Bankruptcy Prediction in Taiwanese Companies using Financial Ratios and Corporate Governance Indicators: A Machine Learning Approach

Ben Wynia

The goal of this project is to predict whether a company will go bankrupt or not, based on a set of financial ratios and corporate governance indicators. The dataset used in this project was collected from the Taiwan Economic Journal for the years 1999 to 2009, and company bankruptcy was defined based on the business regulations of the Taiwan Stock Exchange. The dataset contains 95 input features (X1 to X95), which include financial ratios such as return on assets, gross profit margin, debt ratio, and inventory turnover rate, as well as corporate governance indicators such as board size, CEO duality, and audit committee independence.

To achieve this goal, we will use machine learning algorithms to build a predictive model that can accurately classify companies as bankrupt or non-bankrupt based on their financial and governance data. We will use a variety of classification algorithms, including logistic regression, decision trees, random forests, and support vector machines, to identify the best-performing model for this dataset. We will also use feature selection techniques to identify the most important features for predicting bankruptcy, and to improve the performance of our model.

The results of this project could be useful for investors, creditors, and other stakeholders who are interested in assessing the financial health and viability of companies. By accurately predicting bankruptcy, they can make informed decisions about investing in or lending to a particular company, and take appropriate measures to manage their risk exposure. Additionally, the insights gained from this project could help regulators and policymakers to improve their understanding of the factors that contribute to corporate bankruptcy, and to develop more effective policies and regulations to mitigate this risk.

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
from sklearn.feature_selection import RFE

# 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
from PIL import Image
```

```
# 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

The data used in this project was collected from the Taiwan Economic Journal for the years 1999 to 2009. The company bankruptcy data was defined based on the business regulations of the Taiwan Stock Exchange. The dataset contains 96 columns, of which the first column is the class label (Y) indicating whether the company went bankrupt or not, and the remaining columns (X1 to X95) represent the input features which include financial ratios and corporate governance indicators.

To preprocess the data, it is important to perform data cleaning to ensure that the data is consistent and accurate. This involves removing duplicates, correcting errors, and handling missing or invalid values. Feature engineering is also an important preprocessing step, as it can improve the performance of the machine learning model by creating new features that capture the underlying patterns in the data. Feature engineering techniques such as normalization, scaling, and transformation can be applied to the data to ensure that the features are on a similar scale and have similar ranges.

Handling missing values or outliers is another important preprocessing step. Missing values can be imputed using techniques such as mean imputation, regression imputation, or k-nearest neighbors imputation. Outliers can be detected using statistical methods such as z-score analysis or the interquartile range (IQR) method, and can be removed or replaced with a more appropriate value. It is important to keep in mind that preprocessing steps can have a significant impact on the performance of the machine learning model, and therefore it is important to carefully evaluate the impact of each step and to select the best preprocessing pipeline.

In [2]:
```
current_directory = os.getcwd()
data_dictionary = pd.read_csv(f"{current_directory}/data/data_dictionary.csv")
data_dictionary.columns = ['Name', 'Definition']
df = pd.read_csv(f"{current_directory}/data/company_data.csv", low_memory=False)
```

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

Out[3]:

	Name	Definition
0	X1	ROA(C) before interest and depreciation before...
1	X2	ROA(A) before interest and % after tax: Return...
2	X3	ROA(B) before interest and depreciation after ...
3	X4	Operating Gross Margin: Gross Profit/Net Sales
4	X5	Realized Sales Gross Margin: Realized Gross Pr...
...	...	...
90	X91	Liability to Equity
91	X92	Degree of Financial Leverage (DFL)
92	X93	Interest Coverage Ratio (Interest expense to E...
93	X94	Net Income Flag: 1 if Net Income is Negative f...
94	X95	Equity to Liability

95 rows × 2 columns

Section 3. Data Cleanup and Exploratory Data Analysis

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

```
The dataset has 96 columns and 6819 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_de
Bankrupt?	6819	0	int64	0.032263	0.000000	0.000000	1.0	0.031226	0.17671
ROA(C) before interest and depreciation before interest	6819	0	float64	0.505180	0.502706	0.490128	1.0	0.003683	0.06068
ROA(A) before interest and % after tax	6819	0	float64	0.558625	0.559802	0.559693	1.0	0.004306	0.06562
ROA(B) before interest and depreciation after tax	6819	0	float64	0.553589	0.552278	0.538787	1.0	0.003794	0.06159
Operating Gross Margin	6819	0	float64	0.607948	0.605997	0.598956	1.0	0.000287	0.01693
...	...	...	...	...	...	...	...	...	
Liability to Equity	6819	0	float64	0.280365	0.278778	0.275746	1.0	0.000209	0.01446
Degree of Financial Leverage (DFL)	6819	0	float64	0.027541	0.026808	0.026791	1.0	0.000245	0.01566
Interest Coverage Ratio (Interest expense to EBIT)	6819	0	float64	0.565358	0.565252	0.565158	1.0	0.000175	0.01321
Net Income Flag	6819	0	int64	1.000000	1.000000	1.000000	0.0	0.000000	0.00000
Equity to Liability	6819	0	float64	0.047578	0.033798	0.000000	1.0	0.002501	0.05001

96 rows × 14 columns

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

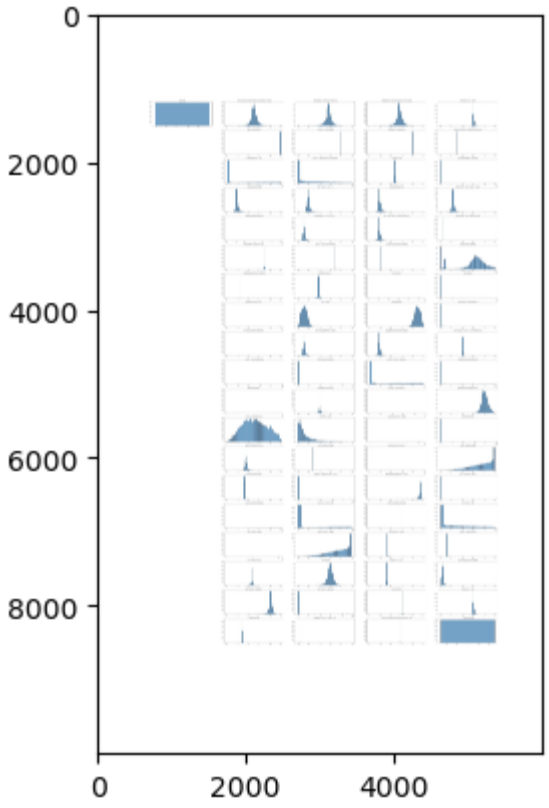
Example Usage:

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]:
```
if os.path.isfile('histograms.jpg'):
    img = plt.imread('histograms.jpg')
    plt.imshow(img)
    plt.show()
else:
    histograms.create_histogram(df, figsize=(60, 100))
```



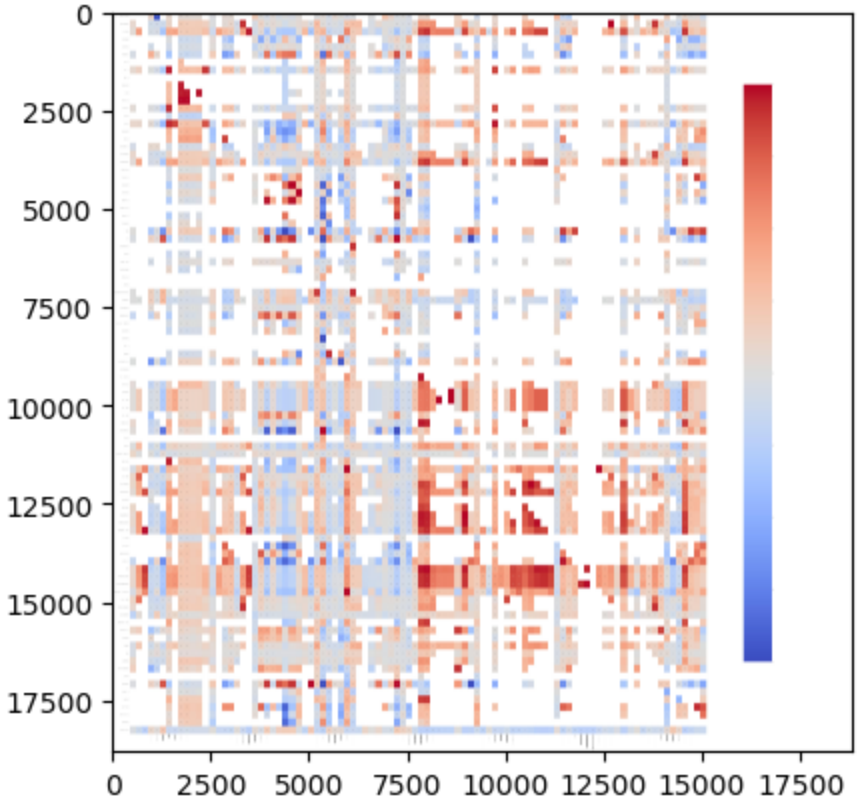
Correlation Heatmap

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]:
```
try:
    if os.path.isfile('corr_plot.jpg'):
        img = plt.imread('corr_plot.jpg')
        plt.imshow(img)
        plt.show()
    else:
        corr_map.correlation_heatmap(df)
except:
    corr_map.correlation_heatmap(df)
```



Flag Categorical and Orginal Variables

In [8]:
```
def identify_categorical_ordinal(df, threshold=10, include_ordinal=True):
    categorical_vars = []
    ordinal_vars = []

    for column in df.columns:
        unique_values = df[column].nunique()

        if unique_values <= threshold:
            categorical_vars.append(column)

            if include_ordinal:
                if df[column].dtype in ['int64', 'float64']:
                    sorted_unique_values = sorted(df[column].dropna().unique())
                    if all(sorted_unique_values[i] < sorted_unique_values[i+1] for i i
                        ordinal_vars.append(column)

    return categorical_vars, ordinal_vars
```

In [9]:
```
categorical_vars, ordinal_vars = identify_categorical_ordinal(df,10,False)

print("Categorical Variables:", categorical_vars)
print("Ordinal Variables:", ordinal_vars)
```

```
Categorical Variables: ['Bankrupt?', ' Liability-Assets Flag', ' Net Income Flag']
Ordinal Variables: []
```

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.

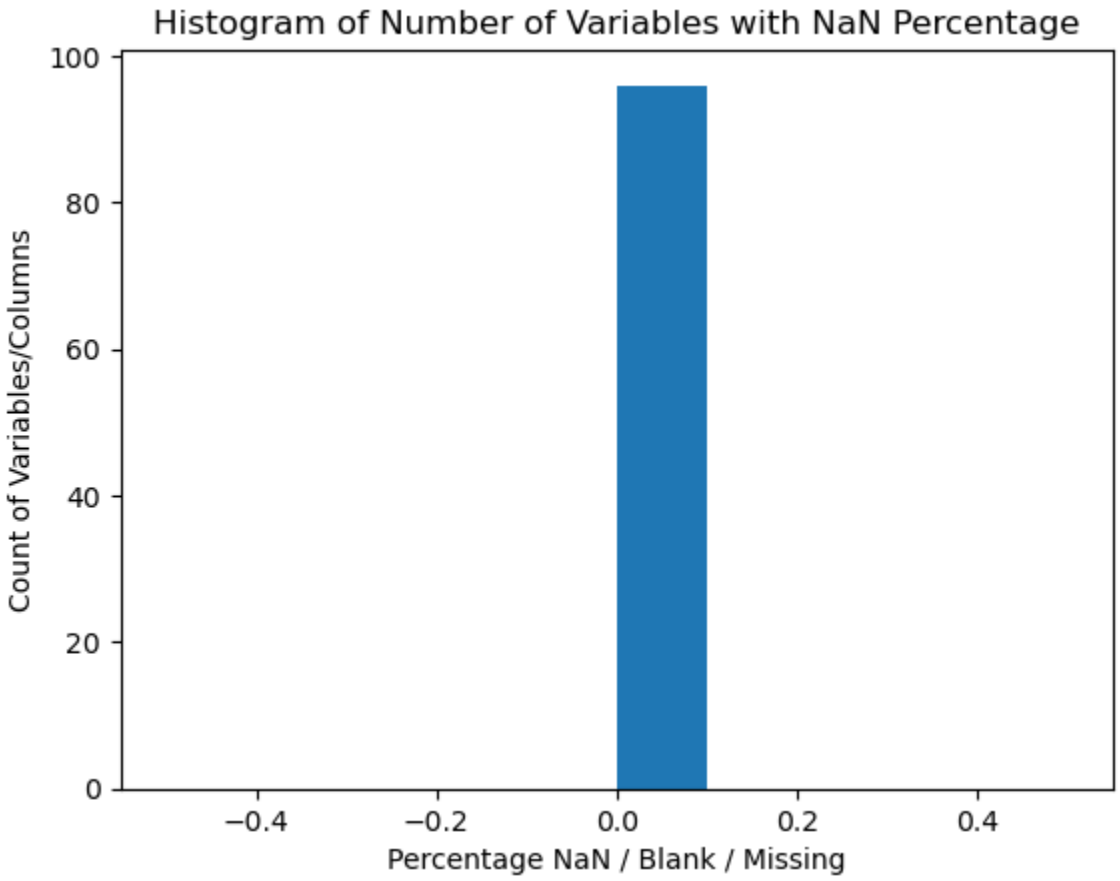
Output: A tuple containing the resulting data frame and data dictionary.

Required Libraries:

pandas numpy matplotlib.pyplot seaborn

In [10]:
```
columns_to_keep=list(df.columns)
columns_to_recode=[' Liability-Assets Flag', ' Net Income Flag']
breakpoint=0.95
df, data_dictionary2 = data_cleanup.run_data_cleanup_functions(df, data_dictionary, co
```

```
Original dataframe---
The dataset has 96 columns and 6819 rows
```



```
After cleaning up columns with missing data---
The dataset has 96 columns and 6819 rows
Number of rows before filtering: 6819
Number of rows after filtering: 6819
Percentage of rows dropped: 0.0%
After cleaning up rows with missing data---
The dataset has 96 columns and 6819 rows
After recoding categorical variables into one-hot variables---
The dataset has 97 columns and 6819 rows
```

Define the Dependent Variable

In [11]:
```
target = 'Bankrupt?'
target
```

Out[11]:
```
'Bankrupt?'
```

Define the Independent Variables

In [12]:
```
features = [col for col in df.columns if col != target]
features
```

Out[12]:
```
[' ROA(C) before interest and depreciation before interest',
 ' ROA(A) before interest and % after tax',
 ' ROA(B) before interest and depreciation after tax',
 ' Operating Gross Margin',
 ' Realized Sales Gross Margin',
 ' Operating Profit Rate',
 ' Pre-tax net Interest Rate',
 ' After-tax net Interest Rate',
 ' Non-industry income and expenditure/revenue',
 ' Continuous interest rate (after tax)',
 ' Operating Expense Rate',
 ' Research and development expense rate',
 ' Cash flow rate',
 ' Interest-bearing debt interest rate',
 ' Tax rate (A)',
 ' Net Value Per Share (B)',
 ' Net Value Per Share (A)',
 ' Net Value Per Share (C)',
 ' Persistent EPS in the Last Four Seasons',
 ' Cash Flow Per Share',
 ' Revenue Per Share (Yuan ¥)',
 ' Operating Profit Per Share (Yuan ¥)',
 ' Per Share Net profit before tax (Yuan ¥)',
 ' Realized Sales Gross Profit Growth Rate',
 ' Operating Profit Growth Rate',
 ' After-tax Net Profit Growth Rate',
 ' Regular Net Profit Growth Rate',
 ' Continuous Net Profit Growth Rate',
 ' Total Asset Growth Rate',
 ' Net Value Growth Rate',
 ' Total Asset Return Growth Rate Ratio',
 ' Cash Reinvestment %',
 ' Current Ratio',
 ' Quick Ratio',
 ' Interest Expense Ratio',
 ' Total debt/Total net worth',
 ' Debt ratio %',
 ' Net worth/Assets',
 ' Long-term fund suitability ratio (A)',
 ' Borrowing dependency',
 ' Contingent liabilities/Net worth',
 ' Operating profit/Paid-in capital',
 ' Net profit before tax/Paid-in capital',
 ' Inventory and accounts receivable/Net value',
 ' Total Asset Turnover',
 ' Accounts Receivable Turnover',
 ' Average Collection Days',
 ' Inventory Turnover Rate (times)',
 ' Fixed Assets Turnover Frequency',
 ' Net Worth Turnover Rate (times)',
 ' Revenue per person',
 ' Operating profit per person',
 ' Allocation rate per person',
 ' Working Capital to Total Assets',
 ' Quick Assets/Total Assets',
 ' Current Assets/Total Assets',
 ' Cash/Total Assets',
 ' Quick Assets/Current Liability',
 ' Cash/Current Liability',
 ' Current Liability to Assets',
```

```
 ' Operating Funds to Liability',
 ' Inventory/Working Capital',
 ' Inventory/Current Liability',
 ' Current Liabilities/Liability',
 ' Working Capital/Equity',
 ' Current Liabilities/Equity',
 ' Long-term Liability to Current Assets',
 ' Retained Earnings to Total Assets',
 ' Total income/Total expense',
 ' Total expense/Assets',
 ' Current Asset Turnover Rate',
 ' Quick Asset Turnover Rate',
 ' Working capitcal Turnover Rate',
 ' Cash Turnover Rate',
 ' Cash Flow to Sales',
 ' Fixed Assets to Assets',
 ' Current Liability to Liability',
 ' Current Liability to Equity',
 ' Equity to Long-term Liability',
 ' Cash Flow to Total Assets',
 ' Cash Flow to Liability',
 ' CFO to Assets',
 ' Cash Flow to Equity',
 ' Current Liability to Current Assets',
 ' Net Income to Total Assets',
 ' Total assets to GNP price',
 ' No-credit Interval',
 ' Gross Profit to Sales',
 " Net Income to Stockholder's Equity",
 ' Liability to Equity',
 ' Degree of Financial Leverage (DFL)',
 ' Interest Coverage Ratio (Interest expense to EBIT)',
 ' Equity to Liability',
 '1h_ Liability-Assets Flag_0',
 '1h_ Liability-Assets Flag_1',
 '1h_ Net Income Flag_1']
```

In [13]:
```
df[target].value_counts()
```

Out[13]:
```
0    6599
1     220
Name: Bankrupt?, 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 [14]:
```
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 [15]:
```
correlations = view_independent_variable_correlations(df, target)
correlations
```

Out[15]:
```
 Debt ratio %                                              0.250161
 Current Liability to Assets                               0.194494
 Borrowing dependency                                      0.176543
 Current Liability to Current Assets                       0.171306
 Liability to Equity                                       0.166812
                                                               ...
 ROA(C) before interest and depreciation before interest  -0.260807
 ROA(B) before interest and depreciation after tax        -0.273051
 ROA(A) before interest and % after tax                   -0.282941
 Net Income to Total Assets                               -0.315457
1h_ Net Income Flag_1                                           NaN
Name: Bankrupt?, Length: 96, 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 [16]:
```
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 [17]:
```
class_weights, distribution = compute_class_weights(df, target)
```

```
The distribution of the dependent variable is: 0    6599
1     220
Name: Bankrupt?, dtype: int64
The resuling class weights are: {0: 0.03333838460372784, 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 [18]:
```
labels = ['Not Bankrupt','Bankrupt']
train_data, test_data, val_data = train_test_validate.split_data(df, labels, target)
```

```
Train data dependent distribution:
    3962 (0 - Not Bankrupt) 129(1 - Bankrupt)
Test data dependent distribution:
    1312 (0 - Not Bankrupt) 52(1 - Bankrupt)
Validation data dependent distribution:
    1325 (0 - Not Bankrupt) 39(1 - Bankrupt)
```

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 [19]:
```
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\corporate_bankrupty_prediction/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 [20]:
```
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, scor

        # 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 [21]:
```
def select_features_rfe(X, y, n_features_to_select=10):
    # Create a logistic regression model
    logistic_regression = LogisticRegression(solver='lbfgs', max_iter=1000)

    # Create the RFE selector with the logistic regression model and desired number of
    rfe_selector = RFE(estimator=logistic_regression, n_features_to_select=n_features_

    # Fit the RFE selector to the data
    rfe_selector.fit(X, y)

    # Get the column indices of the selected features
    selected_columns = np.where(rfe_selector.support_ == True)[0]

    return selected_columns
```

In [22]:
```
feature_subset = select_features_rfe(train_data[features], train_data[target], n_featu
feature_subset = df.iloc[:, feature_subset].copy().columns
feature_subset
```

Out[22]:
```
Index([' Continuous interest rate (after tax)', ' Operating Expense Rate',
       ' Cash flow rate', ' Cash Flow Per Share',
       ' Continuous Net Profit Growth Rate', ' Total Asset Growth Rate',
       ' Current Ratio', ' Interest Expense Ratio', ' Total Asset Turnover',
       ' Accounts Receivable Turnover', ' Average Collection Days',
       ' Net Worth Turnover Rate (times)', ' Operating profit per person',
       ' Cash/Total Assets', ' Quick Assets/Current Liability',
       ' Inventory/Working Capital', ' Current Liabilities/Equity',
       ' Total expense/Assets', ' Working capitcal Turnover Rate',
       ' Net Income to Total Assets'],
      dtype='object')
```

In [23]:
```
# 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_method = "RandomizedSearchCV" # "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'],
        'max_iter': [1000],
        'class_weight': [class_weights]
    },
    {
        'penalty': ['l2'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'fit_intercept': [True, False],
        'solver': ['newton-cg', 'lbfgs', 'sag'],
        'max_iter': [1000],
        'class_weight': [class_weights]
    }
]

# 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.
{'solver': 'liblinear', 'penalty': 'l1', 'max_iter': 1000, 'fit_intercept': False, 'c
lass_weight': {0: 0.03333838460372784, 1: 1}, 'C': 10}
0.8967812203909304
```

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 [24]:
```
# 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.R', 'base_estimator': DecisionTreeClassifier(class_weight={0: 0.
03333838460372784, 1: 1}, max_depth=2), 'learning_rate': 0.01, 'n_estimators': 100,
'random_state': 7}
0.9108116324267017
```

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 [25]:
```
# 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': {0: 0.03333838460372784, 1: 1}, 'criterion': 'entropy', 'max_depth':
3, 'max_features': 'log2', 'min_samples_leaf': 2, 'min_samples_split': 10, 'splitte
r': 'best'}
0.8931006985168398
```

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 [26]:
```
# 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, 'n_jobs': -1, 'n_estimators': 200, 'min_samples_split': 5, 'min_
samples_leaf': 5, 'max_features': 'sqrt', 'max_depth': 20, 'criterion': 'entropy', 'c
lass_weight': None, 'bootstrap': False}
0.9454789323049095
```

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 [27]:
```
# 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': 1, 'n_neighbors': 11, 'n_jobs': -1, 'metric': 'manhatta
n', 'leaf_size': 30, 'algorithm': 'auto'}
0.5903330888671368
```

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 [28]:
```
# 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]
}

# Train and save MLP classifier model - need to come back and undersample or oversampl
mlp_trained_model, mlp_best_params, mlp_best_score, mlp_elapsed_time = train_and_save_
```

```
Loaded existing model and hyperparameter results.
{'warm_start': False, 'verbose': False, 'validation_fraction': 0.1, 'tol': 0.0001, 's
olver': 'adam', 'shuffle': True, 'random_state': 42, 'power_t': 0.75, 'nesterovs_mome
ntum': True, 'n_iter_no_change': 20, 'momentum': 0.9, 'max_iter': 200, 'learning_rate
_init': 0.01, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100,), 'epsilon': 1
e-08, 'early_stopping': False, 'beta_2': 0.999, 'beta_1': 0.95, 'batch_size': 'auto',
'alpha': 0.1, 'activation': 'tanh'}
0.6133209912197932
```

Section 5: Model Evaluation

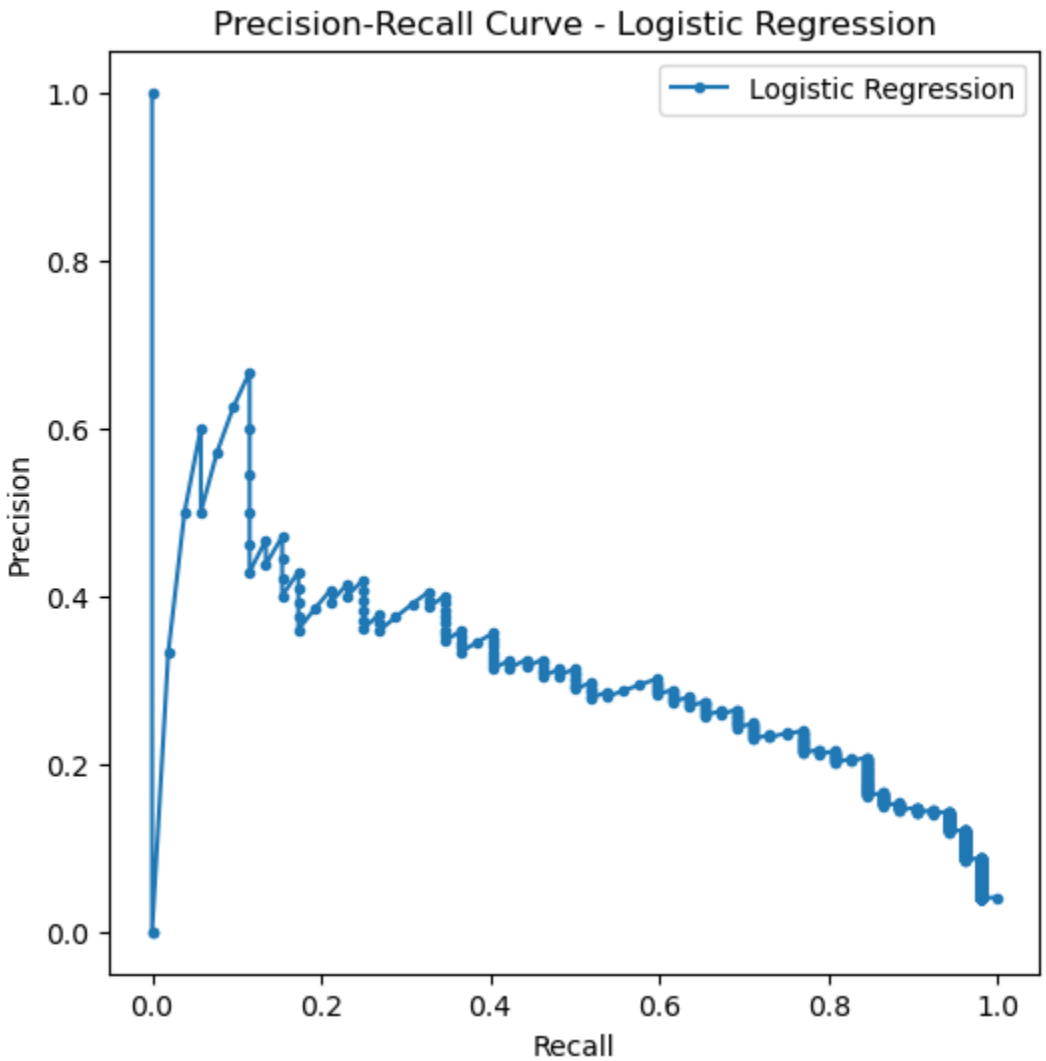
Global Model Variables

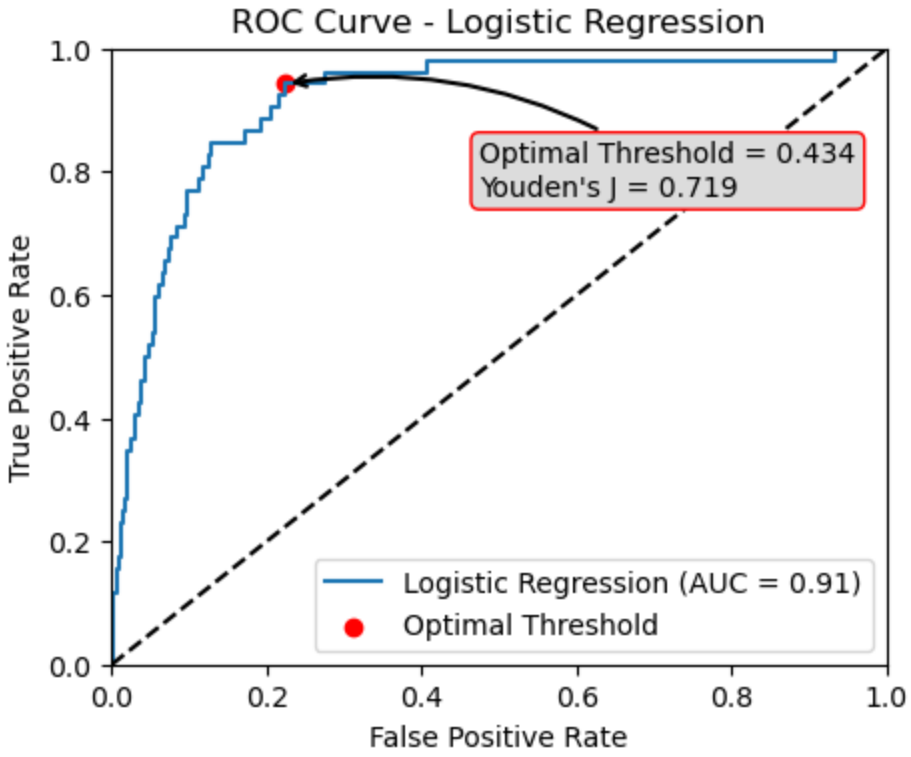
In [29]:
```
# Initialize a dataframe to store the results
results_frame = pd.DataFrame(columns=['Model Name',
                                      'Confusion Matrix',
                                      'Accuracy',
                                      'Precision',
                                      'Recall',
                                      'Specificity',
                                      'F1 Score',
                                      'Youden J Stat',
```

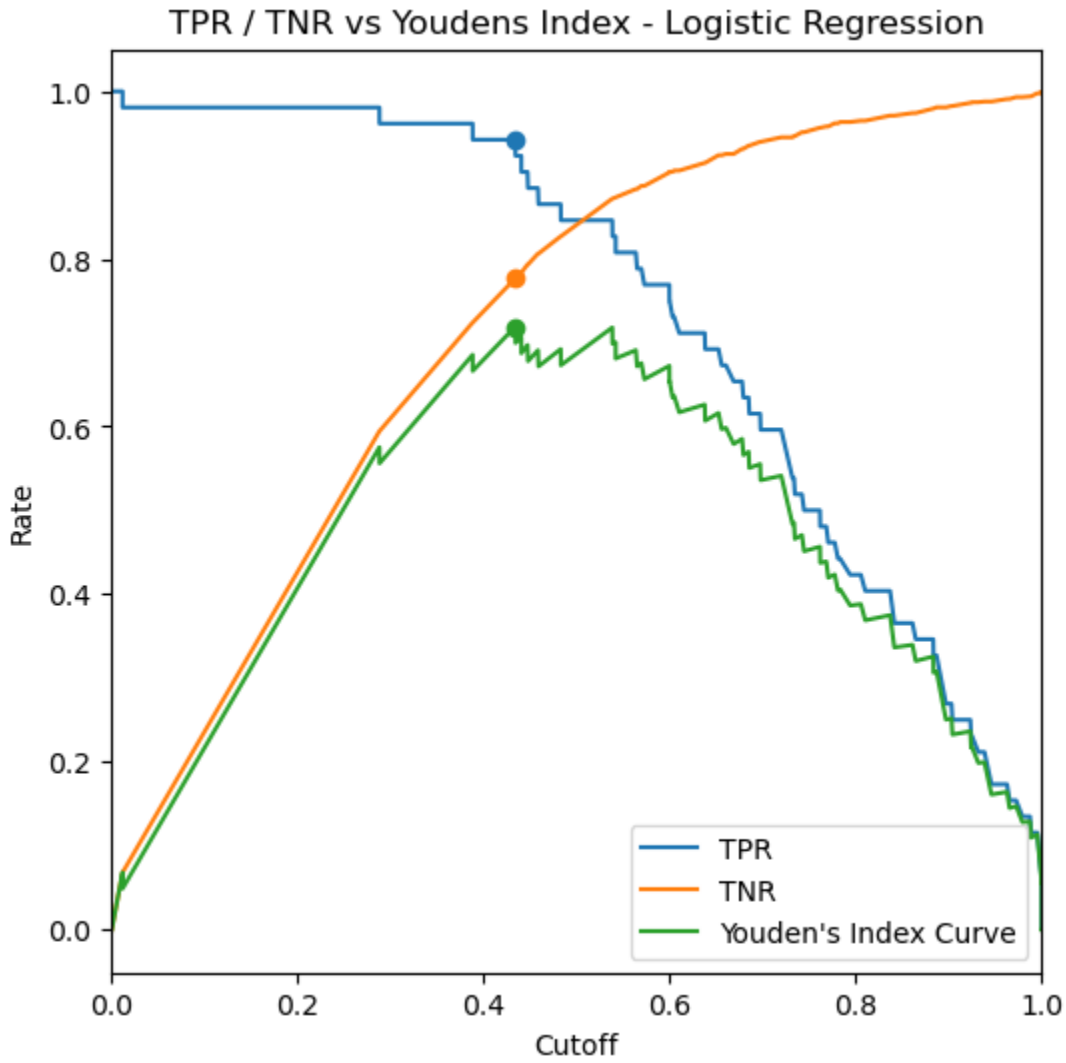
```
                                        'Optimal P Threshold'])

# Bundle datasets for model training and evaluation
datasets = [train_data, test_data, features, target]
datasets2 = [train_data, test_data, feature_subset, target]

# Define class_names
class_names = ["Solvent (0)", "Bankrupt (1)"]
```

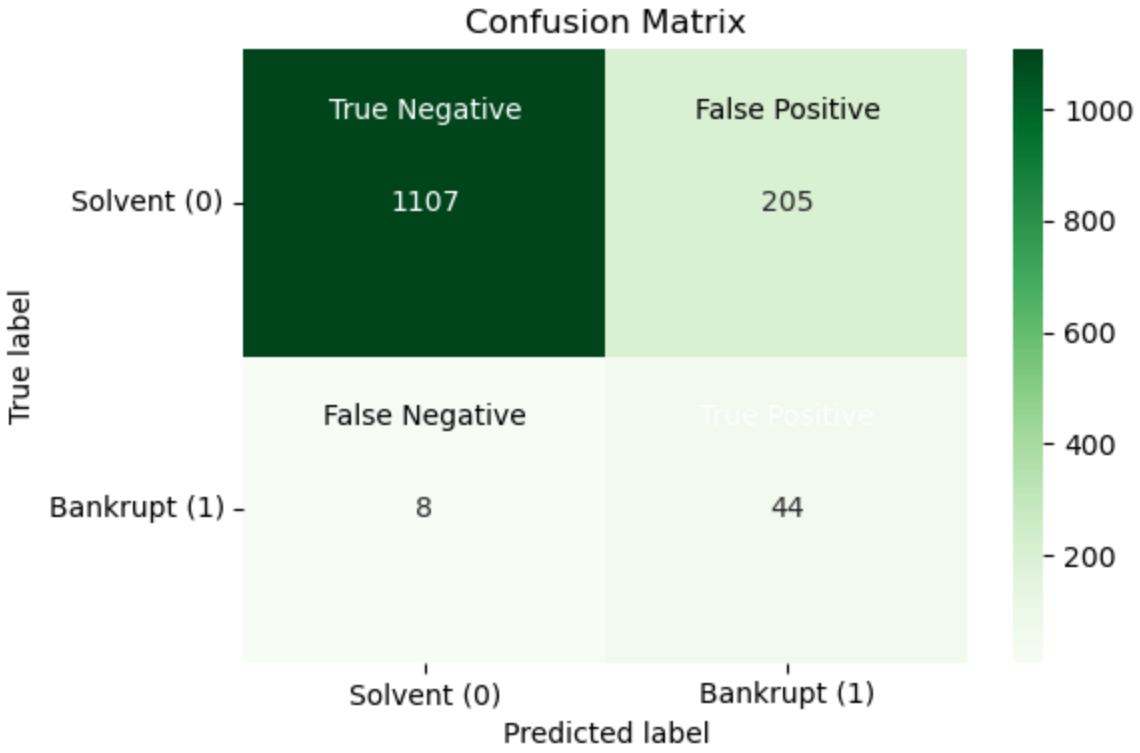
Logistic Regression

In [30]:
```
logreg_model_name, logreg_pr_curve_plot, logreg_roc_curve_plot, logreg_tpr_tnr_plot, l
                                'Logistic Regression',
                                class_names,
                                datasets2,
                                "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)
```

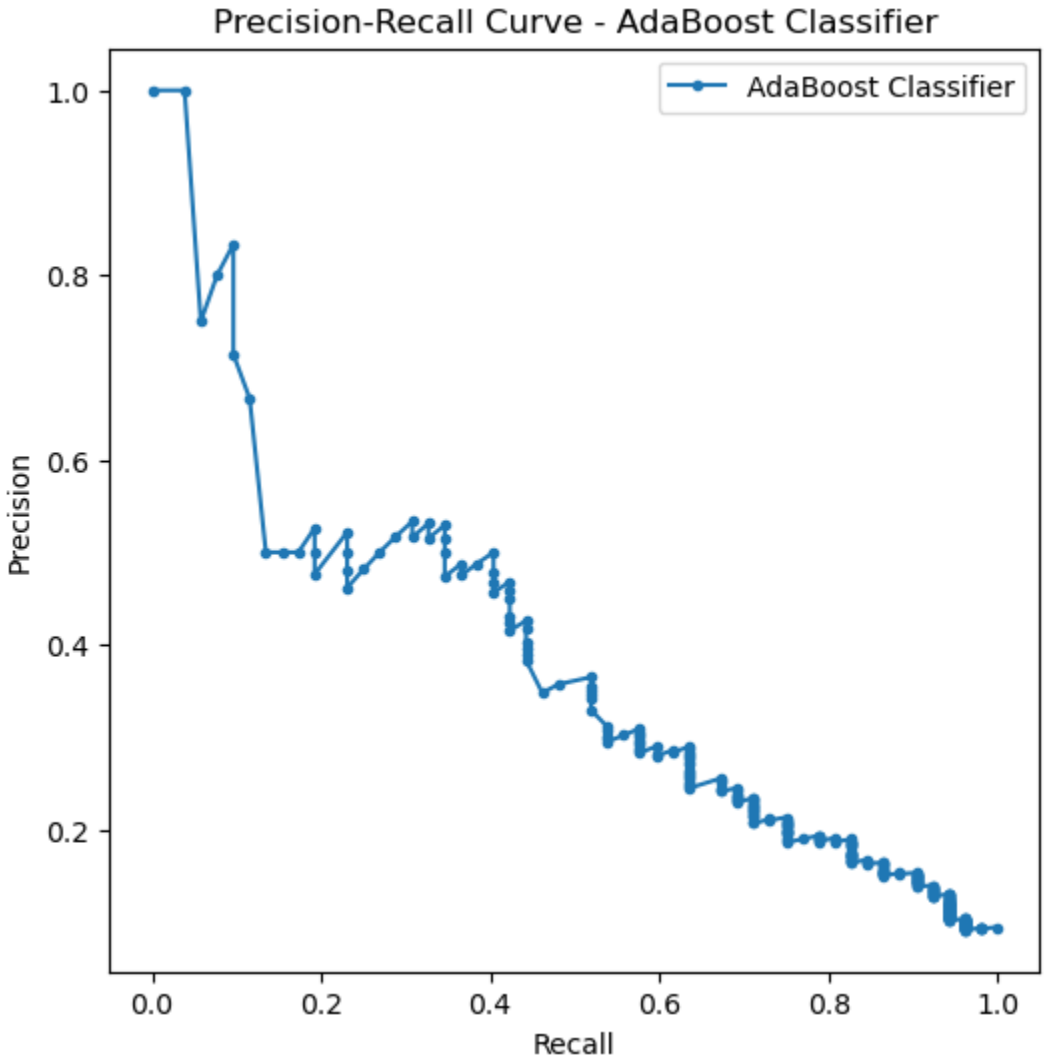


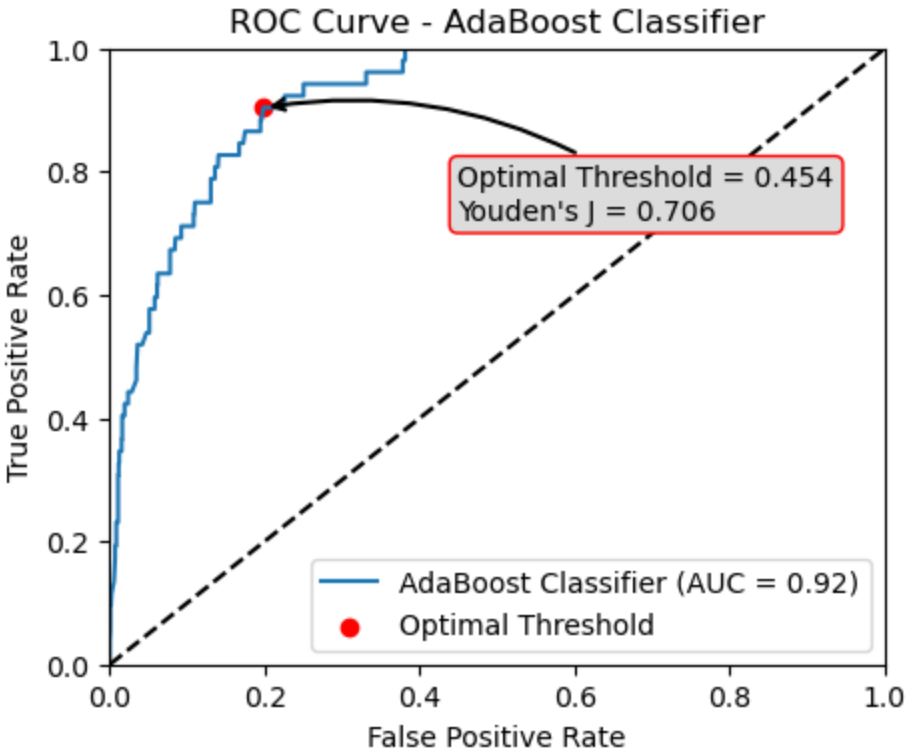

ROC Curve - Logistic Regression
Optimal Threshold = 0.434
Youden's J = 0.719
Logistic Regression (AUC = 0.91)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

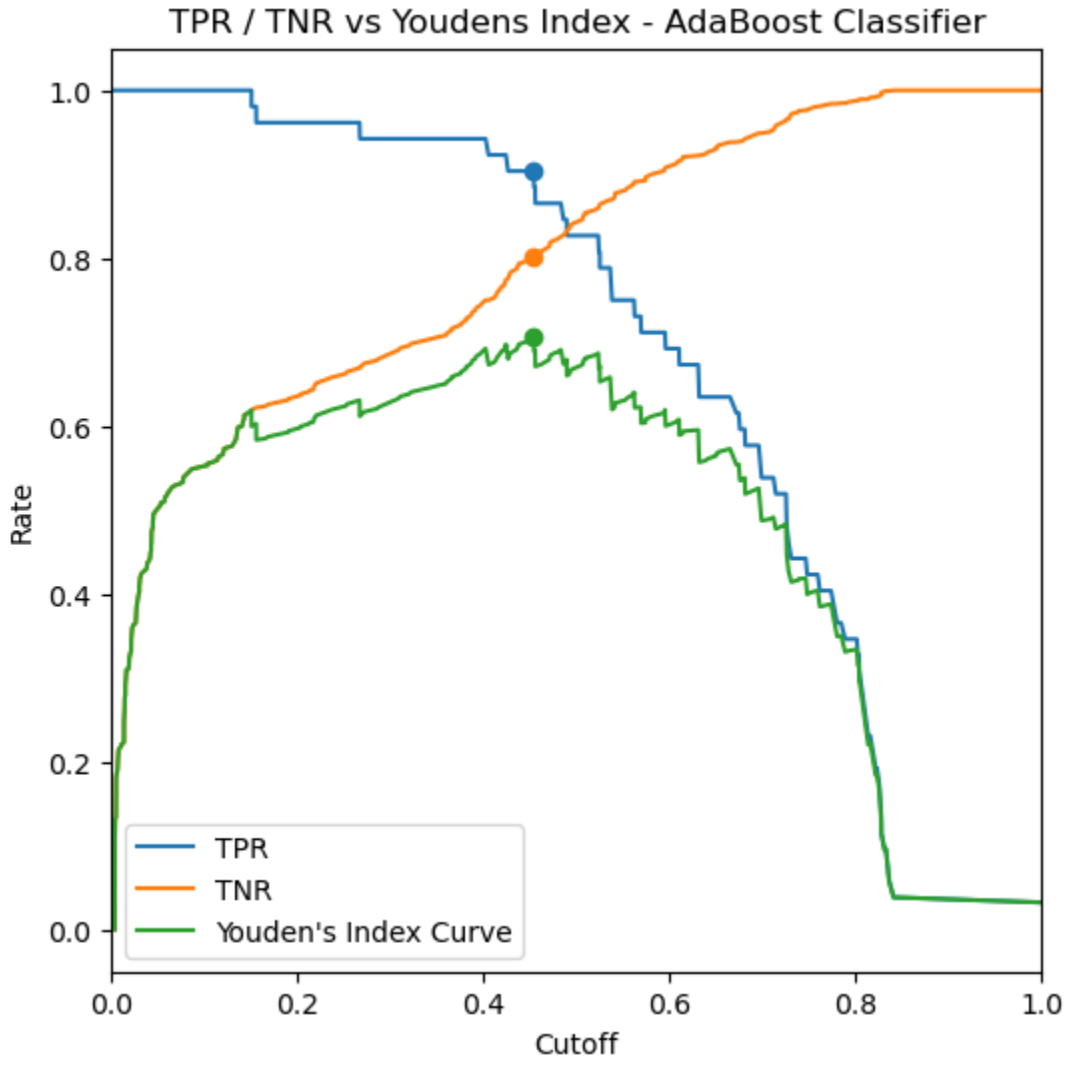

TPR / TNR vs Youdens Index - Logistic Regression
TPR
TNR
Youden's Index Curve
Rate
Cutoff
0.0
0.2
0.4
0.6
0.8
1.0

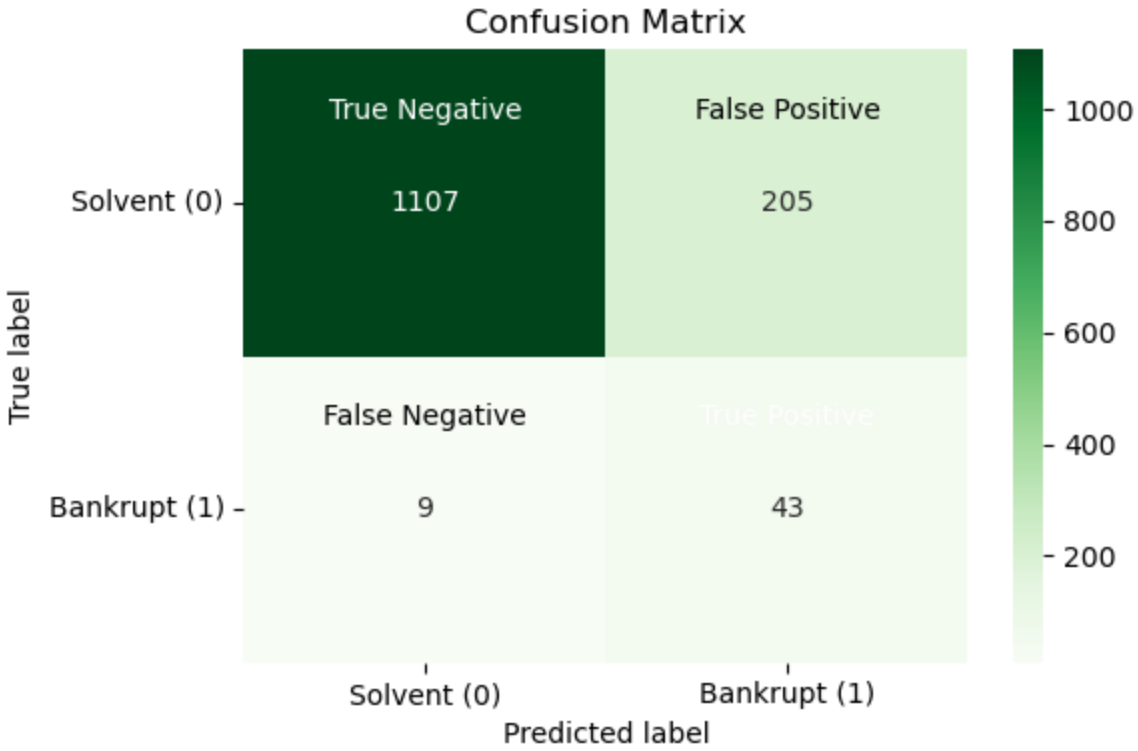


In [31]:
```
abc_model_name, abc_pr_curve_plot, abc_roc_curve_plot, abc_tpr_tnr_plot, abc_conf_matr
                                'AdaBoost Classifier',
                                class_names,
                                datasets2,
                                "full")

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)
```

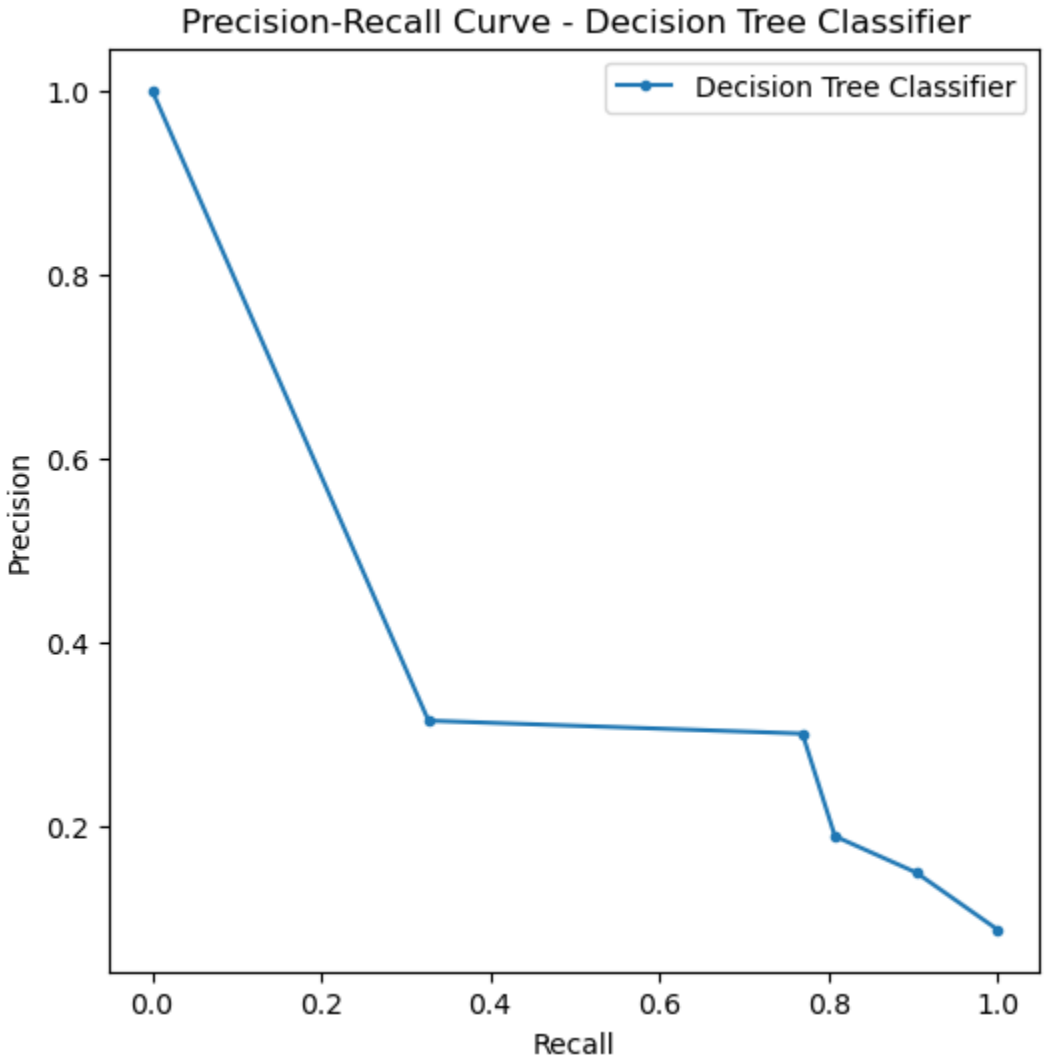

Precision-Recall Curve - AdaBoost Classifier
AdaBoost Classifier
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0

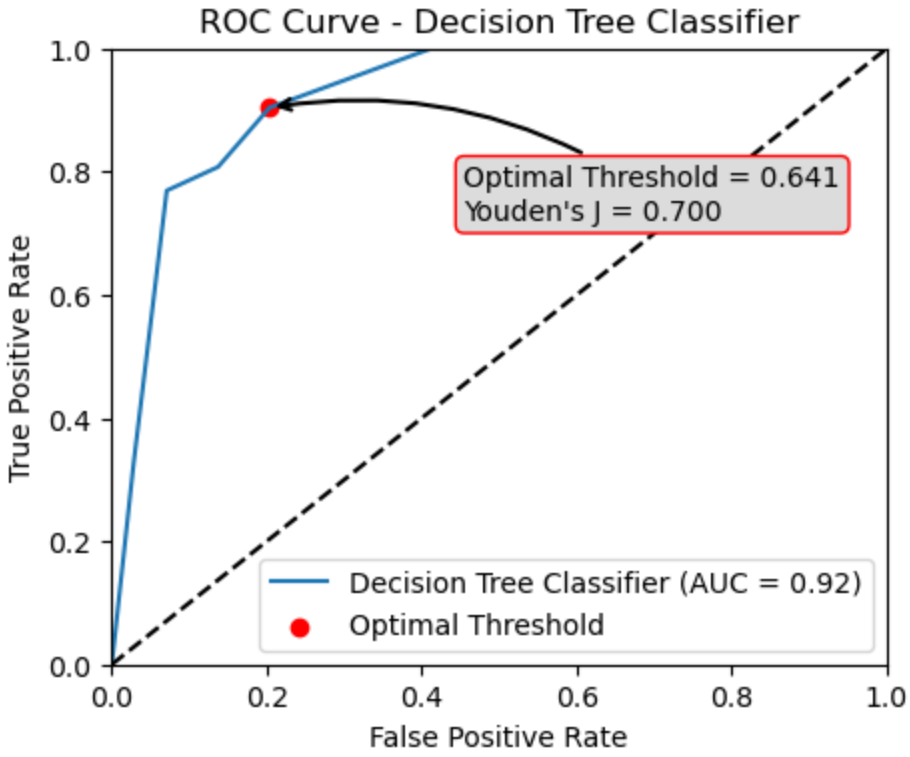

ROC Curve - AdaBoost Classifier
Optimal Threshold = 0.454
Youden's J = 0.706
AdaBoost Classifier (AUC = 0.92)
Optimal Threshold
True Positive Rate
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

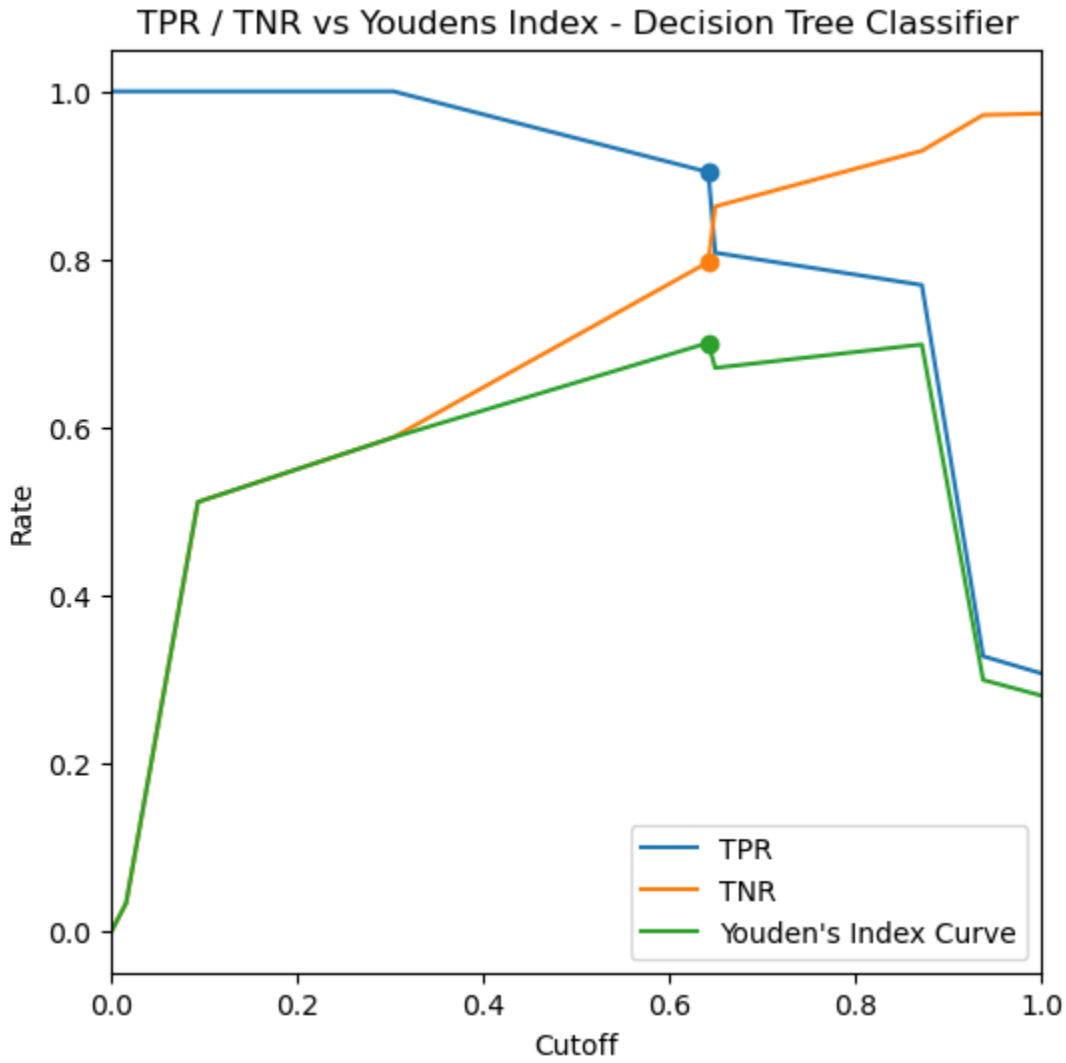

TPR / TNR vs Youdens Index - AdaBoost Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

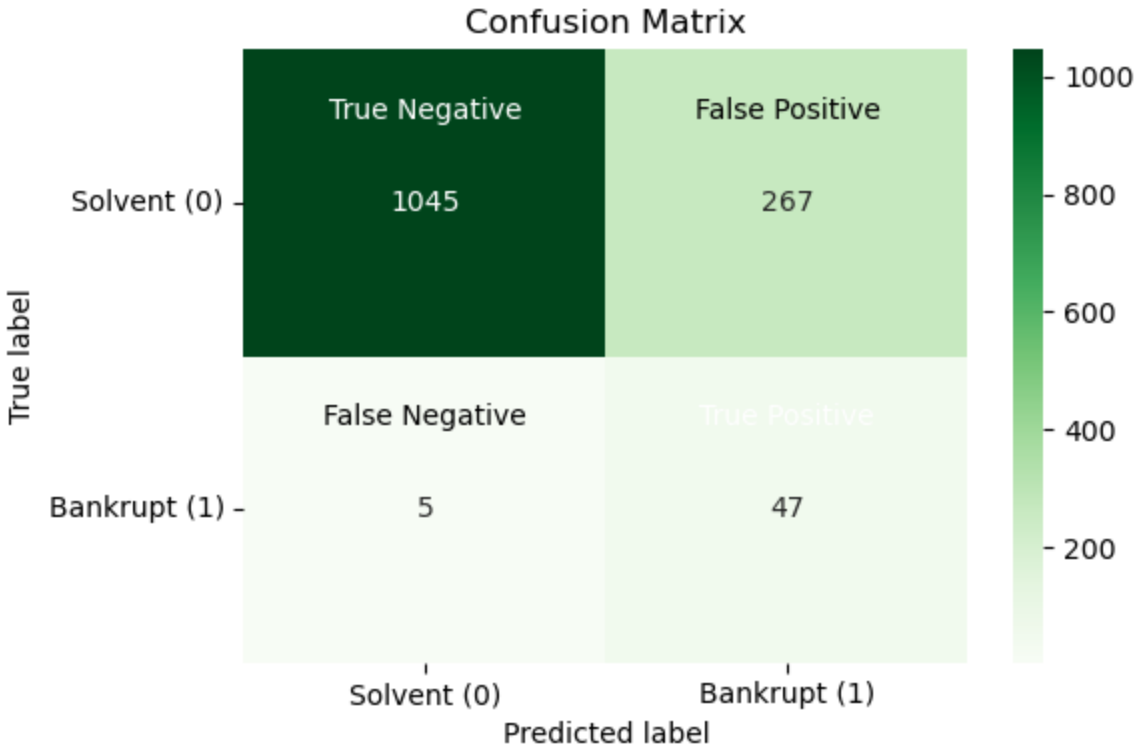

Confusion Matrix
True Negative
False Positive
1107
205
False Negative
True Positive
9
43
Solvent (0)
Bankrupt (1)
True label
Predicted label
1000
800
600
400
200

In [32]:
```
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)
```

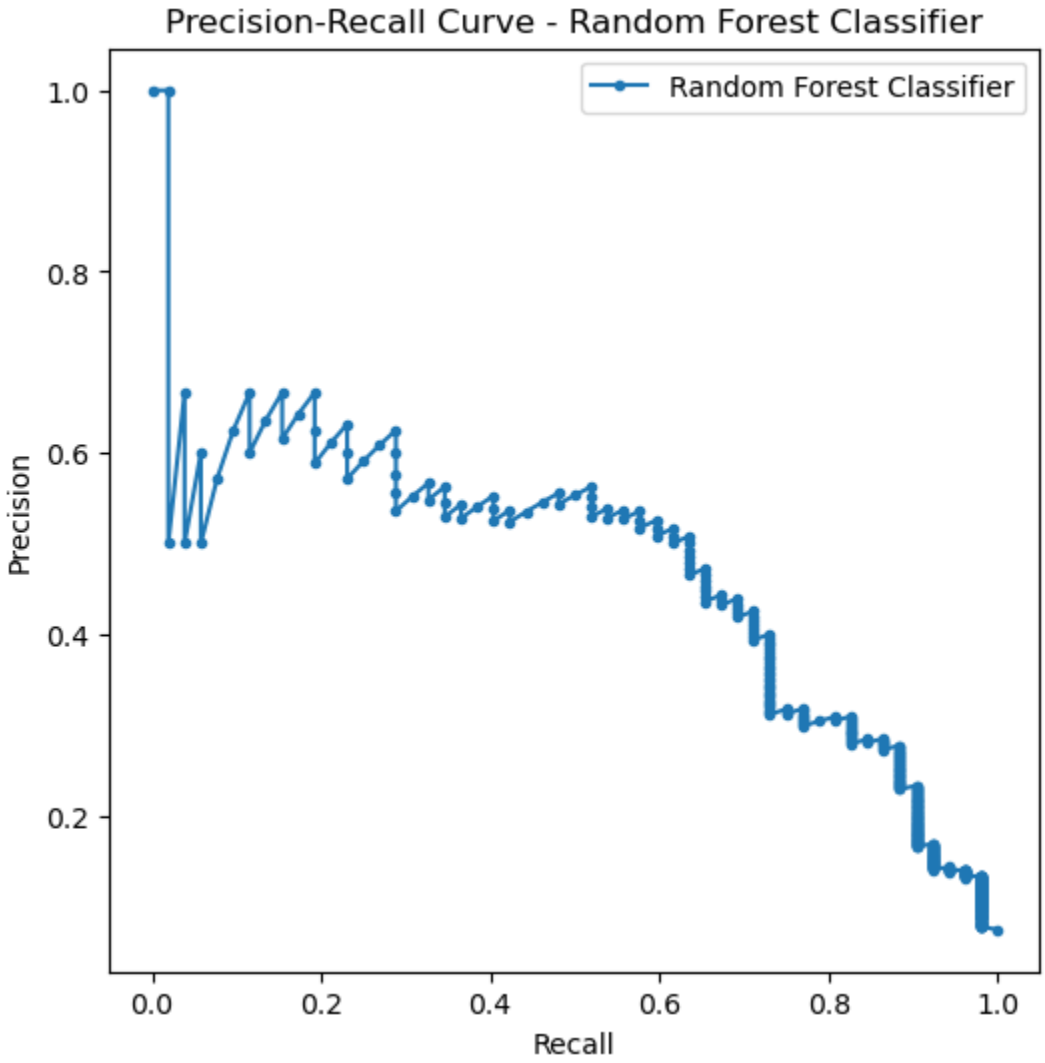


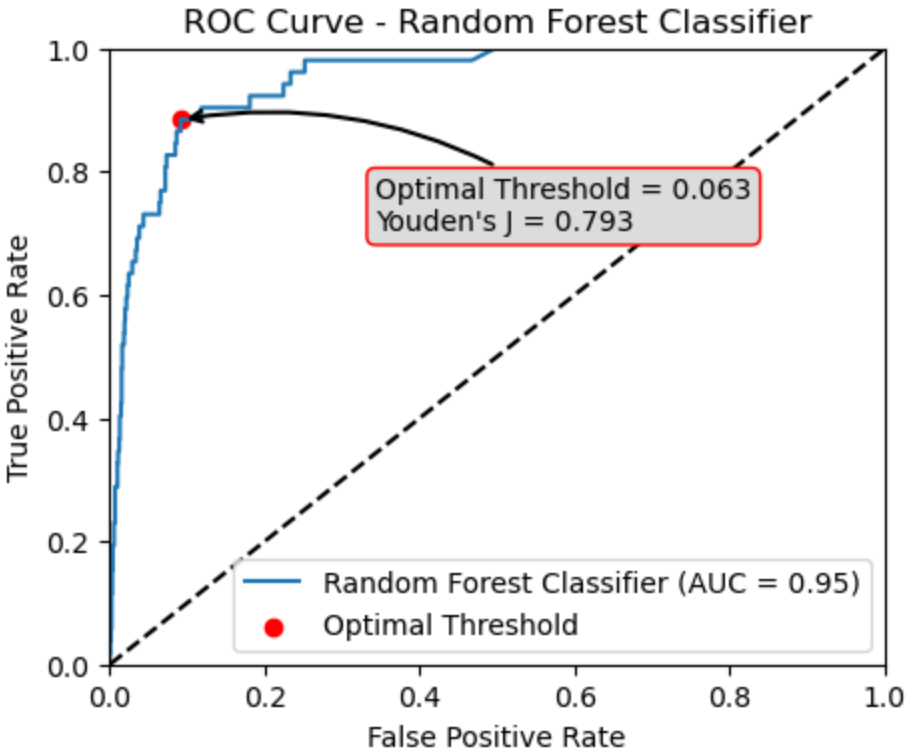

ROC Curve - Decision Tree Classifier
Optimal Threshold = 0.641
Youden's J = 0.700
Decision Tree Classifier (AUC = 0.92)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

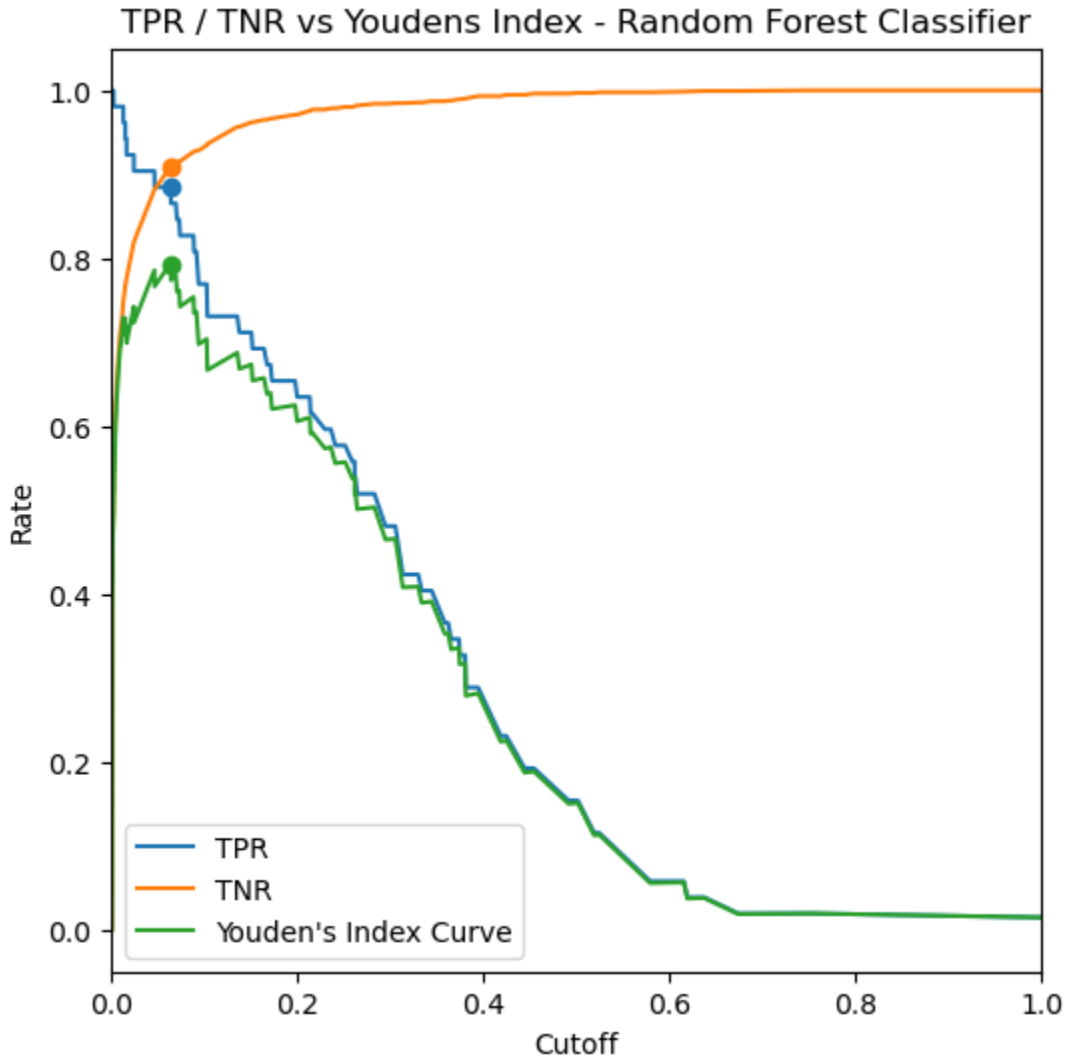

TPR / TNR vs Youdens Index - Decision Tree Classifier
TPR
TNR
Youden's Index Curve
Rate
Cutoff
0.0
0.2
0.4
0.6
0.8
1.0

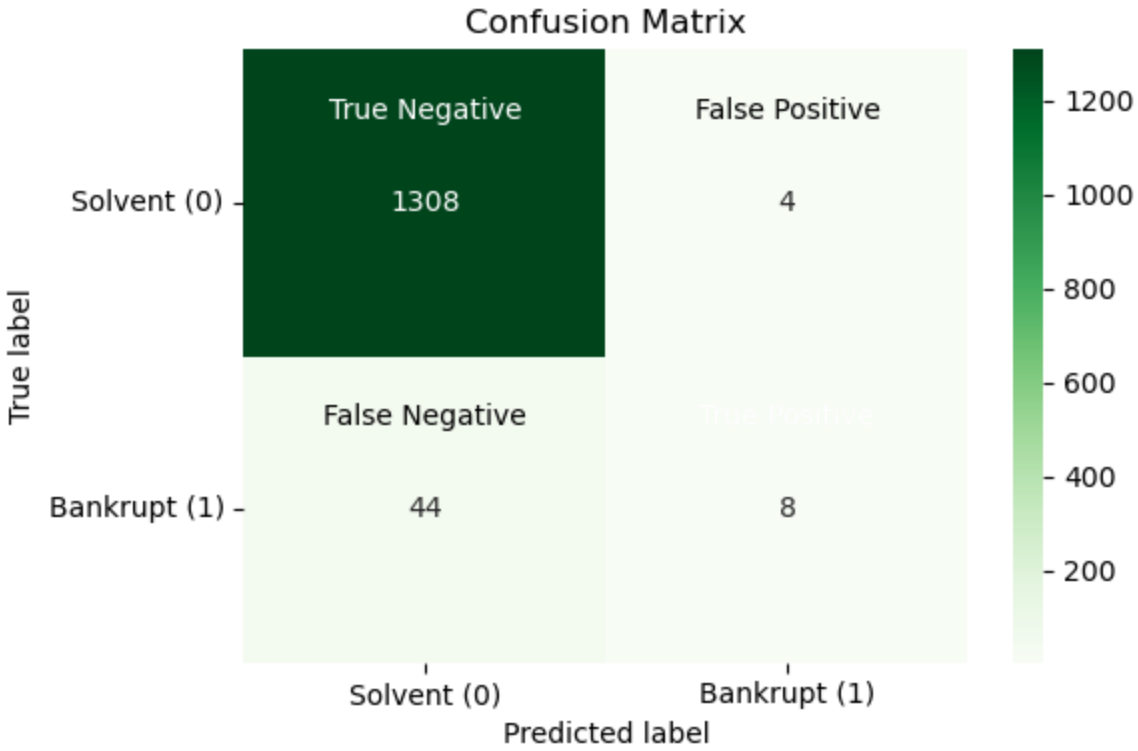


In [33]:
```
rfc_model_name, rfc_pr_curve_plot, rfc_roc_curve_plot, rfc_tpr_tnr_plot, rfc_conf_matr
                                'Random Forest Classifier',
                                class_names,
                                datasets,
                                "full")

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)
```


Precision-Recall Curve - Random Forest Classifier
Random Forest Classifier
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0

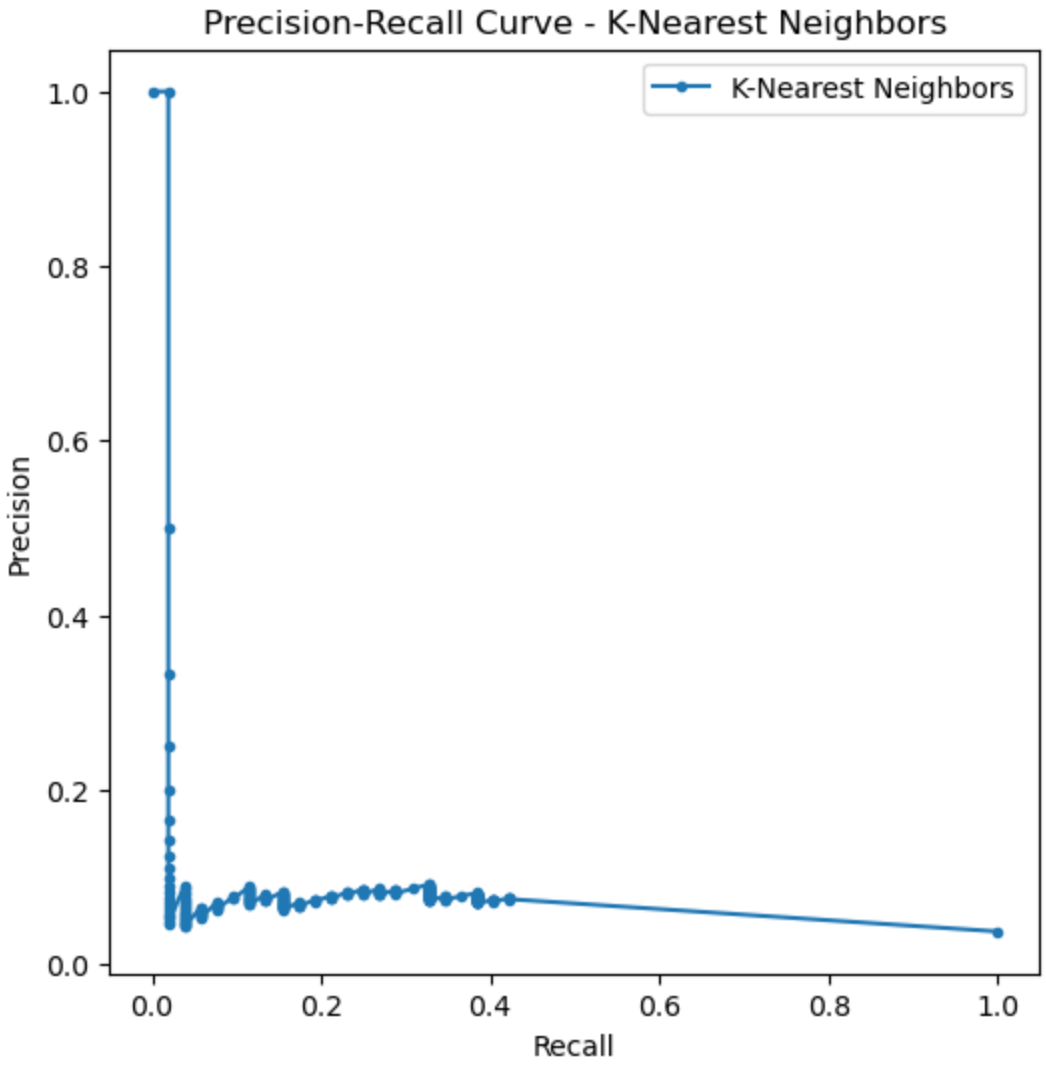

ROC Curve - Random Forest Classifier
Optimal Threshold = 0.063
Youden's J = 0.793
Random Forest Classifier (AUC = 0.95)
Optimal Threshold
True Positive Rate
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

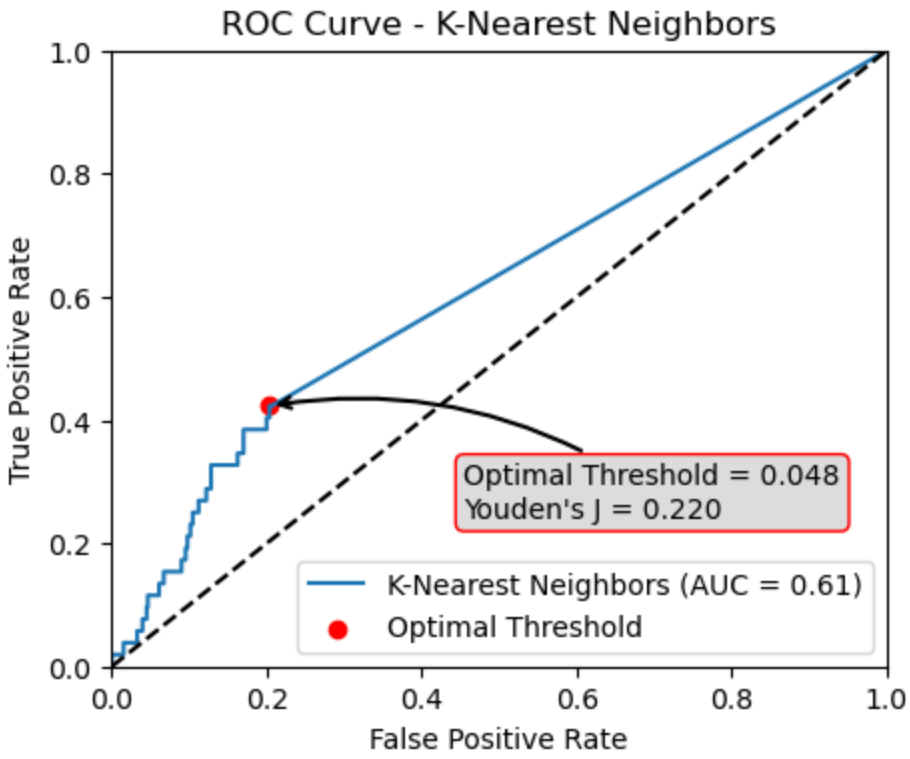

TPR / TNR vs Youdens Index - Random Forest Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

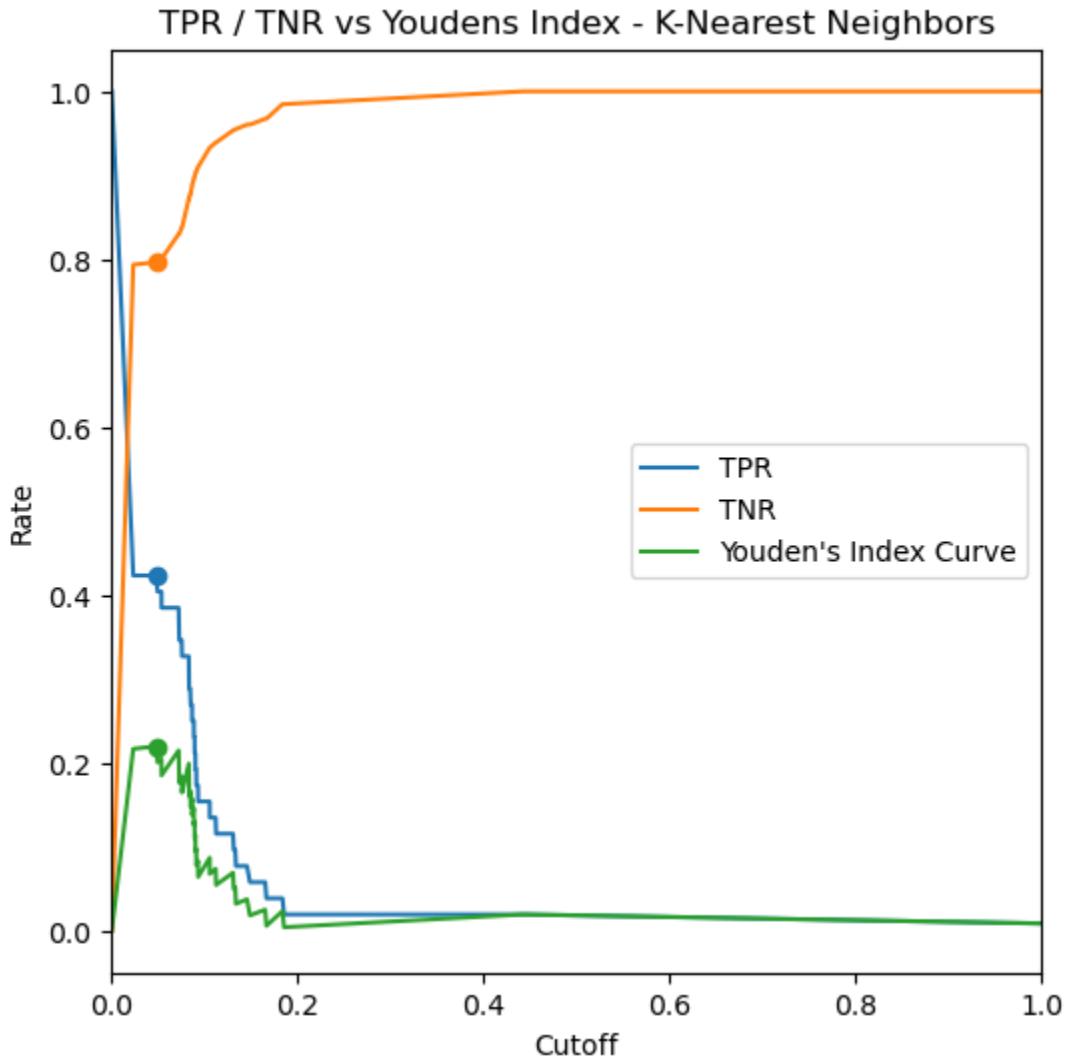

Confusion Matrix
True Negative
False Positive
1308
4
False Negative
True Positive
44
8
Solvent (0)
Bankrupt (1)
True label
Predicted label
1200
1000
800
600
400
200

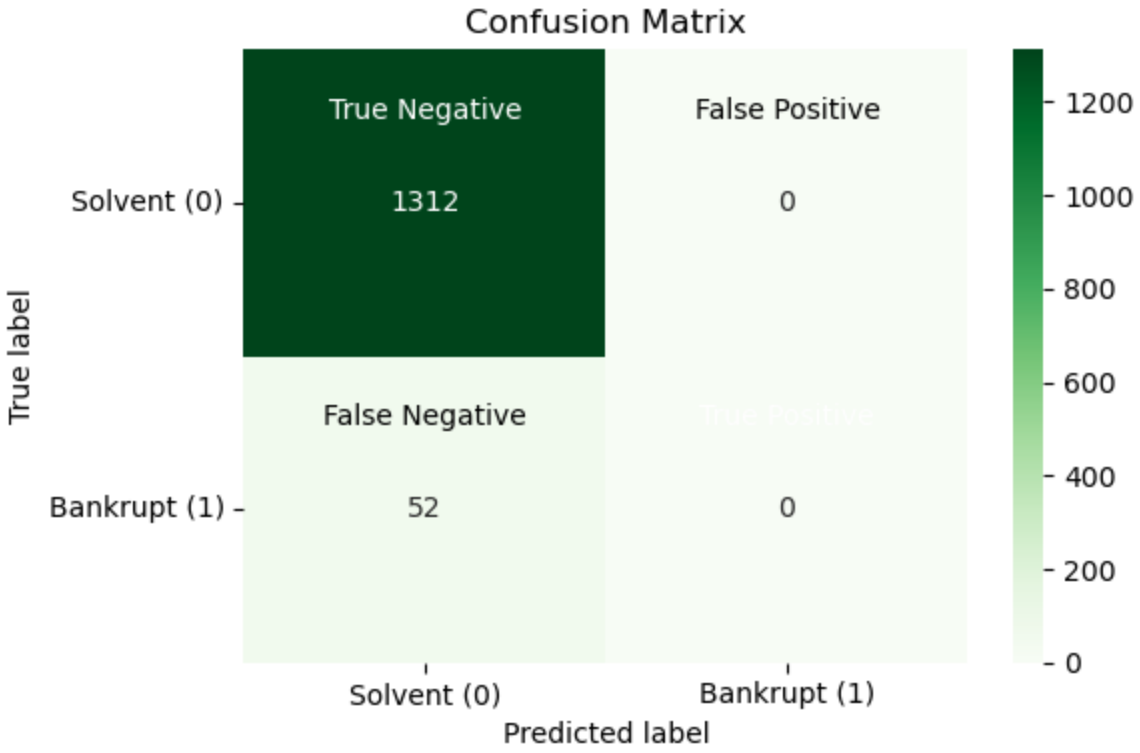
In [34]:
```
knn_model_name, knn_pr_curve_plot, knn_roc_curve_plot, knn_tpr_tnr_plot, knn_conf_matr
                                'K-Nearest Neighbors',
                                class_names,
                                datasets,
                                "full")

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)
```

```
C:\Users\bwynia\Anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predi
cted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```



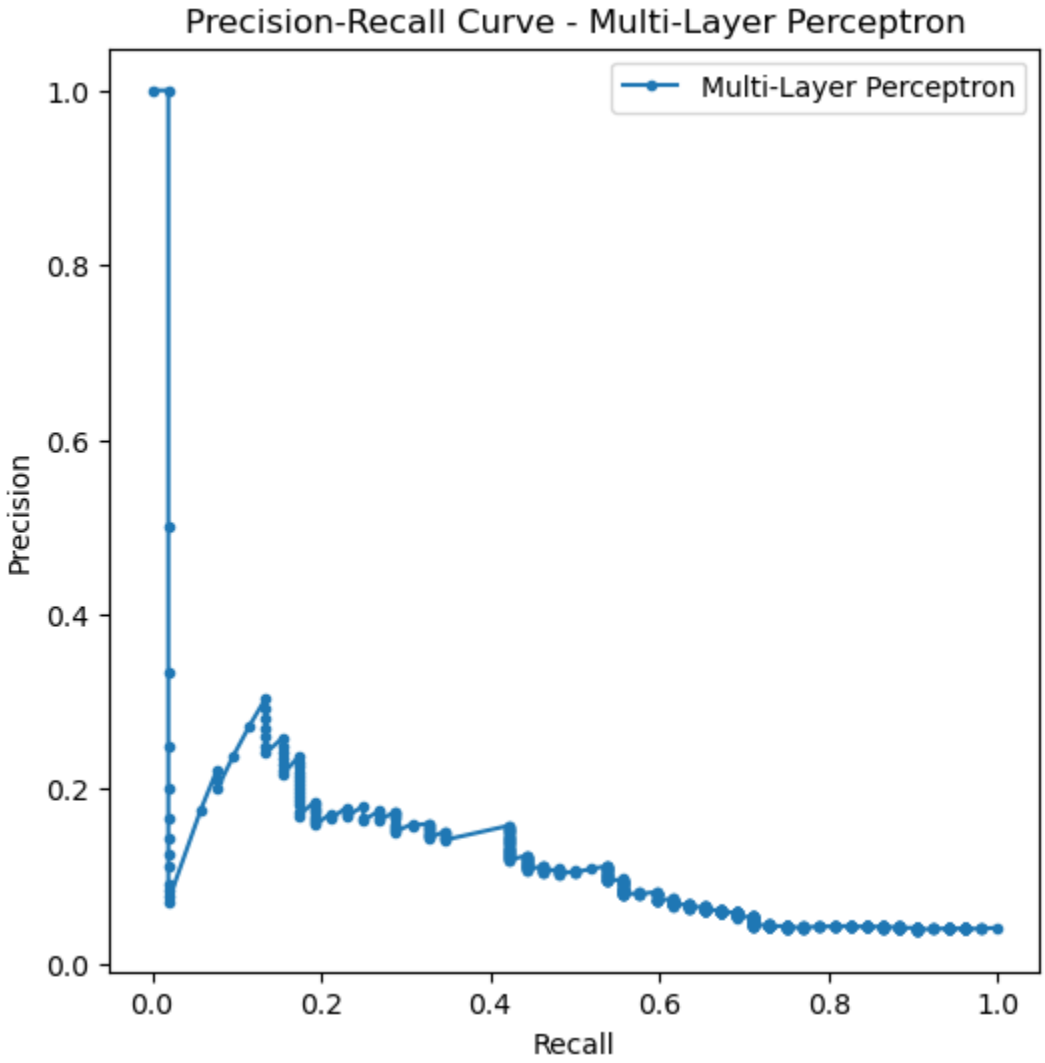

ROC Curve - K-Nearest Neighbors
True Positive Rate
False Positive Rate
Optimal Threshold = 0.048
Youden's J = 0.220
K-Nearest Neighbors (AUC = 0.61)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

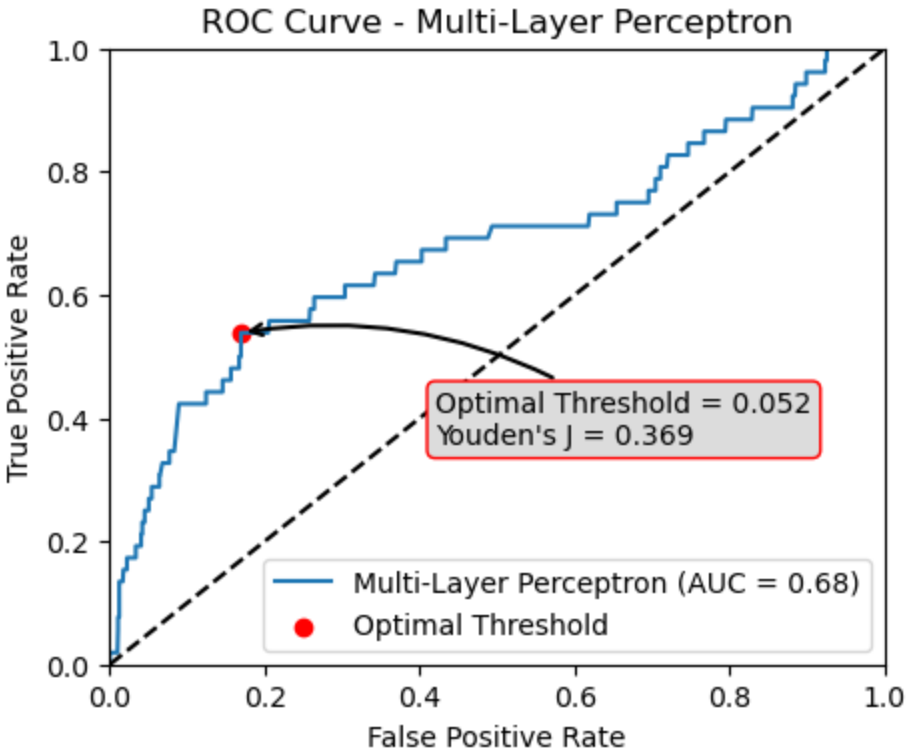

TPR / TNR vs Youdens Index - K-Nearest Neighbors
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

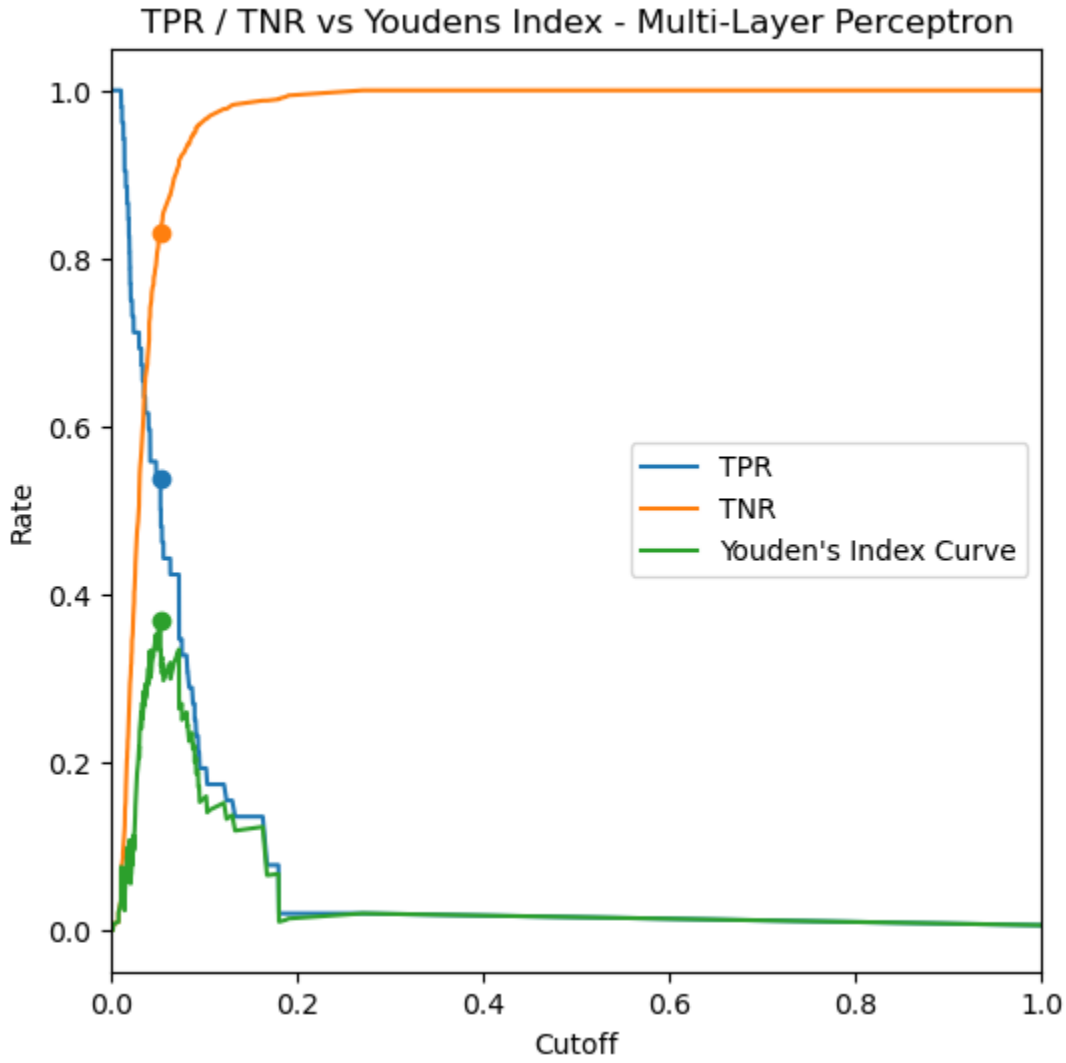


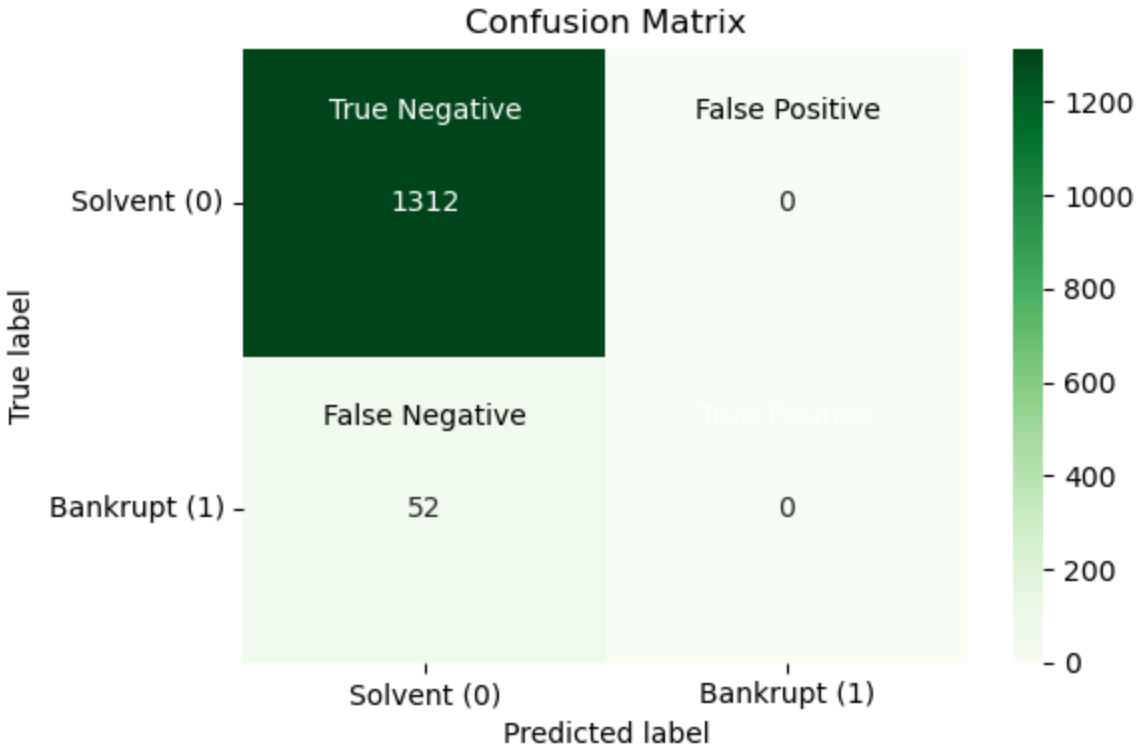
In [35]:
```
mlp_model_name, mlp_pr_curve_plot, mlp_roc_curve_plot, mlp_tpr_tnr_plot, mlp_conf_matr
                                'Multi-Layer Perceptron',
                                class_names,
                                datasets,
                                "Full")
def list_to_paragraph(value_list):
    style = getSampleStyleSheet()["BodyText"]
    return [Paragraph(str(value), style) for value in value_list]

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

```
C:\Users\bwynia\Anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predi
cted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```


Precision-Recall Curve - Multi-Layer Perceptron
Multi-Layer Perceptron
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


ROC Curve - Multi-Layer Perceptron
True Positive Rate
False Positive Rate
Optimal Threshold = 0.052
Youden's J = 0.369
Multi-Layer Perceptron (AUC = 0.68)
Optimal Threshold
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 - Multi-Layer Perceptron
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
1312
0
False Negative
True Positive
52
0
Solvent (0)
Bankrupt (1)
True label
Predicted label
1200
1000
800
600
400
200
0

Section 6: Analysis Summary

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	[[1107, 205], [8, 44]]	0.843842	0.176707	0.846154	0.843750	0.292359	0.718985	0.433734
1	AdaBoost Classifier	[[1107, 205], [9, 43]]	0.843109	0.173387	0.826923	0.843750	0.286667	0.705675	0.453807
2	Decision Tree Classifier	[[1045, 267], [5, 47]]	0.800587	0.149682	0.903846	0.796494	0.256831	0.700340	0.641413
3	Random Forest Classifier	[[1308, 4], [44, 8]]	0.964809	0.666667	0.153846	0.996951	0.250000	0.793152	0.062857
4	K-Nearest Neighbors	[[1312, 0], [52, 0]]	0.961877	0.000000	0.000000	1.000000	0.000000	0.219571	0.047582
5	Multi-Layer Perceptron	[[1312, 0], [52, 0]]	0.961877	0.000000	0.000000	1.000000	0.000000	0.369254	0.052276

Model 0: Logistic Regression

- **Strengths:**
  - Good balance between recall (0.846) and specificity (0.843), leading to a relatively high Youden J Stat (0.719).
  - Moderate F1 score (0.292) compared to other models.
- **Weaknesses:**
  - Lowest precision among all models (0.177).
  - Suboptimal accuracy (0.843).
  - Overfitting: Not evident.

Model 1: AdaBoost Classifier

- **Strengths:**
  - Similar performance to logistic regression in terms of accuracy, recall, and specificity.
  - Slightly higher optimal P threshold (0.454) compared to logistic regression.
- **Weaknesses:**

- Lower precision (0.173) and F1 score (0.287) than logistic regression.
- Lower Youden J Stat (0.706) than logistic regression.
- Overfitting: Not evident.

Model 2: Decision Tree Classifier

- **Strengths:**
  - Highest recall (0.904) among all models.
  - Moderate Youden J Stat (0.700).
- **Weaknesses:**
  - Lowest accuracy (0.801) and specificity (0.796) among all models.
  - Low precision (0.150) and F1 score (0.257).

Model 3: Random Forest Classifier

- **Strengths:**
  - Highest accuracy (0.965) and specificity (0.997) among all models.
  - High precision (0.667).
- **Weaknesses:**
  - Lowest recall (0.154) and F1 score (0.250) among all models.
  - Lower Youden J Stat (0.793) compared to logistic regression.

Model 4: K-Nearest Neighbors Not usable.

Model 5: Multi-Layer Perceptron Not usable. Does not accept a class_weights argument and defaults to a false negative.

Best Model: Logistic Regression (Model 0) seems to be the best model, given its balance between recall and specificity, and a relatively high Youden J Stat. However, it has low precision, which should be considered when assessing its utility. **Usefulness and Applicability:** The underlying models show varying performance in predicting company bankruptcy, with some models having low recall (e.g., Random Forest) or low precision (e.g., Logistic Regression). These weaknesses limit the models' applicability in real-world scenarios. In general, the models need further improvement by addressing these issues, possibly through feature engineering, parameter tuning, or exploring other classification models. Additionally, it's important to consider the potential consequences of false positives and false negatives when assessing the applicability of these models in practice.

Conclusion

The models developed in this project could be leveraged in several ways within the context of predicting company bankruptcy:

1. Risk Assessment: Investors and creditors can use the models to assess the bankruptcy risk of companies before making investment or lending decisions. This can help them manage their risk exposure and make more informed decisions.

2. Early Warning System: Companies can use the models as part of an early warning system to identify potential financial distress. By monitoring the most important features for predicting bankruptcy, management can take proactive measures to address financial issues and avoid bankruptcy.

3. Regulatory Oversight: Regulators and policymakers can use the models to identify companies with a higher risk of bankruptcy and take appropriate actions, such as increasing oversight or requiring additional disclosures, to protect investors and maintain market stability.

4. Benchmarking: The models can be used as a benchmark to compare the financial health of companies within the same industry, helping stakeholders identify industry trends and best practices.

However, there are several limitations to consider when leveraging these models:

1. Data Quality: The performance of the models depends on the quality and representativeness of the dataset used for training. If the dataset contains errors, biases, or is not representative of the population of interest, the model predictions may be inaccurate.

2. Temporal Stability: The models are trained on data from 1999 to 2009, and their performance may not be stable over time due to changes in market conditions, industry trends, or regulations. It is essential to update the models with more recent data and validate their performance periodically.

3. Model Interpretability: Some of the models, such as random forests or support vector machines, may be more difficult to interpret than simpler models like logistic regression. This can make it challenging for stakeholders to understand the rationale behind the model's predictions and trust its results.

4. Generalizability: The models are trained on data from Taiwan, which may limit their generalizability to other countries or regions with different economic conditions, accounting standards, or regulatory frameworks.

5. Imbalanced Data: The dataset may be imbalanced, with a much higher number of non-bankrupt companies than bankrupt ones. This can lead to models that are biased towards predicting the majority class, resulting in poor performance for the minority class (i.e., bankrupt companies). Techniques such as oversampling or undersampling can be used to address this issue.

In summary, while these models can provide valuable insights for predicting company bankruptcy, it is essential to be aware of their limitations and to use them in conjunction with

other sources of information and expert judgment when making decisions.

In [ ]: