Assessing Loan Quality: Developing Machine Learning Models for Loan Default Classification and Profit Optimization in Lending Club

Ben Wynia

This project aimed to use machine learning models to evaluate the quality of loan applications received by the LendingClub. The LendingClub is the largest peer-to-peer lending platform in the world, and it provides a marketplace for borrowers to obtain personal and business loans from individual investors. The primary objective of this analysis was to develop a model that could accurately classify loan applicants into those likely to "pay off" or "charge off" (default) on their loan.

To achieve this goal, the team used a dataset that contained a range of financial indicators across major categories such as debt, income, credit history, charge-offs, and others. Using this data, the team trained a machine learning model to classify loan applicants based on their likelihood of defaulting on their loan. In addition to classification, the team also aimed to optimize the acceptance point to maximize the average profit per applicant.

The outcome of this project has implications for both lenders and borrowers. By accurately predicting the likelihood of default, lenders can make more informed decisions about whether to approve a loan application. On the other hand, borrowers can benefit from a better understanding of the factors that influence their creditworthiness, which can help them make more informed financial decisions in the future. Overall, this project demonstrates the value of machine learning in the lending industry and highlights the potential benefits of using data-driven approaches to improve lending practices.

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

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/loan_data.gzip", low_memory=False)
```

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

Out[3]:

	Name	Definition
0	acc_now_delinq	The number of accounts on which the borrower i...
1	acc_open_past_24mths	Number of trades opened in past 24 months.
2	addr_state	The state provided by the borrower in the loan...
3	all_util	Balance to credit limit on all trades
4	annual_inc	The self-reported annual income provided by th...
...	...	...
148	settlement_amount	The loan amount that the borrower has agreed t...
149	settlement_percentage	The settlement amount as a percentage of the p...
150	settlement_term	The number of months that the borrower will be...
151	NaN	NaN
152	NaN	* Employer Title replaces Employer Name for al...

153 rows × 2 columns

Section 3. Data Cleanup and Exploratory Data Analysis

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.

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:

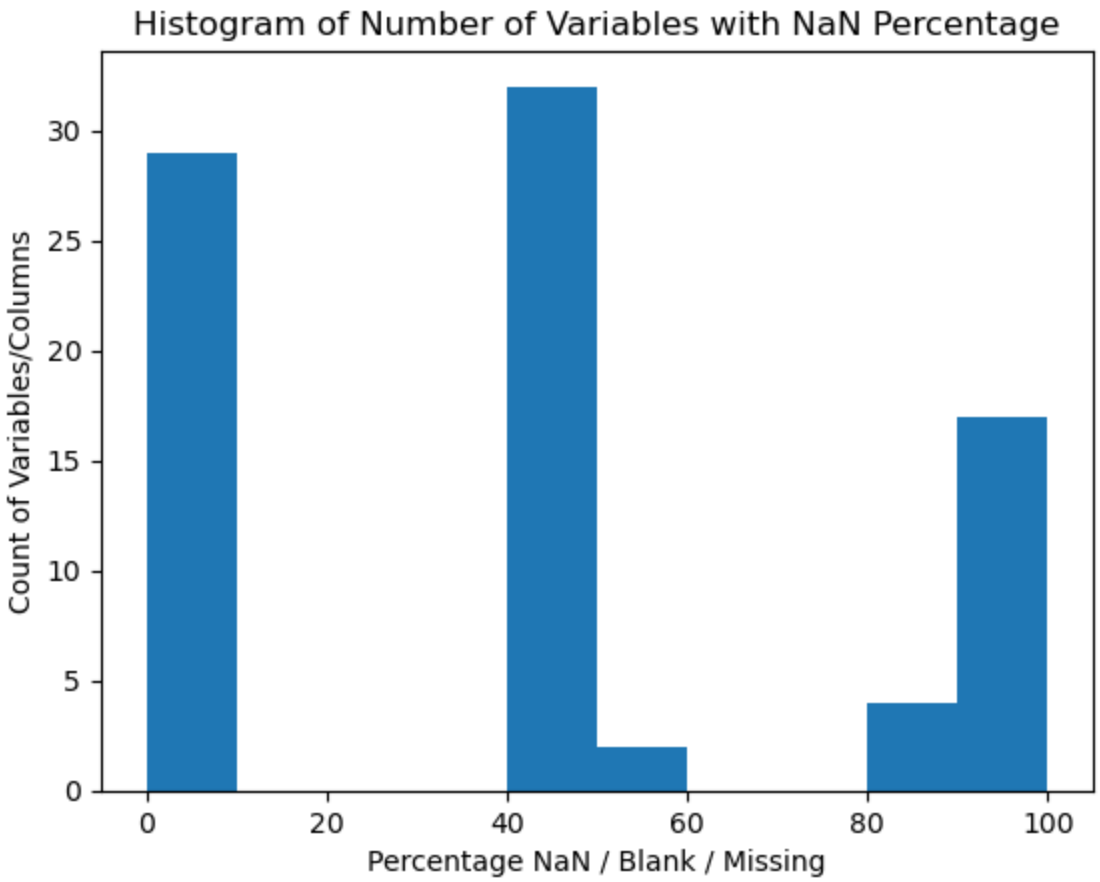
pandas numpy matplotlib.pyplot seaborn

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

```
The dataset has 142 columns and 88876 rows
```

In [5]:
```
# Keep any columns that store information that appears to be "known" at the time an or
columns_to_keep = ['loan_status','acc_open_past_24mths', 'all_util', 'annual_inc', 'ar
columns_to_recode = ['home_ownership', 'verification_status', 'purpose']
breakpoint = 0.95
df, data_dictionary2 = data_cleanup.run_data_cleanup_functions(df, data_dictionary, cc
```

```
Original dataframe---
The dataset has 142 columns and 88876 rows
```




```
After cleaning up columns with missing data---
The dataset has 28 columns and 88876 rows
Number of rows before filtering: 88876
Number of rows after filtering: 84232
Percentage of rows dropped: 5.23%
After cleaning up rows with missing data---
The dataset has 28 columns and 84232 rows
After recoding categorical variables into one-hot variables---
The dataset has 46 columns and 84232 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 [6]:
```
descriptives.descriptive_statistics(df)
```

Out[6]:

	count	missing_values	data_type	freq_dist	rel_freq	
loan_status	84232	0	object	{'Fully Paid': 70154, 'Charged Off': 12034, 'D...	{'Fully Paid': 83.28663690758856, 'Charged Off...	
annual_inc	84232	0	float64	None	None	60
chargeoff_within_12_mths	84232	0	float64	None	None	
collections_12_mths_ex_med	84232	0	float64	None	None	
delinq_2yrs	84232	0	float64	None	None	
delinq_amnt	84232	0	float64	None	None	
dti	84232	0	float64	None	None	
earliest_cr_line	84232	0	object	{'Oct-1999': 706, 'Oct-2000': 688, 'Nov-1998':...	{'Oct-1999': 0.8381612688764365, 'Oct-2000': 0...	
emp_length	84232	0	object	{'10+ years': 25100, '2 years': 8403, '< 1 yea...	{'10+ years': 29.798651343907306, '2 years': 9...	
fico_range_high	84232	0	float64	None	None	
fico_range_low	84232	0	float64	None	None	
funded_amnt	84232	0	float64	None	None	10
funded_amnt_inv	84232	0	float64	None	None	10
inq_last_6mths	84232	0	float64	None	None	
installment	84232	0	float64	None	None	
int_rate	84232	0	object	{' 10.99%': 3870, ' 11.99%': 3620, ' 12.99%': ...	{' 10.99%': 4.594453414379333, ' 11.99%': 4.29...	1
last_fico_range_high	84232	0	float64	None	None	
last_fico_range_low	84232	0	float64	None	None	
open_acc	84232	0	float64	None	None	
out_prncp	84232	0	float64	None	None	

	count	missing_values	data_type	freq_dist	rel_freq	
out_prncp_inv	84232	0	float64	None	None	
pub_rec	84232	0	float64	None	None	
pub_rec_bankruptcies	84232	0	float64	None	None	
term	84232	0	object	{' 36 months': 61281, ' 60 months': 22951}	{' 36 months': 72.75263557792762, ' 60 months'...	m
total_acc	84232	0	float64	None	None	
1h_home_ownership_MORTGAGE	84232	0	uint8	{0: 42319, 1: 41913}	{0: 50.24100104473359, 1: 49.75899895526641}	
1h_home_ownership_OTHER	84232	0	uint8	{0: 84096, 1: 136}	{0: 99.83854117200114, 1: 0.1614588279988603}	
1h_home_ownership_OWN	84232	0	uint8	{0: 77424, 1: 6808}	{0: 91.9175610219394, 1: 8.082438978060596}	
1h_home_ownership_RENT	84232	0	uint8	{0: 48857, 1: 35375}	{0: 58.002896761325864, 1: 41.997103238674136}	
1h_verification_status_Not Verified	84232	0	uint8	{0: 56253, 1: 27979}	{0: 66.78340773102859, 1: 33.21659226897141}	
1h_verification_status_Source Verified	84232	0	uint8	{0: 59196, 1: 25036}	{0: 70.27732928103335, 1: 29.722670718966665}	
1h_verification_status_Verified	84232	0	uint8	{0: 53015, 1: 31217}	{0: 62.93926298793807, 1: 37.06073701206192}	
1h_purpose_car	84232	0	uint8	{0: 82418, 1: 1814}	{0: 97.84642416183873, 1: 2.153575838161269}	
1h_purpose_credit_card	84232	0	uint8	{0: 67464, 1: 16768}	{0: 80.09307626555228, 1: 19.906923734447716}	
1h_purpose_debt_consolidation	84232	0	uint8	{1: 45304, 0: 38928}	{1: 53.78478487985564, 0: 46.215215120144364}	
1h_purpose_educational	84232	0	uint8	{0: 83873, 1: 359}	{0: 99.57379618197359, 1: 0.4262038180264033}	
1h_purpose_home_improvement	84232	0	uint8	{0: 79239, 1:	{0: 94.07232405736538, 1:	

	count	missing_values	data_type	freq_dist	rel_freq
				4993}	5.927675942634628}
1h_purpose_house	84232	0	uint8	{0: 83692, 1: 540}	{0: 99.35891347706335, 1: 0.6410865229366511}
1h_purpose_major_purchase	84232	0	uint8	{0: 81361, 1: 2871}	{0: 96.59155665305347, 1: 3.4084433469465285}
1h_purpose_medical	84232	0	uint8	{0: 83202, 1: 1030}	{0: 98.77718681736157, 1: 1.2228131826384272}
1h_purpose_moving	84232	0	uint8	{0: 83466, 1: 766}	{0: 99.09060689524172, 1: 0.9093931047582866}
1h_purpose_other	84232	0	uint8	{0: 78358, 1: 5874}	{0: 93.02640326716687, 1: 6.973596732833127}
1h_purpose_renewable_energy	84232	0	uint8	{0: 84118, 1: 114}	{0: 99.86465951182448, 1: 0.13534048817551525}
1h_purpose_small_business	84232	0	uint8	{0: 81938, 1: 2294}	{0: 97.27656947478393, 1: 2.7234305252160698}
1h_purpose_vacation	84232	0	uint8	{0: 83705, 1: 527}	{0: 99.37434704150442, 1: 0.6256529584955837}
1h_purpose_wedding	84232	0	uint8	{0: 83254, 1: 978}	{0: 98.83892107512584, 1: 1.161078924874157}

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

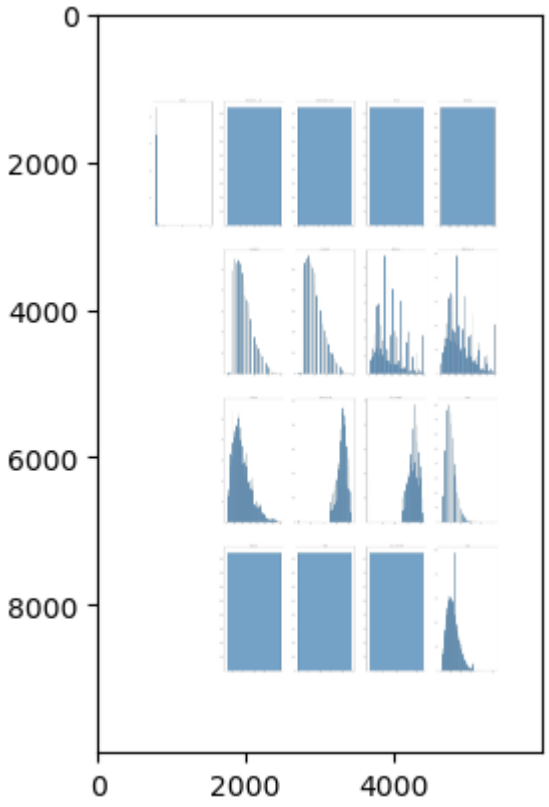
Example Usage:

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 [7]:
```
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

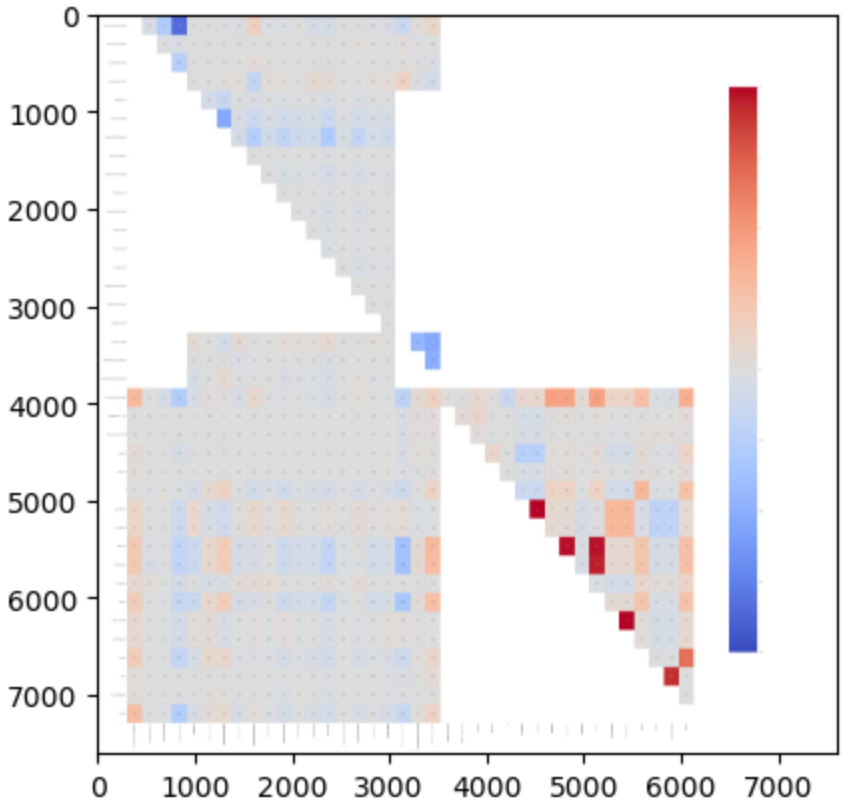
Function: Correlation Heatmap This function computes and visualizes pairwise correlation matrix of numeric variables in a Pandas dataframe using Spearman's rank correlation method. It drops variables that have too few samples or zero variance before computing the correlation

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 [8]:
```
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 Ordinal Variables

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

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

```
Categorical Variables: ['loan_status', 'chargeoff_within_12_mths', 'collections_12_mt
hs_ex_med', 'out_prncp', 'out_prncp_inv', 'pub_rec_bankruptcies', 'term', '1h_home_ow
nership_MORTGAGE', '1h_home_ownership_OTHER', '1h_home_ownership_OWN', '1h_home_owner
ship_RENT', '1h_verification_status_Not Verified', '1h_verification_status_Source Ver
ified', '1h_verification_status_Verified', '1h_purpose_car', '1h_purpose_credit_car
d', '1h_purpose_debt_consolidation', '1h_purpose_educational', '1h_purpose_home_impro
vement', '1h_purpose_house', '1h_purpose_major_purchase', '1h_purpose_medical', '1h_p
urpose_moving', '1h_purpose_other', '1h_purpose_renewable_energy', '1h_purpose_small_
business', '1h_purpose_vacation', '1h_purpose_wedding']
Ordinal Variables: []
```

In [11]:
```
"""
Some of the variables are ordinal or strings and not categorical.
These require manual handling, which is accomplished in the lines below
"""
# Create a dictionary to map emp_length categories to their corresponding numeric valu
emp_length_dict = {'10+ years': 10,
                   '9 years': 9,
                   '8 years': 8,
                   '7 years': 7,
                   '6 years': 6,
                   '5 years': 5,
                   '4 years': 4,
                   '3 years': 3,
                   '2 years': 2,
                   '1 year': 1,
                   '< 1 year': 0.5,
                   np.nan: 0}

# Replace emp_length categories with their corresponding numeric values
df['emp_length'] = df['emp_length'].map(emp_length_dict)

# Clean up the interest rate field
df['int_rate'] = df['int_rate'].str.strip('%').astype(float) / 100

# Convert the earliest credit line date to a number of months relative to the newest r
df['earliest_cr_line'] = pd.to_datetime(df['earliest_cr_line'], format='%b-%Y')
fixed_point = df['earliest_cr_line'].max()
df['earliest_cr_line'] = df['earliest_cr_line'].apply(lambda x: (fixed_point.year - x.

# Define a dictionary to map terms to the number of months
term_map = {
    ' 36 months': 36,
    ' 60 months': 60
}
```

```
# Map the terms to the number of months and store the result in a new column
df['term'] = df['term'].map(term_map)

# Use replace() and map() to refactor the values in the loan_status column into a bina
df['loan_status'] = df['loan_status'].replace({'Does not meet the credit policy. Statu
df['loan_status'] = df['loan_status'].map({'Fully Paid': 1, 'Charged Off': 0})

# View new distribution of target (dependent) variable
df["loan_status"].value_counts()
```

Out[11]:
```
1    71668
0    12564
Name: loan_status, dtype: int64
```

Define the Dependent Variable

In [12]:
```
target = 'loan_status'
target
```

Out[12]:
```
'loan_status'
```

Define the Independent Variables

In [13]:
```
features = [col for col in df.columns if col != target]
features
```

Out[13]:
```
['annual_inc',
 'chargeoff_within_12_mths',
 'collections_12_mths_ex_med',
 'delinq_2yrs',
 'delinq_amnt',
 'dti',
 'earliest_cr_line',
 'emp_length',
 'fico_range_high',
 'fico_range_low',
 'funded_amnt',
 'funded_amnt_inv',
 'inq_last_6mths',
 'installment',
 'int_rate',
 'last_fico_range_high',
 'last_fico_range_low',
 'open_acc',
 'out_prncp',
 'out_prncp_inv',
 'pub_rec',
 'pub_rec_bankruptcies',
 'term',
 'total_acc',
 '1h_home_ownership_MORTGAGE',
 '1h_home_ownership_OTHER',
 '1h_home_ownership_OWN',
 '1h_home_ownership_RENT',
 '1h_verification_status_Not Verified',
 '1h_verification_status_Source Verified',
 '1h_verification_status_Verified',
 '1h_purpose_car',
 '1h_purpose_credit_card',
 '1h_purpose_debt_consolidation',
 '1h_purpose_educational',
 '1h_purpose_home_improvement',
 '1h_purpose_house',
 '1h_purpose_major_purchase',
 '1h_purpose_medical',
 '1h_purpose_moving',
 '1h_purpose_other',
 '1h_purpose_renewable_energy',
 '1h_purpose_small_business',
 '1h_purpose_vacation',
 '1h_purpose_wedding']
```

In [14]:
```
df[target].value_counts()
```

Out[14]:
```
1    71668
0    12564
Name: loan_status, 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 [15]:
```
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 [16]:
```
correlations = view_independent_variable_correlations(df, target)
correlations
```

Out[16]:
```
last_fico_range_high                        0.466954
last_fico_range_low                         0.423546
fico_range_low                              0.118879
fico_range_high                             0.118879
1h_verification_status_Not Verified         0.046156
annual_inc                                  0.045908
1h_purpose_credit_card                      0.036063
1h_home_ownership_MORTGAGE                  0.032840
earliest_cr_line                            0.029779
1h_purpose_major_purchase                   0.019696
1h_purpose_car                              0.016661
1h_purpose_home_improvement                 0.014360
1h_purpose_wedding                          0.013961
total_acc                                   0.013490
chargeoff_within_12_mths                    0.008411
1h_purpose_vacation                         0.000679
1h_home_ownership_OWN                      -0.001898
1h_purpose_renewable_energy                -0.002716
1h_purpose_moving                          -0.003772
1h_purpose_medical                         -0.004356
delinq_amnt                                -0.004947
1h_purpose_house                           -0.006035
emp_length                                 -0.006132
1h_home_ownership_OTHER                    -0.007233
open_acc                                   -0.007954
1h_verification_status_Source Verified     -0.008359
1h_purpose_educational                     -0.008416
delinq_2yrs                                -0.009880
pub_rec                                    -0.014909
collections_12_mths_ex_med                 -0.015148
1h_purpose_other                           -0.015287
pub_rec_bankruptcies                       -0.020372
1h_purpose_debt_consolidation              -0.021422
installment                                -0.029509
1h_home_ownership_RENT                     -0.031633
1h_verification_status_Verified            -0.037101
funded_amnt_inv                            -0.042927
funded_amnt                                -0.050860
1h_purpose_small_business                  -0.054017
dti                                        -0.063312
inq_last_6mths                             -0.073957
term                                       -0.156333
int_rate                                   -0.208274
out_prncp                                        NaN
out_prncp_inv                                    NaN
Name: loan_status, 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 [17]:
```
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 [18]:
```
class_weights, distribution = compute_class_weights(df, target)
```

```
The distribution of the dependent variable is: 1    71668
0    12564
Name: loan_status, dtype: int64
The resuling class weights are: {0: 5.7042343202801655, 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 [19]:
```
labels = ['Charged Off','Paid Off']
train_data, test_data, val_data = train_test_validate.split_data(df, labels, target)
```

```
Train data dependent distribution:
    7666 (0 - Charged Off) 42873(1 - Paid Off)
Test data dependent distribution:
    2454 (0 - Charged Off) 14393(1 - Paid Off)
Validation data dependent distribution:
    2444 (0 - Charged Off) 14402(1 - Paid Off)
```

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 [20]:
```
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\lending_club_loan_defaults/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 [21]:
```
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 [22]:
```
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 [23]:
```
#feature_subset = select_features_rfe(train_data[features], train_data[target], n_feat
#feature_subset = df.iloc[:, feature_subset].copy().columns
#feature_subset
```

In [24]:
```
# 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]
    }
]

# 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': 'newton-cg', 'penalty': 'l2', 'max_iter': 1000, 'fit_intercept': False, 'c
lass_weight': {0: 5.7042343202801655, 1: 1}, 'C': 0.1}
0.8755403085822107
```

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 [25]:
```
# 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: 5.
7042343202801655, 1: 1}, max_depth=2), 'learning_rate': 0.1, 'n_estimators': 200, 'ra
ndom_state': 7}
0.8797117884918443
```

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 [26]:
```
# 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': None, 'criterion': 'entropy', 'max_depth': 5, 'max_features': None,
'min_samples_leaf': 10, 'min_samples_split': 2, 'splitter': 'best'}
0.8714493182534323
```

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 [27]:
```
# 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': 10, 'min
_samples_leaf': 10, 'max_features': 'auto', 'max_depth': 20, 'criterion': 'gini', 'cl
ass_weight': {0: 5.7042343202801655, 1: 1}, 'bootstrap': False}
0.8817041552932604
```

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 [28]:
```
# 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': 50, 'algorithm': 'kd_tree'}
0.7194070492684629
```

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 [29]:
```
# 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,)],
    'activation': ['logistic', 'relu'],
    'solver': ['lbfgs'],
    'alpha': [0.01, 0.1]
}
```

```
# 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_
```

```
C:\Users\bwynia\Anaconda3\lib\site-packages\sklearn\model_selection\_search.py:292: U
serWarning: The total space of parameters 8 is smaller than n_iter=100. Running 8 ite
rations. For exhaustive searches, use GridSearchCV.
  warnings.warn(
Trained model and hyperparameter results saved to files.
{'solver': 'lbfgs', 'hidden_layer_sizes': (100,), 'alpha': 0.01, 'activation': 'logis
tic'}
0.7343506165220705
C:\Users\bwynia\Anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percep
tron.py:549: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

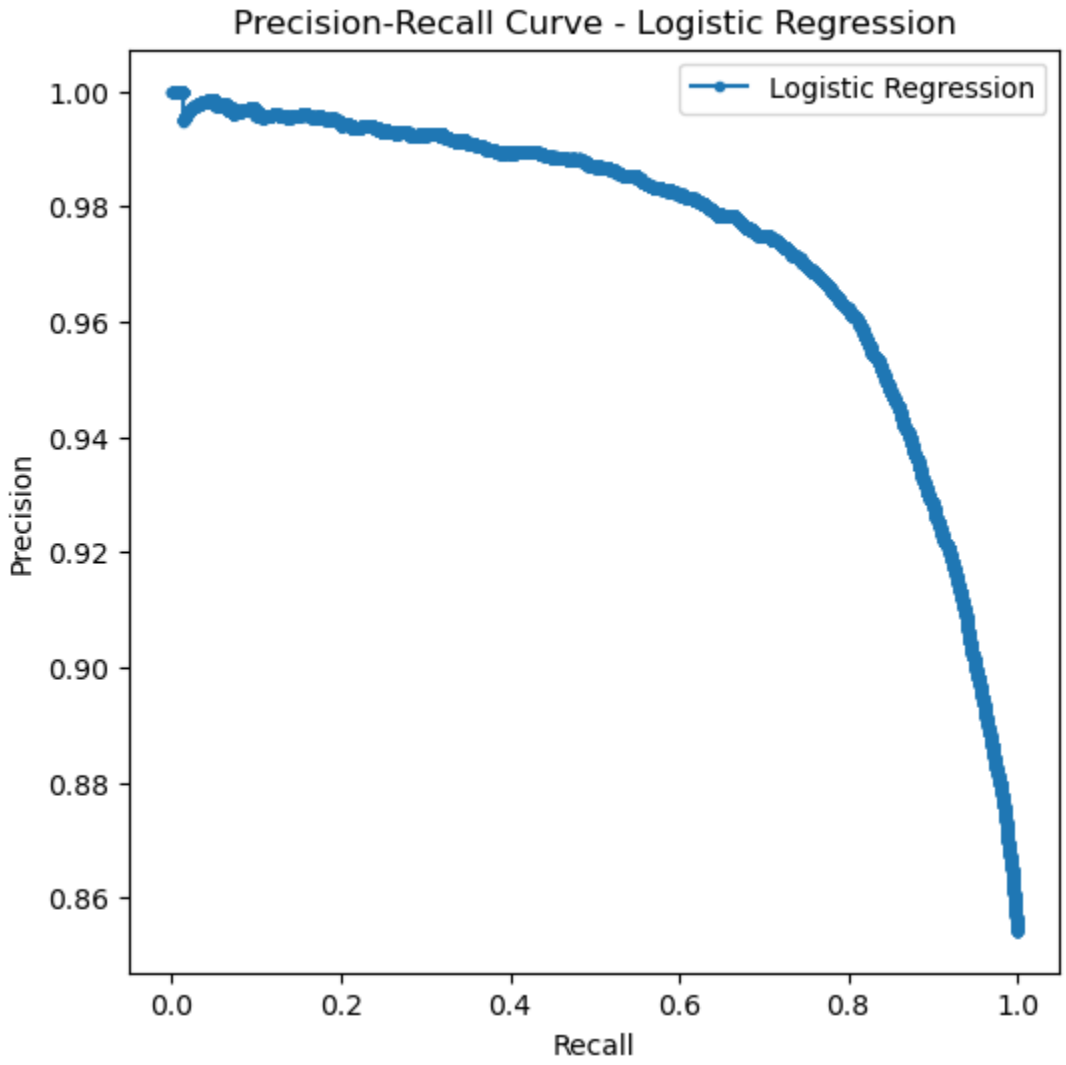
Section 5: Model Evaluation

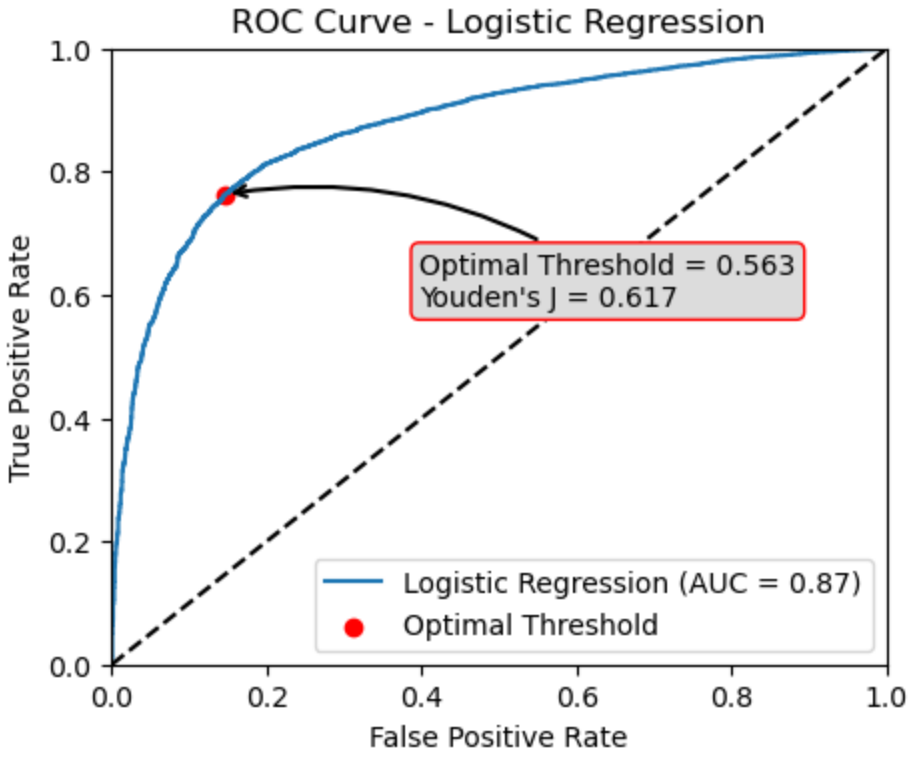
Global Model Variables

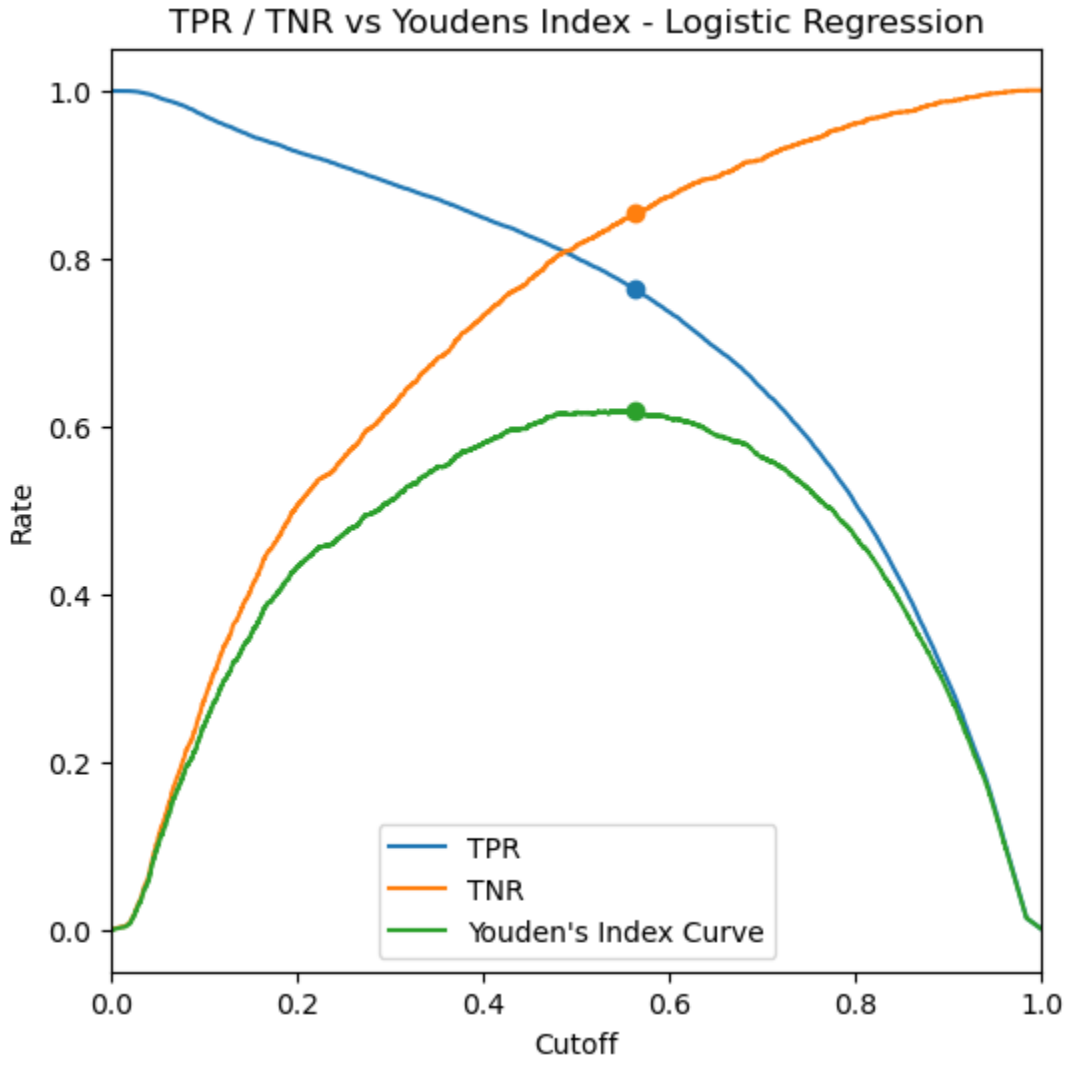
In [35]:
```
# Initialize a dataframe to store the results
results_frame = pd.DataFrame(columns=['Model Name',
                                      'Confusion Matrix',
                                      'Accuracy',
                                      'Precision',
                                      '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 = ["Charge Off (0)", "Paid Off (1)"]
```

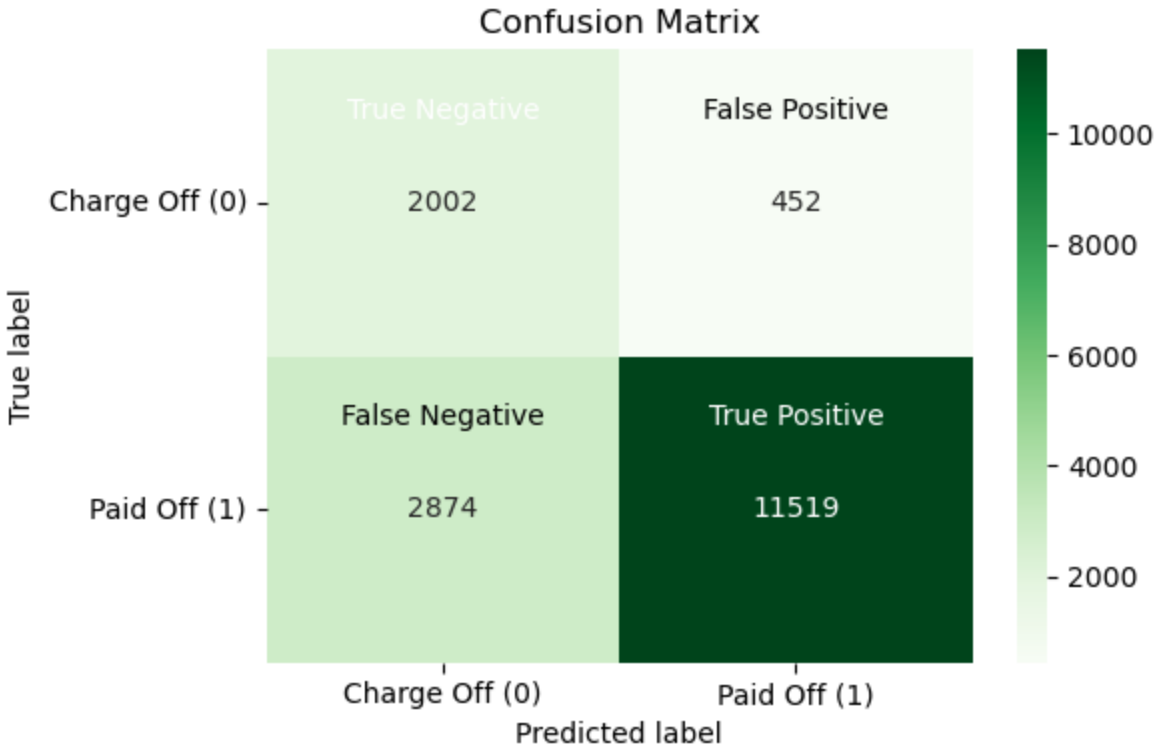
Logistic Regression

In [36]:
```
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)
```

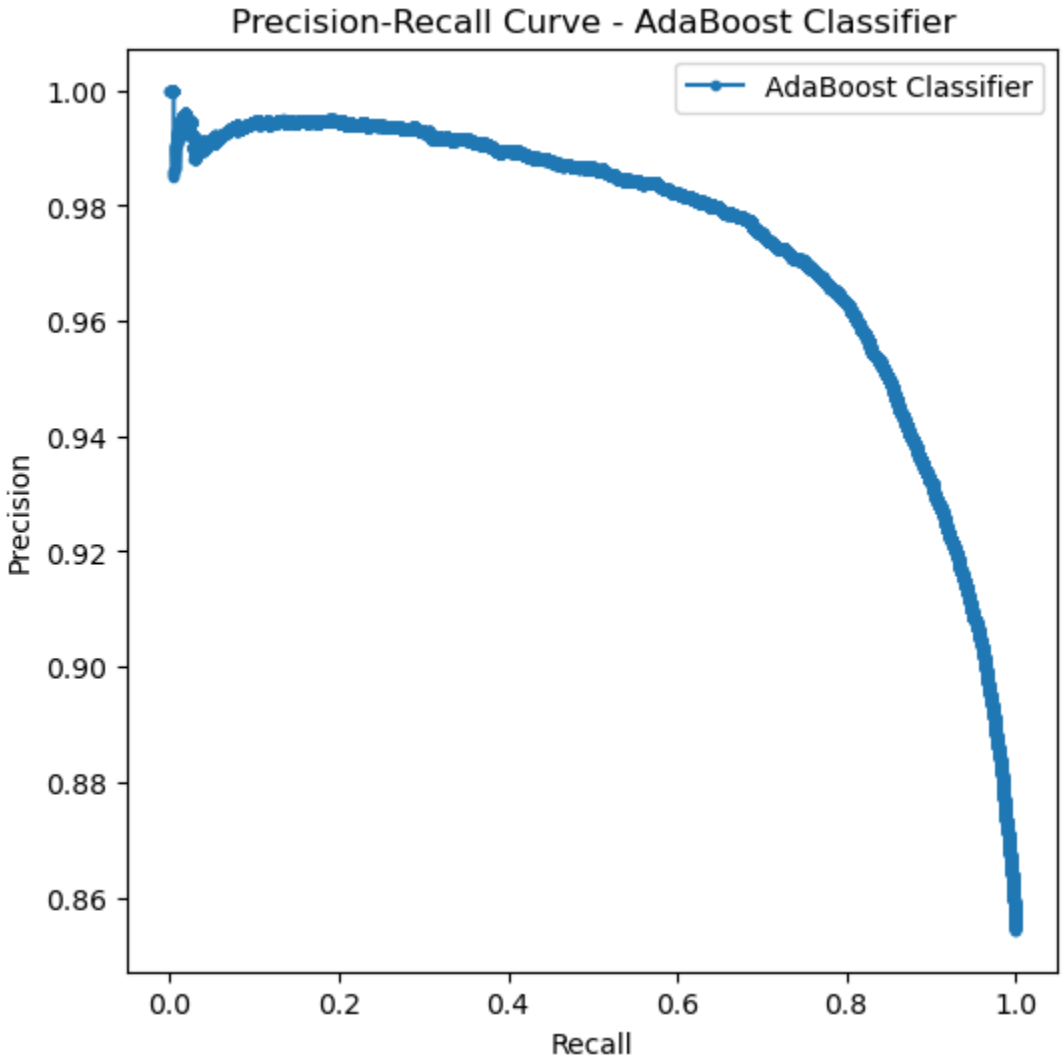

Precision-Recall Curve - Logistic Regression
Logistic Regression
Precision
Recall
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.0
0.2
0.4
0.6
0.8
1.0

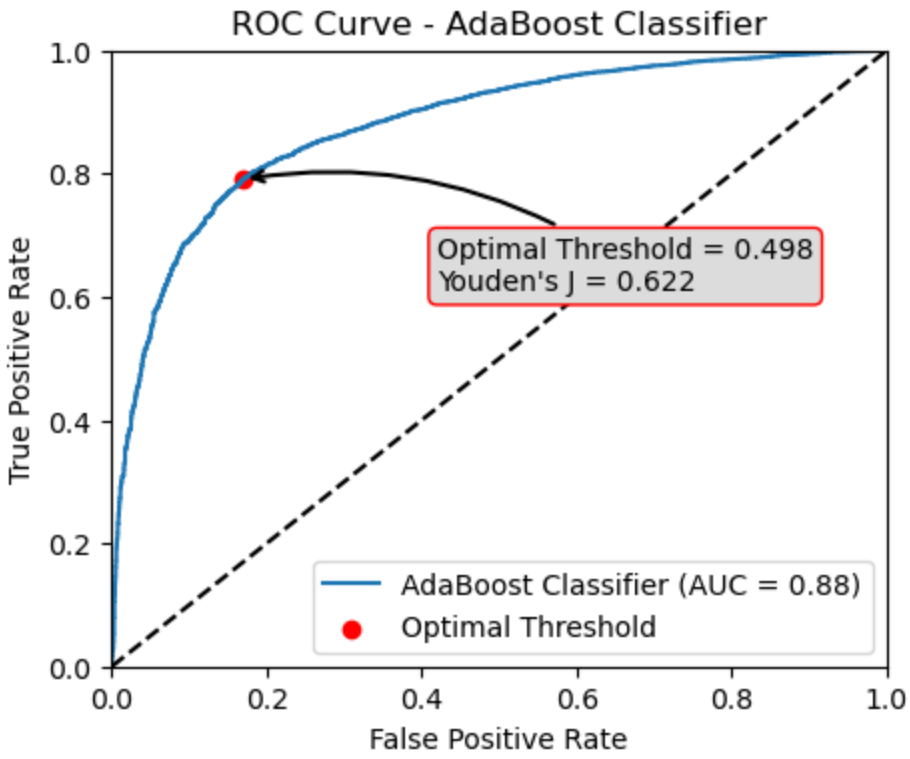

ROC Curve - Logistic Regression
Optimal Threshold = 0.563
Youden's J = 0.617
Logistic Regression (AUC = 0.87)
Optimal Threshold
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

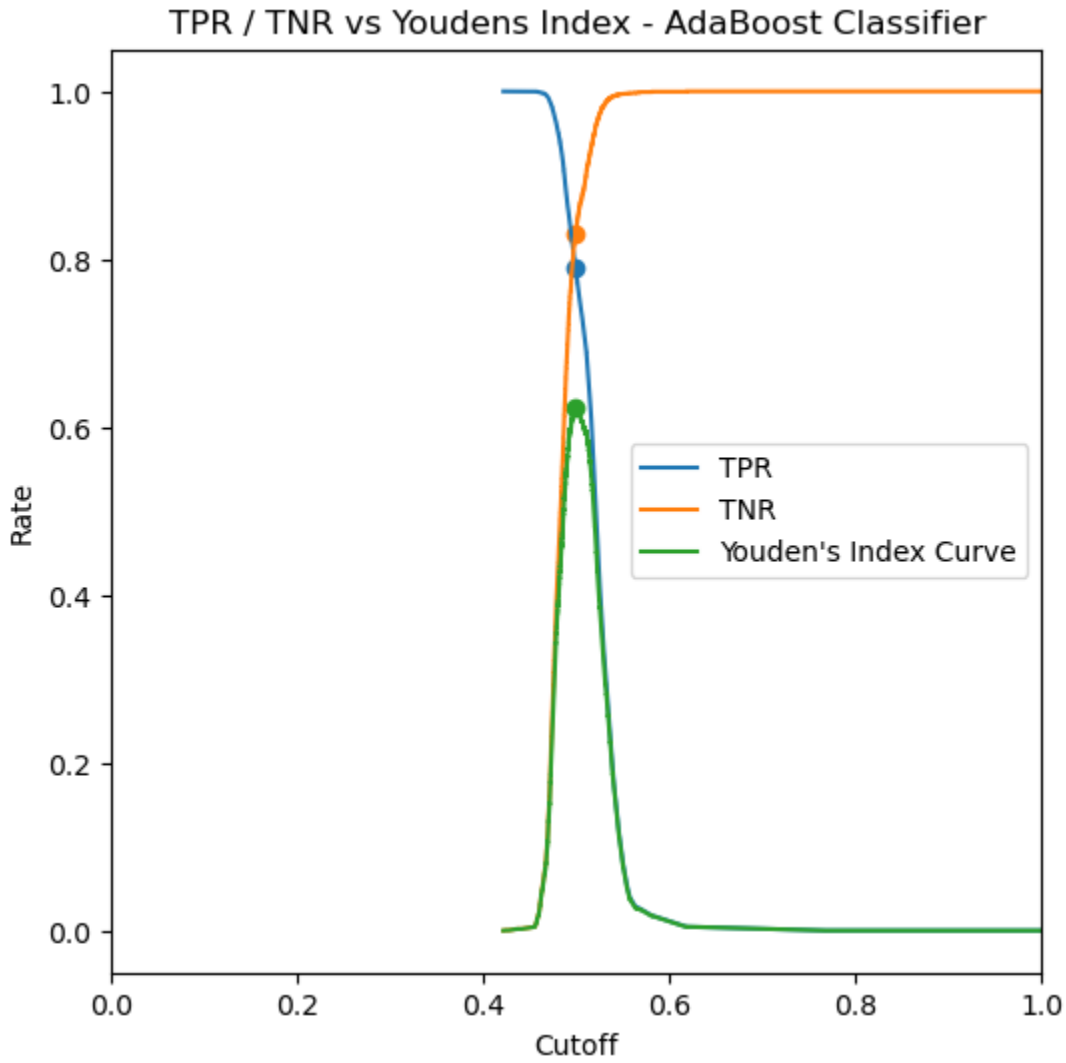

TPR / TNR vs Youdens Index - Logistic Regression
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

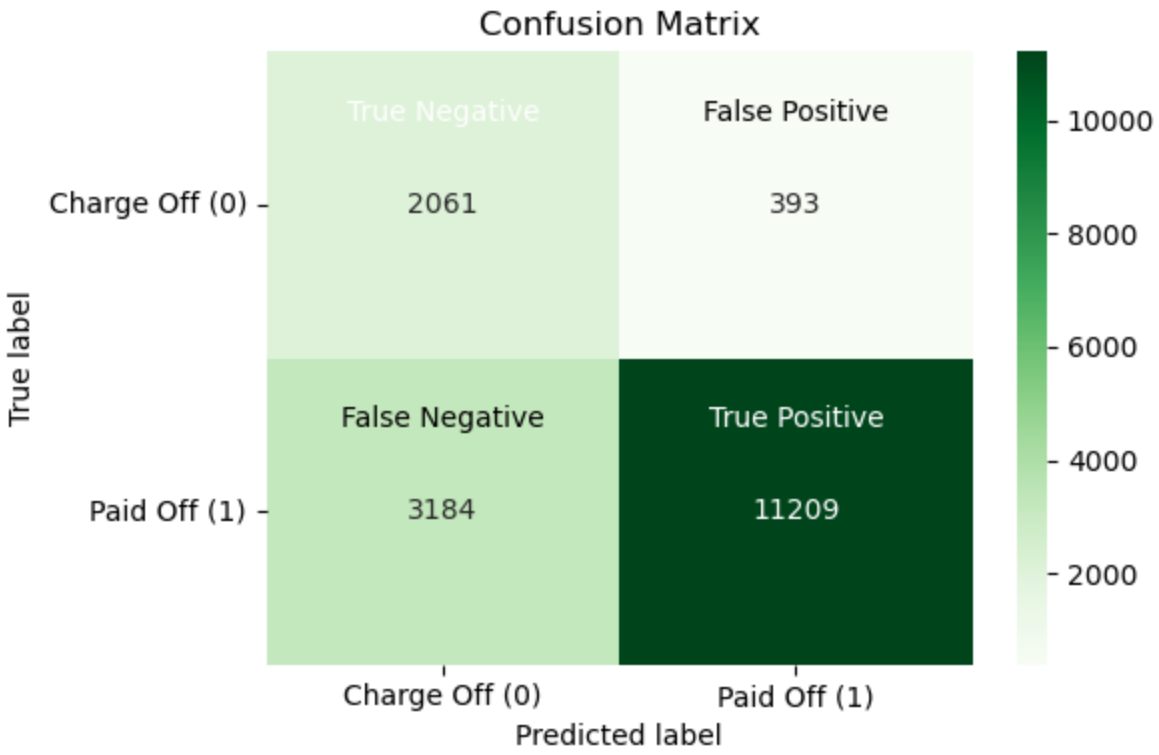

Confusion Matrix
True Negative
False Positive
2002
452
False Negative
True Positive
2874
11519
Charge Off (0)
Paid Off (1)
True label
Predicted label
10000
8000
6000
4000
2000

In [37]:
```
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)
```

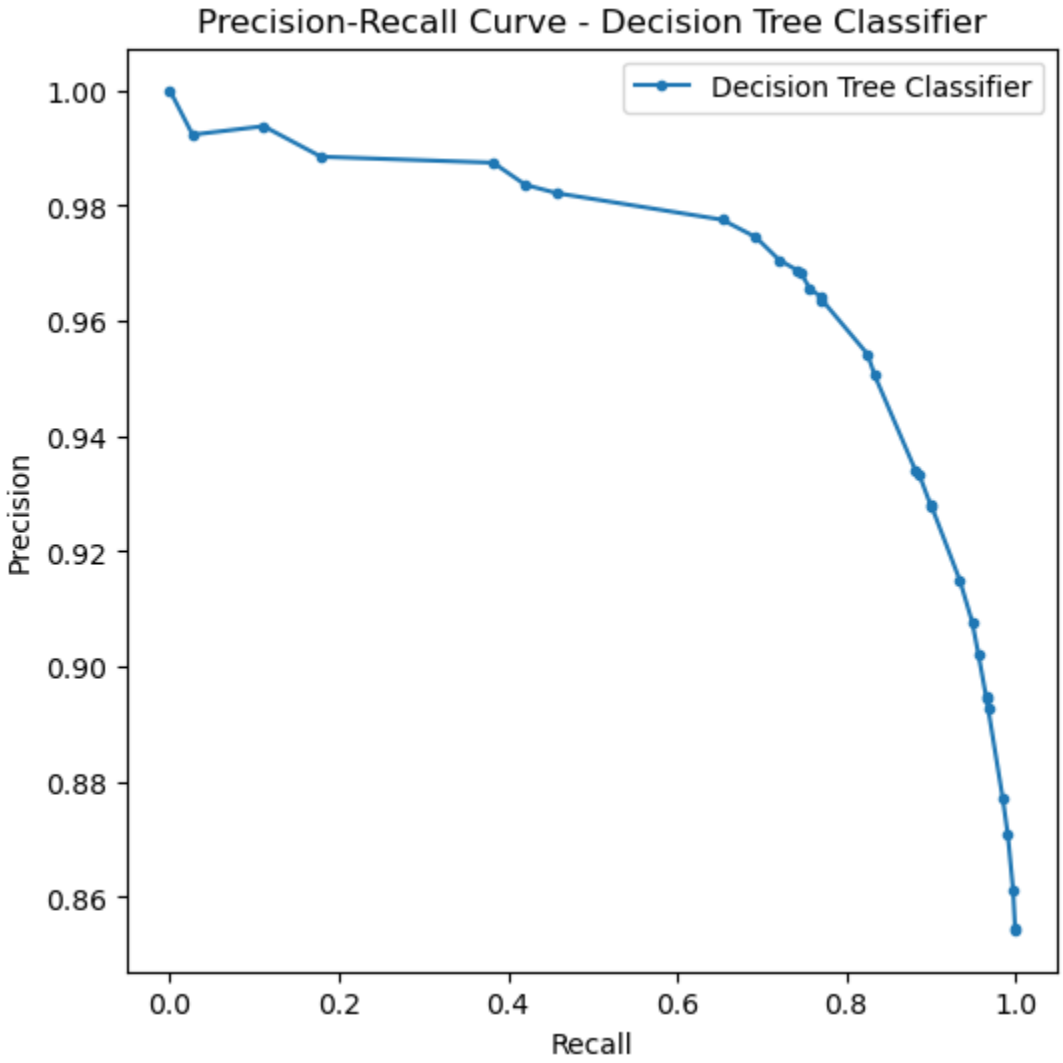


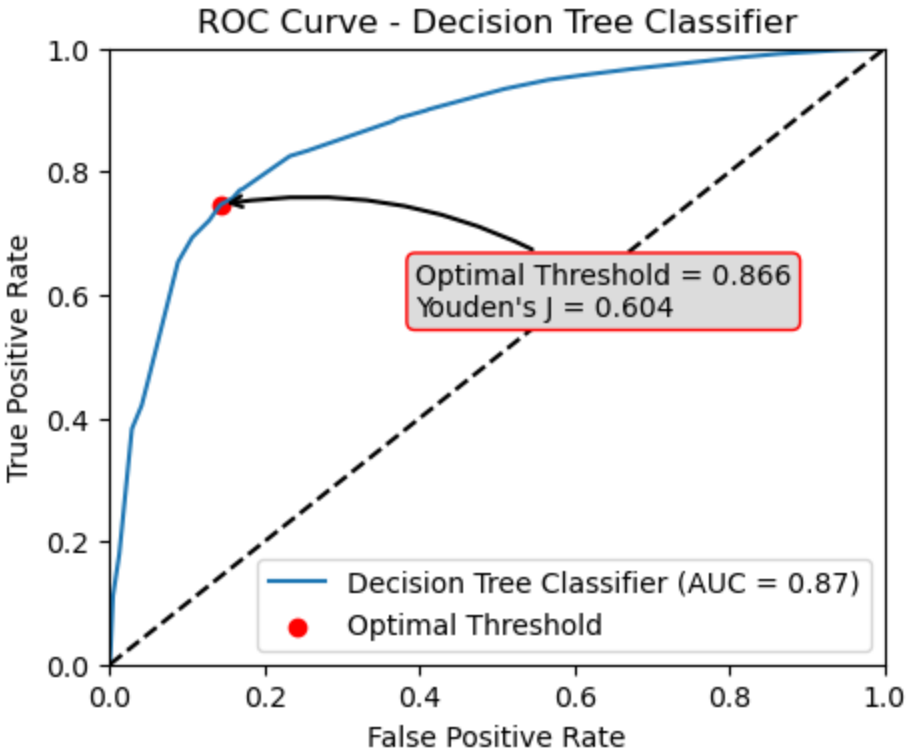

ROC Curve - AdaBoost Classifier
True Positive Rate
False Positive Rate
Optimal Threshold = 0.498
Youden's J = 0.622
AdaBoost Classifier (AUC = 0.88)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

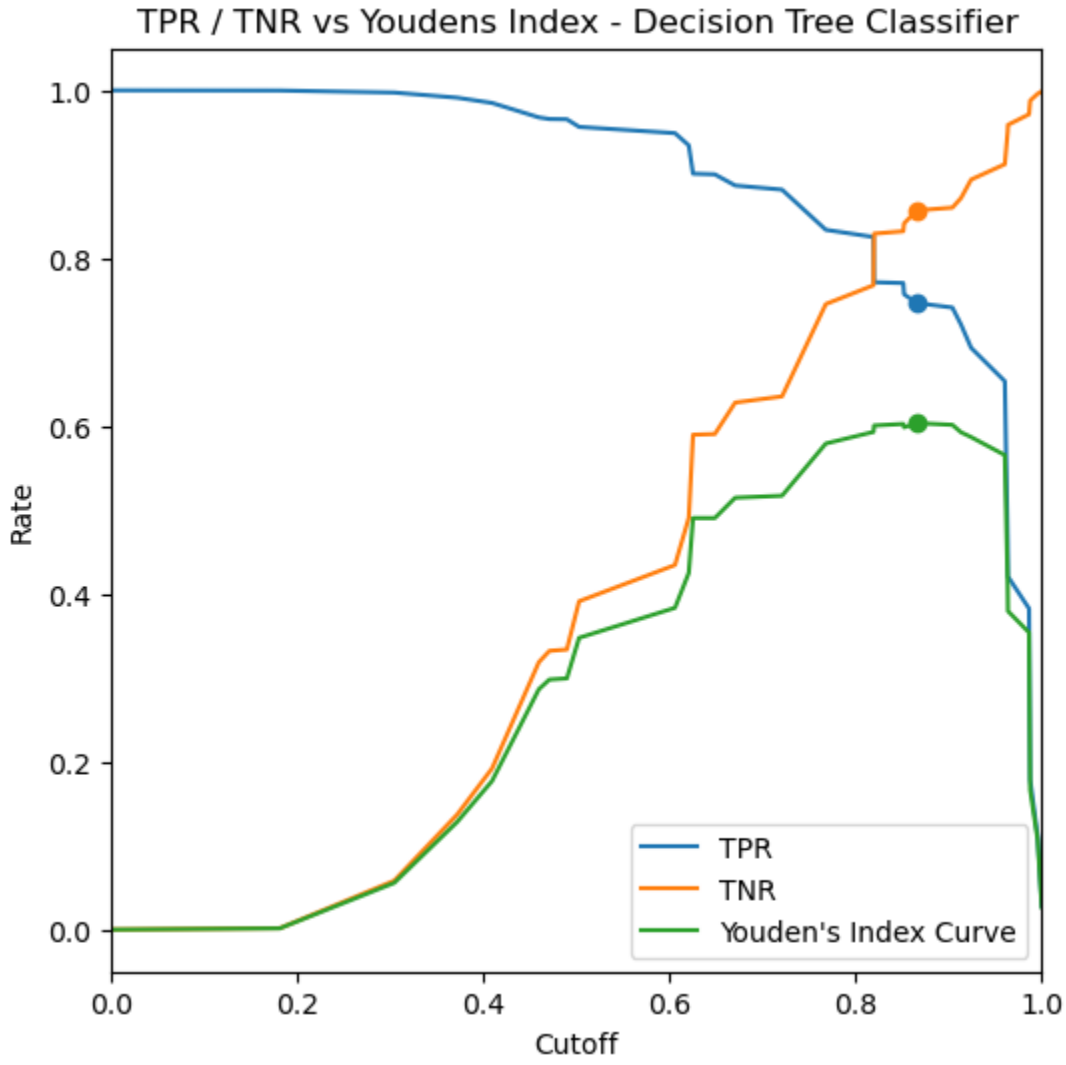

TPR / TNR vs Youdens Index - AdaBoost Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

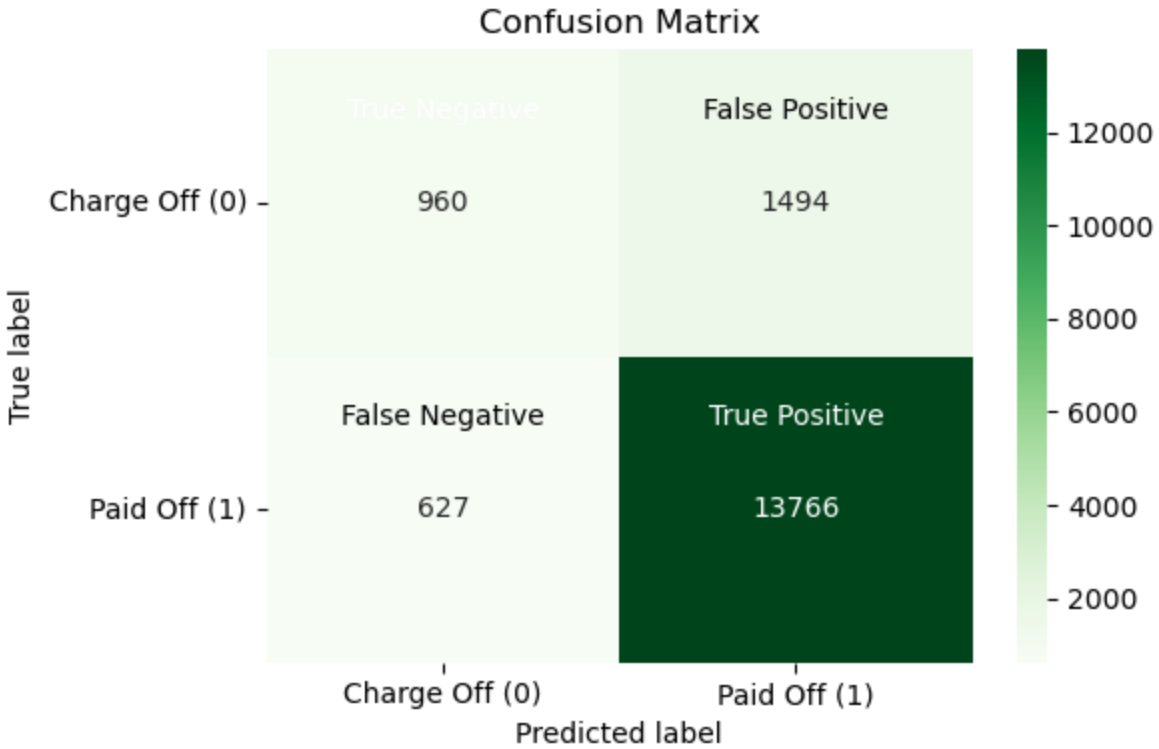


In [38]:
```
dt_model_name, dt_pr_curve_plot, dt_roc_curve_plot, dt_tpr_tnr_plot, dt_conf_matrix, d
                                'Decision Tree Classifier',
                                class_names,
                                datasets,
                                "full")

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)
```

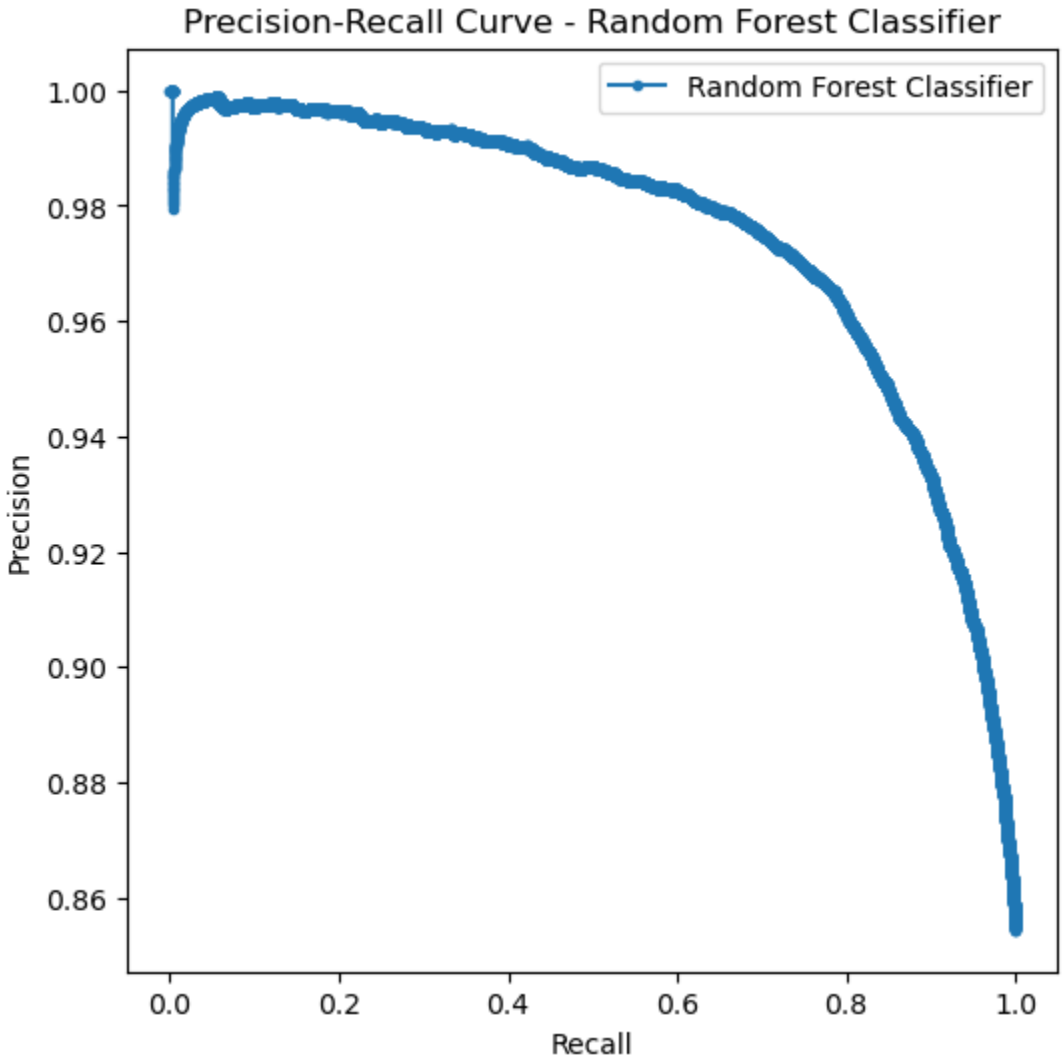

Precision-Recall Curve - Decision Tree Classifier
Decision Tree Classifier
Precision
Recall
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.0
0.2
0.4
0.6
0.8
1.0

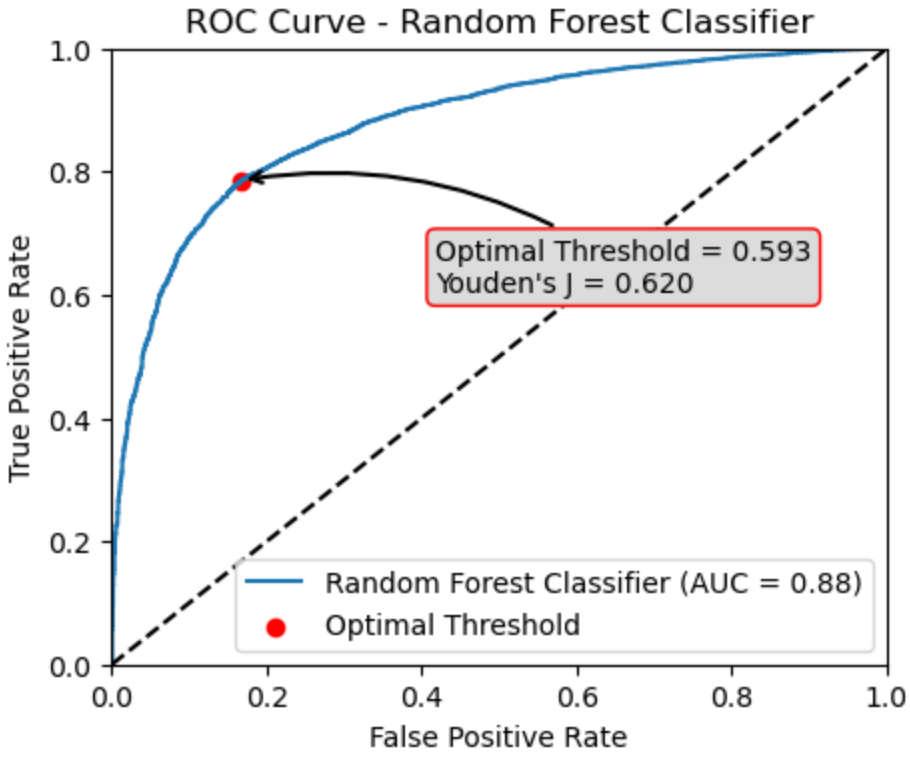

ROC Curve - Decision Tree Classifier
True Positive Rate
False Positive Rate
Optimal Threshold = 0.866
Youden's J = 0.604
Decision Tree Classifier (AUC = 0.87)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

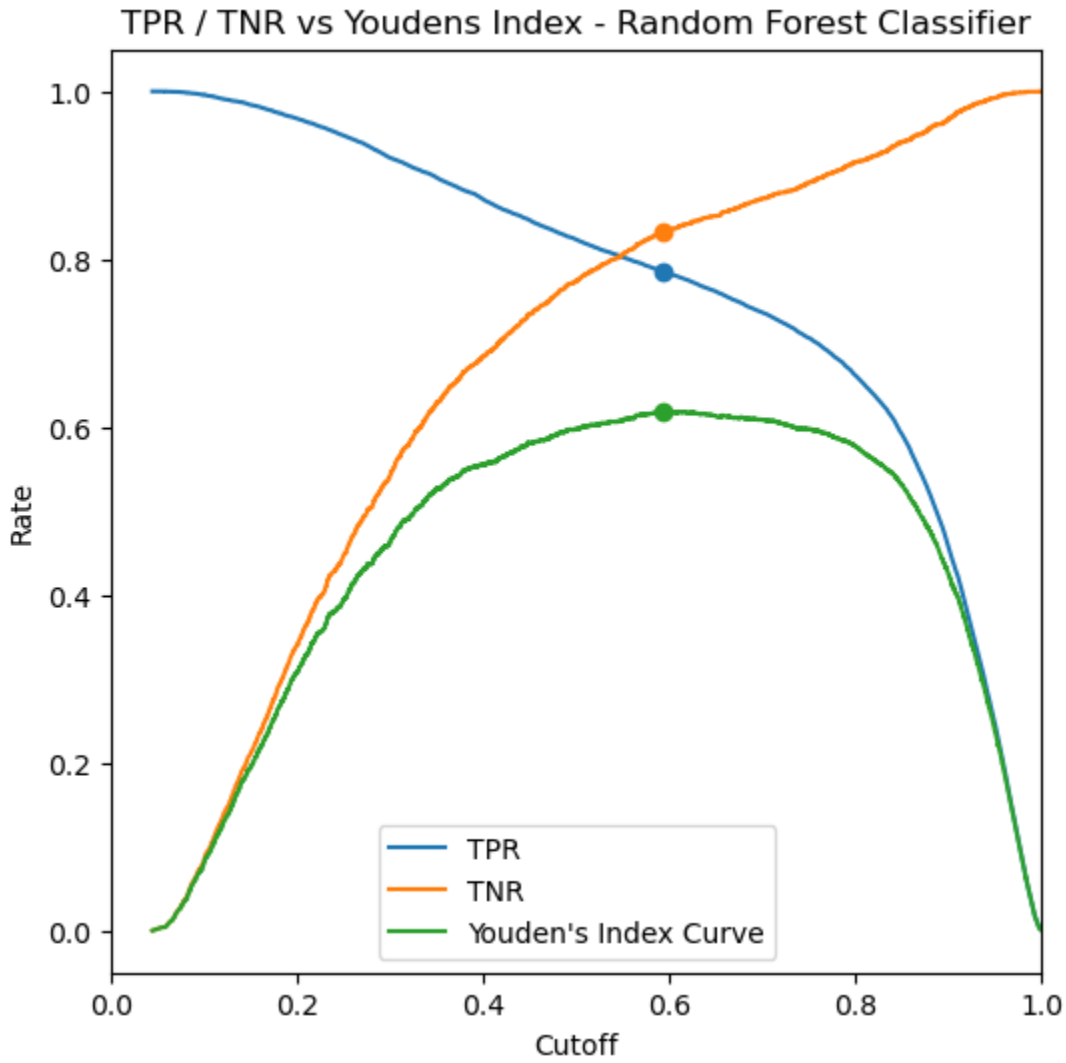

TPR / TNR vs Youdens Index - Decision Tree Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

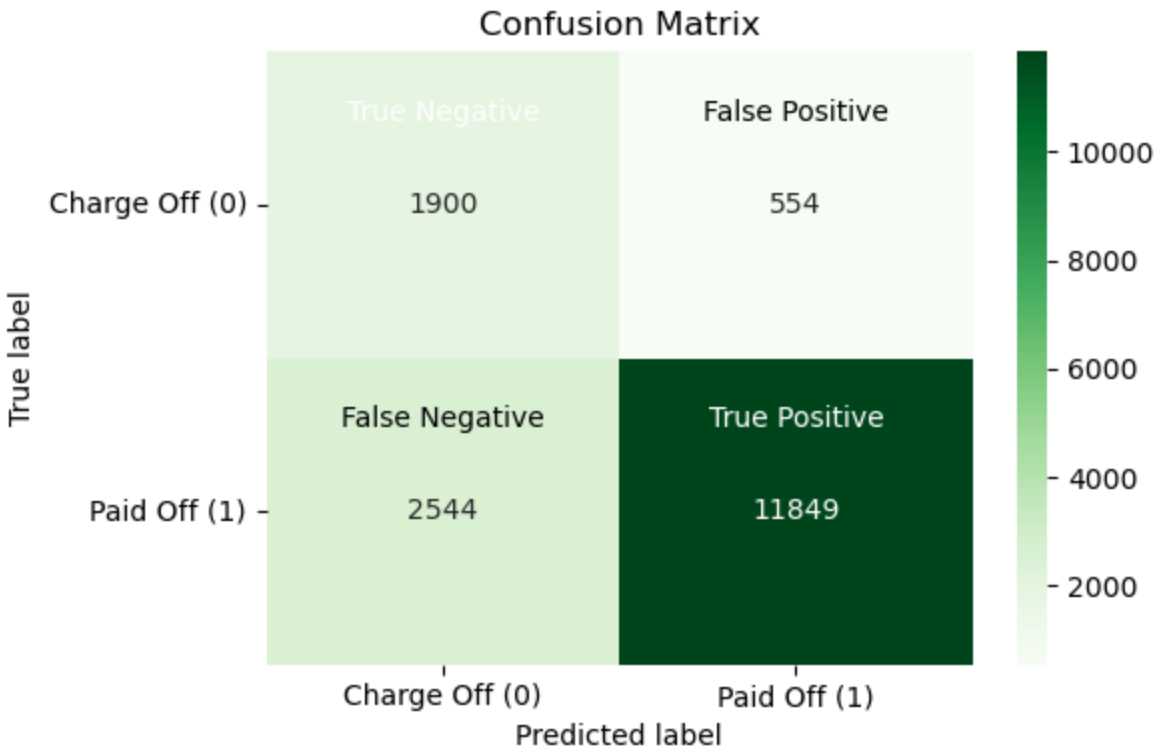

Confusion Matrix
True Negative
False Positive
Charge Off (0)
960
1494
False Negative
True Positive
Paid Off (1)
627
13766
True label
Charge Off (0)
Paid Off (1)
Predicted label
12000
10000
8000
6000
4000
2000

In [39]:
```
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)
```

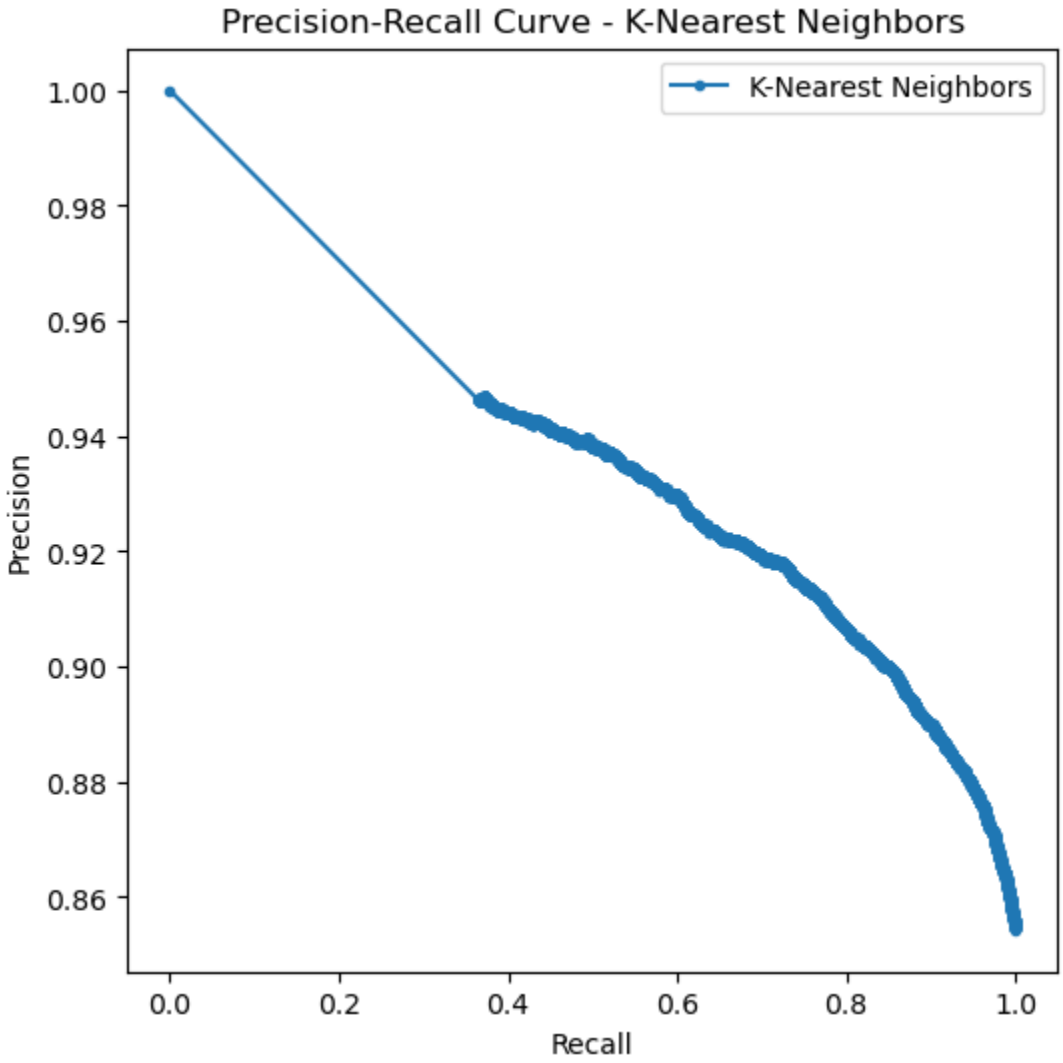


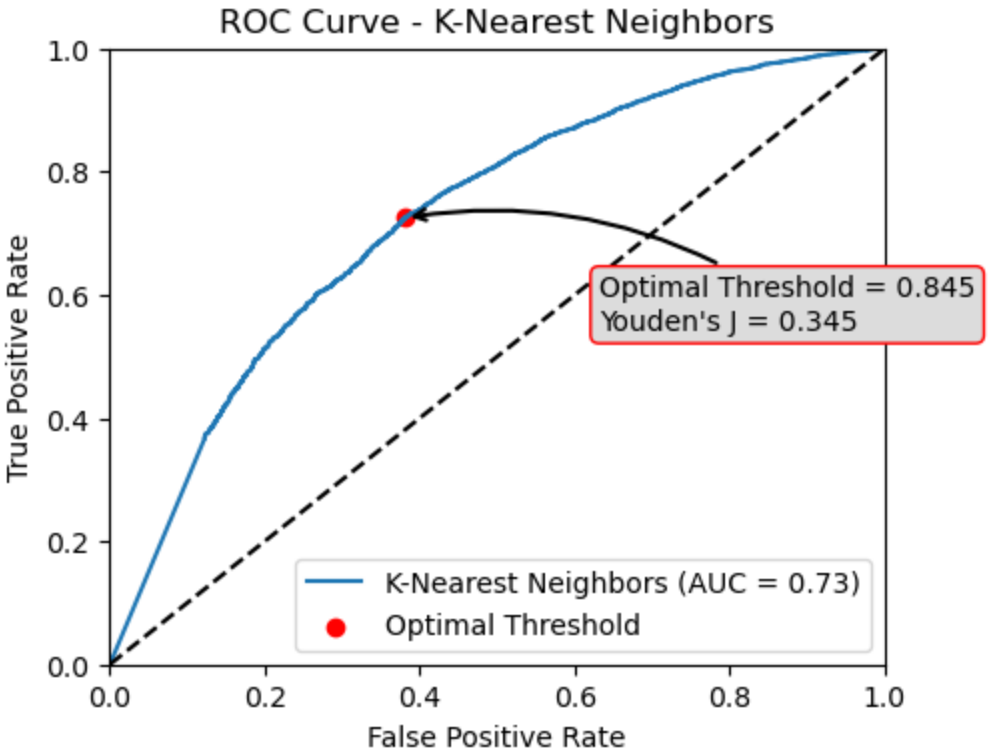

ROC Curve - Random Forest Classifier
True Positive Rate
False Positive Rate
Optimal Threshold = 0.593
Youden's J = 0.620
Random Forest Classifier (AUC = 0.88)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

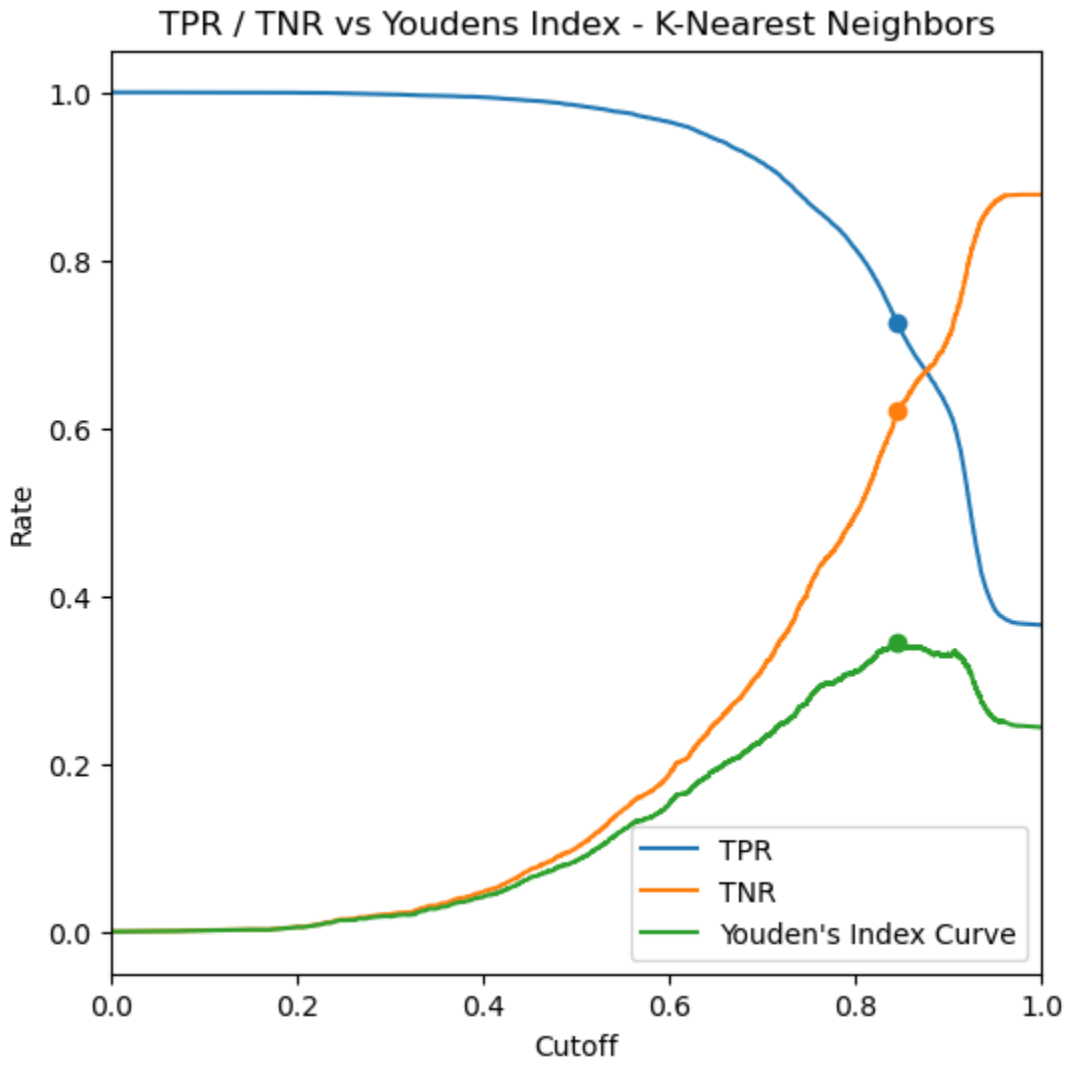

TPR / TNR vs Youdens Index - Random Forest Classifier
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

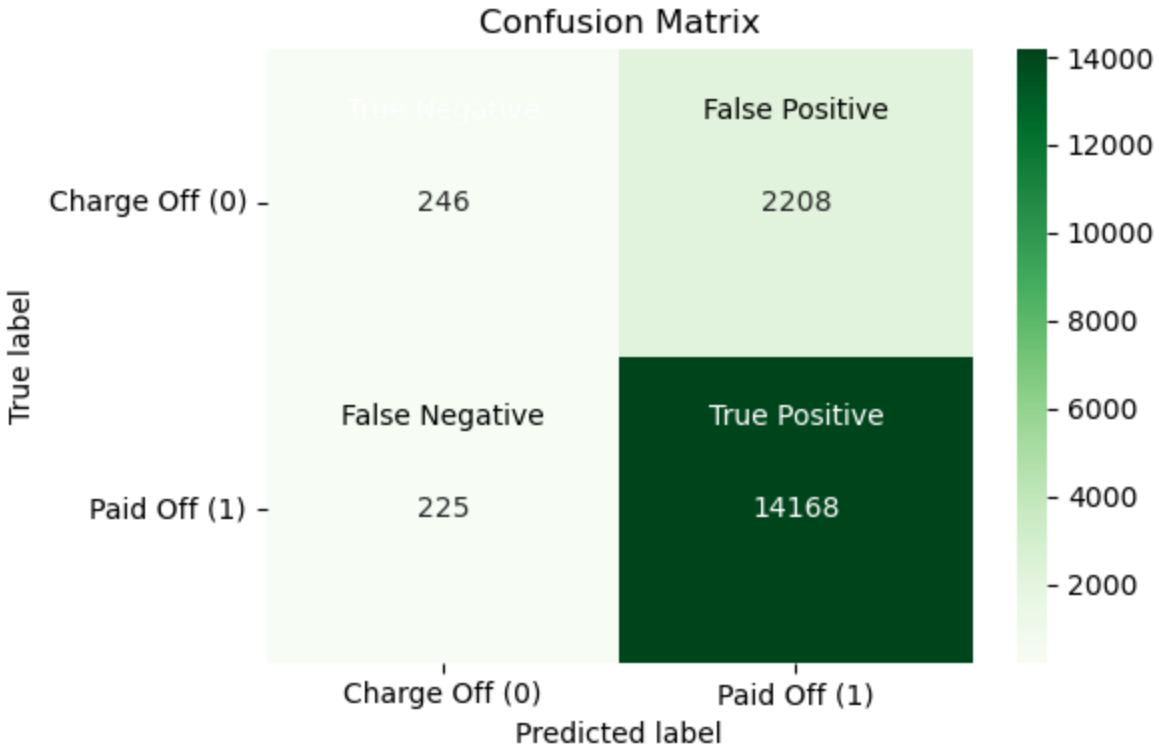


In [40]:
```
knn_model_name, knn_pr_curve_plot, knn_roc_curve_plot, knn_tpr_tnr_plot, knn_conf_matr
                                'K-Nearest Neighbors',
                                class_names,
                                datasets,
                                "full")

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)
```

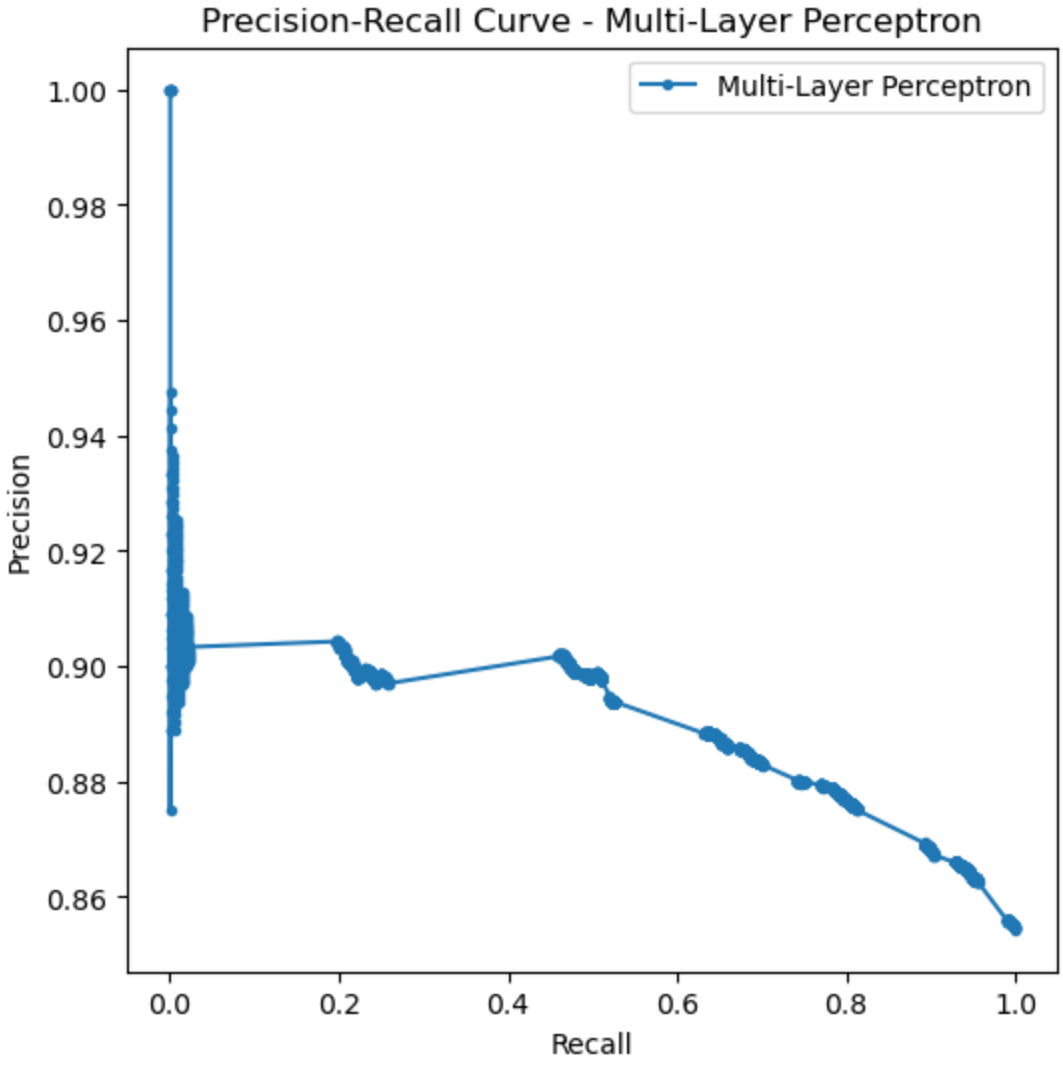

Precision-Recall Curve - K-Nearest Neighbors
K-Nearest Neighbors
Precision
Recall
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.0
0.2
0.4
0.6
0.8
1.0

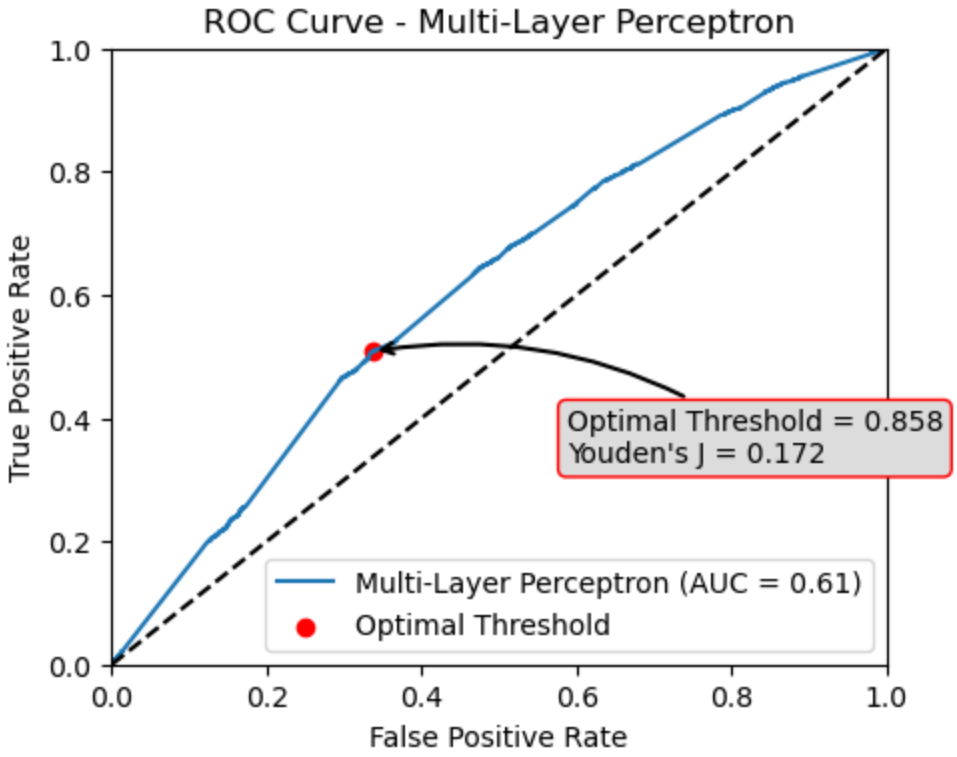

ROC Curve - K-Nearest Neighbors
True Positive Rate
False Positive Rate
Optimal Threshold = 0.845
Youden's J = 0.345
K-Nearest Neighbors (AUC = 0.73)
Optimal Threshold
0.0
0.2
0.4
0.6
0.8
1.0

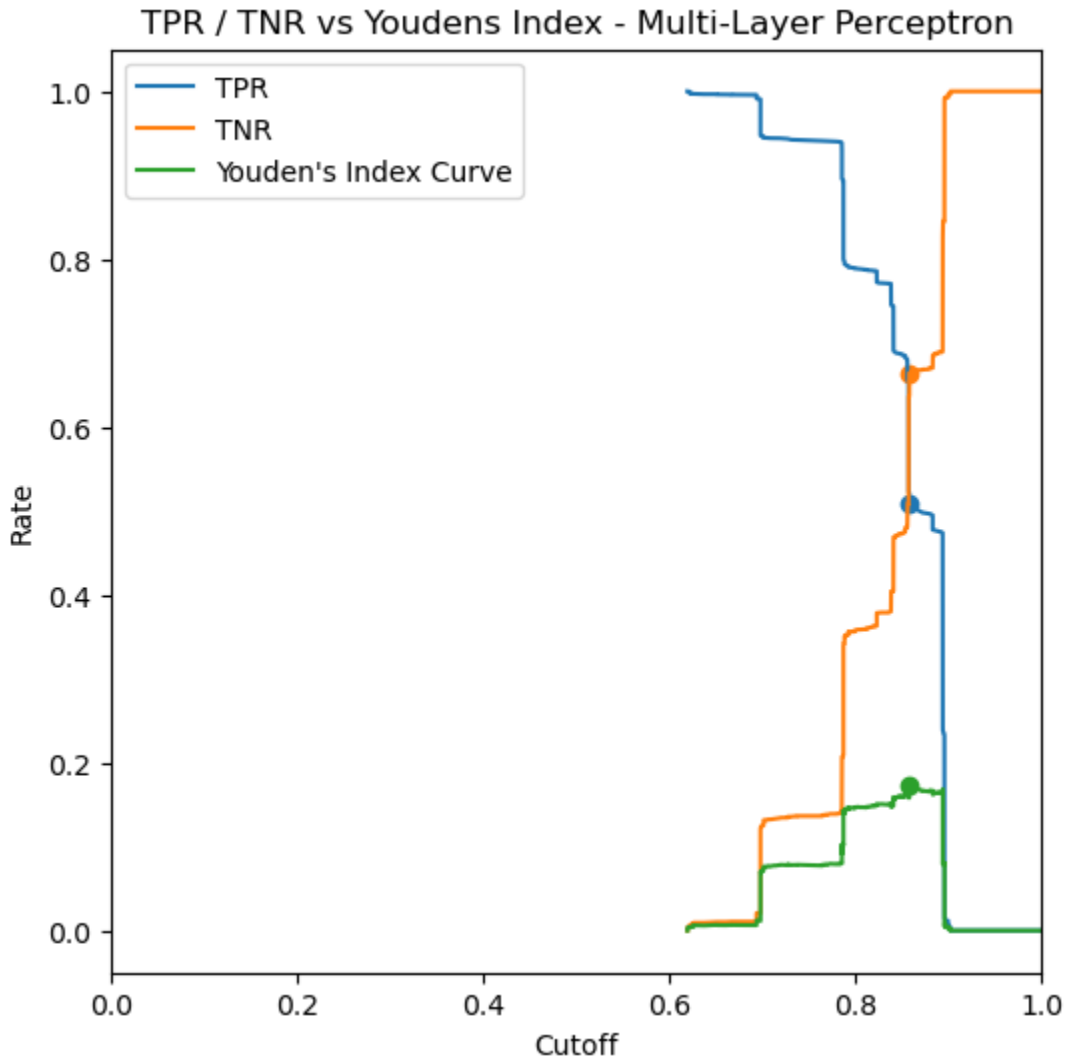

TPR / TNR vs Youdens Index - K-Nearest Neighbors
Rate
Cutoff
TPR
TNR
Youden's Index Curve
0.0
0.2
0.4
0.6
0.8
1.0

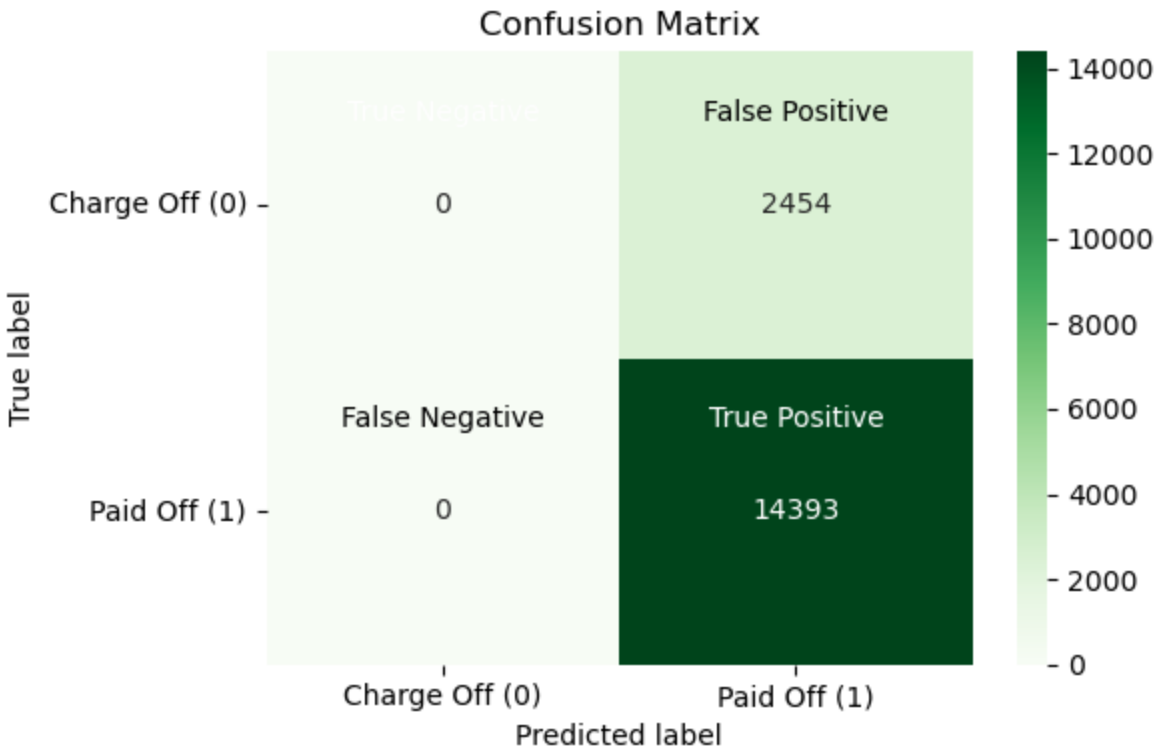

Confusion Matrix
True Negative
False Positive
246
2208
False Negative
True Positive
225
14168
Charge Off (0)
Paid Off (1)
True label
Predicted label
14000
12000
10000
8000
6000
4000
2000

In [41]:
```
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
True Positive Rate
False Positive Rate
Optimal Threshold = 0.858
Youden's J = 0.172
Multi-Layer Perceptron (AUC = 0.61)
Optimal Threshold
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



Section 6: Analysis Summary

In [42]:
```
results_frame
```

Out[42]:

	Model Name	Confusion Matrix	Accuracy	Precision	Recall	Specificity	F1 Score	Youden J Stat	Optimal P Threshold
0	Logistic Regression	[[2002, 452], [2874, 11519]]	0.802576	0.962242	0.800320	0.815811	0.873843	0.617403	0.562818
1	AdaBoost Classifier	[[2061, 393], [3184, 11209]]	0.787677	0.966127	0.778781	0.839853	0.862397	0.621958	0.498417
2	Decision Tree Classifier	[[960, 1494], [627, 13766]]	0.874102	0.902097	0.956437	0.391198	0.928473	0.603512	0.865613
3	Random Forest Classifier	[[1900, 554], [2544, 11849]]	0.816110	0.955333	0.823247	0.774246	0.884386	0.619539	0.593091
4	K-Nearest Neighbors	[[246, 2208], [225, 14168]]	0.855583	0.865169	0.984367	0.100244	0.920927	0.344740	0.844695
5	Multi-Layer Perceptron	[[0, 2454], [0, 14393]]	0.854336	0.854336	1.000000	0.000000	0.921447	0.172316	0.857649

Conclusion

Based on the given model results, let's analyze the strengths and weaknesses of each model, check for overfitting, and determine the best model for this problem:

1. Logistic Regression: • Strengths: Good accuracy (0.802576), high precision (0.962242), and a balanced trade-off between recall and specificity. • Weaknesses: Recall (0.800320) is relatively lower compared to some other models. • Overfitting: Not evident from the provided information. • Assessment: A good candidate for this problem but might not be the best.

2. AdaBoost Classifier: • Strengths: High precision (0.966127) and specificity (0.839853). • Weaknesses: Lower accuracy (0.787677) and recall (0.778781) compared to other models. • Overfitting: Not evident from the provided information. • Assessment: Not the best choice due to lower accuracy and recall.

3. Decision Tree Classifier: • Strengths: High recall (0.956437) and F1 score (0.928473). • Weaknesses: Low accuracy (0.874102), specificity (0.391198), and poor performance in identifying true negatives (high false positives). • Overfitting: Not evident from the provided information, but decision trees are prone to overfitting in general. • Assessment: Not suitable for this problem due to poor specificity.

4. Random Forest Classifier: • Strengths: Good accuracy (0.816110) and high precision (0.955333). • Weaknesses: Lower recall (0.823247) compared to some other models. • Overfitting: Not evident from the provided information. • Assessment: A good candidate for this problem, but there might be better options.

5. K-Nearest Neighbors: • Strengths: High recall (0.984367) and F1 score (0.920927). • Weaknesses: Poor accuracy (0.855583), low specificity (0.100244), and very high false positives. • Overfitting: Not evident from the provided information. • Assessment: Not suitable for this problem due to poor specificity.

6. Multi-Layer Perceptron: • Strengths: High recall (1.000000) and F1 score (0.921447). • Weaknesses: Poor specificity (0.000000), meaning the model is unable to correctly identify any true negatives. • Overfitting: Not evident from the provided information, but neural networks can be prone to overfitting. • Assessment: Not suitable for this problem due to poor specificity.

Based on the assessment, the Logistic Regression model appears to be the best option, as it has a good balance between accuracy, precision, recall, and specificity. The Random Forest Classifier is also a good candidate, but its recall value is lower than that of the Logistic Regression model.

The underlying model seems to be useful and applicable, as it demonstrates the value of machine learning in the lending industry by accurately predicting the likelihood of default. This can help both lenders and borrowers make better financial decisions. However, it's important to consider other factors, such as the model's interpretability and the potential impact on fairness, before deploying it in a real-world setting.