "rb") as f:
            best_params, best_score, elapsed_time = pickle.load(f)
        print("Loaded existing model and hyperparameter results.")
        print(best_params)
        print(best_score)

    except FileNotFoundError:
        # Build classifier
        clf = classifier()

        # Build parameter grid
        param_dist = param_grid

        # Define tuning method
        if method == "GridSearchCV":
            tuning_method = GridSearchCV(clf, param_dist, cv=5, n_jobs=-1, scoring='r
        elif method == "RandomizedSearchCV":
            tuning_method = RandomizedSearchCV(clf, param_dist, cv=5, n_jobs=-1, scori

        # Train classifier and find best hyperparameters
        tuning_method.fit(features, target)
        trained_model = tuning_method.best_estimator_
        best_params = tuning_method.best_params_
        best_score = tuning_method.best_score_
        elapsed_time = tuning_method.refit_time_

        # Save trained model and hyperparameter results to files
        with open(model_file_path, "wb") as f:
            pickle.dump(trained_model, f)
        with open(results_file_path, "wb") as f:
            pickle.dump((best_params, best_score, elapsed_time), f)
        print("Trained model and hyperparameter results saved to files.")
        print(best_params)
        print(best_score)

    return trained_model, best_params, best_score, elapsed_time
```

Logistic Regression

Logistic Regression Hyperparameter Grid

The following hyperparameter grid is used for training a logistic regression model with GridSearchCV or RandomizedSearchCV.

- `penalty` : {'l1', 'l2', 'elasticnet', 'none'}
  - `l1` : Lasso regularization
  - `l2` : Ridge regularization
  - `elasticnet` : Elastic Net regularization
  - `none` : No regularization
- `C` : float, default=1.0
  - Inverse of regularization strength; smaller values specify stronger regularization.
- `fit_intercept` : bool, default=True
  - Specifies whether or not to calculate the intercept for this model.
- `solver` : {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'
  - Algorithm to use in the optimization problem.
- `l1_ratio` : float, default=None
  - Only used if penalty='elasticnet'; specifies the mixing parameter for L1 and L2 regularization.

Training a Logistic Regression Model

The following code defines a logistic regression model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
  - File path for saving/loading the trained model.
- `results_file_path` : str
  - File path for saving/loading the hyperparameter tuning results.
- `logreg_model` : LogisticRegression object
  - Base logistic regression model to be tuned.
- `logreg_method` : {'GridSearchCV', 'RandomizedSearchCV'}
  - Method to use for hyperparameter tuning.
- `logreg_param_grid` : list of dicts
  - Hyperparameter grid to search.
- `train_data[features]` : pandas DataFrame
  - Features to use for training the model.
- `train_data[target]` : pandas Series
  - Target variable to predict.

In [19]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/logreg_trained_model.pkl"
results_file_path = f"{directory_name}/logreg_hyperparameter_results.pkl"

# Define base model and tuning methodology
logreg_model = LogisticRegression(max_iter=10000, class_weight=class_weights)
logreg_method = "GridSearchCV" # "RandomizedSearchCV"
```

```
# Build parameter grid
logreg_param_grid = [
    {
        'penalty': ['l1', 'l2'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'fit_intercept': [True, False],
        'solver': ['liblinear']
    },
    {
        'penalty': ['l2'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'fit_intercept': [True, False],
        'solver': ['newton-cg', 'lbfgs', 'sag']
    },
    {
        'penalty': ['l2', 'none'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'fit_intercept': [True, False],
        'solver': ['saga']
    },
    {
        'penalty': ['elasticnet'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'fit_intercept': [True, False],
        'solver': ['saga'],
        'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9]
    },
    {
        'penalty': ['none'],
        'fit_intercept': [True, False],
        'solver': ['newton-cg', 'lbfgs', 'sag']
    }
]

# Train and save logistic regression model
logreg_trained_model, logreg_best_params, logreg_best_score, logreg_elapsed_time = tra
```

```
Loaded existing model and hyperparameter results.
{'C': 1, 'fit_intercept': True, 'penalty': 'l1', 'solver': 'liblinear'}
0.8454954954954955
```

AdaBoost Hyperparameter Grid

The following hyperparameter grid is used for training an AdaBoost classifier model with GridSearchCV or RandomizedSearchCV.

- `base_estimator` : DecisionTreeClassifier object or list of DecisionTreeClassifier objects
  - Base estimator(s) from which the boosted ensemble is built. The decision tree(s) can be either a single decision tree or a list of decision trees with different depths.
- `n_estimators` : int

    - The maximum number of estimators at which boosting is terminated. Too large a value can lead to overfitting.
- `learning_rate` : float
    - The contribution of each classifier in the final combination is multiplied by this learning rate. Smaller values require more estimators to reach the same level of accuracy as larger values.
- `algorithm` : {'SAMME', 'SAMME.R'}
    - The boosting algorithm to use. 'SAMME' stands for Stagewise Additive Modeling using a Multiclass Exponential loss function, while 'SAMME.R' stands for SAMME.R for Real. SAMME.R is a variant of SAMME that relies on class probabilities rather than class labels and generally performs better.
- `random_state` : int or RandomState
    - Seed for random number generator to ensure reproducibility of results.

Training an AdaBoost Classifier Model

The following code defines an AdaBoost classifier model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
    - File path for saving/loading the trained model.
- `results_file_path` : str
    - File path for saving/loading the hyperparameter tuning results.
- `abc_trained_model` : AdaBoostClassifier object
    - Base AdaBoost classifier model to be tuned.
- `abc_method` : {'GridSearchCV', 'RandomizedSearchCV'}
    - Method to use for hyperparameter tuning.
- `abc_param_grid` : dict
    - Hyperparameter grid to search.
- `train_data[features]` : pandas DataFrame
    - Features to use for training the model.
- `train_data[target]` : pandas Series
    - Target variable to predict.

In [20]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/abc_trained_model.pkl"
results_file_path = f"{directory_name}/abc_hyperparameter_results.pkl"

# Define tuning methodology
abc_method = "GridSearchCV"

# Build parameter grid
abc_param_grid = {
    'base_estimator': [DecisionTreeClassifier(class_weight=class_weights, max_depth=d)
    'n_estimators': [10, 50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.5, 1, 2],
    'algorithm': ['SAMME', 'SAMME.R'],
```

```
    'random_state': [7]
}

# Train and save AdaBoost classifier model
abc_trained_model, abc_best_params, abc_best_score, abc_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'algorithm': 'SAMME', 'base_estimator': DecisionTreeClassifier(class_weight={0: 1.19
56521739130435, 1: 1}, max_depth=1), 'learning_rate': 0.5, 'n_estimators': 50, 'rando
m_state': 7}
0.9347781217750258
```

Decision Tree Hyperparameter Grid

The following hyperparameter grid is used for training a decision tree classifier model with GridSearchCV or RandomizedSearchCV.

- `criterion` : {'gini', 'entropy'}
  - The function to measure the quality of a split. 'gini' uses the Gini impurity, while 'entropy' uses the information gain.

- `splitter` : {'best', 'random'}
  - The strategy used to choose the split at each node. 'best' chooses the best split, while 'random' chooses the best random split.

  - `max_depth` : int or None
    - The maximum depth of the tree. A larger value generally leads to overfitting, while a smaller value can lead to underfitting. If None, the nodes are expanded until all the leaves contain less than min_samples_split samples.

    - `min_samples_split` : int or float
      - The minimum number of samples required to split an internal node. If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

      - `min_samples_leaf` : int or float
        - The minimum number of samples required to be at a leaf node. If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

- `max_features` : {'auto', 'sqrt', 'log2'} or None
  - The number of features to consider when looking for the best split. If None, then all features are considered. 'auto' chooses sqrt(n_features) features, while 'sqrt' and 'log2' choose sqrt(n_features) and log2(n_features) features, respectively.

  - `class_weight` : dict, 'balanced', or None
    - Weights associated with classes. If None, all classes are supposed to have weight one. 'balanced' uses the values of y to automatically adjust weights inversely proportional to class frequencies, while a dictionary assigns weight to each class. It is also possible to pass class_weights calculated externally to the model as a parameter.

Training a Decision Tree Classifier Model

The following code defines a decision tree classifier model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
  - File path for saving/loading the trained model.

- `results_file_path` : str
  - File path for saving/loading the hyperparameter tuning results.

  - `dt_trained`

In [21]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/dt_trained_model.pkl"
results_file_path = f"{directory_name}/dt_hyperparameter_results.pkl"

# Define base model and tuning methodology
dt_method = "GridSearchCV"

# Build parameter grid
dt_param_grid = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
    'max_depth': [None, 3, 5, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5, 10],
```

```
    'max_features': [None, 'auto', 'sqrt', 'log2'],
    'class_weight': [None, 'balanced',class_weights]
}

# Train and save decision tree classifier model
dt_trained_model, dt_best_params, dt_best_score, dt_elapsed_time = train_and_save_mode
```

```
Loaded existing model and hyperparameter results.
{'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 10, 'max_features': 'l
og2', 'min_samples_leaf': 5, 'min_samples_split': 10, 'splitter': 'random'}
0.8999484004127967
```

Random Forest Hyperparameter Grid

The following hyperparameter grid is used for training a random forest classifier model with GridSearchCV or RandomizedSearchCV.

- **General Parameters**
  - `n_estimators` : int
    - The number of trees in the forest.

  - `criterion` : {'gini', 'entropy'}
    - The function to measure the quality of a split. 'gini' uses the Gini impurity, while 'entropy' uses the information gain.

    - `max_depth` : int or None
      - The maximum depth of the tree. A larger value generally leads to overfitting, while a smaller value can lead to underfitting. If None, the nodes are expanded until all the leaves contain less than min_samples_split samples.

      - `min_samples_split` : int or float
        - The minimum number of samples required to split an internal node. If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

        - `min_samples_leaf` : int or float
          - The minimum number of samples required to be at a leaf node. If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

- `max_features` : {'auto', 'sqrt', 'log2'} or None
  - The number of features to consider when looking for the best split. If None, then all features are considered. 'auto' chooses sqrt(n_features) features, while 'sqrt' and 'log2' choose sqrt(n_features) and log2(n_features) features, respectively.

  - `class_weight` : dict, 'balanced', 'balanced_subsample', or None
    - Weights associated with classes. If None, all classes are supposed to have weight one. 'balanced' uses the values of y to automatically adjust weights inversely proportional to class frequencies, while 'balanced_subsample' is the same as 'balanced' but computed on a per-tree basis. A dictionary assigns weight to each class. It is also possible to pass class_weights calculated externally to the model as a parameter.

    - `n_jobs` : int or None
      - The number of jobs to run in parallel for both fit and predict. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors.

      - `random_state` : int or RandomState
        - Seed for random number generator to ensure reproducibility of results.

        - **Bootstrap Sampling Parameters**
        - `bootstrap` : bool
          - Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

          - `oob_score` : bool
            - Whether to use out-of-bag samples to estimate the generalization accuracy. If True, then the score of each tree is computed using samples that are not used for training that tree.

In [22]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/rfc_trained_model.pkl"
results_file_path = f"{directory_name}/rfc_hyperparameter_results.pkl"
```

```
# Define base model and tuning methodology
rfc_method = "RandomizedSearchCV"

# Build parameter grid
rfc_param_grid = [
    {
        'n_estimators': [10, 50, 100, 200],
        'criterion': ['gini', 'entropy'],
        'max_depth': [None, 3, 5, 10, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 5, 10],
        'max_features': ['auto', 'sqrt', 'log2'],
        'bootstrap': [True],
        'class_weight': [None, 'balanced', 'balanced_subsample', class_weights],
        'oob_score': [False, True],
        'n_jobs': [-1],
        'random_state': [42]
    },
    {
        'n_estimators': [10, 50, 100, 200],
        'criterion': ['gini', 'entropy'],
        'max_depth': [None, 3, 5, 10, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 5, 10],
        'max_features': ['auto', 'sqrt', 'log2'],
        'bootstrap': [False],
        'class_weight': [None, 'balanced', 'balanced_subsample', class_weights],
        'n_jobs': [-1],
        'random_state': [42]
    }
]

# Train and save random forest classifier model
rfc_trained_model, rfc_best_params, rfc_best_score, rfc_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'random_state': 42, 'oob_score': False, 'n_jobs': -1, 'n_estimators': 200, 'min_samp
les_split': 10, 'min_samples_leaf': 10, 'max_features': 'log2', 'max_depth': 5, 'crit
erion': 'entropy', 'class_weight': None, 'bootstrap': False}
0.9344341245270037
```

K-Nearest Neighbors (KNN) Hyperparameter Grid

The following hyperparameter grid is used for training a KNN model with GridSearchCV or RandomizedSearchCV.

- `n_neighbors` : int
  - Number of neighbors to use.
- `weights` : {'uniform', 'distance'}

    - Weight function used in prediction. 'uniform' weights all points in the neighborhood equally, while 'distance' weights points by the inverse of their distance.
- `algorithm` : {'auto', 'ball_tree', 'kd_tree', 'brute'}
    - Algorithm used to compute the nearest neighbors.
- `leaf_size` : int
    - Leaf size passed to BallTree or KDTree.
- `p` : int
    - Power parameter for the Minkowski metric. When p=1, this is equivalent to using the Manhattan distance, and when p=2, it is equivalent to using the Euclidean distance.
- `metric` : {'euclidean', 'manhattan', 'minkowski'}
    - Distance metric to use for the tree.
- `n_jobs` : int
    - Number of CPU cores to use for the computation.

Training a KNN Classifier Model

The following code defines a KNN classifier model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
    - File path for saving/loading the trained model.
- `results_file_path` : str
    - File path for saving/loading the hyperparameter tuning results.
- `knn_model` : KNeighborsClassifier object
    - Base KNN classifier model to be tuned.
- `knn_method` : {'GridSearchCV', 'RandomizedSearchCV'}
    - Method to use for hyperparameter tuning.
- `knn_param_grid` : dict
    - Hyperparameter grid to search.
- `train_data[features]` : pandas DataFrame
    - Features to use for training the model.
- `train_data[target]` : pandas Series
    - Target variable to predict.

In [23]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/knn_trained_model.pkl"
results_file_path = f"{directory_name}/knn_hyperparameter_results.pkl"

# Define base model and tuning methodology
knn_method = "RandomizedSearchCV"

# Build parameter grid
knn_param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 30, 50],
```

```
    'p': [1, 2],
    'metric': ['euclidean', 'manhattan', 'minkowski'],
    'n_jobs': [-1]
}

# Train and save KNN classifier model
knn_trained_model, knn_best_params, knn_best_score, knn_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'weights': 'distance', 'p': 2, 'n_neighbors': 11, 'n_jobs': -1, 'metric': 'manhatta
n', 'leaf_size': 10, 'algorithm': 'auto'}
0.6866185070519436
```

MLP Classifier Hyperparameter Grid

The following hyperparameter grid is used for training an MLP classifier model with GridSearchCV or RandomizedSearchCV.

- `hidden_layer_sizes` : tuple, default=(100,)
  - The ith element represents the number of neurons in the ith hidden layer.
- `activation` : {'identity', 'logistic', 'tanh', 'relu'}, default='relu'
  - Activation function for the hidden layer.
- `solver` : {'lbfgs', 'sgd', 'adam'}, default='adam'
  - Algorithm to use in the optimization problem.
- `alpha` : float, default=0.0001
  - L2 penalty (regularization term) parameter.
- `batch_size` : int, default='auto'
  - Size of minibatches for stochastic optimizers.
- `learning_rate` : {'constant', 'invscaling', 'adaptive'}, default='constant'
  - Learning rate schedule for weight updates.
- `learning_rate_init` : float, default=0.001
  - The initial learning rate used.
- `power_t` : float, default=0.5
  - Exponent for inverse scaling learning rate.
- `max_iter` : int, default=200
  - Maximum number of iterations.
- `shuffle` : bool, default=True
  - Whether to shuffle samples in each iteration.
- `random_state` : int, default=42
  - Seed used by the random number generator.
- `tol` : float, default=1e-4
  - Tolerance for optimization.

- `verbose` : bool, default=False
  - Whether to print progress messages to stdout.
- `warm_start` : bool, default=False
  - When set to True, reuse the solution of the previous call to fit as initialization.
- `momentum` : float, default=0.9
  - Momentum for SGD optimizer.
- `nesterovs_momentum` : bool, default=True
  - Whether to use Nesterov's momentum. Only used when `solver='sgd'`.
- `early_stopping` : bool, default=False
  - Whether to use early stopping to terminate training when validation score doesn't improve. If set to True, also requires setting aside some validation data.
- `validation_fraction` : float, default=0.1
  - The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1.
- `beta_1` : float, default=0.9
  - The exponential decay rate for estimating the first moment vector in Adam. Only used when `solver='adam'` or `solver='adamax'`.
- `beta_2` : float, default=0.999
  - The exponential decay rate for estimating the second moment vector in Adam. Only used when `solver='adam'` or `solver='adamax'`.
- `epsilon` : float, default=1e-8
  - Value to avoid division by zero. Only used when `solver='adam'` or `solver='adamax'`.
- `n_iter_no_change` : int, default=10
  - Maximum number of iterations with no improvement to wait before stopping optimization. Only used when `early_stopping=True`.
- `class_weight` : dict, 'balanced', or None, default=None
  - Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. If set to 'balanced', class weights will be inversely proportional to the number of samples in each class. Only used for `solver='sgd'` or `solver='adam'`.

In [24]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/mlp_trained_model.pkl"
results_file_path = f"{directory_name}/mlp_hyperparameter_results.pkl"

# Define tuning methodology
mlp_method = "RandomizedSearchCV"

# Build parameter grid
mlp_param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 100)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'],
    'alpha': [0.0001, 0.001, 0.01, 0.1],
    'batch_size': ['auto', 50, 100, 200],
```

```
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'power_t': [0.5, 0.75],
    'max_iter': [200, 500, 1000],
    'shuffle': [True, False],
    'random_state': [42],
    'tol': [1e-4, 1e-5],
    'verbose': [False],
    'warm_start': [False, True],
    'momentum': [0.9, 0.95],
    'nesterovs_momentum': [True, False],
    'early_stopping': [False, True],
    'validation_fraction': [0.1, 0.2],
    'beta_1': [0.9, 0.95],
    'beta_2': [0.999, 0.99],
    'epsilon': [1e-8, 1e-9],
    'n_iter_no_change': [10, 20],
    'class_weight': ['balanced', class_weights]
}

# Train and save MLP classifier model
mlp_trained_model, mlp_best_params, mlp_best_score, mlp_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'warm_start': True, 'verbose': False, 'validation_fraction': 0.2, 'tol': 0.0001, 'so
lver': 'adam', 'shuffle': False, 'random_state': 42, 'power_t': 0.75, 'nesterovs_mome
ntum': True, 'n_iter_no_change': 20, 'momentum': 0.95, 'max_iter': 1000, 'learning_ra
te_init': 0.01, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (50,), 'epsilo
n': 1e-09, 'early_stopping': True, 'beta_2': 0.99, 'beta_1': 0.95, 'batch_size': 100,
'alpha': 0.01, 'activation': 'identity'}
0.9229446164430684
```

Training an SVM Model

The following code defines an SVM model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
  - File path for saving/loading the trained model.
- `results_file_path` : str
  - File path for saving/loading the hyperparameter tuning results.
- `svm_model` : SVC object
  - `svm_method` : {'GridSearchCV', 'RandomizedSearchCV'}
    - Method to use for hyperparameter tuning.
  - `svm_param_grid` : list of dicts
    - Hyperparameter grid to search.
    - `C` : float, default=1.0

- Penalty parameter of the error term.
- `kernel` : {'linear', 'poly', 'rbf', 'sigmoid'}, default='rbf'
  - Kernel function used in the algorithm.
- `degree` : int, default=3
  - Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
- `gamma` : {'scale', 'auto'} or float, default='scale'
  - Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
- `coef0` : float, default=0.0
  - Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.
- `shrinking` : bool, default=True
  - Whether to use the shrinking heuristic.
- `probability` : bool, default=False
  - Whether to enable probability estimates.
- `tol` : float, default=1e-3
  - Tolerance for stopping criterion.
- `class_weight` : {dict, 'balanced', None}, default=None
  - Weights associated with classes. If not given, all classes are supposed to have weight one.
- `verbose` : bool, default=False
  - Enable verbose output.
- `decision_function_shape` : {'ovr', 'ovo'}, default='ovr'
  - Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

- `train_data[features]` : pandas DataFrame
  - Features to use for training the model.
- `train_data[target]` : pandas Series
  - Target variable to predict.

In [25]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/svm_trained_model.pkl"
results_file_path = f"{directory_name}/svm_hyperparameter_results.pkl"

# Define tuning methodology
svm_method = "RandomizedSearchCV"

# Build parameter grid for SVM
svm_param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': [2, 3, 4],  # only applicable for 'poly' kernel
    'gamma': ['scale', 'auto', 0.1, 1, 10],  # only applicable for 'rbf', 'poly', 'sig
    'coef0': [0, 1, 2],  # only applicable for 'poly' and 'sigmoid' kernels
    'shrinking': [True, False],
    'probability': [True],
    'tol': [1e-3, 1e-4],
    'class_weight': [None, 'balanced',class_weights],
```

```
    'verbose': [False],
    'decision_function_shape': ['ovr', 'ovo']
}

# Train and save SVM classifier model
svm_trained_model, svm_best_params, svm_best_score, svm_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'verbose': False, 'tol': 0.001, 'shrinking': False, 'probability': True, 'kernel':
'linear', 'gamma': 10, 'degree': 3, 'decision_function_shape': 'ovr', 'coef0': 2, 'cl
ass_weight': 'balanced', 'C': 0.1}
0.919642242862057
```

Training a Gradient Boosting Model

The following code defines a Gradient Boosting model and hyperparameter grid, and then trains and saves the model using the `train_and_save_model` function.

- `model_file_path` : str
  - File path for saving/loading the trained model.
- `results_file_path` : str
  - File path for saving/loading the hyperparameter tuning results.
- `gbm_model` : GradientBoostingClassifier object
  - `gbm_method` : {'GridSearchCV', 'RandomizedSearchCV'}
    - Method to use for hyperparameter tuning.
  - `gbm_param_grid` : list of dicts
    - Hyperparameter grid to search.
    - `loss` : {'deviance', 'exponential'}, default='deviance'
      - Loss function to be optimized.
    - `learning_rate` : float, default=0.1
      - Learning rate shrinks the contribution of each tree.
    - `n_estimators` : int, default=100
      - The number of boosting stages to perform.
    - `subsample` : float, default=1.0
      - The fraction of samples to be used for fitting the individual base learners.
    - `criterion` : {'friedman_mse', 'mse', 'mae'}, default='friedman_mse'
      - The function to measure the quality of a split.
    - `min_samples_split` : int or float, default=2
      - The minimum number of samples required to split an internal node.
    - `min_samples_leaf` : int or float, default=1
      - The minimum number of samples required to be at a leaf node.
    - `max_depth` : int, default=3

- The maximum depth of the individual regression estimators.
- `max_features` : {'auto', 'sqrt', 'log2'}, default=None
  - The number of features to consider when looking for the best split.
- `random_state` : int, RandomState instance or None, default=None
  - Controls the randomness of the estimator.
  - `verbose` : int, default=0
    - Controls the verbosity when fitting and predicting.
  - `validation_fraction` : float, default=0.1
    - The proportion of the training data to set aside for validation, to be used for early stopping.
  - `n_iter_no_change` : int or None, default=None
    - Number of iterations with no improvement to wait before stopping.
  - `tol` : float, default=1e-4
    - Tolerance for the early stopping criterion.
  - `ccp_alpha` : non-negative float, default=0.0
    - Complexity parameter used for Minimal Cost-Complexity Pruning.

In [26]:
```
# Define file paths for saving/loading the model and its results
model_file_path = f"{directory_name}/gbm_trained_model.pkl"
results_file_path = f"{directory_name}/gbm_hyperparameter_results.pkl"

# Define tuning methodology
gbm_method = "RandomizedSearchCV"

# Parameter grid for Gradient Boosting Machines (GBM)
gbm_param_grid = {
    'loss': ['deviance', 'exponential'],
    'learning_rate': [0.01, 0.1, 0.5, 1],
    'n_estimators': [10, 50, 100, 200],
    'subsample': [0.5, 0.8, 1.0],
    'criterion': ['friedman_mse', 'mse', 'mae'],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5, 10],
    'max_depth': [3, 5, 10, 20],
    'max_features': ['auto', 'sqrt', 'log2'],
    'random_state': [42],
    'verbose': [0],
    'validation_fraction': [0.1, 0.2],
    'n_iter_no_change': [None, 5, 10],
    'tol': [1e-4, 1e-3],
    'ccp_alpha': [0.0, 0.1, 0.2]
}

# Train and save Gradient Boosting classifier model
gbm_trained_model, gbm_best_params, gbm_best_score, gbm_elapsed_time = train_and_save_
    GradientBoostingClassifier,
    gbm_param_grid,
    gbm_method,
    train_data[features],
    train_data[target],
    model_file_path,
    results_file_path)
```

```
Loaded existing model and hyperparameter results.
{'verbose': 0, 'validation_fraction': 0.2, 'tol': 0.001, 'subsample': 0.8, 'random_st
ate': 42, 'n_iter_no_change': None, 'n_estimators': 50, 'min_samples_split': 5, 'min_
samples_leaf': 5, 'max_features': 'sqrt', 'max_depth': 3, 'loss': 'exponential', 'lea
rning_rate': 0.1, 'criterion': 'friedman_mse', 'ccp_alpha': 0.0}
0.9286549707602338
```

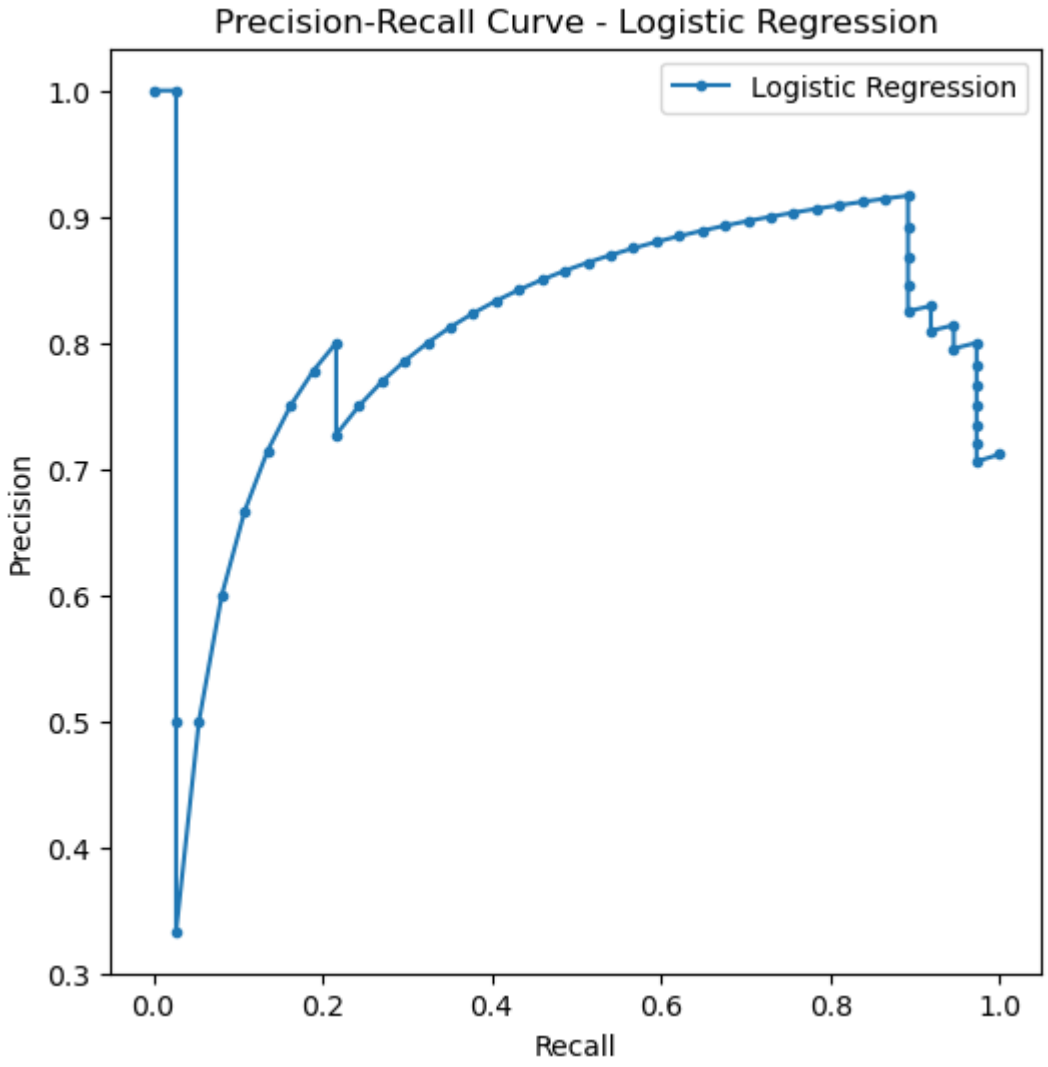
Section 5: Model Evaluation

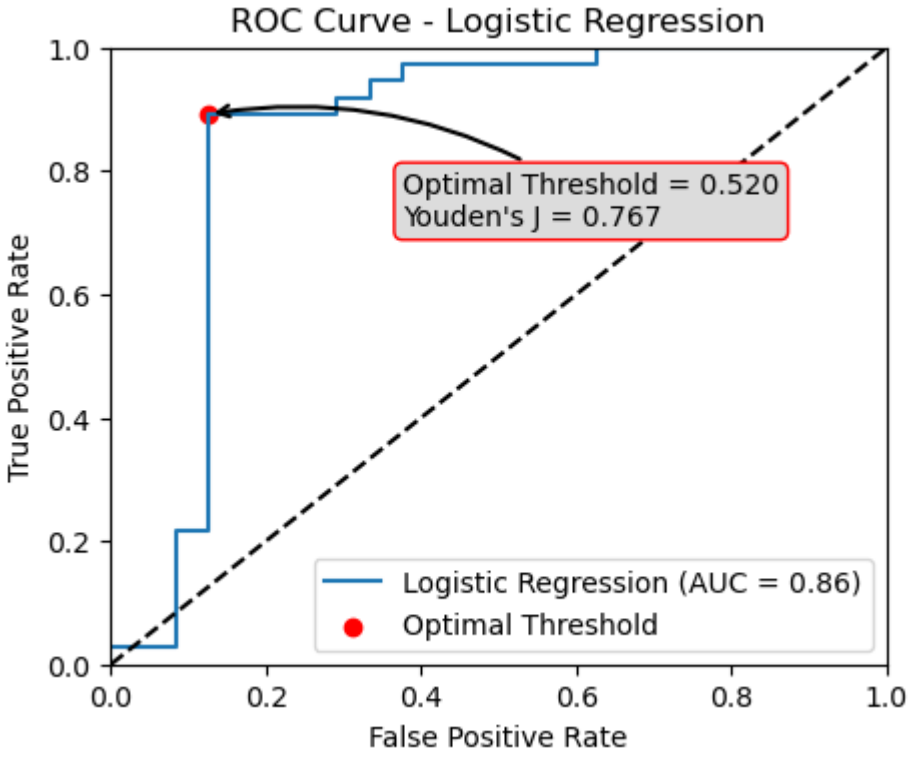
Global Model Variables

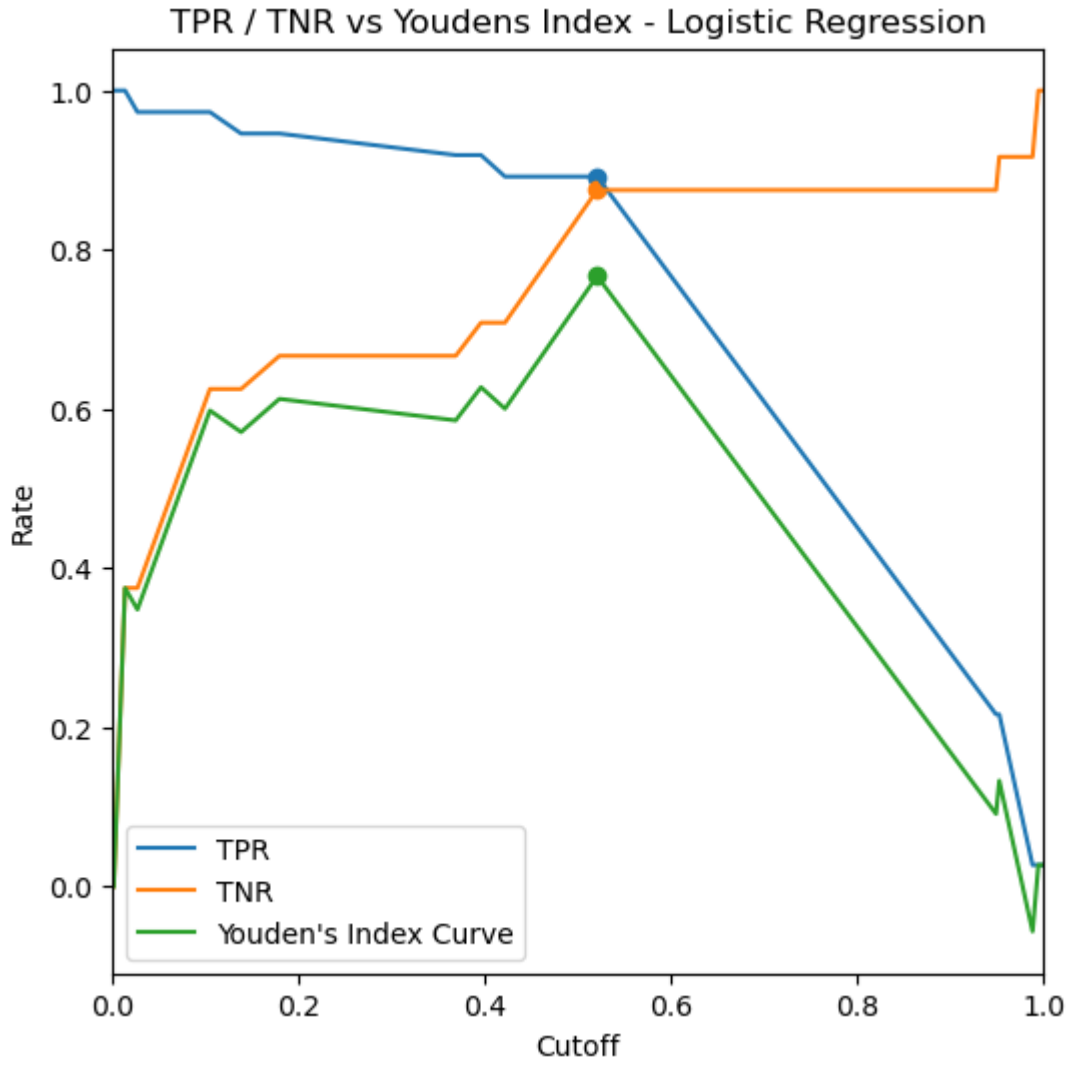
In [27]:
```
# Initialize a dataframe to store the results
results_frame = pd.DataFrame(columns=['Model Name',
                                      'Confusion Matrix',
                                      'Accuracy',
                                      'Precision',
                                      'Recall',
                                      'Specificity',
                                      'F1 Score',
                                      'Youden J Stat',
                                      'Optimal P Threshold'])

# Bundle datasets for model training and evaluation
datasets = [train_data, test_data, features, target]

# Define class_names
class_names = ["Low Heart Risk (0)", "High Heart Risk (1)"]
```

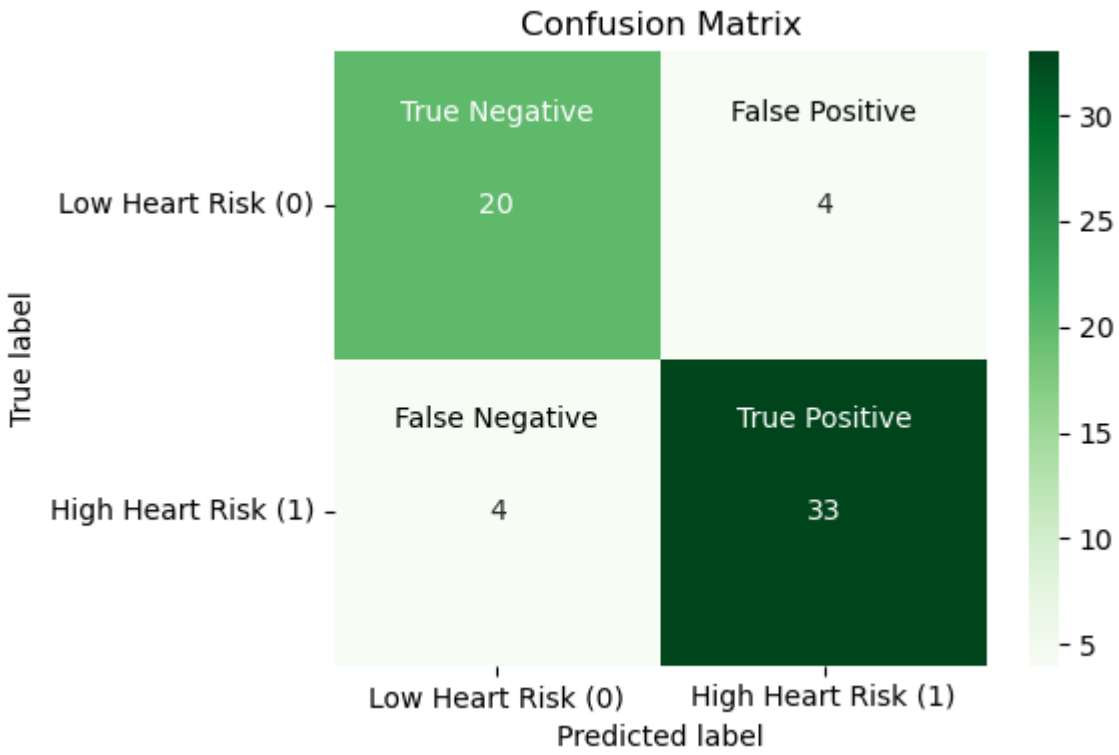
Logistic Regression

In [28]:
```
logreg_model_name, logreg_pr_curve_plot, logreg_roc_curve_plot, logreg_tpr_tnr_plot, l
                                'Logistic Regression',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(logreg_model_name, logreg_pr_curve_plot, lo
results_frame = results_frame.append(logreg_results, ignore_index=True)
```

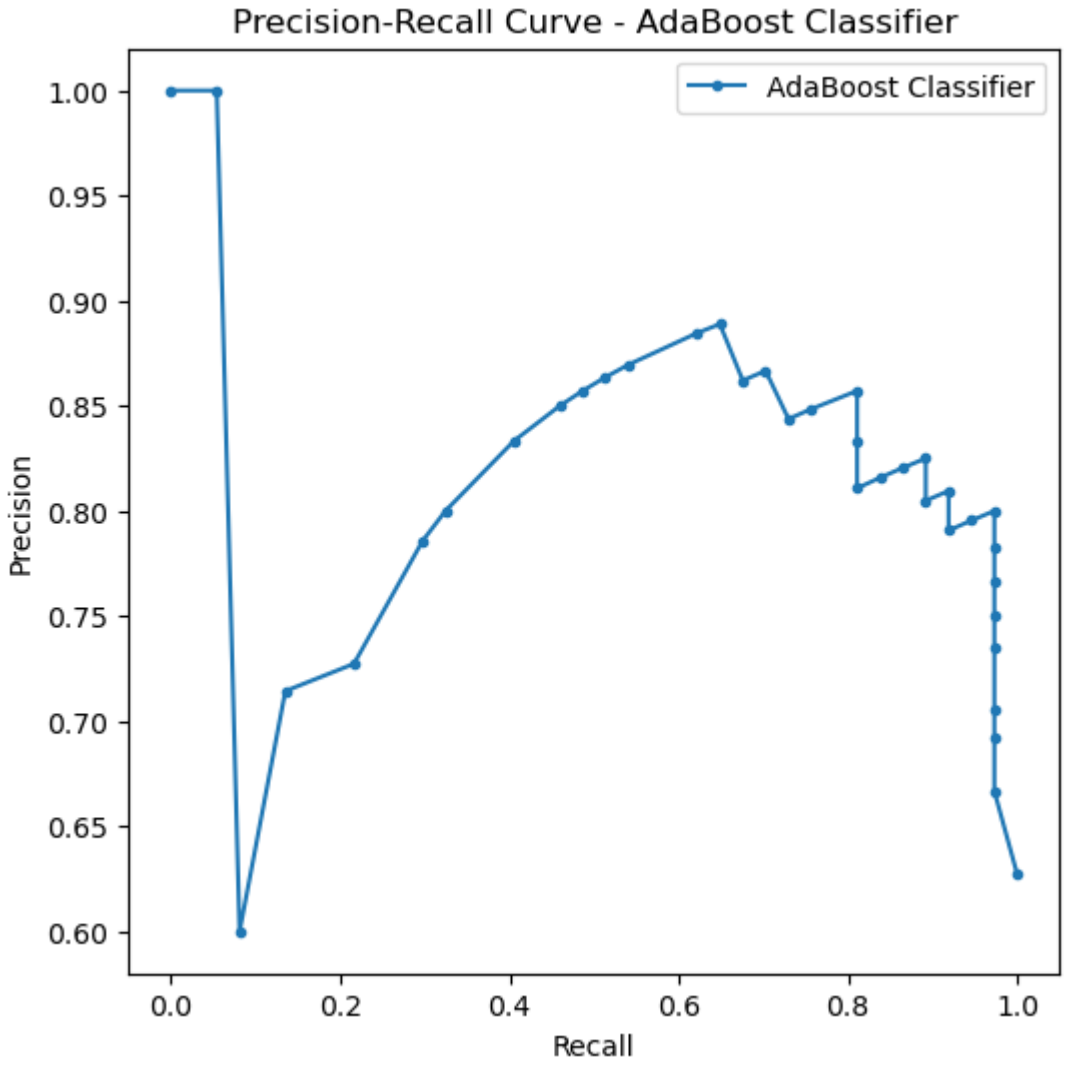

Precision-Recall Curve - Logistic Regression
Logistic Regression
Precision
Recall
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.0
0.2
0.4
0.6
0.8
1.0

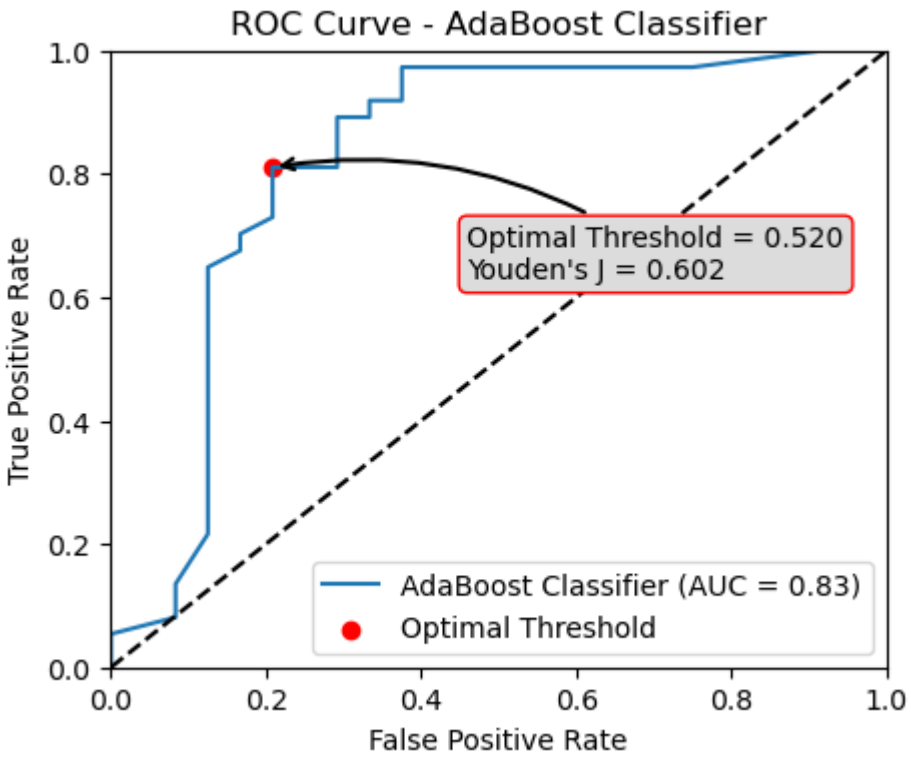

ROC Curve - Logistic Regression
Optimal Threshold = 0.520
Youden's J = 0.767
Logistic Regression (AUC = 0.86)
Optimal Threshold
True Positive Rate
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

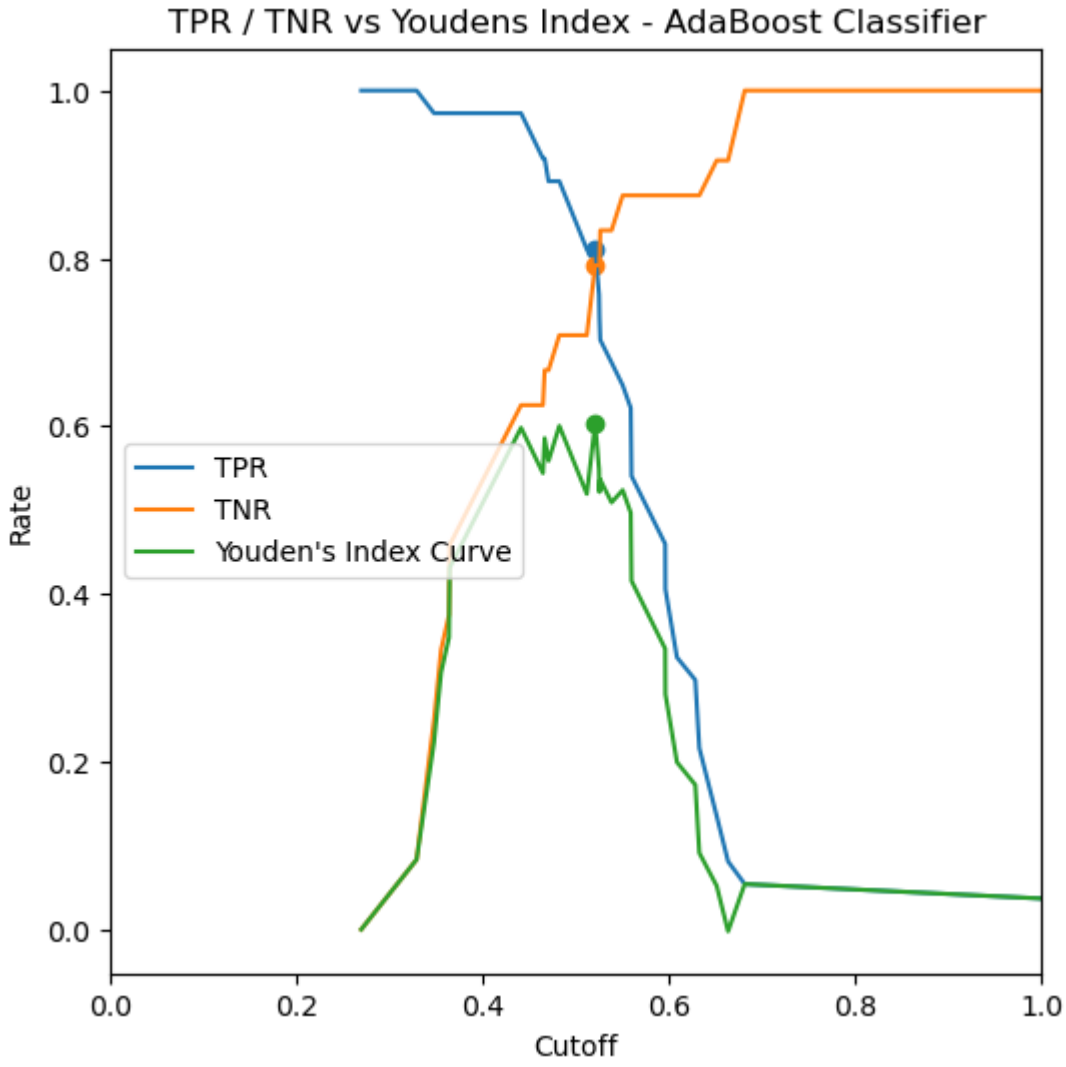

TPR / TNR vs Youdens Index - Logistic Regression
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

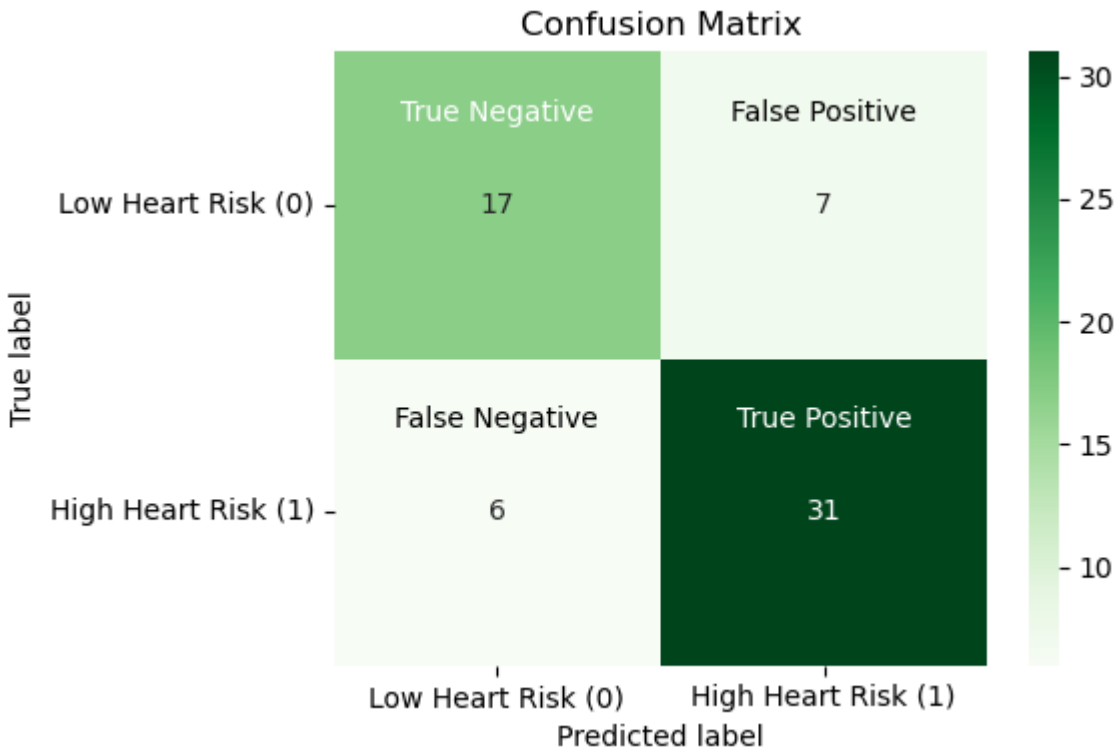

Confusion Matrix
True Negative
False Positive
20
4
False Negative
True Positive
4
33
Low Heart Risk (0)
High Heart Risk (1)
True label
Predicted label
30
25
20
15
10
5

In [29]:
```
abc_model_name, abc_pr_curve_plot, abc_roc_curve_plot, abc_tpr_tnr_plot, abc_conf_matr
                                'AdaBoost Classifier',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(abc_model_name, abc_pr_curve_plot, abc_roc_
results_frame = results_frame.append(abc_results, ignore_index=True)
```

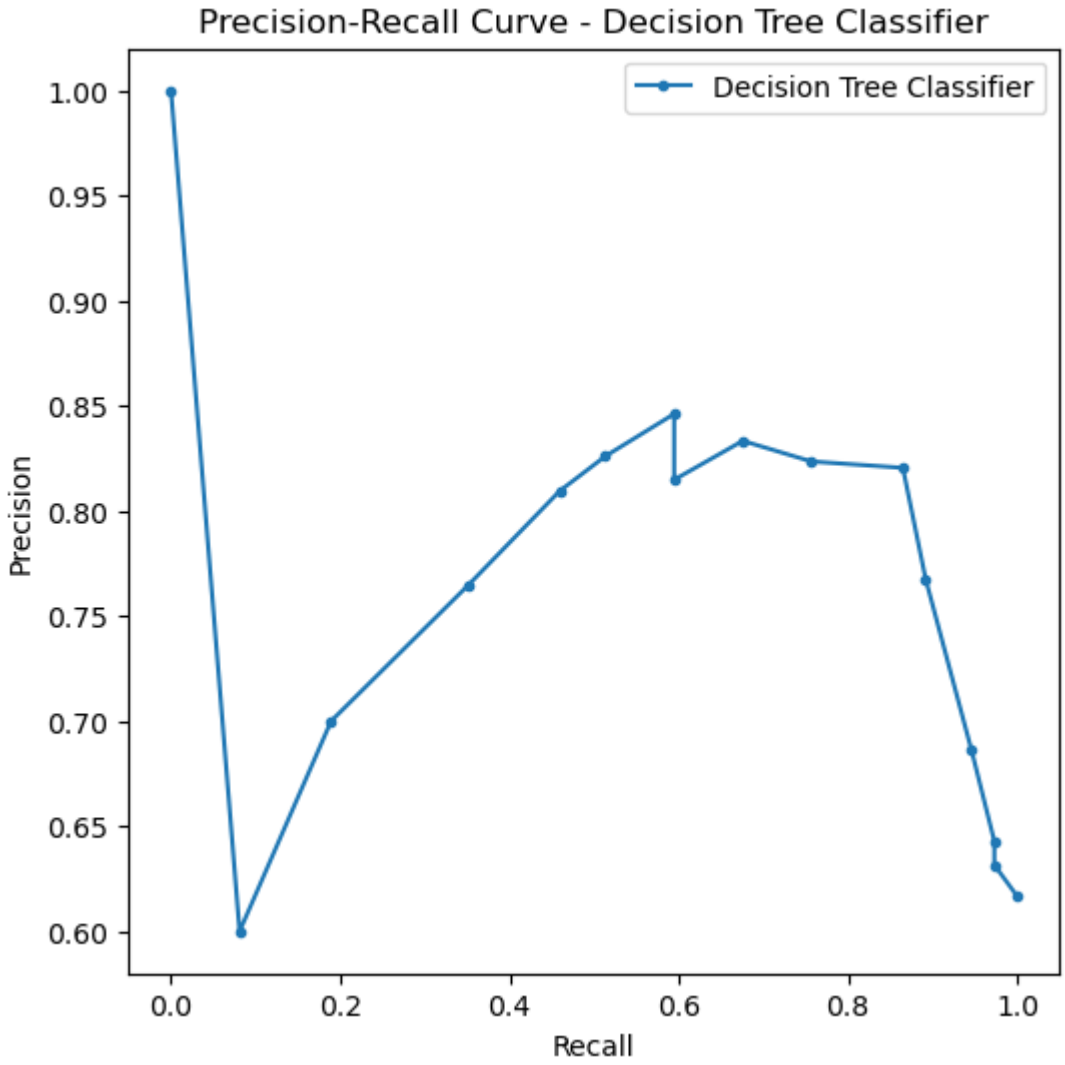


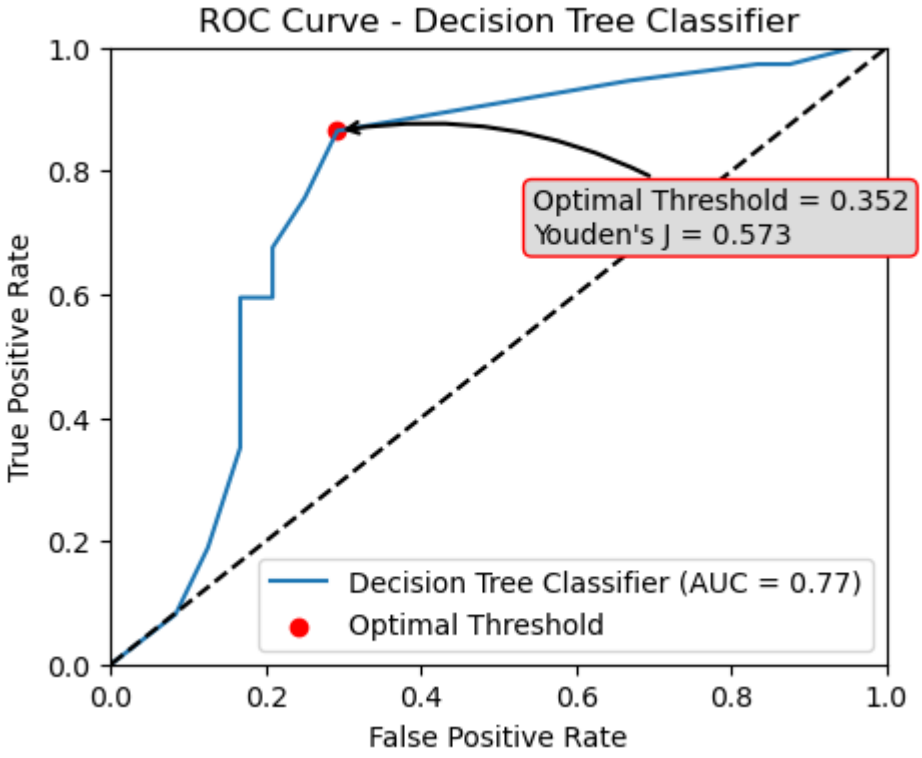

ROC Curve - AdaBoost Classifier
True Positive Rate
False Positive Rate
Optimal Threshold = 0.520
Youden's J = 0.602
AdaBoost Classifier (AUC = 0.83)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

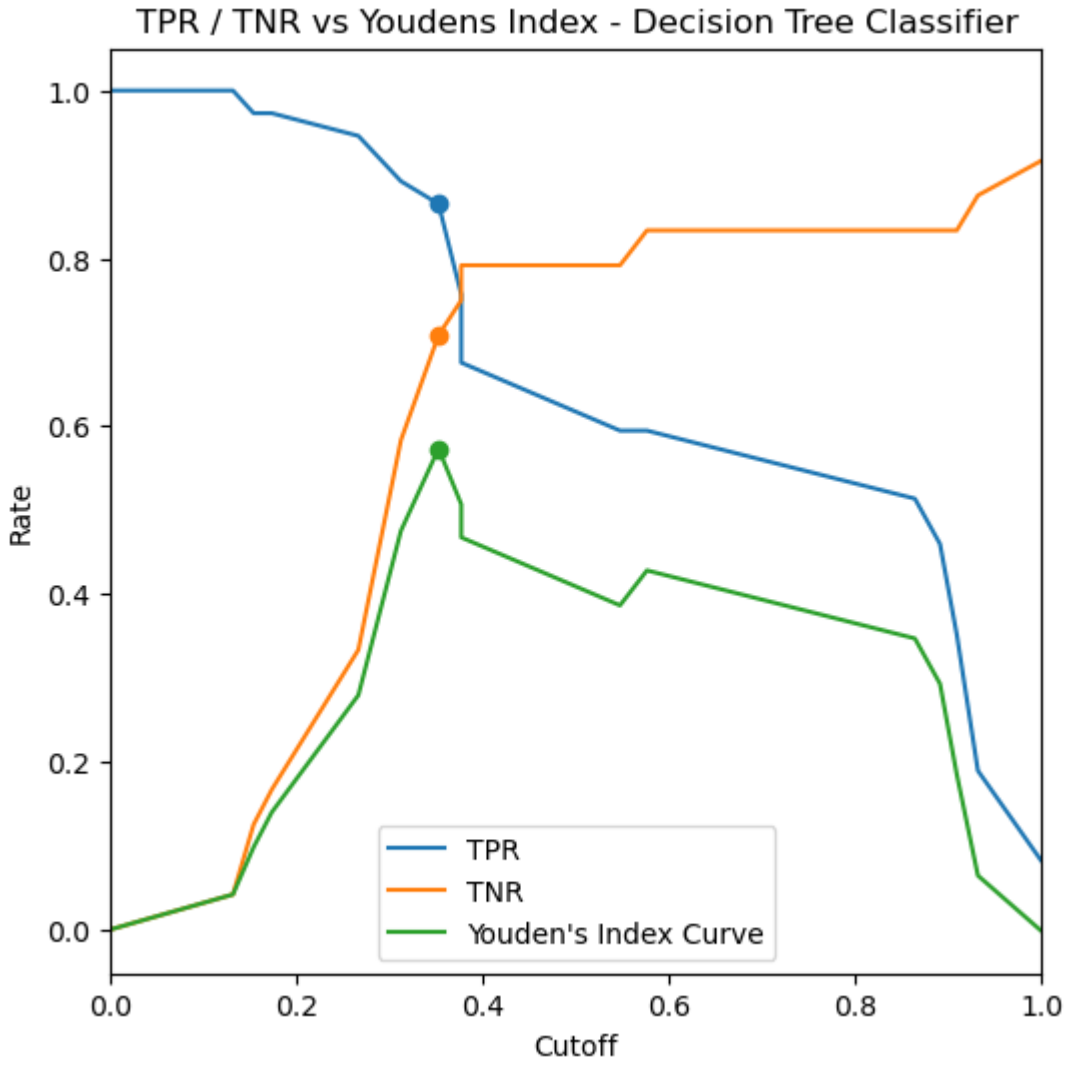

TPR / TNR vs Youdens Index - AdaBoost Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

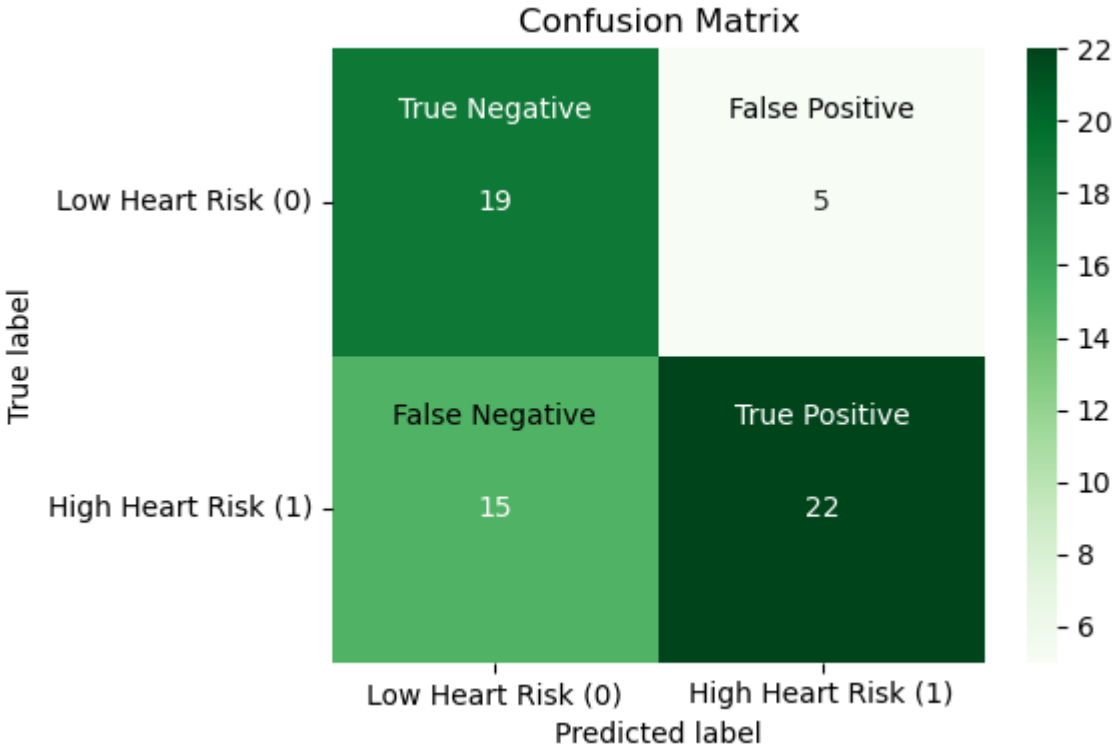


In [30]:
```
dt_model_name, dt_pr_curve_plot, dt_roc_curve_plot, dt_tpr_tnr_plot, dt_conf_matrix, d
                                'Decision Tree Classifier',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(dt_model_name, dt_pr_curve_plot, dt_roc_cur
results_frame = results_frame.append(dt_results, ignore_index=True)
```

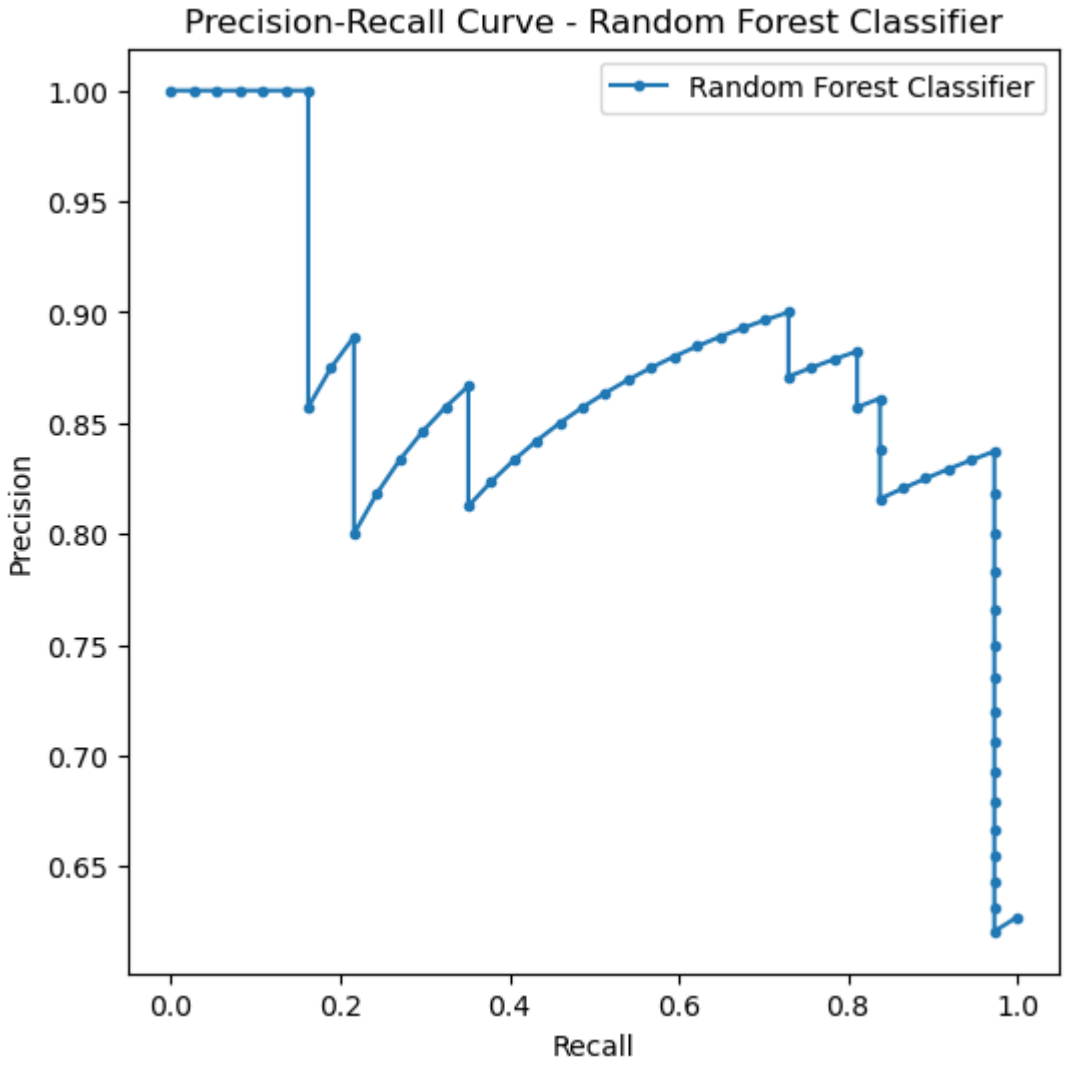

Precision-Recall Curve - Decision Tree Classifier
Decision Tree Classifier
Precision
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
0.2
0.4
0.6
0.8
1.0

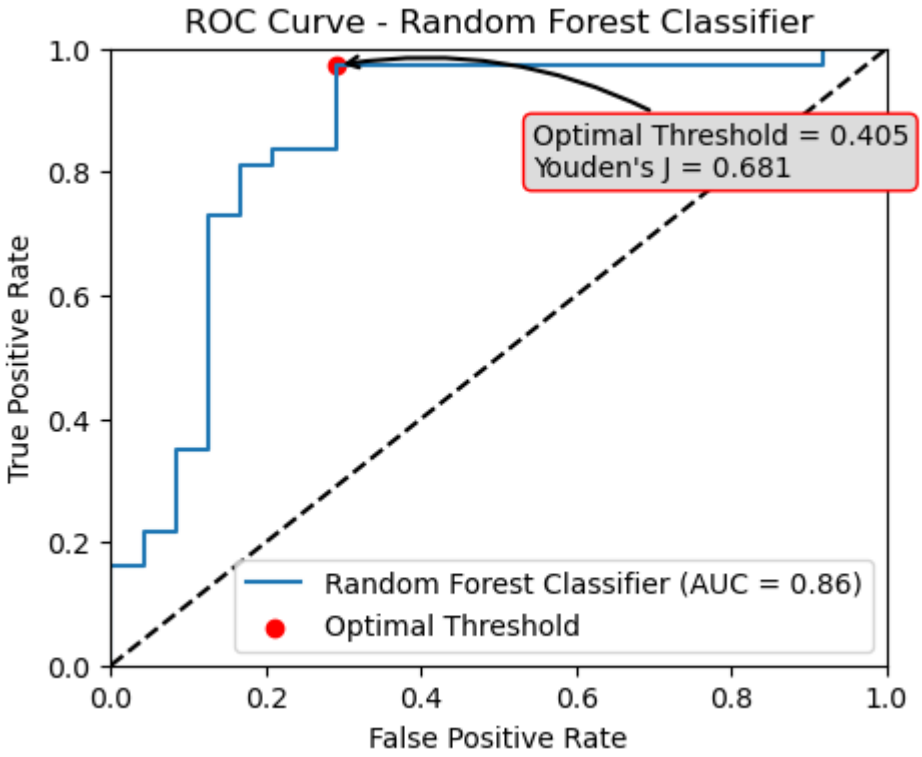

ROC Curve - Decision Tree Classifier
Optimal Threshold = 0.352
Youden's J = 0.573
Decision Tree Classifier (AUC = 0.77)
Optimal Threshold
True Positive Rate
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

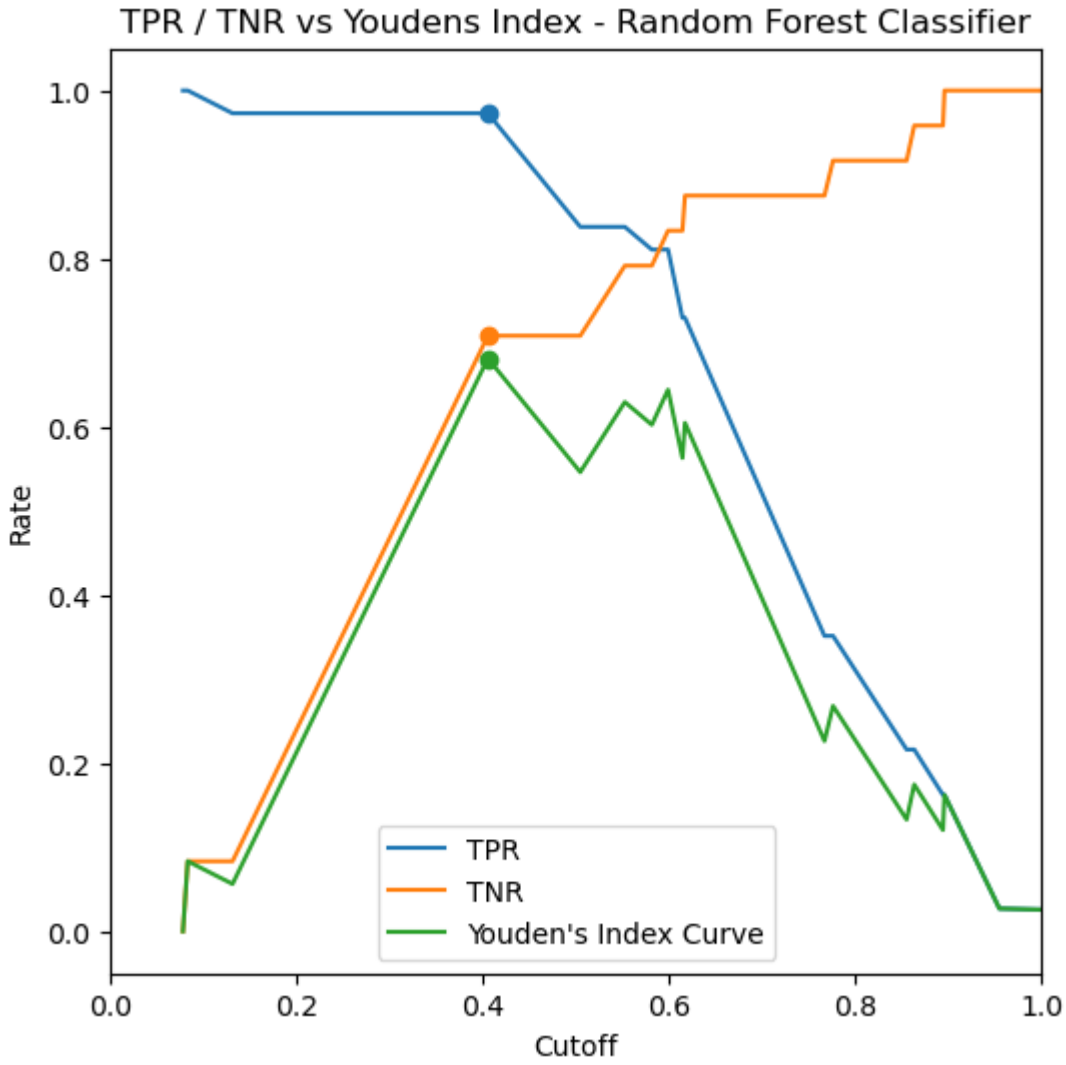

TPR / TNR vs Youdens Index - Decision Tree Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

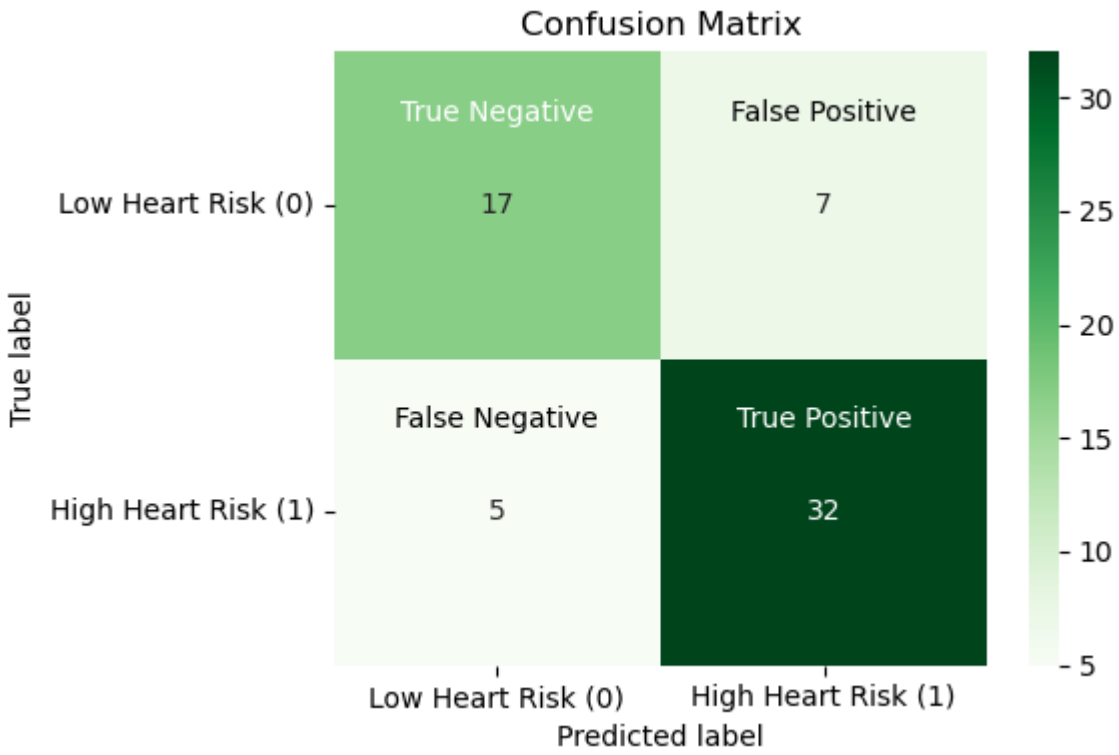

Confusion Matrix
True Negative
False Positive
19
5
False Negative
True Positive
15
22
Low Heart Risk (0)
High Heart Risk (1)
True label
Predicted label
22
20
18
16
14
12
10
8
6

In [31]:
```
rfc_model_name, rfc_pr_curve_plot, rfc_roc_curve_plot, rfc_tpr_tnr_plot, rfc_conf_matr
                                'Random Forest Classifier',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(rfc_model_name, rfc_pr_curve_plot, rfc_roc_
results_frame = results_frame.append(rfc_results, ignore_index=True)
```

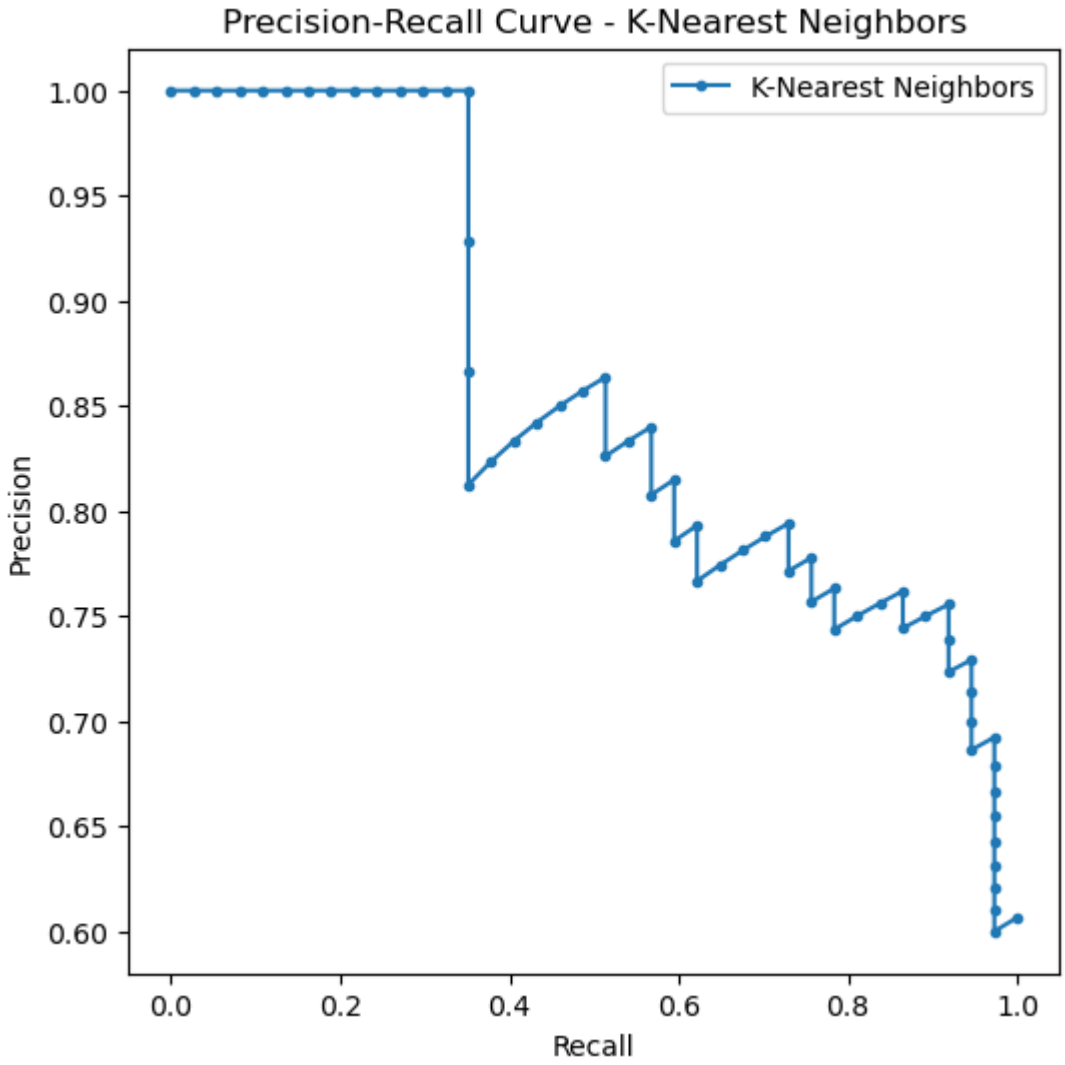


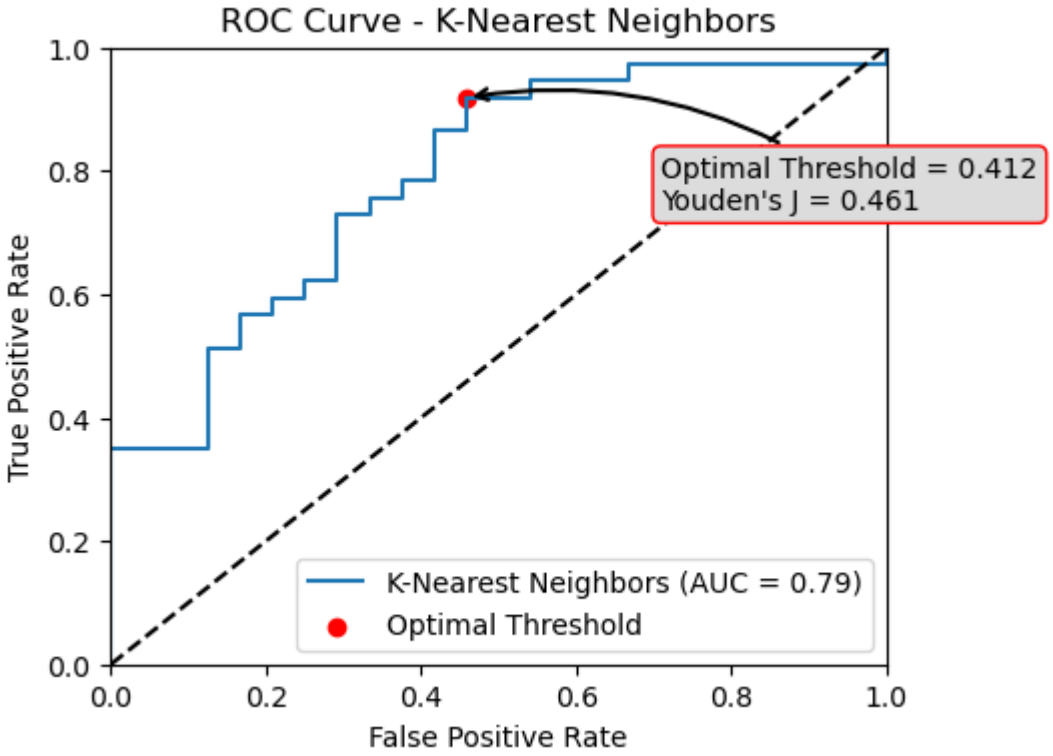

ROC Curve - Random Forest Classifier
Optimal Threshold = 0.405
Youden's J = 0.681
Random Forest Classifier (AUC = 0.86)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

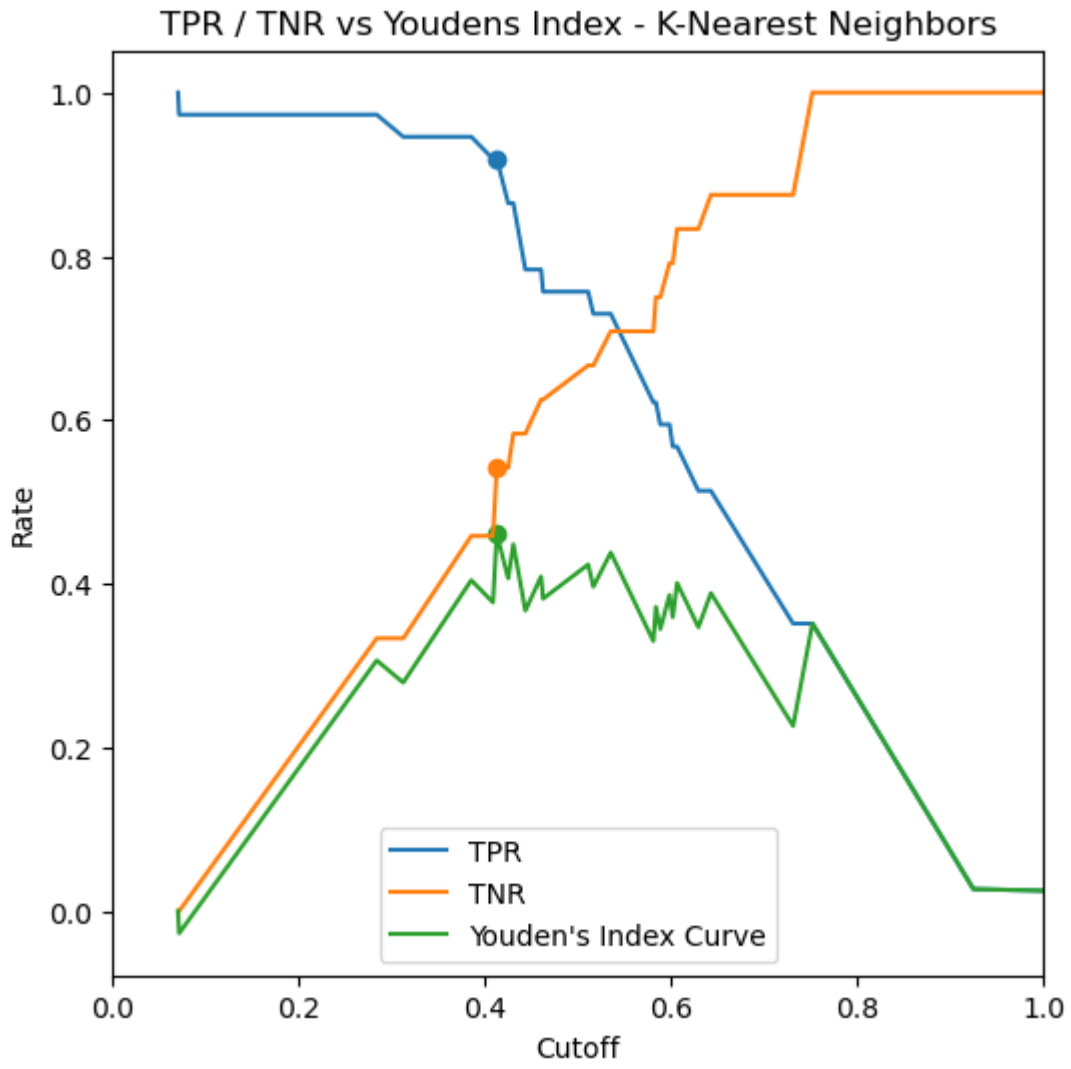

TPR / TNR vs Youdens Index - Random Forest Classifier
TPR
TNR
Youden's Index Curve
Rate
Cutoff
0.0
0.2
0.4
0.6
0.8
1.0

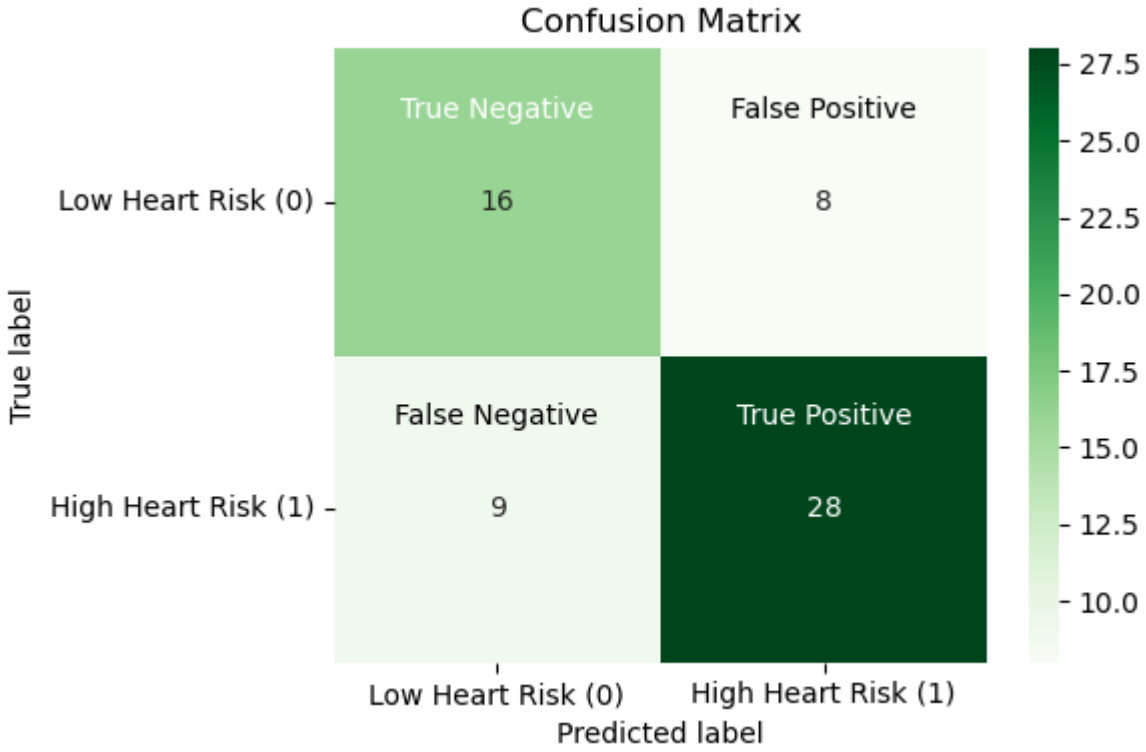


In [32]:
```
knn_model_name, knn_pr_curve_plot, knn_roc_curve_plot, knn_tpr_tnr_plot, knn_conf_matr
                                'K-Nearest Neighbors',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(knn_model_name, knn_pr_curve_plot, knn_roc_
results_frame = results_frame.append(knn_results, ignore_index=True)
```

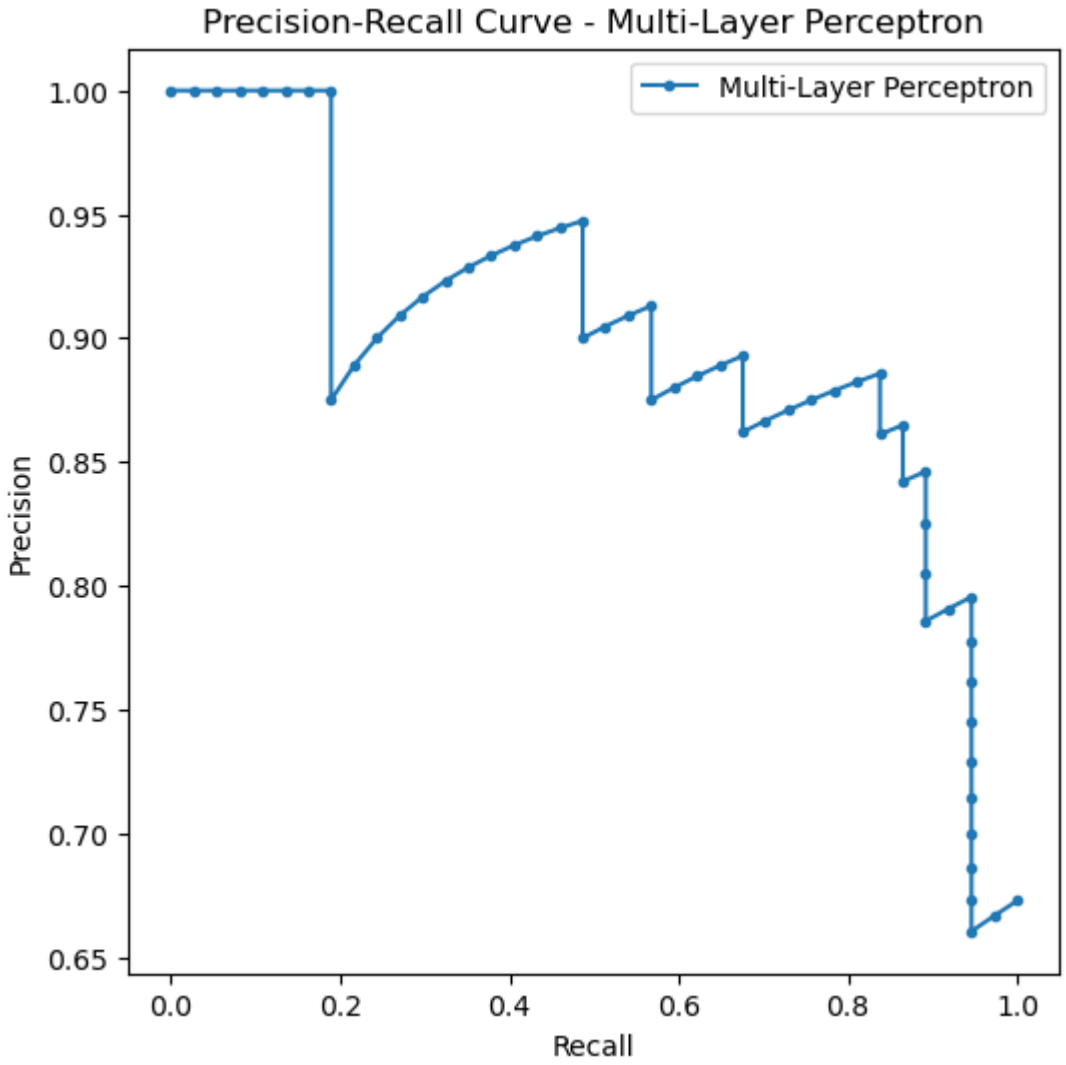

Precision-Recall Curve - K-Nearest Neighbors
K-Nearest Neighbors
Precision
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
0.2
0.4
0.6
0.8
1.0

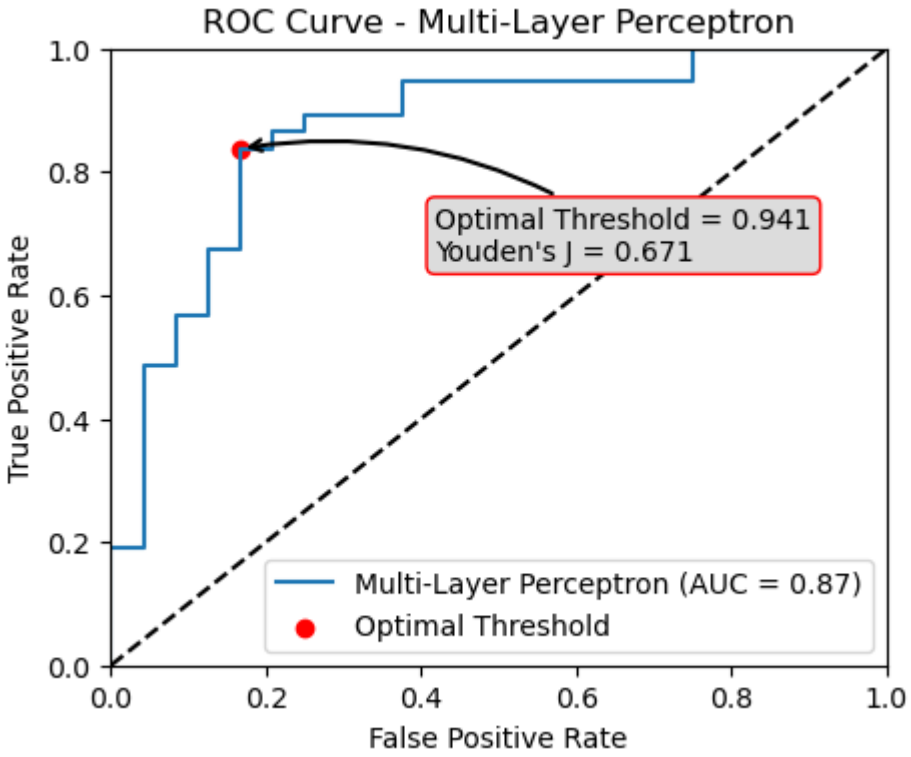

ROC Curve - K-Nearest Neighbors
Optimal Threshold = 0.412
Youden's J = 0.461
K-Nearest Neighbors (AUC = 0.79)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

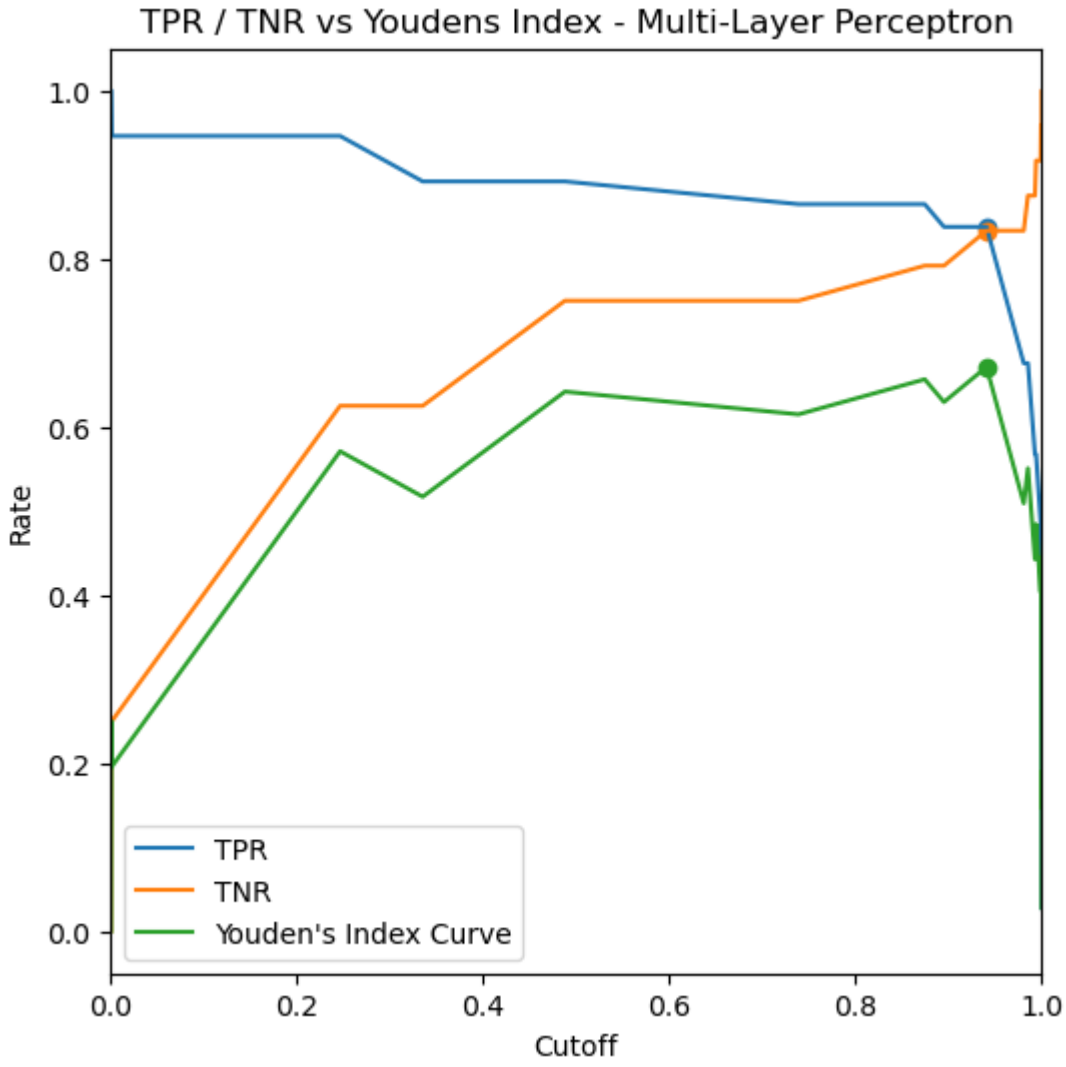

TPR / TNR vs Youdens Index - K-Nearest Neighbors
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

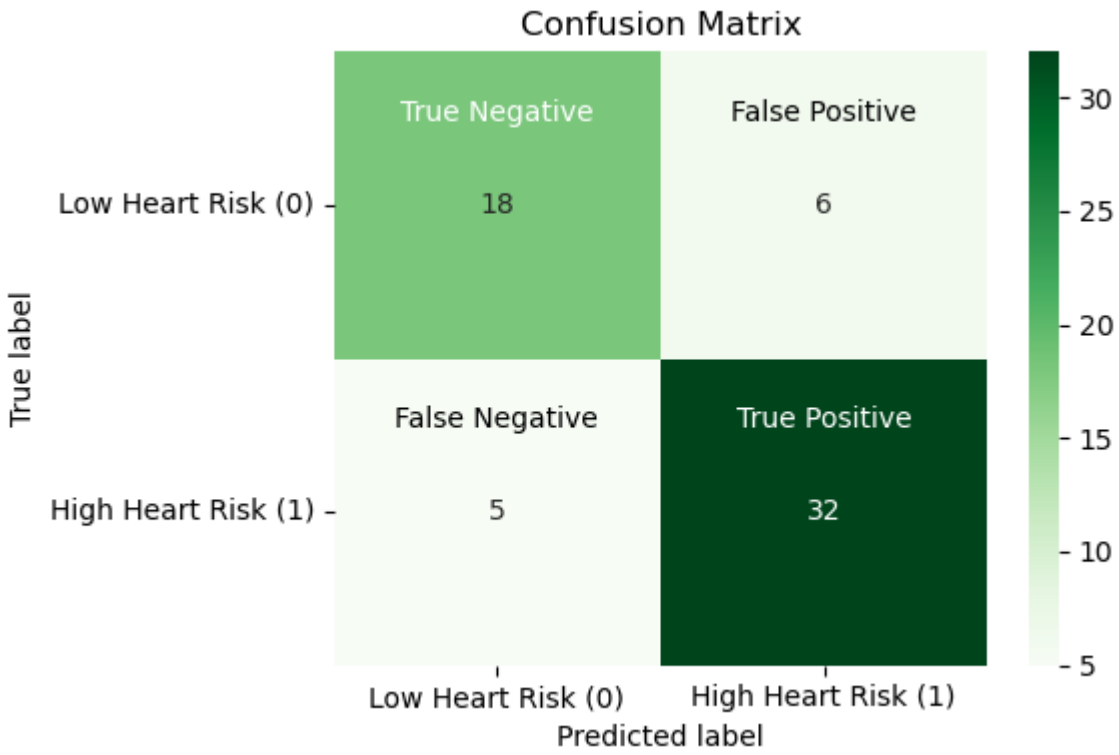

Confusion Matrix
True Negative
False Positive
16
8
False Negative
True Positive
9
28
Low Heart Risk (0)
High Heart Risk (1)
True label
Predicted label
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0

In [33]:
```
mlp_model_name, mlp_pr_curve_plot, mlp_roc_curve_plot, mlp_tpr_tnr_plot, mlp_conf_matr
                                'Multi-Layer Perceptron',
                                class_names,
                                datasets,
                                "Full")
def list_to_paragraph(value_list):
    style = getSampleStyleSheet()["BodyText"]
    return [Paragraph(str(value), style) for value in value_list]

results_frame = results_frame.append(mlp_results, ignore_index=True)
```

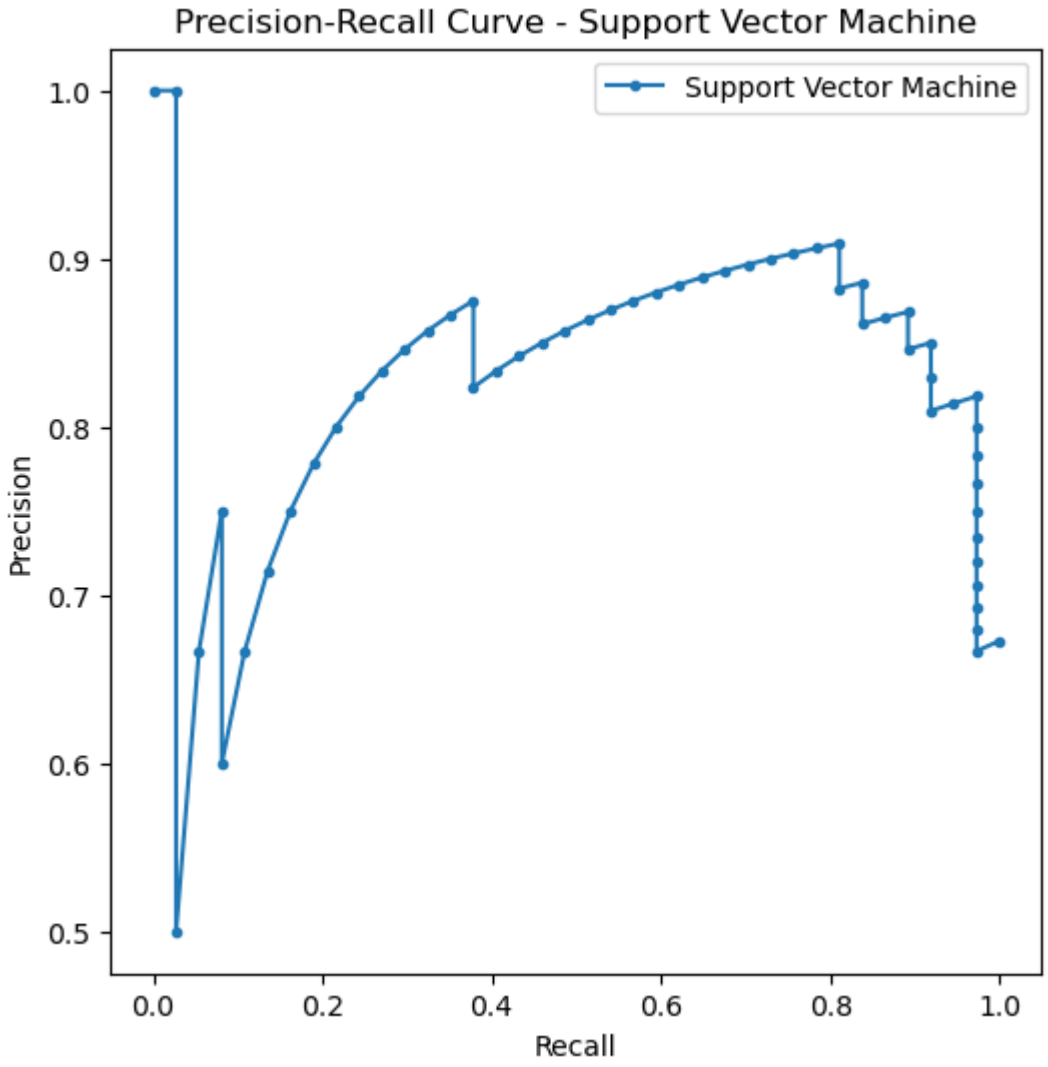


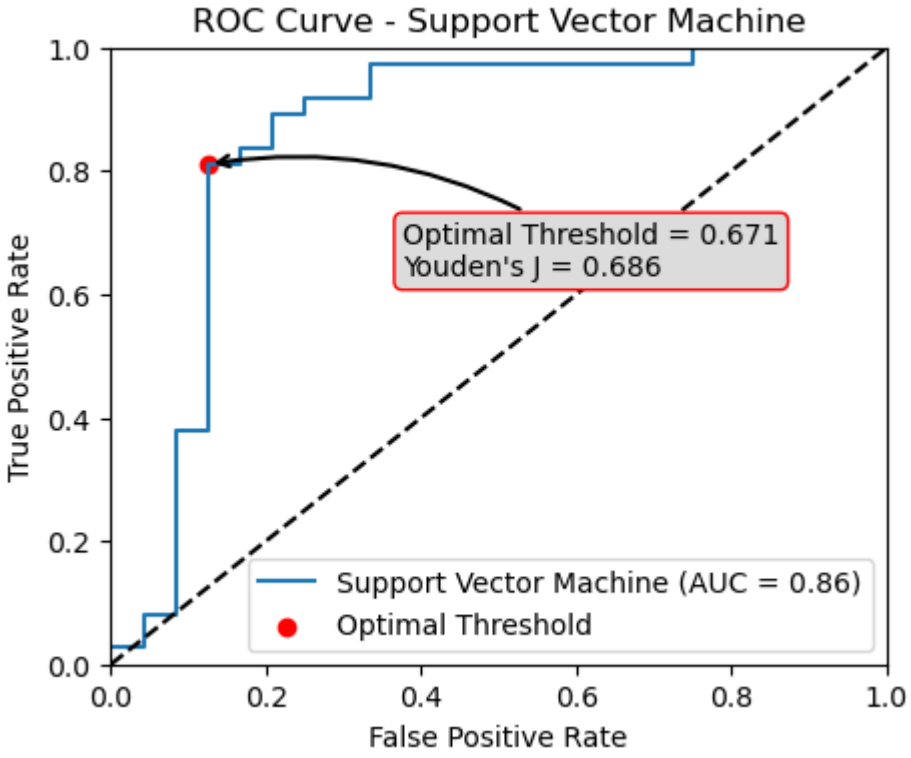

ROC Curve - Multi-Layer Perceptron
Optimal Threshold = 0.941
Youden's J = 0.671
Multi-Layer Perceptron (AUC = 0.87)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

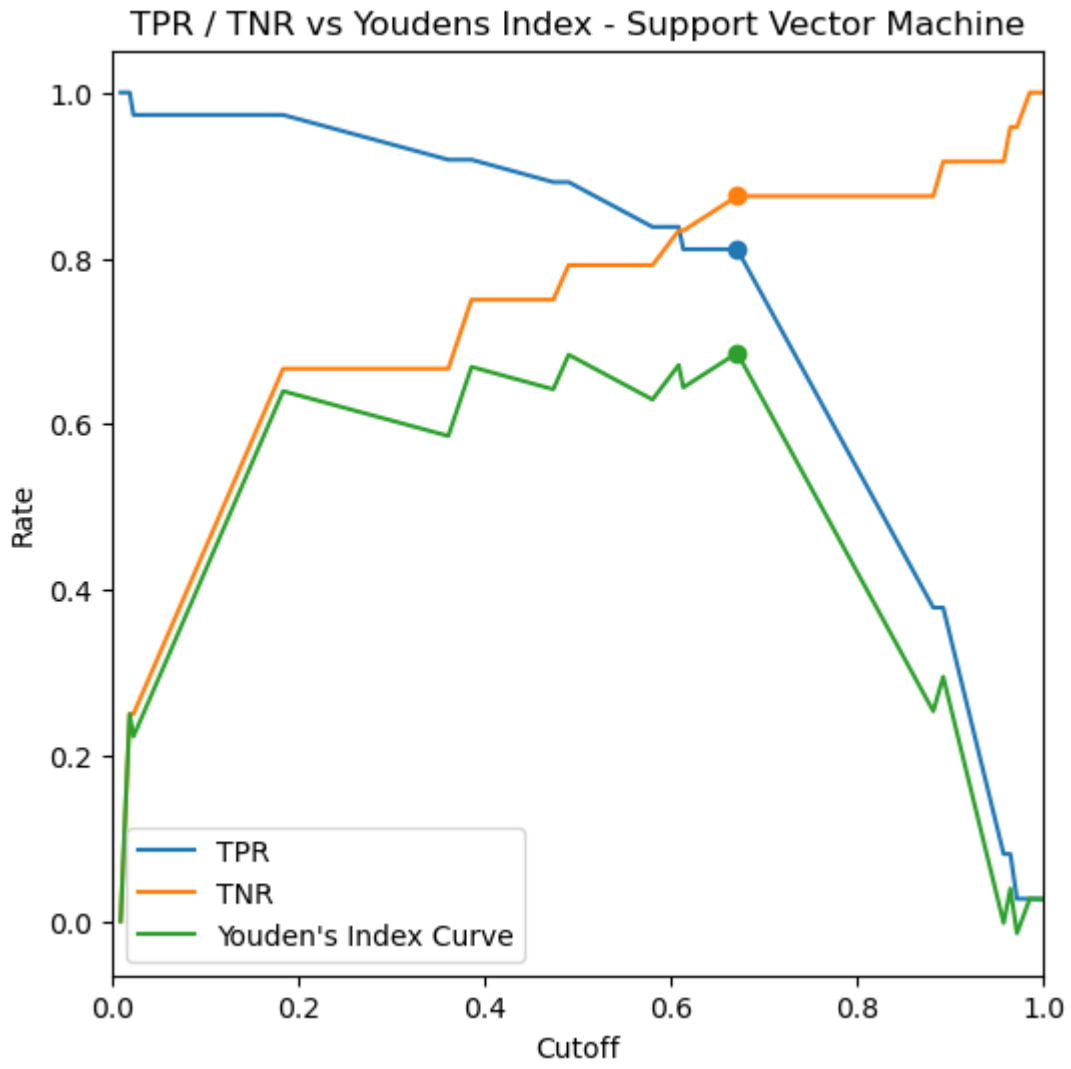

TPR / TNR vs Youdens Index - Multi-Layer Perceptron
TPR
TNR
Youden's Index Curve
Rate
Cutoff
0.0
0.2
0.4
0.6
0.8
1.0

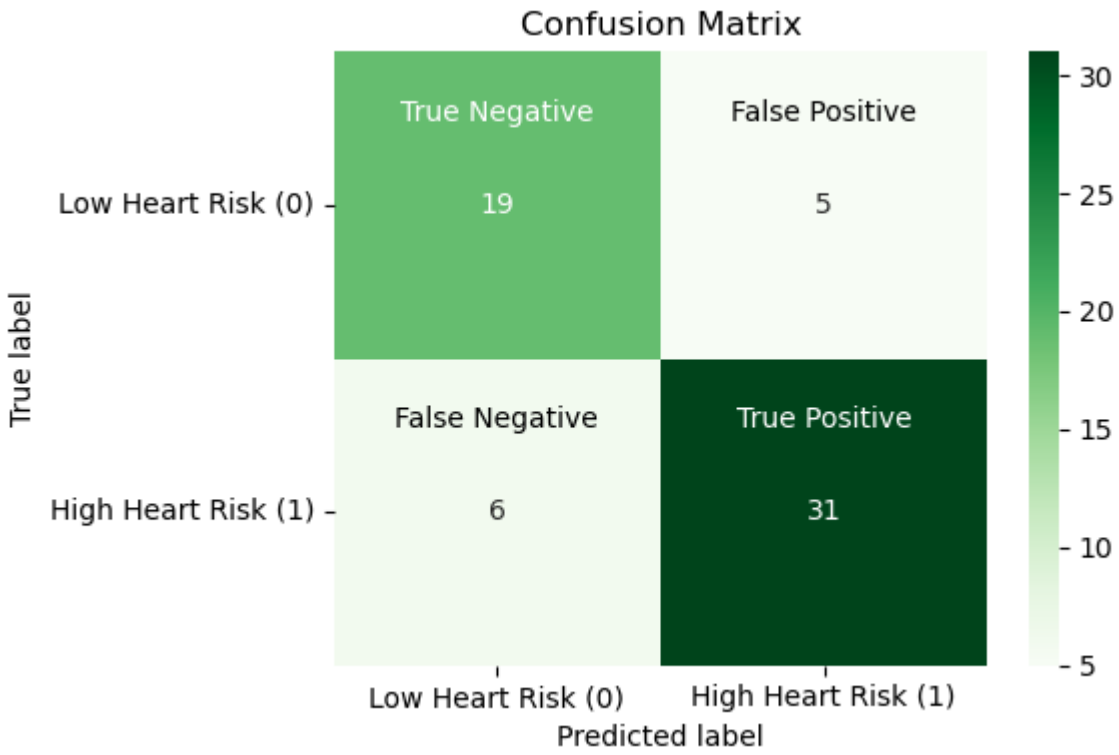


In [34]:
```
svm_model_name, svm_pr_curve_plot, svm_roc_curve_plot, svm_tpr_tnr_plot, svm_conf_matr
                                'Support Vector Machine',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(svm_model_name, svm_pr_curve_plot, svm_roc_
results_frame = results_frame.append(svm_results, ignore_index=True)
```


Precision-Recall Curve - Support Vector Machine
Support Vector Machine
Precision
Recall
1.0
0.9
0.8
0.7
0.6
0.5
0.0
0.2
0.4
0.6
0.8
1.0


ROC Curve - Support Vector Machine
Optimal Threshold = 0.671
Youden's J = 0.686
Support Vector Machine (AUC = 0.86)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


TPR / TNR vs Youdens Index - Support Vector Machine
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix
True Negative
False Positive
19
5
False Negative
True Positive
6
31
Low Heart Risk (0)
High Heart Risk (1)
True label
Predicted label
30
25
20
15
10
5

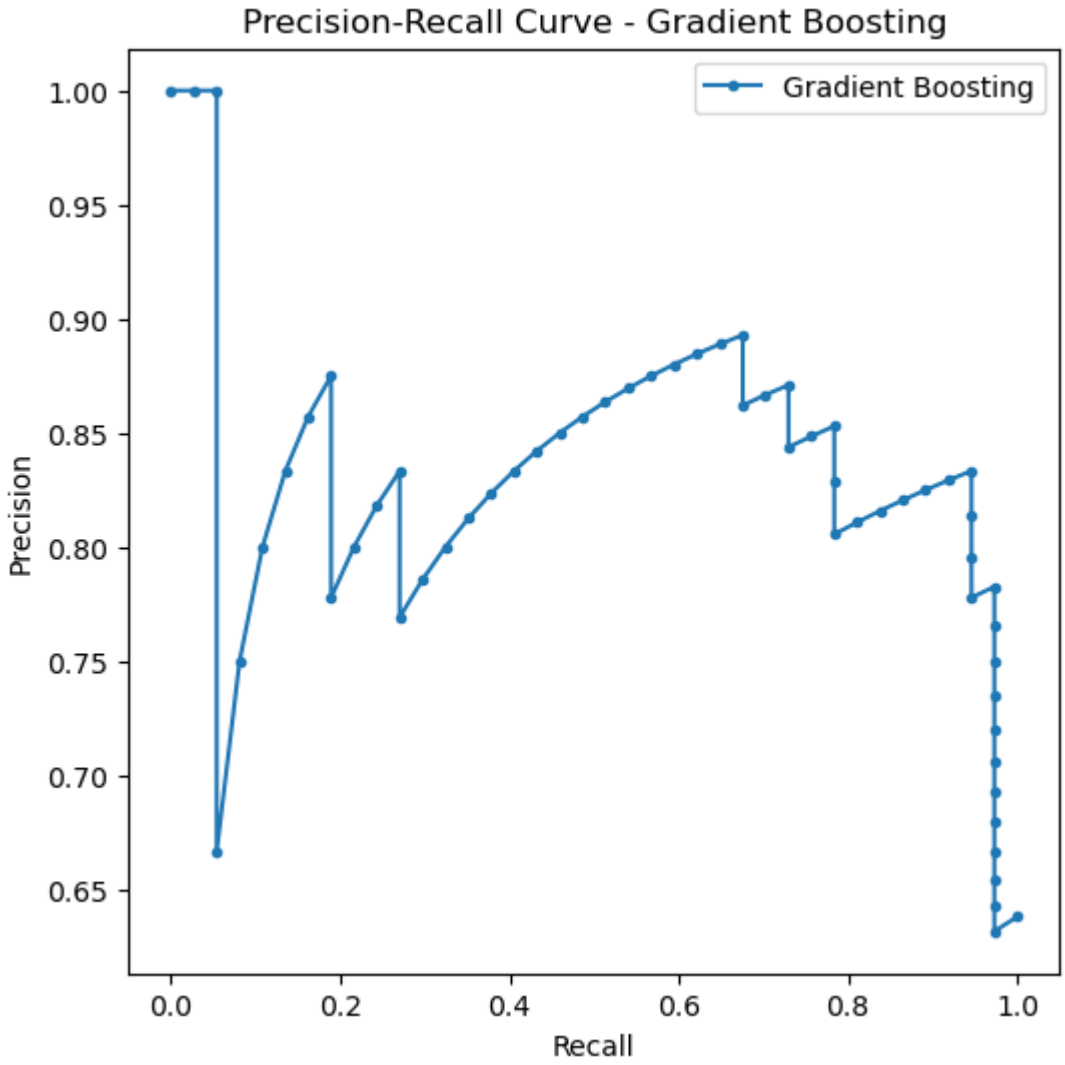
Gradient Boost Results

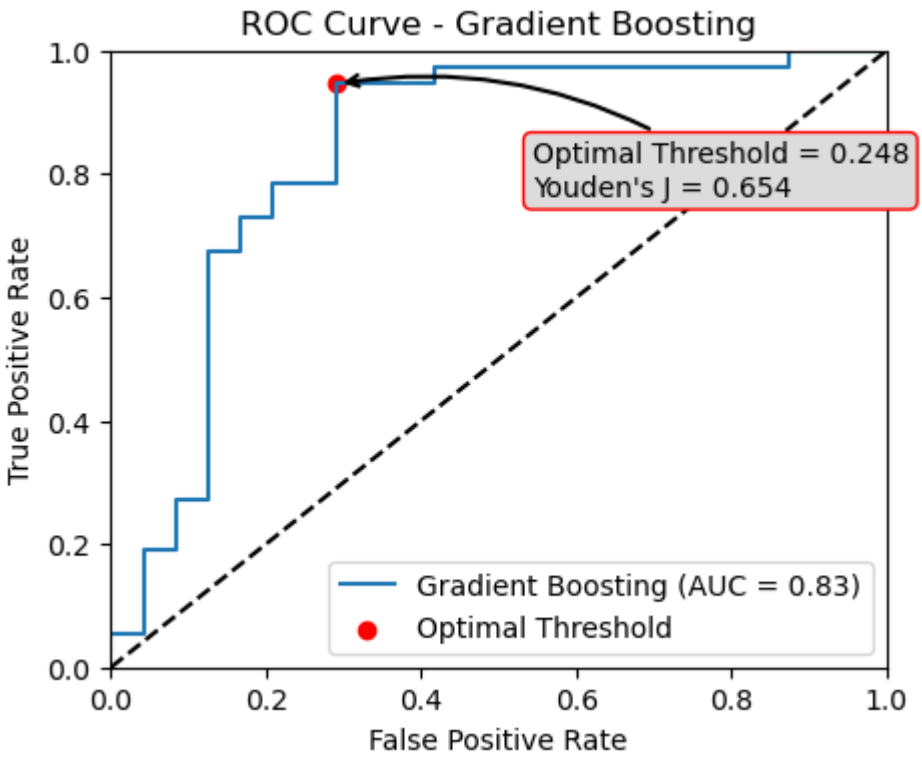
Based on the confusion matrix, the gradient boost classifier correctly predicted 17 true negatives and 32 true positives, while misclassifying 5 false negatives and 7 false positives. This resulted in an accuracy score of 0.803279, indicating that the model correctly classified 80.3% of the total observations.

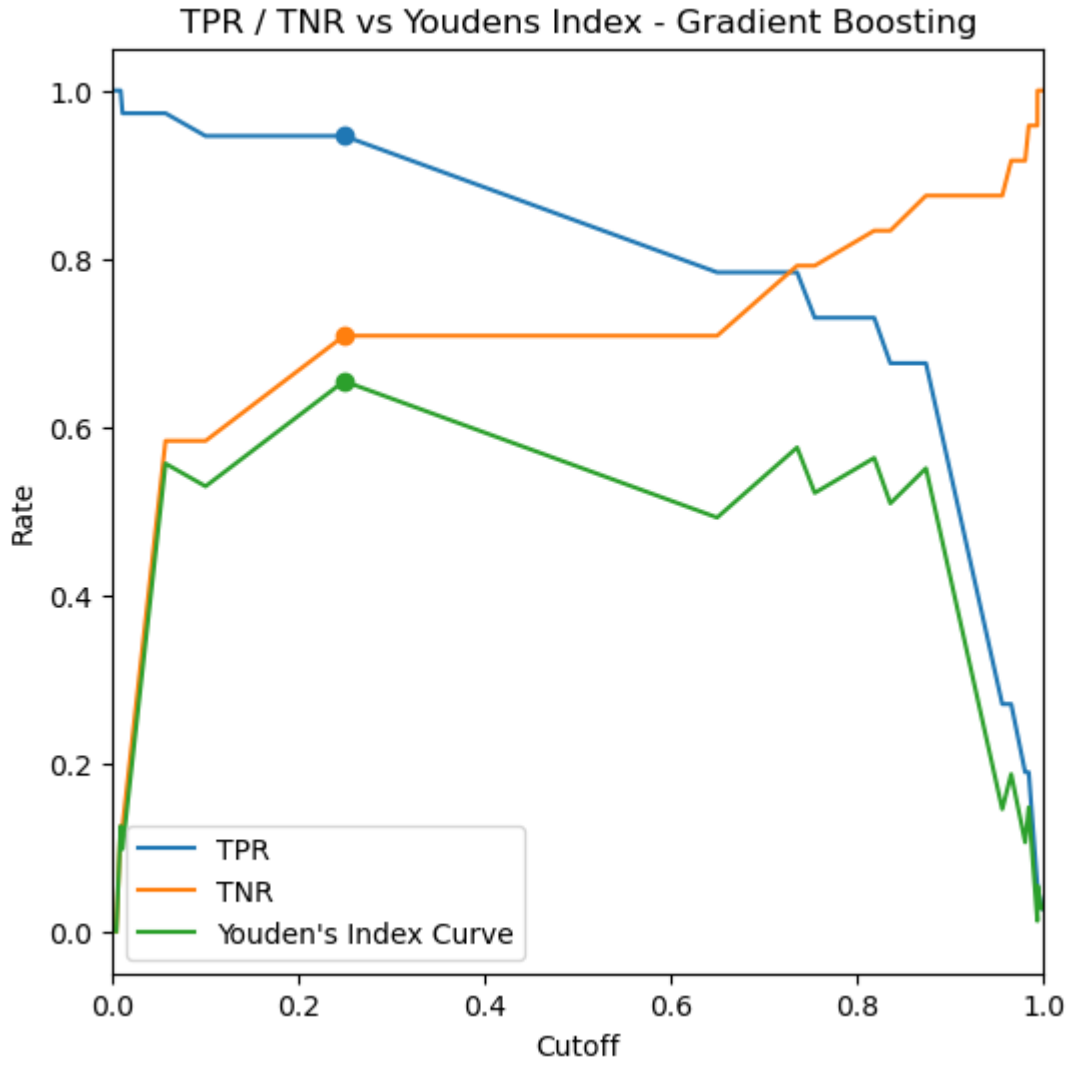
The precision score of 0.820513 indicates that out of all the positive predictions made by the model, 82.05% were actually true positives. The recall score of 0.864865 indicates that the model correctly identified 86.49% of all actual positives. The F1 score of 0.842105, which is the harmonic mean of precision and recall, suggests that the model achieved a good balance between these two metrics.

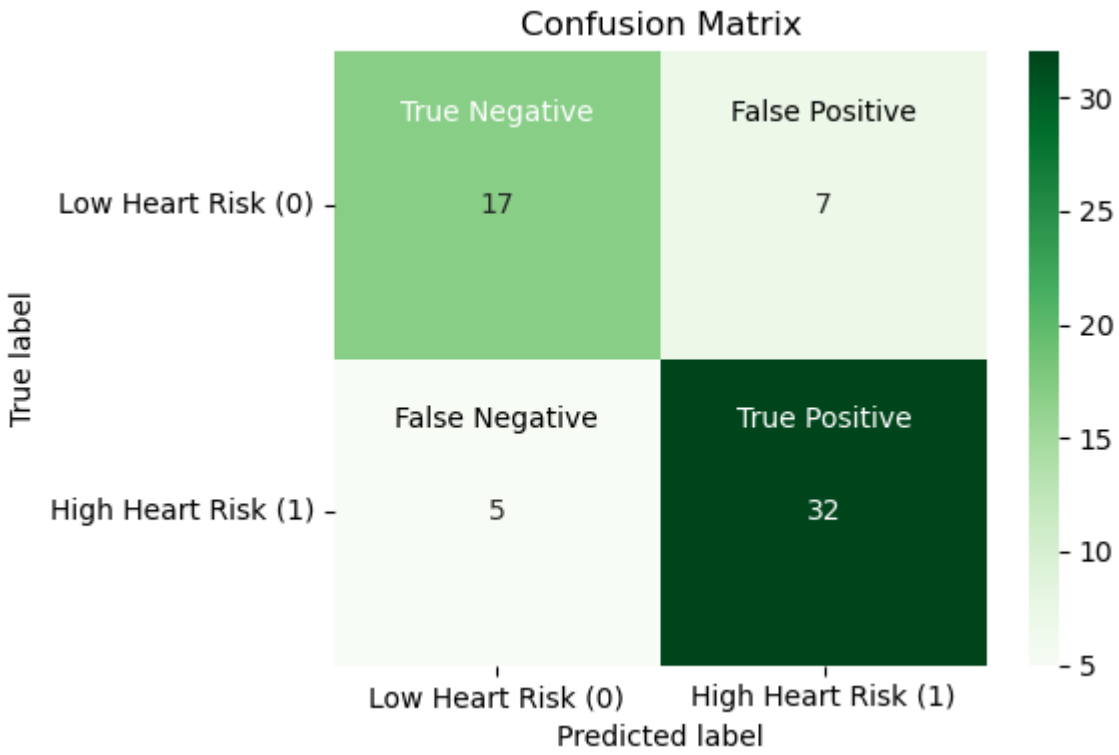
The Youden J statistic of 0.654279 indicates that the model has a good balance between the true positive rate and the false positive rate. The optimal probability threshold of 0.248343 suggests that a lower threshold should be used to increase the sensitivity of the model. Overall, the gradient boost classifier performs reasonably well with a balanced precision and recall score, and can be used for further analysis and prediction.

In [35]:
```
gbm_model_name, gbm_pr_curve_plot, gbm_roc_curve_plot, gbm_tpr_tnr_plot, gbm_conf_matr
                                'Gradient Boosting',
                                class_names,
                                datasets,
                                "full")

model_pdf_report.save_model_results_to_pdf(gbm_model_name, gbm_pr_curve_plot, gbm_roc_
results_frame = results_frame.append(gbm_results, ignore_index=True)
```


Precision-Recall Curve - Gradient Boosting
Gradient Boosting
Precision
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.0
0.2
0.4
0.6
0.8
1.0


ROC Curve - Gradient Boosting
Optimal Threshold = 0.248
Youden's J = 0.654
Gradient Boosting (AUC = 0.83)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


TPR / TNR vs Youdens Index - Gradient Boosting
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix
True Negative
False Positive
17
7
False Negative
True Positive
5
32
Low Heart Risk (0)
High Heart Risk (1)
True label
Predicted label
30
25
20
15
10
5

Section 6: Analysis Summary

The table presents the performance of eight different binary classification models. Among the models, Logistic Regression performs the best in terms of Accuracy, Precision, Recall, F1 Score, and Youden J Stat. It has an accuracy of 86.89%, precision of 89.19%, recall of 89.19%, and F1 score of 89.19%. The Youden J Stat, which is a combined measure of sensitivity (recall) and specificity, is 76.69% for the Logistic Regression model. The second-best model in terms of these performance metrics is the Support Vector Machine, which has an accuracy of 81.97%, precision of 86.11%, recall of 83.78%, F1 score of 84.93%, and a Youden J Stat of 68.58%.

Looking closely at the performance of the best model, Logistic Regression, the confusion matrix shows that it correctly classified 20 true negative and 33 true positive instances, with 4 false positives and 4 false negatives. This demonstrates a good balance between specificity (83.33%) and sensitivity (89.19%), which is important for a reliable binary classification model. The optimal probability threshold for this model is 0.52, which is close to the standard threshold of 0.5, indicating that the model's default decision boundary is already well-calibrated. Overall, the Logistic Regression model's high accuracy, precision, recall, and F1 score demonstrate its efficacy in handling this binary classification task and make it the most suitable choice among the tested models.

In [36]:
```
results_frame
```

Out[36]:

	Model Name	Confusion Matrix	Accuracy	Precision	Recall	Specificity	F1 Score	Youden J Stat	Optimal P Threshold
0	Logistic Regression	[[20, 4], [4, 33]]	0.868852	0.891892	0.891892	0.833333	0.891892	0.766892	0.520191
1	AdaBoost Classifier	[[17, 7], [6, 31]]	0.786885	0.815789	0.837838	0.708333	0.826667	0.602477	0.520231
2	Decision Tree Classifier	[[19, 5], [15, 22]]	0.672131	0.814815	0.594595	0.791667	0.687500	0.573198	0.351978
3	Random Forest Classifier	[[17, 7], [5, 32]]	0.803279	0.820513	0.864865	0.708333	0.842105	0.681306	0.405020
4	K-Nearest Neighbors	[[16, 8], [9, 28]]	0.721311	0.777778	0.756757	0.666667	0.767123	0.460586	0.412430
5	Multi-Layer Perceptron	[[18, 6], [5, 32]]	0.819672	0.842105	0.864865	0.750000	0.853333	0.671171	0.940643
6	Support Vector Machine	[[19, 5], [6, 31]]	0.819672	0.861111	0.837838	0.791667	0.849315	0.685811	0.670729
7	Gradient Boosting	[[17, 7], [5, 32]]	0.803279	0.820513	0.864865	0.708333	0.842105	0.654279	0.248343