proj

June 6, 2024

1 Loan Tap Dataset

Project presented by - *Landi Francesco (2103413)* : he worked on cleaning the data and predicting the data. - *Niero Riccardo (2125392)* : he worked on commenting the report, graphs, data prediction and slides. - *Parolin Gianvittorio (2120003)* : he worked on the code and on gathering information and different techniques used.

Loan Tap is an online platform committed to delivering customized loan products to millennials. They innovate in an otherwise dull loan segment, to deliver instant, flexible loans on consumer friendly terms to salaried professionals and businessmen. The dataset contains nearly 400.000 observations each one for a single loan's borrower.

1.1 Loading data

[ ]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
from datetime import datetime
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.corpus import wordnet as wn

# Load data from the provided URL into a DataFrame
url = "https://d2beiqkhq929f0.cloudfront.net/public_assets/assets/000/003/549/
 ↪original/logistic_regression.csv"
df = pd.read_csv(url)

# Display the first few rows of the DataFrame
# pd.set_option('display.max_columns', None)
display(df.head())
```

```
   loan_amnt        term  int_rate  installment grade sub_grade  \
0    10000.0   36 months     11.44       329.48     B        B4
1     8000.0   36 months     11.99       265.68     B        B5
2    15600.0   36 months     10.49       506.97     B        B3
```

```
3     7200.0   36 months      6.49       220.65      A         A2
4    24375.0   60 months     17.27       609.33      C         C5

                 emp_title emp_length home_ownership  annual_inc  …  \
0                Marketing  10+ years           RENT    117000.0  …
1           Credit analyst    4 years       MORTGAGE     65000.0  …
2             Statistician   < 1 year           RENT     43057.0  …
3          Client Advocate    6 years           RENT     54000.0  …
4  Destiny Management Inc.    9 years       MORTGAGE     55000.0  …

  open_acc pub_rec revol_bal revol_util total_acc  initial_list_status  \
0     16.0     0.0   36369.0       41.8      25.0                    w
1     17.0     0.0   20131.0       53.3      27.0                    f
2     13.0     0.0   11987.0       92.2      26.0                    f
3      6.0     0.0    5472.0       21.5      13.0                    f
4     13.0     0.0   24584.0       69.8      43.0                    f

  application_type  mort_acc  pub_rec_bankruptcies  \
0       INDIVIDUAL       0.0                   0.0
1       INDIVIDUAL       3.0                   0.0
2       INDIVIDUAL       0.0                   0.0
3       INDIVIDUAL       0.0                   0.0
4       INDIVIDUAL       1.0                   0.0

                                              address
0     0174 Michelle Gateway\r\nMendozaberg, OK 22690
1  1076 Carney Fort Apt. 347\r\nLoganmouth, SD 05113
2  87025 Mark Dale Apt. 269\r\nNew Sabrina, WV 05113
3            823 Reid Ford\r\nDelacruzside, MA 00813
4             679 Luna Roads\r\nGreggshire, VA 11650

[5 rows x 27 columns]
```

[ ]:
```python
# Function to check values in the DataFrame (used also later)
def check_values(df):
    data = []
    for column in df.columns:
        data.append([
            column,
            df[column].dtype,
            df[column].isna().sum(),
            round(100 * (df[column].isna().sum() / len(df)), 2),
            df[column].nunique(),
            df[column].shape[0]
        ])
    return pd.DataFrame(columns=['Data features', 'Type', 'Null', '% Null',␣
↪'Unique values', 'Length'], data=data)
```

```
# Check the dataframe to see the characteristics
check_values(df)
```

```
[ ]:          Data features     Type   Null  % Null  Unique values  Length
    0              loan_amnt  float64      0    0.00           1397  396030
    1                   term   object      0    0.00              2  396030
    2               int_rate  float64      0    0.00            566  396030
    3            installment  float64      0    0.00          55706  396030
    4                  grade   object      0    0.00              7  396030
    5              sub_grade   object      0    0.00             35  396030
    6              emp_title   object  22927    5.79         173105  396030
    7             emp_length   object  18301    4.62             11  396030
    8         home_ownership   object      0    0.00              6  396030
    9             annual_inc  float64      0    0.00          27197  396030
    10   verification_status   object      0    0.00              3  396030
    11               issue_d   object      0    0.00            115  396030
    12           loan_status   object      0    0.00              2  396030
    13               purpose   object      0    0.00             14  396030
    14                 title   object   1756    0.44          48816  396030
    15                   dti  float64      0    0.00           4262  396030
    16      earliest_cr_line   object      0    0.00            684  396030
    17              open_acc  float64      0    0.00             61  396030
    18               pub_rec  float64      0    0.00             20  396030
    19             revol_bal  float64      0    0.00          55622  396030
    20            revol_util  float64    276    0.07           1226  396030
    21             total_acc  float64      0    0.00            118  396030
    22   initial_list_status   object      0    0.00              2  396030
    23      application_type   object      0    0.00              3  396030
    24              mort_acc  float64  37795    9.54             33  396030
    25  pub_rec_bankruptcies  float64    535    0.14              9  396030
    26               address   object      0    0.00         393700  396030
```

Data Dictionary

- loan_amnt: Listed amount of the loan applied for by the borrower.
- term: Number of payments on the loan (in months), can be 36 or 60.
- int_rate: Interest Rate on the loan.
- installment: Monthly payment owed by the borrower if the loan originates.
- grade: LoanTap assigned loan grade.
- sub_grade: LoanTap assigned loan subgrade.
- emp_title: Job title supplied by the borrower when applying for the loan.
- emp_length: Employment length in years.
- home_ownership: Home ownership status provided by the borrower.
- annual_inc: Self-reported annual income provided by the borrower.
- verification_status: Indicates if income was verified by LoanTap.
- issue_d: Month in which the loan was funded.

- loan_status: Current status of the loan (Target Variable).
- purpose: Category provided by the borrower for the loan request.
- title: Loan title provided by the borrower.
- dti: Ratio calculated using borrowers total monthly debt payments.
- earliest_cr_line: Month borrower's earliest reported credit line was opened.
- open_acc: Number of open credit lines in the borrower's credit file.
- pub_rec: Number of derogatory public records.
- revol_bal: The total amount of credit that a borrower has outstanding on revolving credit accounts.
- revol_util: Percentage of available credit that a borrower is currently using.
- total_acc: Total number of credit lines currently in the borrower's credit file.
- initial_list_status: The manner in which loans are initially presented for funding on the platform.
- application_type: Indicates whether the loan is an individual or joint application.
- mort_acc: Number of mortgage accounts.
- pub_rec_bankruptcies: Number of public record bankruptcies.
- address: Address of the individual.

1.2 Objective

Our goal is to find the value of Loan_Status by processing the other variables. Loan_Status can take on 2 values: "fully paid" or "charged off". A debt is canceled ("charged off") when it cannot be repaid and is written into the losses of the balance sheet. At a utility level, it would be useful for the company Loan_tap to understand a priori whether a person will more likely or not to pay his debt.

1.3 Handling missing values

The missing percentage of these columns is less then 1%, so we decided to fill them with the median value of the columns.

The other column that has NaN values is *mort_acc*, the percentage here is around 10%. For this reason, and because we have other columns representing the total number of credit lines, we will drop this column because it can't be filled with median value.

[ ]:
```
# Fill missing values for 'pub_rec_bankruptcies' with the median
median_pub_rec_bankruptcies = df['pub_rec_bankruptcies'].median()
df['pub_rec_bankruptcies'].fillna(median_pub_rec_bankruptcies, inplace=True)

# Handle remaining missing values in 'revol_util' with the median
median_revol_util = df['revol_util'].median()
df['revol_util'].fillna(median_revol_util, inplace=True)
```

1.3.1 Feature Engineering

[ ]:
```
# Create a copy of the DataFrame to avoid modifying the original data
df_fe = df.copy()
```

```python
# Initialize Porter Stemmer
stemmer = PorterStemmer()

# Downlod 'wordnet' database to find common job titles
nltk.download('wordnet')

# Function to find common job titles using WordNet
def find_job_titles():
    job_titles = set()
    for synset in wn.all_synsets(wn.NOUN):
        for lemma in synset.lemmas():
            if 'job' in synset.definition() or 'profession' in synset.
 ↪definition():
                job_titles.add(lemma.name().capitalize())
    return job_titles

common_job_titles = find_job_titles()

# Function to extract job title, if not able to extract use the original␣
 ↪emp_title
def extract_job_title(title):
    if pd.isna(title):
        return 'Unknown'  # Return 'Unknown' if title is NaN (missing value)
    if not isinstance(title, str):
        return 'Unknown'  # Return 'Unknown' if title is not a string
    words = title.capitalize().split()
    for word in words:
        if word == 'Rn' or word == 'Nurse' : # Trying to aggregate the work␣
 ↪classes that we found out are the same
          return 'Registered nurse'
        if word in common_job_titles:
          return word
    return title.capitalize()  # Return the original emp_title if no match found

# Apply function to extract job title and delete old variable
df_fe['job_title'] = df_fe['emp_title'].apply(extract_job_title).
 ↪astype('category')

# Remap 'term' as Short (36 months) and Long (60 months), and change its type␣
 ↪to category
df_fe['term'] = df_fe['term'].map({' 36 months': 'Short', ' 60 months': 'Long'})
df_fe['term'] = df_fe['term'].astype('category')

# Remap 'grade' and change type to category. Drop 'sub_grade' to simplify the␣
 ↪dataset
category_order = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

```
df_fe['grade'] = pd.Categorical(df_fe['grade'], categories=category_order,␣
↪ordered=True)

# Map 'emp_length' to categorical values and handle missing values
emp_length_mapping = {
    '< 1 year': '0-3 years',
    '1 year': '0-3 years',
    '2 years': '0-3 years',
    '3 years': '0-3 years',
    '4 years': '4-6 years',
    '5 years': '4-6 years',
    '6 years': '4-6 years',
    '7 years': '7-9 years',
    '8 years': '7-9 years',
    '9 years': '7-9 years',
    '10+ years': '10+ years',
    np.nan: 'Unknown'  # Handle missing values
}
df_fe['emp_length'] = df_fe['emp_length'].map(emp_length_mapping)
df_fe['emp_length'] = pd.Categorical(df_fe['emp_length'], categories=['0-3␣
↪years', '4-6 years', '7-9 years', '10+ years', 'Unknown'], ordered=True)

# Get the current year
current_year = datetime.now().year

# Split 'issue_d' column into separate columns for month and year, and␣
↪categorize
df_fe['month_issue'] = df_fe['issue_d'].str.split('-').str[0].astype('category')
df_fe['year_since_issue'] = current_year - pd.to_datetime(df_fe['issue_d'].str.
↪split('-').str[1]).dt.year

# Rewrite 'loan_status' to have specific categories
#loan_status_mapping = ['Fully Paid', 'Charged Off']
#df_fe['loan_status'] = df_fe['loan_status'].astype('category')

# Convert 'loan_status' column to 'category' data type
df_fe['loan_status'] = df_fe['loan_status'].astype('category')

# Map 'Fully Paid' as 0 and 'Charged Off' as 1
df_fe['loan_status'] = df_fe['loan_status'].cat.rename_categories({
    'Fully Paid': 0,
    'Charged Off': 1
})

# Categorize 'initial_list_status'
df_fe['initial_list_status'] =df_fe['initial_list_status'].str.replace("w",␣
↪"Whole").str.replace("f", "Fractional").astype('category')
```

```
# Categorize 'application_type'
# "Direct Pay" is a term that may refer to a specific type of loan repayment␣
↪arrangement.
# In this context, it could indicate that loan payments are made directly from␣
↪a designated source, such as an employer or government agency, rather than␣
↪from the borrower's personal accounts.
df_fe['application_type'] = df_fe['application_type'].str.capitalize().str.
↪replace('_', ' ').astype('category')

# Extract year from 'earliest_cr_line' and drop the original column
df_fe['years_since_first_credit'] = current_year - pd.
↪to_datetime(df_fe['earliest_cr_line']).dt.year

# Convert other categorical variables
df_fe['verification_status'] = df_fe['verification_status'].astype('category')
df_fe['home_ownership'] = df_fe['home_ownership'].str.capitalize().
↪astype('category')
# Map 'home_ownership' to categorical values and handle missing values
home_ownership_mapping = {
    'Any': 'Other',
    'None': 'Other',
    'Rent': 'Rent',
    'Own': 'Own',
    'Mortgage': 'Mortgage',
    'Other': 'Other',
    np.nan: 'Other'  # Handle missing values
}
df_fe['home_ownership'] = df_fe['home_ownership'].map(home_ownership_mapping)
df_fe['purpose'] = df_fe['purpose'].str.capitalize().str.replace('_', ' ').
↪astype('category')

# Convert some float columns to integer because they cannot be floats
columns_to_convert = ['pub_rec', 'total_acc', 'pub_rec_bankruptcies',␣
↪'open_acc']
df_fe[columns_to_convert] = df_fe[columns_to_convert].astype('int32')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data…
<ipython-input-4-25eacaaad1d9>:93: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure
parsing is consistent and as-expected, please specify a format.
  df_fe['years_since_first_credit'] = current_year -
pd.to_datetime(df_fe['earliest_cr_line']).dt.year
```

Flag columns for *pub_rec* and *open_acc*, this will be a dummy which will be 1 if the columns have 1 or more entries 0 otherwise.

Introduction of ratios: - Loan-to-Income Ratio: Elevated values imply a substantial loan compared to income, signaling increased financial strain. - Revolving Balance to Income Ratio: Increased values signify a considerable proportion of income allocated to revolving debt, indicating financial pressure. - Loan Amount per Open Account: Elevated values indicate large loan amounts spread thinly across accounts, indicating heightened risk exposure. - Total Accounts per Year: High values indicate a substantial number of accounts opened within a short period, highlighting potential risk factors.

```
[ ]: # Flag columns
     flag_columns = ['pub_rec', 'open_acc']

     for col in flag_columns:
         df_fe[f'{col}_flag'] = df_fe[col].apply(lambda x: True if x >= 1 else False)

     df_ratio = df_fe.copy()

     # Calculate Loan-to-Income Ratio
     df_ratio['loan_to_income_ratio'] = df_fe['loan_amnt'] / df_fe['annual_inc']

     # Calculate Revolving Balance to Income Ratio
     df_ratio['revol_bal_to_income_ratio'] = df_fe['revol_bal'] / df_fe['annual_inc']

     # Calculate Loan Amount per Open Account
     df_ratio['loan_amnt_per_open_acc'] = df_fe['loan_amnt'] / df_fe['open_acc']

     # Calculate Installment to Loan Amount Ratio
     df_ratio['installment_to_loan_amnt_ratio'] = df_fe['installment'] /␣
      ↪df_fe['loan_amnt']
```

Deleting the original variables that are used to generate these ratios and keeping only the ratios can simplify the model, but it comes with both advantages and disadvantages.

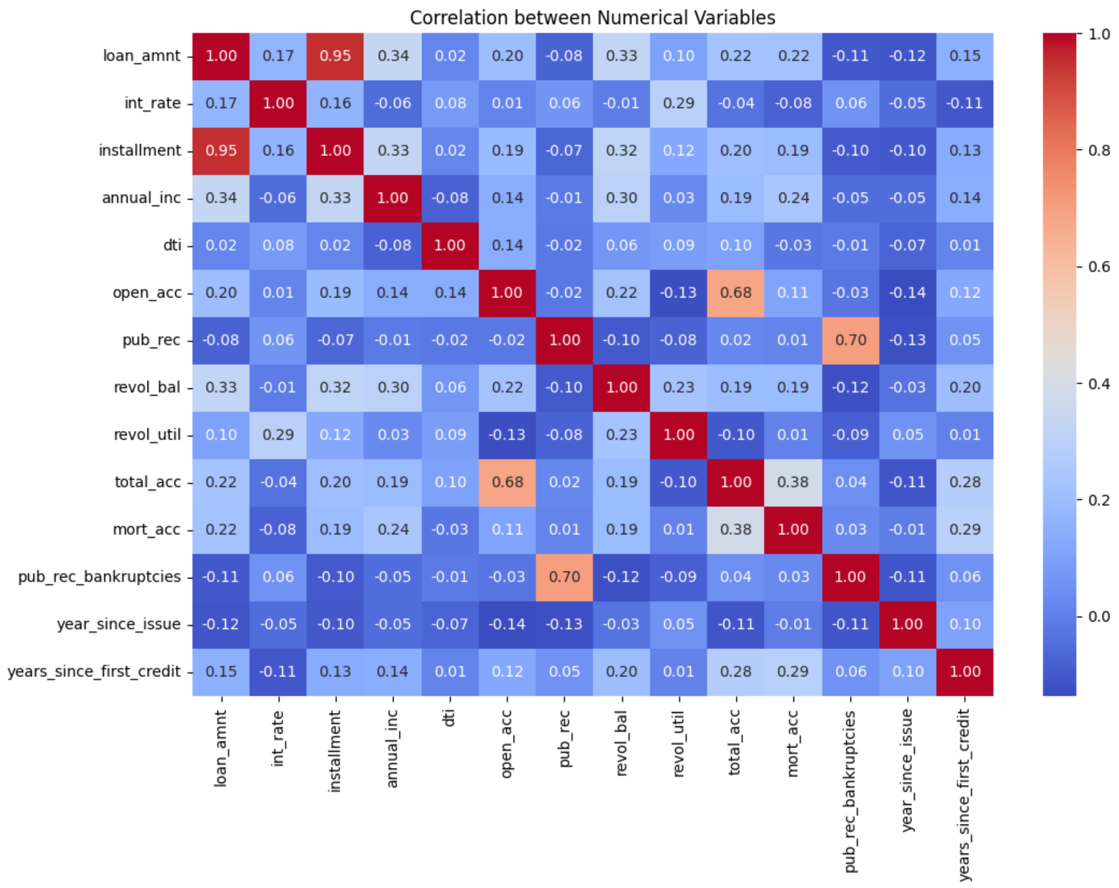
Advantages:

- Simplicity: Reduces the number of features, making the model simpler and potentially easier to interpret. Simplified models can be less prone to overfitting, especially if the original dataset is large and complex.
- Relevance: Ratios can capture the relative importance of different variables, potentially providing more direct insights into financial health and risk.

Disadvantages: - Loss of Granularity: Original variables provide detailed information that might be lost when only ratios are kept. Important nuances in the data might be overlooked, potentially reducing model accuracy. - Multicollinearity: Some of the original variables might be highly correlated, which ratios can help mitigate. However, removing all original variables means you lose the ability to identify and address multicollinearity effectively. - Interpretability: While ratios can simplify interpretation, they can also obscure the underlying reasons for a prediction if detailed information is needed. Regulatory or business requirements might necessitate keeping some original variables for transparency and compliance.

Correlation matrix

```
[ ]: corr_matrix = df_fe.select_dtypes(include=['number']).corr()

     # Plot the correlation matrix
     plt.figure(figsize=(12, 8))  # Set the figure size
     sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")  # Create a heatmap with annotations
     plt.title("Correlation between Numerical Variables")  # Set the title of the plot
     plt.show()  # Display the plot
```



We can see 3 major (positive) correlations: - the one between loan_amount and installment: it makes sense because they are proportional measures, the first is about the size of the loan and the second is about the size of the repayment that the borrower owes. - the one between open_acc and total_acc: it makes sense because they're about the same variable. - the one between pub-rec and pub_rec_bankruptcies: it makes sense because they're about the same variables, indicating that puc_rec are more common in case of default.

Drop raw and useless/similar columns to avoid multicollinearity

Reasons

- sub_grade: it has 35 unique values compared to 7 of grade, the trade-off this time goes for simplicity and not granularity
- emp_title: it has been substituted by *job_title*
- issue_d: it has been splitted in *month_issue* and *year_since_issue*
- title: it's very similar to *purpose* but prone to errors because inserted "by hand" by the client
- earliest_cr_line: transformed to *years_since_first_credit*
- address: dropped because anonymized and we have no information about this column
- mort_acc: dropped because it has almost 10% of NaN values, it's also a subset of *total_acc* which we decided to use because it has no missing values
- open_acc, pub_rec, pub_rec_bankruptcies: we substitute them with flag columns, we don't care about how much are they but just if they have open accounts and public records (not only bankruptcies)
- installment: higly (0.95%) correlated to *loan_amt*

```
[ ]: columns_to_drop = ['sub_grade', 'emp_title', 'issue_d', 'title',␣
      ↪'earliest_cr_line', 'address', 'mort_acc', 'open_acc', 'pub_rec',␣
      ↪'pub_rec_bankruptcies', 'installment']
     df_fe.drop(columns_to_drop, axis=1, inplace=True)
```

The new dataframe contains these columns:

```
[ ]: check_values(df_fe)
```

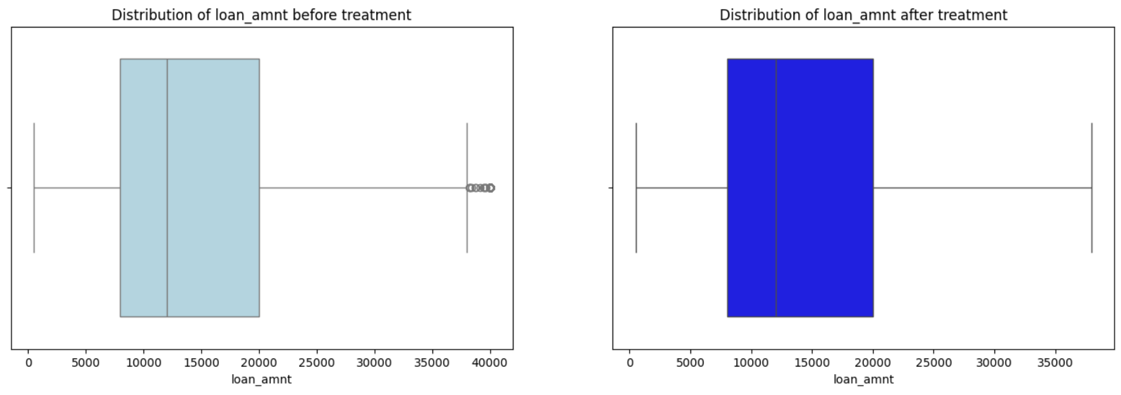
```
[ ]:            Data features      Type  Null  % Null  Unique values  Length
     0                loan_amnt   float64     0     0.0           1397  396030
     1                     term  category     0     0.0              2  396030
     2                 int_rate   float64     0     0.0            566  396030
     3                    grade  category     0     0.0              7  396030
     4               emp_length  category     0     0.0              5  396030
     5           home_ownership    object     0     0.0              4  396030
     6               annual_inc   float64     0     0.0          27197  396030
     7      verification_status  category     0     0.0              3  396030
     8              loan_status  category     0     0.0              2  396030
     9                  purpose  category     0     0.0             14  396030
     10                     dti   float64     0     0.0           4262  396030
     11               revol_bal   float64     0     0.0          55622  396030
     12              revol_util   float64     0     0.0           1226  396030
     13               total_acc     int32     0     0.0            118  396030
     14     initial_list_status  category     0     0.0              2  396030
     15        application_type  category     0     0.0              3  396030
     16               job_title  category     0     0.0         152097  396030
     17             month_issue  category     0     0.0             12  396030
     18        year_since_issue     int32     0     0.0             10  396030
     19  years_since_first_credit     int32     0     0.0             65  396030
     20            pub_rec_flag      bool     0     0.0              2  396030
     21           open_acc_flag      bool     0     0.0              2  396030
```

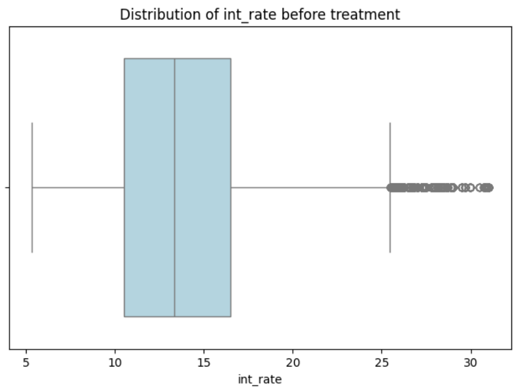
1.4 EDA with/without outliers

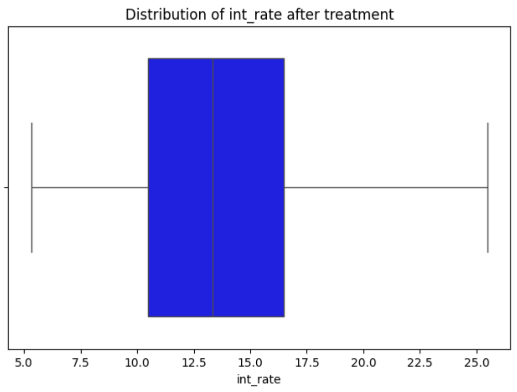
We decided to remove the outliers from our quantitative variables. Outliers are data points that significantly differ from the rest of the data. Outliers can affect statistical analyses and modeling, so identifying and handling them appropriately is important because they could bring too much noise in our analysis.

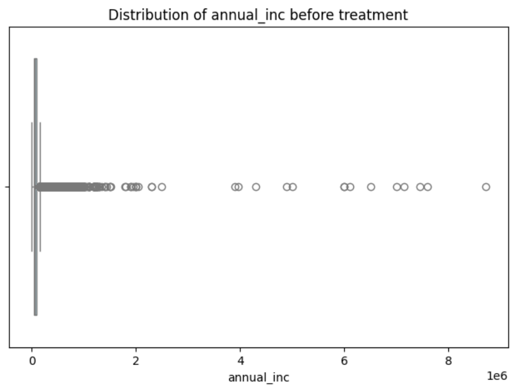
```
[ ]: # Function to cut outliers based on IQR method
def cut_outliers(df):
    treated_df = df.copy()
    for col in treated_df.select_dtypes(include=['number']).columns:
        q1 = treated_df[col].quantile(0.25)
        q3 = treated_df[col].quantile(0.75)
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr
        treated_df[col] = treated_df[col].clip(lower_bound, upper_bound)
    return treated_df

df_fe_robust = cut_outliers(df_fe)
```

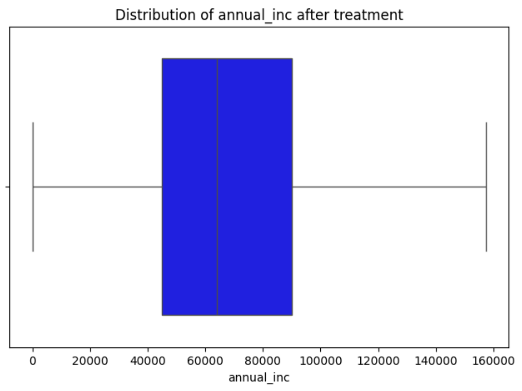
```
[ ]: # Plot the distribution of variables before and after outlier treatment
for col in df_fe.select_dtypes(include=['number']).columns:
    plt.figure(figsize=(17, 5))
    plt.subplot(1, 2, 1)
    sns.boxplot(x=df_fe[col], color="lightblue")
    plt.title(f'Distribution of {col} before treatment')
    plt.subplot(1, 2, 2)
    sns.boxplot(x=df_fe_robust[col], color="blue")
    plt.title(f'Distribution of {col} after treatment')
    plt.show()
```

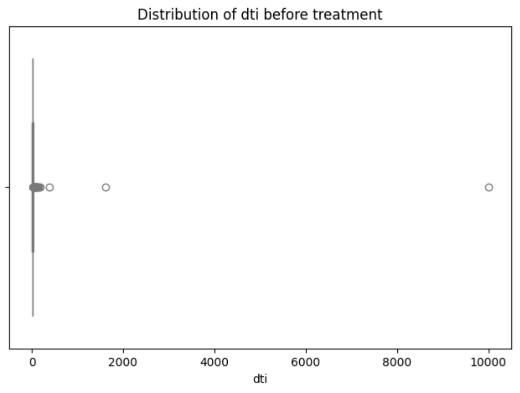


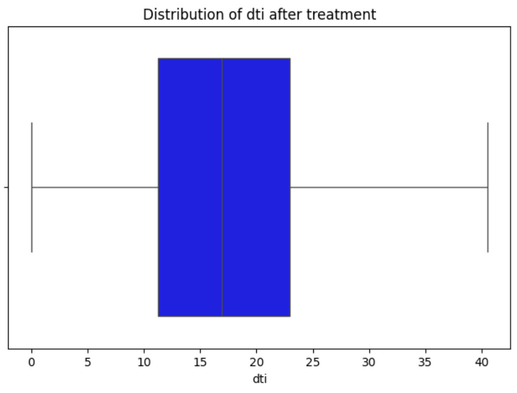

Distribution of int_rate before treatment
5
10
15
20
25
30
int_rate

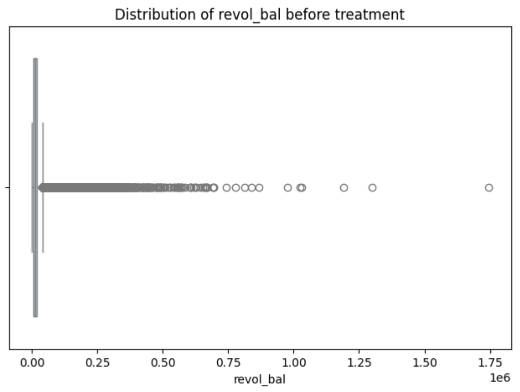

Distribution of int_rate after treatment
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
int_rate

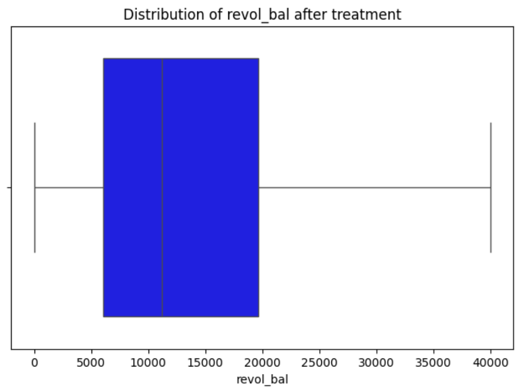

Distribution of annual_inc before treatment
0
2
4
6
8
1e6
annual_inc

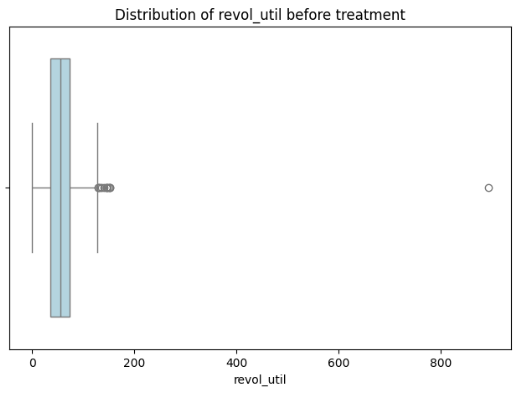

Distribution of annual_inc after treatment
0
20000
40000
60000
80000
100000
120000
140000
160000
annual_inc

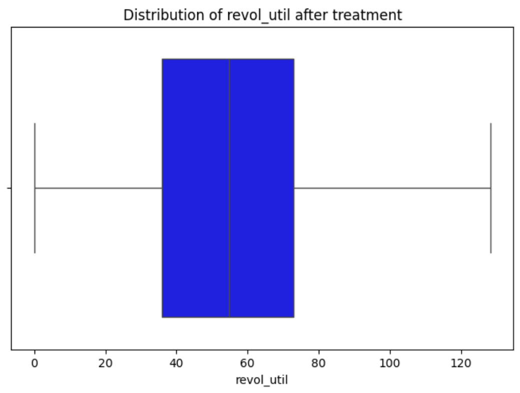

Distribution of dti before treatment
0
2000
4000
6000
8000
10000
dti

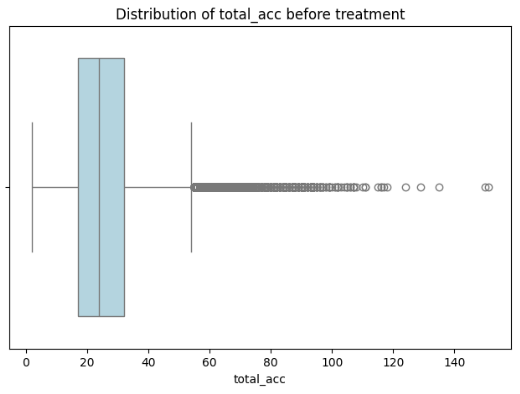

Distribution of dti after treatment
0
5
10
15
20
25
30
35
40
dti

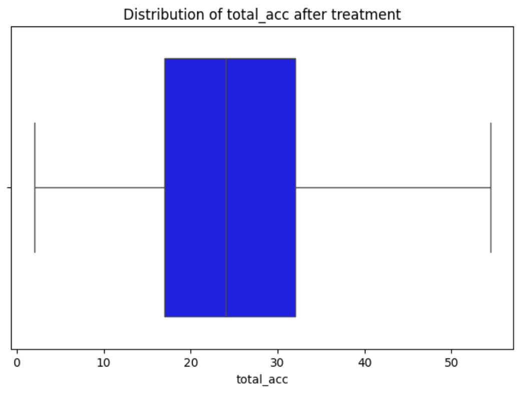

Distribution of revol_bal before treatment
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
1e6
revol_bal

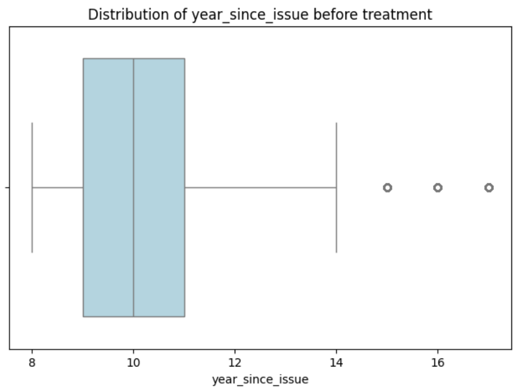

Distribution of revol_bal after treatment
0
5000
10000
15000
20000
25000
30000
35000
40000
revol_bal

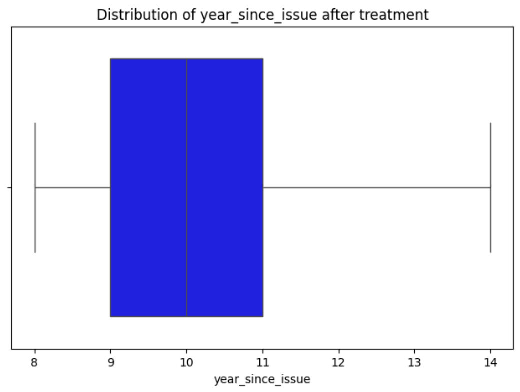

Distribution of revol_util before treatment
0
200
400
600
800
revol_util

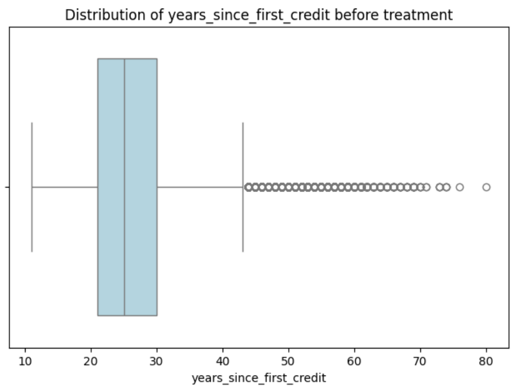

Distribution of revol_util after treatment
0
20
40
60
80
100
120
revol_util

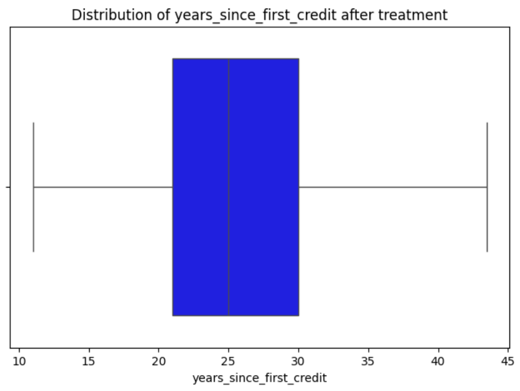

Distribution of total_acc before treatment
0
20
40
60
80
100
120
140
total_acc


Distribution of total_acc after treatment
0
10
20
30
40
50
total_acc







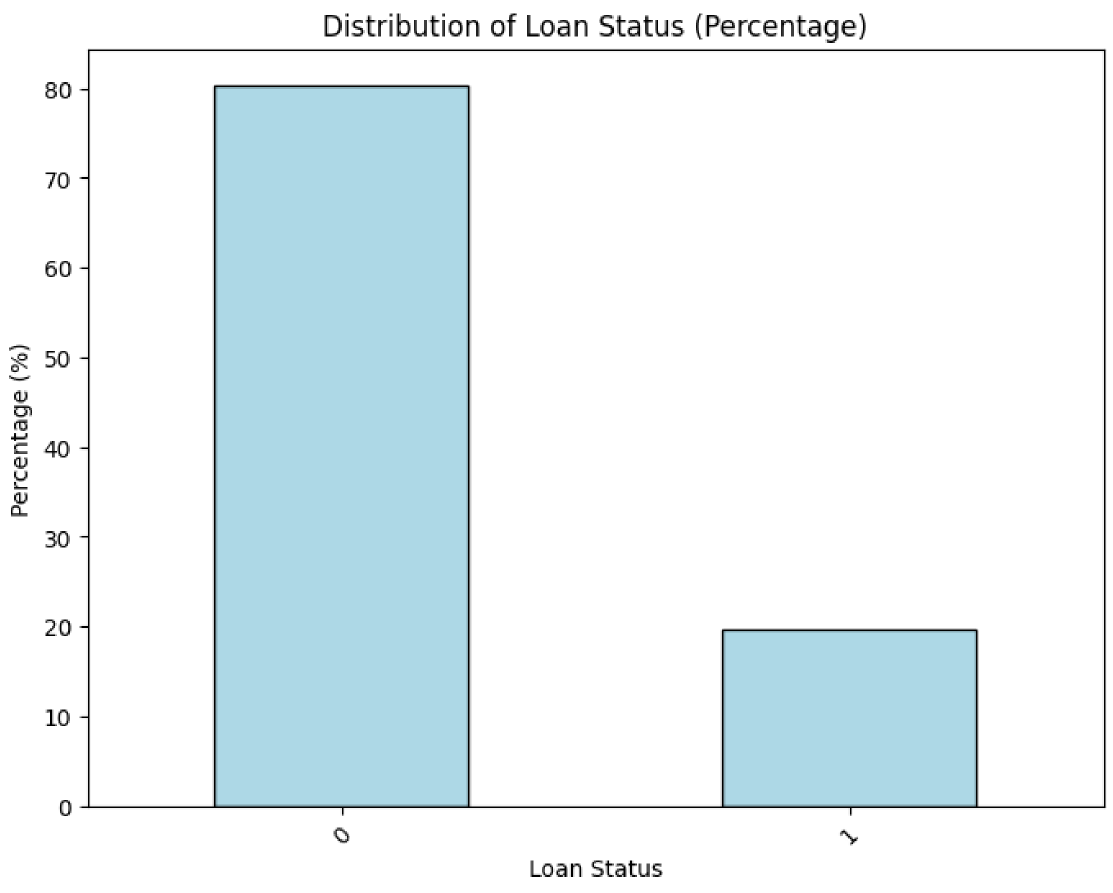


We decided to remove the outliers from our quantitative variables in order to see if they influence or not the prediction.

After removing the outliers and it's clearly seeable than the distribution after the distribution is much less compressed making the data more interpretable.

Distribution of the class that is going to be predicted (loan_status)

```
[ ]: # Count occurrences of each loan status
loan_status_counts = df_fe['loan_status'].value_counts()

# Calculate percentages
loan_status_percentages = (loan_status_counts / loan_status_counts.sum()) * 100

# Create the bar plot
plt.figure(figsize=(8, 6))
loan_status_percentages.plot(kind='bar', color='lightblue', edgecolor='black')

# Add title and labels
plt.title('Distribution of Loan Status (Percentage)')
plt.xlabel('Loan Status')
plt.ylabel('Percentage (%)')

# Show the plot
plt.xticks(rotation=45)
plt.show()
```

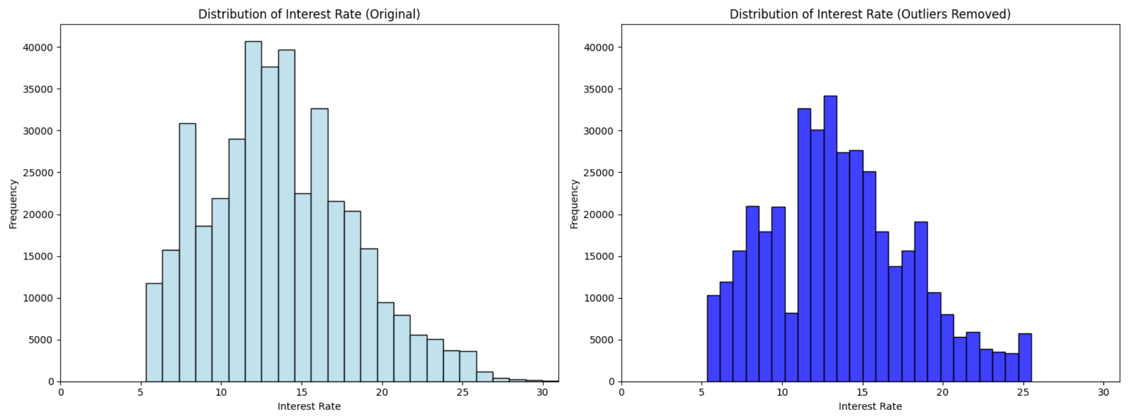


As expected, the prediction variable is not balanced at all. 20% are 1 (Charged off/Default), 80% are 0 (Fully Paid/Non-default). This is ok for a company which lends loans because a greater number of "Charged off" will lead the bank to financial troubles.

Distribution Plot for Interest Rate

```
[ ]: # Set up the figure with subplots
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Determine the limits for the x axes based on the combined data, starting from␣
↪0
x_limits = [0, max(df_fe['int_rate'].max(), df_fe_robust['int_rate'].max())]
```

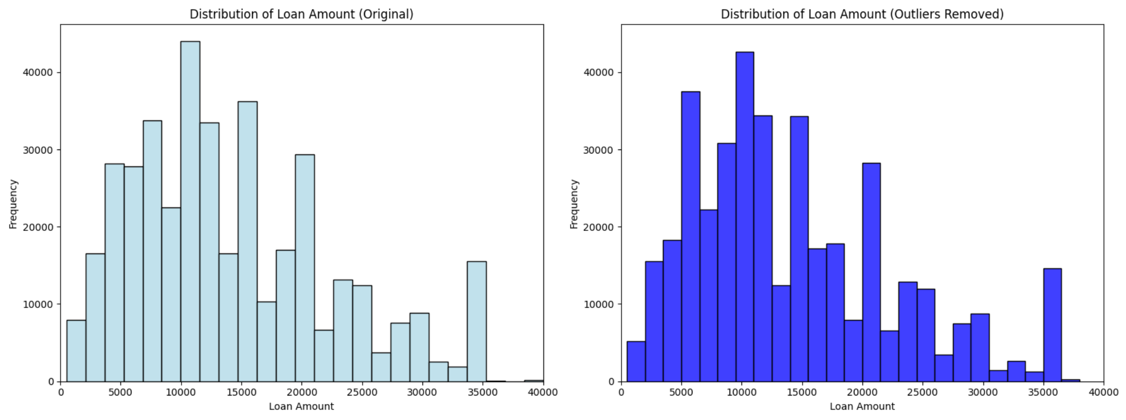
```
# Plot for original DataFrame df_fe
hist1 = sns.histplot(df_fe['int_rate'], bins=25, color='lightblue', ax=axes[0])
axes[0].set_title("Distribution of Interest Rate (Original)")
axes[0].set_xlabel("Interest Rate")
axes[0].set_ylabel("Frequency")
axes[0].set_xlim(x_limits)  # Set x-axis limits

# Capture the y-axis limit after plotting the first histogram
y_limits = axes[0].get_ylim()

# Plot for DataFrame after outlier removal df_fe_robust
hist2 = sns.histplot(df_fe_robust['int_rate'], bins=25, color='blue',␣
 ↪ax=axes[1])
axes[1].set_title("Distribution of Interest Rate (Outliers Removed)")
axes[1].set_xlabel("Interest Rate")
axes[1].set_ylabel("Frequency")
axes[1].set_xlim(x_limits)  # Set x-axis limits

# Update y-axis limit if the second histogram exceeds the first one's limits
y_limits = [0, max(y_limits[1], axes[1].get_ylim()[1])]
axes[0].set_ylim(y_limits)  # Apply updated y-limits to both plots
axes[1].set_ylim(y_limits)

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



Interest rate distribution is assumed as a normal distribution with a right skewness. This is a characteristic of a distribution in which the tail of the right side is longer or fatter than the left side. The left side of the distribution is truncated near 5 because there is no interest rate below this value. For the original distribution there is a positive peak near 7. The one without the outliers has a negative peak near 11 and a fatter right tail.

Distribution of loan_amnt

```
[ ]: # Set up the figure with subplots
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Determine the limits for the x axes based on the combined data, starting from␣
↪0
x_limits = [0, max(df_fe['loan_amnt'].max(), df_fe_robust['loan_amnt'].max())]

# Plot for original DataFrame df_fe
hist1 = sns.histplot(df_fe['loan_amnt'], bins=25, color='lightblue', ax=axes[0])
axes[0].set_title("Distribution of Loan Amount (Original)")
axes[0].set_xlabel("Loan Amount")
axes[0].set_ylabel("Frequency")
axes[0].set_xlim(x_limits)  # Set x-axis limits

# Capture the y-axis limit after plotting the first histogram
y_limits = axes[0].get_ylim()

# Plot for DataFrame after outlier removal df_fe_robust
hist2 = sns.histplot(df_fe_robust['loan_amnt'], bins=25, color='blue',␣
↪ax=axes[1])
axes[1].set_title("Distribution of Loan Amount (Outliers Removed)")
axes[1].set_xlabel("Loan Amount")
axes[1].set_ylabel("Frequency")
axes[1].set_xlim(x_limits)  # Set x-axis limits

# Update y-axis limit if the second histogram exceeds the first one's limits
y_limits = [0, max(y_limits[1], axes[1].get_ylim()[1])]
axes[0].set_ylim(y_limits)  # Apply updated y-limits to both plots
axes[1].set_ylim(y_limits)

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



Tables of term, home ownership and purpose by loan_status in percentage

```
[ ]: # Create crosstabs and normalize to get percentages, then round to 2 decimal␣
      ↪places
     term_table = (pd.crosstab(df_fe['term'], df_fe['loan_status'],␣
      ↪normalize='index') * 100).round(2)
     home_ownership_table = (pd.crosstab(df_fe['home_ownership'],␣
      ↪df_fe['loan_status'], normalize='index') * 100).round(2)
     purpose_table = (pd.crosstab(df_fe['purpose'], df_fe['loan_status'],␣
      ↪normalize='index') * 100).round(2)

     # Print the tables
     print("Term Table (Percentage):")
     print(term_table)
     print("\nHome Ownership Table (Percentage):")
     print(home_ownership_table)
     print("\nPurpose Table (Percentage):")
     print(purpose_table)
```

```
Term Table (Percentage):
loan_status      1      0
term
Long         31.94  68.06
Short        15.77  84.23

Home Ownership Table (Percentage):
loan_status         1      0
home_ownership
Mortgage        16.96  83.04
Other           15.75  84.25
Own             20.68  79.32
Rent            22.66  77.34

Purpose Table (Percentage):
loan_status             1      0
purpose
Car                 13.48  86.52
Credit card         16.71  83.29
Debt consolidation  20.74  79.26
Educational         16.34  83.66
Home improvement    17.01  82.99
House               19.72  80.28
Major purchase      16.47  83.53
Medical             21.71  78.29
Moving              23.48  76.52
Other               21.22  78.78
Renewable energy    23.40  76.60
Small business      29.45  70.55
```

```
Vacation            18.92  81.08
Wedding             12.09  87.91
```

In the “term” table we can see that the amount of *defult* in long loans is double the amount in short loans.

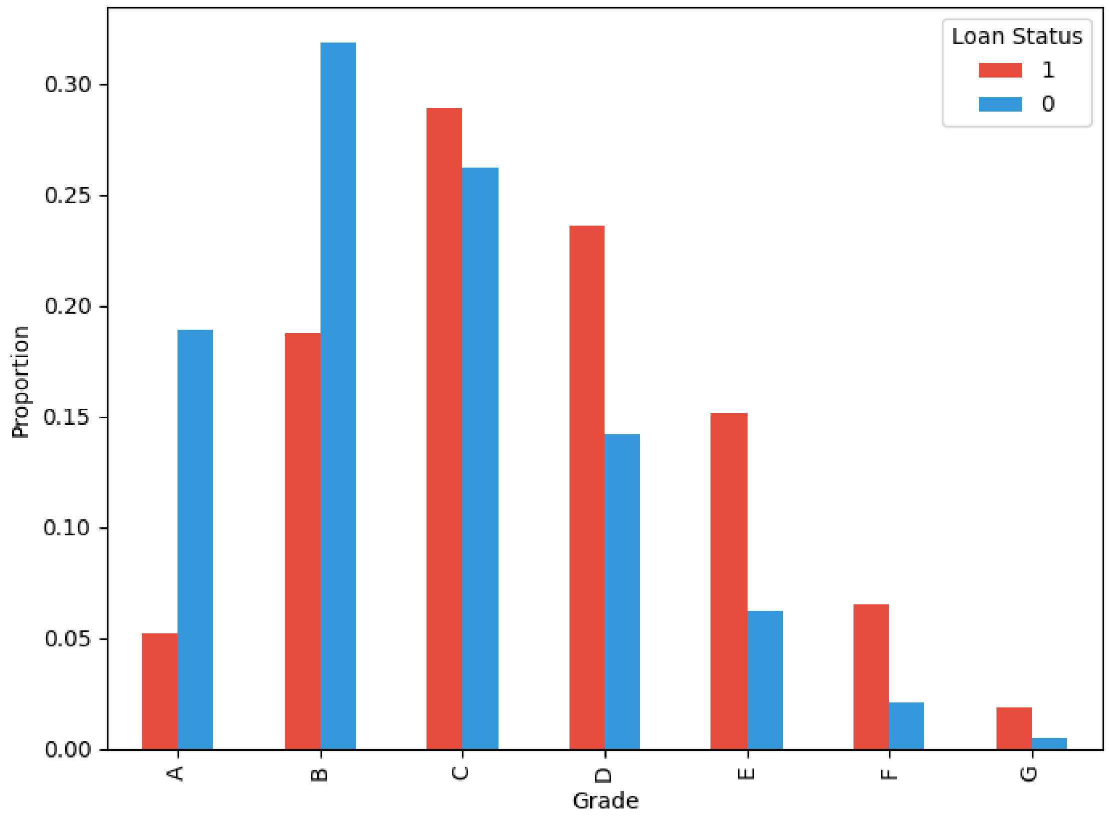
In the “home ownership” table we can see that the amount of *default* in every category is more or less the same but “rent” which is a bit higher.

In the “purpose” table the variable with the highest percentage of *default* is “small business” and the one with the lowest is “weddings”.

This make sense because the loan amount for weddings is usually low and can be repaid easily; on the other hand the loan amount for small businesses is usually higher and it’s possible for the business to not be able to meet the expectations of the initial investment, financed by the loan.

Proportion of loan status by grades

```
[ ]: # Create a new figure
     fig = plt.figure(figsize=(8, 6))

     # Create a new axis object
     ax = fig.add_subplot(1, 1, 1)

     # Set the color palette
     sns.set_palette(sns.color_palette(["#e74c3c", "#3498db"]))

     # Plot the data
     pd.crosstab(index=df_fe["grade"], columns=df_fe["loan_status"],␣
      ↪normalize="columns").plot(kind="bar", ax=ax)

     # Set the plot title, labels, and legend
     ax.set_xlabel("Grade")
     ax.set_ylabel("Proportion")
     _ = ax.legend(title='Loan Status')
```



For the highest grade loans (A), the proportion of non-defaulted loans (blue bars) is much higher than the proportion of defaulted loans (red bars). As the loan grade decreases (moving from A to G), the proportion of defaulted loans increases, and the proportion of non-defaulted loans decreases.

This trend suggests that lower-grade loans carry a higher risk of default compared to higher-grade loans. The graph clearly illustrates that as the loan grade deteriorates, the likelihood of a borrower defaulting on the loan increases substantially.

Percentage of fully paid or charged off by purpose of the loan

```
[ ]: round(pd.crosstab(index = df_fe["purpose"],
                 columns= df_fe["loan_status"],normalize= "index", margins =␣
      ↪True)*100,2)
```

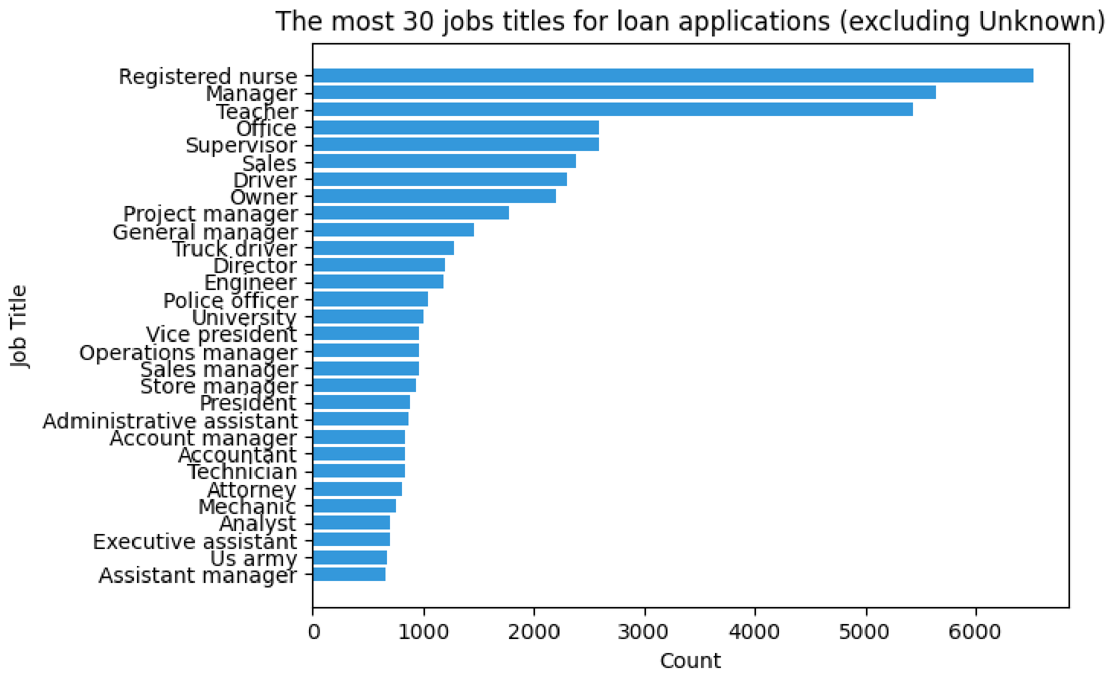
```
[ ]: loan_status            1      0
     purpose
     Car                13.48  86.52
     Credit card        16.71  83.29
     Debt consolidation 20.74  79.26
     Educational        16.34  83.66
     Home improvement   17.01  82.99
     House              19.72  80.28
```

```
Major purchase      16.47  83.53
Medical             21.71  78.29
Moving              23.48  76.52
Other               21.22  78.78
Renewable energy    23.40  76.60
Small business      29.45  70.55
Vacation            18.92  81.08
Wedding             12.09  87.91
All                 19.61  80.39
```

From the table of loans in base of the purposes , it's evident that loans for weddings and cars have the lowest default rates, at 12.09% and 13.48% respectively. On the other hand, small business loans exhibit the highest default rate, with 29.45%, followed by loans for renewable energy at 23.40% and for moving at 23.48%.However, it's worth noting that loans for home purchases have a default rate of 19.72%.

First 30 most frequent Job Titles (excluding unknown ones)

```
[ ]: plt.barh(df_fe['job_title'].value_counts()[1:31].index, df_fe['job_title'].
      ↪value_counts()[1:31], color="#3498db")
     plt.title("The most 30 jobs titles for loan applications (excluding Unknown)")
     plt.xlabel("Count")
     plt.ylabel("Job Title")
     plt.gca().invert_yaxis()  # Invert y-axis to have the highest count at the top
     plt.show()
```



The frequency graph on job_title shows the 30 most common jobs among our graphs. In the dataset, the most common occupation is registered nurse, followed by managers and teachers, each with a count ranging between 5000 and 6000. However, there is a considerable drop in frequency when we move below the fourth most common job, with a count below 3000. This position includes various roles such as office workers, supervisors, sales and drivers.

Percentage of available credit that a borrower is currently using grouped by status

```
[ ]: df_fe.groupby("loan_status")["revol_util"].median()
```

```
[ ]: loan_status
     1    59.3
     0    53.7
     Name: revol_util, dtype: float64
```

As expected if the person defaulted he/she was using more credit 59% vs 53%.

So, a higher amount of used loan may influence the default.

1.4.1 Possible pitfalls:

- Test Snooping: when substituting the missing values with the median computed using the entire values if you compute this average using the entire dataset instead of only the trading dataset, you introduce leakage.
- Selective Snooping: when filtering the data using "appropriate" techniques. You might "remove" outliers that are causing problems at testing time, cheating in the overall model performance.
- Biased Parameter Selection while this operation is legitimate, "observing different models performance" should be just that. While it is ok to observe how different models behave in the testing set, you should not explicitly pick conclusions. If you need to pick only one model, use a different holding partition If you only need to observe and compare different performances, the operation is legitimate.
- Base Rate Fallacy: the data analysis reveals a significant class imbalance within the 'loan_status' column, with the majority class "Fully Paid" comprising approximately 80.38% of the dataset, while the minority class "Charged Off" constitutes only about 19.62%. This imbalance heavily skews the model towards predicting the majority class, resulting in an overestimation of overall performance metrics such as accuracy and weighted averages. Consequently, the model struggles with detecting instances of the minority class, leading to extremely low values of precision, recall, and F1-score for class 1. This scenario exemplifies the Base Rate Fallacy, where the misleading interpretation of results due to class imbalance undermines the model's effectiveness in accurately classifying both classes.

Mitigation:

- Test/Selective Snooping: in the prediction we also build a model that contains the dataset without the feature engineering (so with outliers but without NaN values)

- To handle the Biased Parameter Selection: we implement different models such as Random Forest and Perceptron alongside Logistic Regression. We compare these models to determine which one performs better in terms of accuracy, precision, recall, F1 score, and other relevant metrics.

- To deal with Base Rate Fallacy we try to implement the logistic regression with the parameter “class_weight=‘balanced’ “ discussed later in the details. We also use the techniques SMOTE, in the Perceptron case, to oversample the minority class.

1.5 Prediction

Removal of column ‘job_title’:
However we decided to exclude the column “job_title” because it has too many distinct values resulting in high complexity that makes one-hot encoding computationally expensive.

One-hot encoding:
We use one-hot encoding, with the dummy variables, to handle categorical variables by transforming them into a numerical format that can be used in the predictions.

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import LogisticRegression, Perceptron
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import GridSearchCV,  train_test_split
```

[ ]:
```
# Create a copy of the DataFrame
df_encoded = df_fe_robust.copy()

# Drop the 'job_title' column due to its high cardinality which makes one-hot␣
 ↪encoding computationally expensive
df_encoded.drop("job_title", axis=1, inplace=True)

# Automatically identify categorical features for one-hot encoding, excluding␣
 ↪'loan_status'
categorical_features = df_encoded.select_dtypes(include=['object', 'category']).
 ↪columns.tolist()
if 'loan_status' in categorical_features:
    categorical_features.remove('loan_status')

# Perform one-hot encoding on the identified categorical features
df_encoded = pd.get_dummies(df_encoded, columns=categorical_features,␣
 ↪drop_first=True)

# Display the encoded rows DataFrame to verify
display(df_encoded.iloc[:, 10:].head())
```

```
   pub_rec_flag  open_acc_flag  term_Short  grade_B  grade_C  grade_D  \
0         False           True        True     True    False    False
1         False           True        True     True    False    False
2         False           True        True     True    False    False
3         False           True        True    False    False    False
4         False           True       False    False     True    False
```

```
   grade_E  grade_F  grade_G  emp_length_4-6 years  …  month_issue_Dec  \
0    False    False    False                 False  …            False
1    False    False    False                  True  …            False
2    False    False    False                 False  …            False
3    False    False    False                  True  …            False
4    False    False    False                 False  …            False

   month_issue_Feb  month_issue_Jan  month_issue_Jul  month_issue_Jun  \
0            False             True            False            False
1            False             True            False            False
2            False             True            False            False
3            False            False            False            False
4            False            False            False            False

   month_issue_Mar  month_issue_May  month_issue_Nov  month_issue_Oct  \
0            False            False            False            False
1            False            False            False            False
2            False            False            False            False
3            False            False             True            False
4            False            False            False            False

   month_issue_Sep
0            False
1            False
2            False
3            False
4            False

[5 rows x 45 columns]
```

We have splitted the data using the stratified method, we havethree sets that consist of training, validation and test. Dataset division was such that training took 80% while the remaining 20% was left for the purpose of testing. Then, out of this set aside for training, 25% was used for validation making sure that it represented 20% out of all the available data (25%= 80%). This ensures that the size of the validation set remains proportional to that of both the training set and test set.

We ensure by doing so that our validation is representative of the data distribution we can evaluate our model during its training reliably. In addition to this, we don't touch on our test dataset hence giving an unbiased estimation on how well our final model will perform against unseen data.

[ ]:
```python
# X contains all features except 'loan_status'
X = df_encoded.drop('loan_status', axis=1)

# y contains the target variable 'loan_status'
y = df_encoded['loan_status']

## Train-test split
```

```
# Splitting the dataset into training and testing sets
# Test size is 20% of the entire dataset
# Stratify=y ensures that the class distribution is preserved in both train and␣
 ↪test sets
# Random state ensures reproducibility of the split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪stratify=y, random_state=42)

# Validation set
# Further splitting the training set into training and validation sets
# Validation size is 25% of the training set, i.e., 20% of the entire dataset
# Stratify=y_train ensures that the class distribution is preserved in the␣
 ↪validation set
# Random state ensures reproducibility of the split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪25, stratify=y_train, random_state=42)

# Printing the shapes of the resulting datasets
print("Training set shape:", X_train.shape, y_train.shape)
print("Validation set shape:", X_val.shape, y_val.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (237618, 54) (237618,)
Validation set shape: (79206, 54) (79206,)
Test set shape: (79206, 54) (79206,)
```

1.5.1 Model 1: Logistic Regression

Advantages:
- Simplicity and interpretability. - Computationally efficient. - Well-understood and widely used.

Disadvantages: - Assumes linear relationship between features and log-odds of the outcome. - Sensitive to outliers.

[ ]:
```
# MODEL 1
m1 = LogisticRegression(random_state=123, max_iter=1000)
m1.fit(X_train, y_train)

# Predicting on the training set
y_hat_train_m1 = m1.predict(X_train)
train_accuracy_m1 = accuracy_score(y_train, y_hat_train_m1)

# Predicting on the validation set
y_hat_val_m1 = m1.predict(X_val)
val_accuracy_m1 = accuracy_score(y_val, y_hat_val_m1)

# Predicting on the test set
y_hat_test_m1 = m1.predict(X_test)
```

```
test_accuracy_m1 = accuracy_score(y_test, y_hat_test_m1)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m1 = round(train_accuracy_m1, 4)
val_accuracy_m1 = round(val_accuracy_m1, 4)
test_accuracy_m1 = round(test_accuracy_m1, 4)

print('Model 1: Logistic Regression\n')
print(f"Training Accuracy: {train_accuracy_m1}")
print(f"Validation Accuracy: {val_accuracy_m1}")
print(f"Test Accuracy: {test_accuracy_m1}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m1, digits=4))
```

```
Model 1: Logistic Regression

Training Accuracy: 0.8012
Validation Accuracy: 0.8004
Test Accuracy: 0.8013

Classification Report:
              precision    recall  f1-score   support

           0     0.8102    0.9830    0.8883     63671
           1     0.4474    0.0564    0.1002     15535

    accuracy                         0.8013     79206
   macro avg     0.6288    0.5197    0.4942     79206
weighted avg     0.7391    0.8013    0.7337     79206
```

- *Precision*: Precision is the ratio of true positive predictions to the total number of positive predictions made by the model. It measures the accuracy of positive predictions, indicating how many of the predicted positive cases are actually positive. High precision means that the model has fewer false positives.
- *Recall*: Recall, also known as sensitivity or true positive rate, is the ratio of true positive predictions to the total number of actual positive instances in the dataset. It measures the ability of the model to correctly identify positive instances, indicating how many of the actual positive cases are correctly predicted by the model. High recall means that the model captures a higher proportion of true positive cases.
- *F1 Score*: The F1 score is the harmonic mean of precision and recall. It provides a single metric that combines both precision and recall into a single value, balancing the trade-off between them. The F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It is a useful metric for evaluating models when there is an imbalance between the positive and negative classes, as it considers both false positives and false negatives.

Training Accuracy, Validation Accuracy and Test Accuracy always result very similar to each other. So for our analysis we take in consideration the Validation Accuracy.

Interpretation The model consistently achieves an 80% accuracy across, validation, and test sets, primarily due to its strong performance in classifying class 0 (the majority class). However, it struggles with class 1, exhibiting low precision (0.4474), recall (0.0564), and F1-score (0.1002).

This suggests a need for action to address class imbalance. The poor recall on class 1 highlights the necessity for strategies like resampling or using complex models to improve performance.

Next step is to find a better model using ElasticNet.

1.5.2 Model 2: ElasticNet

ElasticNet ElasticNet is a regularized regression technique that combines the properties of both Lasso (L1) and Ridge (L2) regression. It is particularly useful when dealing with datasets that have a large number of features or when there are correlations between the features.

[ ]:
```
# MODEL 2
elastic_net = ElasticNet()

# Define the hyperparameter grid
param_grid = {
    'alpha': [0.1, 0.5, 1.0],  # Regularization strength; controls the␣
 ↪magnitude of the penalty term
    'l1_ratio': [0.1, 0.5, 0.9],  # Mixing parameter; determines the balance␣
 ↪between L1 and L2 penalties
    'tol': [0.001, 0.0001, 0.00001]  # Tolerance for the optimization algorithm␣
 ↪to converge
}

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=elastic_net, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_alpha = grid_search.best_params_['alpha']
best_l1_ratio = grid_search.best_params_['l1_ratio']
best_tol = grid_search.best_params_['tol']

# Train the Elastic Net model with the best hyperparameters
m2 = ElasticNet(alpha=best_alpha, l1_ratio=best_l1_ratio, tol=best_tol)
m2.fit(X_train, y_train)
```

[ ]:
```
ElasticNet(alpha=0.1, l1_ratio=0.1, tol=0.001)
```

- *alpha*: This parameter is the regularization strength. It controls the penalty term's magnitude, where higher values of alpha result in stronger regularization. It helps prevent overfitting by penalizing large coefficients.

- *l1_ratio*: This parameter is the mixing parameter that determines the balance between L1 and L2 penalties in the regularization term. A value of 0 corresponds to Ridge (L2 penalty only), a value of 1 corresponds to Lasso (L1 penalty only), and values in between represent a mix of both penalties.
- *tol*: This parameter is the tolerance for the optimization algorithm to converge. It specifies the stopping criterion for the optimization process. The optimization stops when the difference in consecutive objective function values is less than tol.

[ ]:
```
# Threshold probabilities to obtain binary predictions
y_hat_train_m2 = np.where(m2.predict(X_train) >= 0.5, 1, 0)

train_accuracy_m2 = accuracy_score(y_train, y_hat_train_m2)

# Threshold probabilities to obtain binary predictions
y_hat_val_m2 = np.where(m2.predict(X_val) >= 0.5, 1, 0)

val_accuracy_m2 = accuracy_score(y_val, y_hat_val_m2)

# Threshold probabilities to obtain binary predictions
y_hat_test_m2 = np.where(m2.predict(X_test) >= 0.5, 1, 0)

test_accuracy_m2 = accuracy_score(y_test, y_hat_test_m2)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m2 = round(train_accuracy_m2, 4)
val_accuracy_m2 = round(val_accuracy_m2, 4)
test_accuracy_m2 = round(test_accuracy_m2, 4)

print(f"Training Accuracy: {train_accuracy_m2}")
print(f"Validation Accuracy: {val_accuracy_m2}")
print(f"Test Accuracy: {test_accuracy_m2}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m2, digits=4))
```

```
Training Accuracy: 0.8039
Validation Accuracy: 0.804
Test Accuracy: 0.8041

Classification Report:
              precision    recall  f1-score   support

           0     0.8046    0.9990    0.8913     63671
           1     0.5733    0.0055    0.0110     15535

    accuracy                         0.8041     79206
```

```
   macro avg     0.6890    0.5023    0.4511     79206
weighted avg     0.7592    0.8041    0.7186     79206
```

Interpretation The result of this is slightly better then *model 1*, accuracy is higher and also classification of the class 1 is a little small better (F1-score is higher).

Overall the model is not yet sufficient.

Next step is to try to restrict the model using just the significant variables.

```
[ ]: # Get the coefficients of the trained Elastic Net model
     coefficients = m2.coef_

     # Create a DataFrame to store the coefficients along with the corresponding␣
      ↪feature names
     coefficients_df = pd.DataFrame({'Feature': X.columns, 'Coefficient':␣
      ↪coefficients})

     # Sort the coefficients by absolute value to see the most influential features
     coefficients_df['Absolute Coefficient'] = np.abs(coefficients_df['Coefficient'])
     coefficients_df = coefficients_df.sort_values(by='Absolute Coefficient',␣
      ↪ascending=False)

     # Filter coefficients different than zero
     significant_coefficients_df = coefficients_df[coefficients_df['Coefficient'] !=␣
      ↪0]

     # Print the significant coefficients
     print(significant_coefficients_df)
```

```
                    Feature   Coefficient  Absolute Coefficient
1                  int_rate  1.833873e-02          1.833873e-02
7          year_since_issue -6.318333e-03          6.318333e-03
3                       dti  3.886443e-03          3.886443e-03
11               term_Short -1.708044e-03          1.708044e-03
6                 total_acc -5.414929e-04          5.414929e-04
5                revol_util  1.486708e-04          1.486708e-04
8    years_since_first_credit -9.761435e-05          9.761435e-05
0                 loan_amnt  3.091107e-06          3.091107e-06
4                 revol_bal -8.000612e-07          8.000612e-07
2                annual_inc -7.705455e-07          7.705455e-07
```

```
[ ]: # Filter out coefficients that were not shrunk (non-zero coefficients)
     non_shrunk_coeffs = coefficients[coefficients != 0]

     # Filter corresponding feature names
     selected_features = X.columns[coefficients != 0]
```

```
# Select only the columns with non-shrunk coefficients from your original␣
↪dataset
X_selected = X[selected_features]
```

1.5.3 Model 3: Logistic Regression on restricted model (significant variables)

```
[ ]: ## Train-test split
# Splitting the dataset into training and testing sets
# Test size is 20% of the entire dataset
# Stratify=y ensures that the class distribution is preserved in both train and␣
↪test sets
# Random state ensures reproducibility of the split
X_train_restricted, X_test_restricted, y_train_restricted, y_test_restricted =␣
↪train_test_split(X_selected, y, test_size=0.2, stratify=y, random_state=42)

# Validation set
# Further splitting the training set into training and validation sets
# Validation size is 25% of the training set, i.e., 20% of the entire dataset
# Stratify=y_train ensures that the class distribution is preserved in the␣
↪validation set
# Random state ensures reproducibility of the split
X_train_restricted, X_val_restricted, y_train_restricted, y_val_restricted =␣
↪train_test_split(X_train_restricted, y_train_restricted, test_size=0.25,␣
↪stratify=y_train_restricted, random_state=42)


# Printing the shapes of the resulting datasets
print("Training set shape:", X_train_restricted.shape, y_train_restricted.shape)
print("Validation set shape:", X_val_restricted.shape, y_val_restricted.shape)
print("Test set shape:", X_test_restricted.shape, y_test_restricted.shape)
```

```
Training set shape: (237618, 10) (237618,)
Validation set shape: (79206, 10) (79206,)
Test set shape: (79206, 10) (79206,)
```

```
[ ]: # MODEL 3
m3 = LogisticRegression(random_state=123, max_iter=1000)
m3.fit(X_train_restricted, y_train_restricted)

# Predicting on the training set
y_hat_train_m3 = m3.predict(X_train_restricted)
train_accuracy_m3 = accuracy_score(y_train_restricted, y_hat_train_m3)

# Predicting on the validation set
y_hat_val_m3 = m3.predict(X_val_restricted)
```

```
val_accuracy_m3 = accuracy_score(y_val_restricted, y_hat_val_m3)

# Predicting on the test set
y_hat_test_m3 = m3.predict(X_test_restricted)
test_accuracy_m3 = accuracy_score(y_test_restricted, y_hat_test_m3)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m3 = round(train_accuracy_m3, 4)
val_accuracy_m3 = round(val_accuracy_m3, 4)
test_accuracy_m3 = round(test_accuracy_m3, 4)

print('Model 3: Logistic Regression on restricted model\n')
print(f"Training Accuracy: {train_accuracy_m3}")
print(f"Validation Accuracy: {val_accuracy_m3}")
print(f"Test Accuracy: {test_accuracy_m3}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test_restricted, y_hat_test_m3, digits=4))
```

```
Model 3: Logistic Regression on restricted model

Training Accuracy: 0.8012
Validation Accuracy: 0.8004
Test Accuracy: 0.8012

Classification Report:
              precision    recall  f1-score   support

           0     0.8101    0.9831    0.8883     63671
           1     0.4452    0.0555    0.0987     15535

    accuracy                         0.8012     79206
   macro avg     0.6277    0.5193    0.4935     79206
weighted avg     0.7385    0.8012    0.7334     79206
```

Interpretation The model consistently achieves an 80% accuracy across training, validation, and test sets, primarily due to its strong performance in classifying class 0 (the majority class).

The accuracy is similar to *Model 2* but the F1-score jumps from 0.01 to almost 0.1, so this model can classify better the class 1, as does *Model 1*.

Next step is to fix the imbalance of classes.

1.5.4 Model 4: Logistic Regression with balanced classes (full model)

In classification problems, class imbalance occurs when the number of instances in one class is significantly higher than in the other class(es). This can lead to biased models that favor the

majority class and perform poorly on the minority class. To address this issue, various techniques can be employed, one of which is using balanced class weights.
Balanced Class Weight:
When you set the class_weight parameter to ‘balanced’ in a logistic regression model, the algorithm automatically adjusts the weights inversely proportional to the class frequencies in the training data. This means that the minority class will have a higher weight, and the majority class will have a lower weight. The purpose of this adjustment is to penalize misclassifications of the minority class more than those of the majority class, thereby encouraging the model to perform better on the minority class.

```
[ ]: # MODEL 4
m4 = LogisticRegression(random_state=123, max_iter=1000,␣
 ↪class_weight='balanced')
m4.fit(X_train, y_train)

# Predicting on the training set
y_hat_train_m4 = m4.predict(X_train)
train_accuracy_m4 = accuracy_score(y_train, y_hat_train_m4)

# Predicting on the validation set
y_hat_val_m4 = m4.predict(X_val)
val_accuracy_m4 = accuracy_score(y_val, y_hat_val_m4)

# Predicting on the test set
y_hat_test_m4 = m4.predict(X_test)
test_accuracy_m4 = accuracy_score(y_test, y_hat_test_m4)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m4 = round(train_accuracy_m4, 4)
val_accuracy_m4 = round(val_accuracy_m4, 4)
test_accuracy_m4 = round(test_accuracy_m4, 4)

print('Model 4: Logistic Regression with balanced classes (full model)\n')
print(f"Training Accuracy: {train_accuracy_m4}")
print(f"Validation Accuracy: {val_accuracy_m4}")
print(f"Test Accuracy: {test_accuracy_m4}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m4, digits=4))
```

```
Model 4: Logistic Regression with balanced classes (full model)

Training Accuracy: 0.5301
Validation Accuracy: 0.5331
Test Accuracy: 0.5324

Classification Report:
```

```
              precision    recall  f1-score   support

           0     0.8684    0.4931    0.6290     63671
           1     0.2503    0.6939    0.3679     15535

    accuracy                         0.5324     79206
   macro avg     0.5594    0.5935    0.4985     79206
weighted avg     0.7472    0.5324    0.5778     79206
```

Interpretation Because of the calss balancing *Model 4* has an accuracy of almost 53%, the model exhibits a relatively high precision of 0.86 for class 0, indicating that when the model predicts an instance as belonging to class 0, it is correct most of the time. But, as excpected, the class 0 indicators are worse than the ones in the previous models.

Also the F1-score of the class 1 improved from 0.1 to 0.36, but as a trade-off the F1-score of the class 1 degraded to 0.62.

Next step is to try to use the balanced weights on the restricted model with only significant coefficients.

1.5.5 Model 5: Logistic Regression with balanced classes (restricted model)

```
[ ]: # MODEL 5

#Logistic model
m5 = LogisticRegression(random_state=123, max_iter=1000,␣
 ↪class_weight='balanced')
m5.fit(X_train_restricted, y_train_restricted)

# Predicting on the training set
y_hat_train_m5 = m5.predict(X_train_restricted)
train_accuracy_m5 = accuracy_score(y_train_restricted, y_hat_train_m5)

# Predicting on the validation set
y_hat_val_m5 = m5.predict(X_val_restricted)
val_accuracy_m5 = accuracy_score(y_val_restricted, y_hat_val_m5)

# Predicting on the test set
y_hat_test_m5 = m5.predict(X_test_restricted)
test_accuracy_m5 = accuracy_score(y_test_restricted, y_hat_test_m5)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m5 = round(train_accuracy_m5, 4)
val_accuracy_m5 = round(val_accuracy_m5, 4)
test_accuracy_m5 = round(test_accuracy_m5, 4)

print('Model 5: Logistic Regression with balanced classes (restricted model)\n')
print(f"Training Accuracy: {train_accuracy_m5}")
print(f"Validation Accuracy: {val_accuracy_m5}")
print(f"Test Accuracy: {test_accuracy_m5}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test_restricted, y_hat_test_m5, digits=4))
```

```
Model 5: Logistic Regression with balanced classes (restricted model)

Training Accuracy: 0.5301
Validation Accuracy: 0.5332
Test Accuracy: 0.5325

Classification Report:
              precision    recall  f1-score   support

           0     0.8684    0.4931    0.6290     63671
           1     0.2504    0.6939    0.3679     15535

    accuracy                         0.5325     79206
   macro avg     0.5594    0.5935    0.4985     79206
weighted avg     0.7472    0.5325    0.5778     79206
```

Interpretation The model is still biased towards class 0, with no improvement compared to the full model.

All the metrics are less or equal compared to the full model. For this reason, the restricted model won't be considered any more.

Next there will be a comparision on full model using *Random Forest* and *Perceptron* vs *Logistic Regression*

1.5.6 Random Forest & Perceptron vs Model 4

Model 4: Logistic Regression with balanced classes (full model) in comparison with: - Random Forest with balanced classes - Perceptron with SMOTE (Synthetic Minority Over-sampling Technique) is a method to create synthetic samples of the minority class to balance imbalanced datasets.

```
[ ]: #RandomForestClassifier
     rf_classifier = RandomForestClassifier(random_state=123, n_estimators=100,␣
      ↪class_weight='balanced')
     rf_classifier.fit(X_train, y_train)

     # Predicting on the training set
     y_pred_rf_train = rf_classifier.predict(X_train)
     train_accuracy = accuracy_score(y_train, y_pred_rf_train)
```

```
# Predicting on the validation set
y_pred_rf_val = rf_classifier.predict(X_val)
val_accuracy = accuracy_score(y_val, y_pred_rf_val)

# Predicting on the test set
y_pred_rf_test = rf_classifier.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred_rf_test)

# Printing Accuracy
print('RandomForestClassifier\n')
print("Training Accuracy:", round(train_accuracy, 4))
print("Validation Accuracy:", round(val_accuracy, 4))
print("Test Accuracy :", round(test_accuracy, 4))

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rf_test))
```

```
RandomForestClassifier

Training Accuracy: 1.0
Validation Accuracy: 0.8062
Test Accuracy : 0.8061

Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.99      0.89     63671
           1       0.55      0.06      0.11     15535

    accuracy                           0.81     79206
   macro avg       0.68      0.52      0.50     79206
weighted avg       0.76      0.81      0.74     79206
```

```
[ ]: from imblearn.over_sampling import SMOTE

     # Handle imbalance data with SMOTE
     sm = SMOTE(random_state=123)
     X_train_res, y_train_res = sm.fit_resample(X_train, y_train)

     # Train the Perceptron again with resampled data
     perceptron = Perceptron(random_state=123)
     perceptron.fit(X_train_res, y_train_res)

     # Predicting on the training set
     y_perceptron_train_res = perceptron.predict(X_train)
```

```
train_accuracy_perceptron_res = accuracy_score(y_train, y_perceptron_train_res)

# Predicting on the validation set
y_perceptron_val_res = perceptron.predict(X_val)
val_accuracy_perceptron_res = accuracy_score(y_val, y_perceptron_val_res)

# Predicting on the test set
y_perceptron_test_res = perceptron.predict(X_test)
test_accuracy_perceptron_res = accuracy_score(y_test, y_perceptron_test_res)

# Printing Accuracy
print('Handle imbalance data with SMOTE\n')
print("Training Accuracy with SMOTE:", round(train_accuracy_perceptron_res, 4))
print("Validation Accuracy with SMOTE:", round(val_accuracy_perceptron_res, 4))
print("Test Accuracy with SMOTE:", round(test_accuracy_perceptron_res, 4))

# Printing results
print(classification_report(y_test, y_perceptron_test_res))
```

```
Handle imbalance data with SMOTE

Training Accuracy with SMOTE: 0.8037
Validation Accuracy with SMOTE: 0.8038
Test Accuracy with SMOTE: 0.8036
              precision    recall  f1-score   support

           0       0.80      1.00      0.89     63671
           1       0.31      0.00      0.00     15535

    accuracy                           0.80     79206
   macro avg       0.56      0.50      0.45     79206
weighted avg       0.71      0.80      0.72     79206
```

Interpretation Both models offer very bad metrics compared to *Model 4*.

Random forest has a too much high accuracy on the train set that signify overfitting, while perceptron cannot classify class 1 properly so it as an F1-score of 0.

For this reason these models will be excluded from further analysis.

1.5.7 Model 6: Logistic Regression with RATIOs (full model)

```
[ ]: # Remove the original columns used for calculations from df_ratio
columns_to_drop_ratio = ['installment', 'annual_inc', 'loan_amnt', 'revol_bal',␣
 ↪'open_acc', 'sub_grade', 'emp_title', 'issue_d', 'title',␣
 ↪'earliest_cr_line', 'address', 'mort_acc', 'pub_rec', 'pub_rec_bankruptcies']
df_ratio.drop(columns_to_drop_ratio, axis=1, inplace=True)
```

```
# Cut outliers to remove Inf values creaed by RATIOs (Division by 0)
df_ratio = cut_outliers(df_ratio)
```

[ ]:
```
# Create a copy of the DataFrame of df_ratio
df_ratio_encoded = df_ratio.copy()

# Drop the 'job_title' column due to its high cardinality which makes one-hot␣
 ↪encoding computationally expensive
df_ratio_encoded.drop("job_title", axis=1, inplace=True)

# Automatically identify categorical features for one-hot encoding, excluding␣
 ↪'loan_status'
categorical_features = df_ratio_encoded.select_dtypes(include=['object',␣
 ↪'category']).columns.tolist()
if 'loan_status' in categorical_features:
    categorical_features.remove('loan_status')

# Perform one-hot encoding on the identified categorical features
df_ratio_encoded = pd.get_dummies(df_ratio_encoded,␣
 ↪columns=categorical_features, drop_first=True)
```

[ ]:
```
# X_ratio contains all features except 'loan_status'
X_ratio = df_ratio_encoded.drop('loan_status', axis=1)

# y_ratio contains the target variable 'loan_status'
y_ratio = df_ratio_encoded['loan_status']

# Train-test split
# Splitting the dataset into training and testing sets
# Test size is 20% of the entire dataset
# Stratify=y ensures that the class distribution is preserved in both train and␣
 ↪test sets
# Random state ensures reproducibility of the split
X_train, X_test, y_train, y_test = train_test_split(X_ratio, y_ratio,␣
 ↪test_size=0.2, stratify=y_ratio, random_state=42)

# Validation set
# Further splitting the training set into training and validation sets
# Validation size is 25% of the training set, i.e., 20% of the entire dataset
# Stratify=y_train ensures that the class distribution is preserved in the␣
 ↪validation set
# Random state ensures reproducibility of the split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪25, stratify=y_train, random_state=42)
```

```
# Printing the shapes of the resulting datasets
print("Training set shape:", X_train.shape, y_train.shape)
print("Validation set shape:", X_val.shape, y_val.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (237618, 55) (237618,)
Validation set shape: (79206, 55) (79206,)
Test set shape: (79206, 55) (79206,)
```

[ ]:
```
# MODEL 6
m6 = LogisticRegression(random_state=123, max_iter=1000,␣
 ↪class_weight='balanced')
m6.fit(X_train, y_train)

# Predicting on the training set
y_hat_train_m6 = m6.predict(X_train)
train_accuracy_m6 = accuracy_score(y_train, y_hat_train_m6)

# Predicting on the validation set
y_hat_val_m6 = m6.predict(X_val)
val_accuracy_m6 = accuracy_score(y_val, y_hat_val_m6)

# Predicting on the test set
y_hat_test_m6 = m6.predict(X_test)
test_accuracy_m6 = accuracy_score(y_test, y_hat_test_m6)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m6 = round(train_accuracy_m6, 4)
val_accuracy_m6 = round(val_accuracy_m6, 4)
test_accuracy_m6 = round(test_accuracy_m6, 4)

print('Model 6 logistic Regression without scaling')
print(f"Training Accuracy: {train_accuracy_m6}")
print(f"Validation Accuracy: {val_accuracy_m6}")
print(f"Test Accuracy: {test_accuracy_m6}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m6, digits=4))
```

```
Model 6 logistic Regression without scaling
Training Accuracy: 0.6463
Validation Accuracy: 0.648
Test Accuracy: 0.6467

Classification Report:
              precision    recall  f1-score   support
```

```
           0     0.8850    0.6443    0.7457     63671
           1     0.3106    0.6568    0.4217     15535

    accuracy                         0.6467     79206
   macro avg     0.5978    0.6505    0.5837     79206
weighted avg     0.7723    0.6467    0.6821     79206
```

Interpretation *Model 6* has a higher accuracy compared to *Model 4*.

Not only that, the F1-score for the minority class is 0.42 compared to 0.37 and also the F1-score of class 0 goes from 0.63 to 0.74.

The model 6 overall performs better because is able to classify in clearer way the minority class.

1.5.8 Model 7: Logistic Regression on the dataset with outliers

[ ]:
```
# Create a copy of the DataFrame df_fe with outliers and call it df_fe_encoded
df_fe_encoded = df_fe.copy()

# Drop the 'job_title' column due to its high cardinality, which makes one-hot␣
↪encoding computationally expensive
df_fe_encoded.drop("job_title", axis=1, inplace=True)

# Automatically identify categorical features for one-hot encoding, excluding␣
↪'loan_status'
categorical_features = df_fe_encoded.select_dtypes(include=['object',␣
↪'category']).columns.tolist()
if 'loan_status' in categorical_features:
    categorical_features.remove('loan_status')

# Perform one-hot encoding on the identified categorical features
df_fe_encoded = pd.get_dummies(df_fe_encoded, columns=categorical_features,␣
↪drop_first=True)
```

[ ]:
```
# x_outliers contains all features except 'loan_status'
x_outliers = df_fe_encoded.drop('loan_status', axis=1)

# y_outliers contains the target variable 'loan_status'
y_outliers = df_fe_encoded['loan_status']

# Train-test split
# Splitting the dataset into training and testing sets
# Test size is 20% of the entire dataset
# Stratify=y_outliers ensures that the class distribution is preserved in both␣
↪train and test sets
# Random state ensures reproducibility of the split
```

```
x_train, x_test, y_train, y_test = train_test_split(x_outliers, y_outliers,␣
↪test_size=0.2, stratify=y_outliers, random_state=42)

# Validation set
# Further splitting the training set into training and validation sets
# Validation size is 25% of the training set, i.e., 20% of the entire dataset
# Stratify=y_train ensures that the class distribution is preserved in the␣
↪validation set
# Random state ensures reproducibility of the split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.
↪25, stratify=y_train, random_state=42)

# Printing the shapes of the resulting datasets
print("Training set shape:", x_train.shape, y_train.shape)
print("Validation set shape:", x_val.shape, y_val.shape)
print("Test set shape:", x_test.shape, y_test.shape)
```

```
Training set shape: (237618, 54) (237618,)
Validation set shape: (79206, 54) (79206,)
Test set shape: (79206, 54) (79206,)
```

[ ]:
```
# MODEL 7
m7 = LogisticRegression(random_state=123, max_iter=2000)
m7.fit(X_train, y_train)

# Predicting on the training set
y_hat_train_m7 = m7.predict(X_train)
train_accuracy_m7 = accuracy_score(y_train, y_hat_train_m7)

# Predicting on the validation set
y_hat_val_m7 = m7.predict(X_val)
val_accuracy_m7 = accuracy_score(y_val, y_hat_val_m7)

# Predicting on the test set
y_hat_test_m7 = m7.predict(X_test)
test_accuracy_m7 = accuracy_score(y_test, y_hat_test_m7)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m7 = round(train_accuracy_m7, 4)
val_accuracy_m7 = round(val_accuracy_m7, 4)
test_accuracy_m7 = round(test_accuracy_m7, 4)

print('Model 7: Logistic Regression (Dataset with outliers)\n')
print(f"Training Accuracy: {train_accuracy_m7}")
print(f"Validation Accuracy: {val_accuracy_m7}")
print(f"Test Accuracy: {test_accuracy_m7}")
```

```
# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m7, digits=4))
```

```
Model 7: Logistic Regression (Dataset with outliers)

Training Accuracy: 0.8049
Validation Accuracy: 0.8044
Test Accuracy: 0.8048

Classification Report:
              precision    recall  f1-score   support

           0     0.8116    0.9861    0.8904     63671
           1     0.5204    0.0617    0.1103     15535

    accuracy                         0.8048     79206
   macro avg     0.6660    0.5239    0.5003     79206
weighted avg     0.7545    0.8048    0.7374     79206
```

The results are very similar to *Model 1*.

Interpretation

1.5.9 Model 8: Logistic Regression with balanced classes and in the dataset with outliers

```
[ ]: # MODEL 8
m8 = LogisticRegression(random_state=123, max_iter=1000,␣
 ↪class_weight='balanced')
m8.fit(X_train, y_train)

# Predicting on the training set
y_hat_train_m8 = m8.predict(X_train)
train_accuracy_m8 = accuracy_score(y_train, y_hat_train_m8)

# Predicting on the validation set
y_hat_val_m8 = m8.predict(X_val)
val_accuracy_m8 = accuracy_score(y_val, y_hat_val_m8)

# Predicting on the test set
y_hat_test_m8 = m8.predict(X_test)
test_accuracy_m8 = accuracy_score(y_test, y_hat_test_m8)

# Rounding the accuracy scores to 4 decimal places
train_accuracy_m8 = round(train_accuracy_m8, 4)
val_accuracy_m8 = round(val_accuracy_m8, 4)
```

```
test_accuracy_m8 = round(test_accuracy_m8, 4)

print('Model 8: Logistic Regression with balanced classes (full model, with␣
↪outliers)\n')
print(f"Training Accuracy: {train_accuracy_m8}")
print(f"Validation Accuracy: {val_accuracy_m8}")
print(f"Test Accuracy: {test_accuracy_m8}")

# Printing classification report for the test set
print("\nClassification Report:")
print(classification_report(y_test, y_hat_test_m8, digits=4))
```

```
Model 8: Logistic Regression with balanced classes (full model, with outliers)

Training Accuracy: 0.6463
Validation Accuracy: 0.648
Test Accuracy: 0.6467

Classification Report:
              precision    recall  f1-score   support

           0     0.8850    0.6443    0.7457     63671
           1     0.3106    0.6568    0.4217     15535

    accuracy                         0.6467     79206
   macro avg     0.5978    0.6505    0.5837     79206
weighted avg     0.7723    0.6467    0.6821     79206
```

Interpretation With a 65% accuracy the model performs as better as *Model 6* and better then the others.

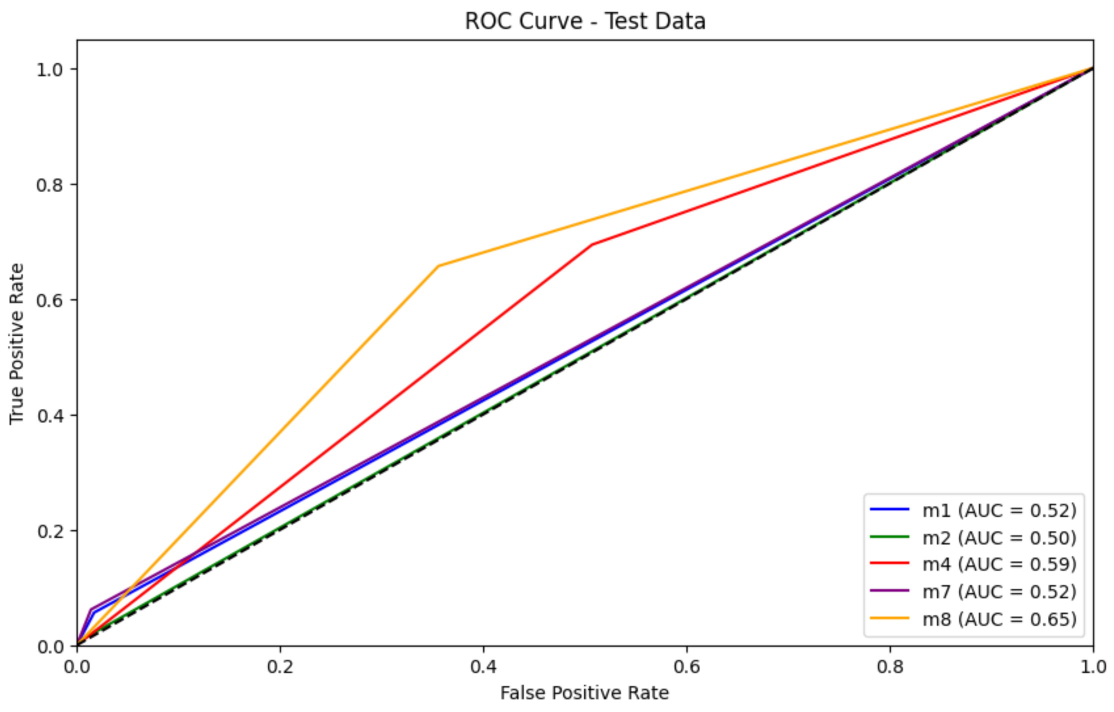
Precision: This represents the proportion of positive predictions that were correct. For class 0, the precision is 0.8850 (88.5%), and for class 1, it is 0.3106 (31.06%). Recall: This represents the proportion of actual positives that were correctly identified. For class 0, the recall is 0.6443 (64.43%), and for class 1, it is 0.6568 (65.68%). F1-score: This is the harmonic mean of precision and recall, providing a balanced measure of both metrics. For class 0, the F1-score is 0.7457 (74.57%), and for class 1, it is 0.4217 (42.17%).

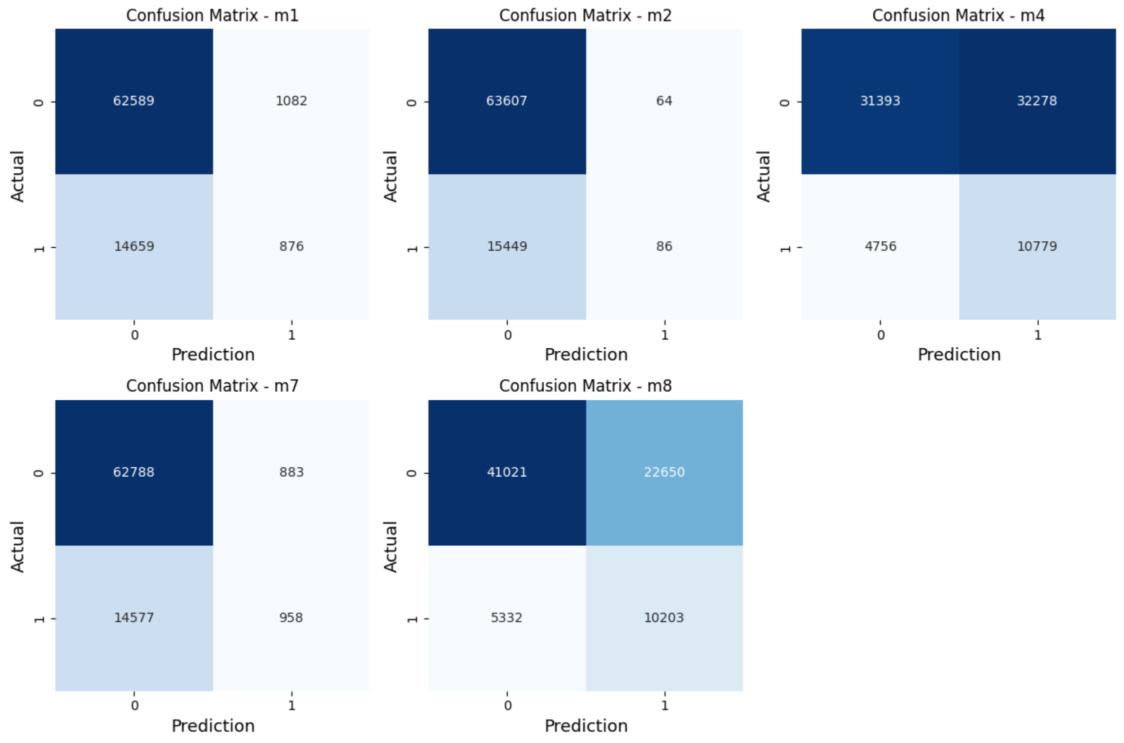
Based on these metrics, the model appears to perform better at classifying instances of class 0 compared to class 1, as evidenced by the higher precision and recall for class 0.

1.6 Model selection

Model 6 is taken no more in cosideration because it has the same results as *Model 8*, this leads us to the conclusion that the substitutions of new ratios instead of the variables with which they were created doesn't bring new and useful information to the prediction.

```
[ ]: from sklearn.metrics import roc_curve, auc, confusion_matrix

     # To evaluate multiple models and compare their performance using ROC curves.

     # Define model names
     models = ['m1', 'm2', 'm4', 'm7', 'm8']

     # Initialize empty dictionaries to store predictions
     y_pred_train = {}
     y_pred_val = {}
     y_pred_test = {}

     # Fill dictionaries with predictions
     for model in models:
         y_pred_train[model] = globals()[f"y_hat_train_{model}"]
         y_pred_val[model] = globals()[f"y_hat_val_{model}"]
         y_pred_test[model] = globals()[f"y_hat_test_{model}"]

     # Define a list of colors for the ROC curves
     colors = ['blue', 'green', 'red', 'purple', 'orange']

     # Plot ROC curves
     plt.figure(figsize=(10, 6))
     for model, color in zip(models, colors):
         fpr, tpr, _ = roc_curve(y_test, y_pred_test[model])
         roc_auc = auc(fpr, tpr)
         plt.plot(fpr, tpr, color=color, label=f'{model} (AUC = {roc_auc:.2f})')

     plt.plot([0, 1], [0, 1], 'k--')
     plt.xlim([0.0, 1.0])
     plt.ylim([0.0, 1.05])
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('ROC Curve - Test Data')
     plt.legend(loc="lower right")
     plt.show()

     # Plot confusion matrices
     plt.figure(figsize=(12, 8))
     for i, model in enumerate(models, start=1):
         plt.subplot(2, 3, i)
         cm = confusion_matrix(y_test, [round(pred) for pred in y_pred_test[model]])
         sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
         plt.title(f'Confusion Matrix - {model}')
         plt.xlabel('Prediction',fontsize=13)
         plt.ylabel('Actual',fontsize=13)
plt.tight_layout()
plt.show()
```





Interpretation

ROC curve The Receiver Operating Characteristic (ROC) curve shows the performance of the three models on the test data. The area under the curve (AUC) is a summary metric that indicates the overall performance of a binary classifier. Higher AUC values are better.

Model 1 and *Model 2* have an AUC of 0.50, which is equivalent to a random classifier and suggests poor performance. *Model 4* has an AUC of 0.59 and *Model 8* has an AUC of 0.65, which are better than random but still not very good.

Confusion matrix

- *Model 1*

True Positives (TP): 876
True Negatives (TN): 62589
False Positives (FP): 1082
False Negatives (FN): 14659

- *Model 2*

True Positives (TP): 86
True Negatives (TN): 63607
False Positives (FP): 64
False Negatives (FN): 15449

- *Model 4*

True Positives (TP): 10779
True Negatives (TN): 31393
False Positives (FP): 32278
False Negatives (FN): 4756

- *Model 7*

True Positives (TP): 958
True Negatives (TN): 62788
False Positives (FP): 883
False Negatives (FN): 14577

- *Model 8*

True Positives (TP): 10203
True Negatives (TN): 41021
False Positives (FP): 22650
False Negatives (FN): 5332

Overall, the confusion matrices reveal that m1, m2 and m7 perform poorly. While, taking in consideration the fact m4 and m8 has better performance but still makes a significant number of misclassifications for both classes. Reducing false negatives is particularly important in scenarios where missing a positive instance can have severe consequences. For example, in loan's business, a false negative means a possible default goes undetected, which can lead to significant financial losses. Therefore models with a higher recall (like m4 and m8) are preferable, even if they come with a higher rate of false positives.

1.6.1 (One possible) Conclusion

In summary, based on the ROC curve, confusion matrices and on the comparison between models using accuracy and other metrics, *Model 4* and *Model 8* to be the best-performing model among the three, but its performance is still not very good, suggesting that further model improvements or different approaches may be necessary for the given task. Logistic regression is influenced by outliers but not as much as other algorithms, giving a slightly better performance in the model with outliers compared to the one without. So they should be maintained in the dataset because they offer useful information for the analysis because these non common values could be default indicators.

```
[ ]: #This code extracts the coefficients from the  model 8, calculates the odds␣
      ↪ratios for each feature
     #Prints the feature names along with their corresponding odds ratios.
     #This helps understand how each feature influences the model's predictions.

     # Get the coefficients and the intercept
     coefficients = m8.coef_[0]

     # Print the coefficients and their corresponding odds ratios
     print("Feature Odds Ratios")

     for feature, coef in zip(x_outliers.columns, coefficients):
         odds_ratio = np.exp(coef)
         print(f"{feature}: {odds_ratio:.2f}")
```

```
Feature Odds Ratios
loan_amnt: 1.10
int_rate: 1.02
annual_inc: 1.00
dti: 0.99
revol_bal: 0.91
revol_util: 0.99
total_acc: 1.11
year_since_issue: 0.82
years_since_first_credit: 1.62
pub_rec_flag: 1.17
open_acc_flag: 1.00
term_Short: 0.99
grade_B: 0.57
grade_C: 1.10
grade_D: 1.29
grade_E: 1.27
```

```
grade_F: 1.14
grade_G: 0.92
emp_length_4-6 years: 1.01
emp_length_7-9 years: 0.90
emp_length_10+ years: 0.92
emp_length_Unknown: 0.89
home_ownership_Other: 1.63
home_ownership_Own: 1.00
home_ownership_Rent: 1.26
verification_status_Source Verified: 1.36
verification_status_Verified: 1.11
purpose_Credit card: 1.11
purpose_Debt consolidation: 0.83
purpose_Educational: 0.94
purpose_Home improvement: 1.01
purpose_House: 1.00
purpose_Major purchase: 0.97
purpose_Medical: 0.96
purpose_Moving: 1.01
purpose_Other: 1.02
purpose_Renewable energy: 0.96
purpose_Small business: 1.01
purpose_Vacation: 1.29
purpose_Wedding: 0.98
initial_list_status_Whole: 0.95
application_type_Individual: 0.94
application_type_Joint: 0.86
month_issue_Aug: 0.96
month_issue_Dec: 0.92
month_issue_Feb: 0.90
month_issue_Jan: 1.01
month_issue_Jul: 0.97
month_issue_Jun: 1.04
month_issue_Mar: 1.02
month_issue_May: 0.97
month_issue_Nov: 1.08
month_issue_Oct: 0.97
month_issue_Sep: 1.00
```

2 Odds Ratio Analysis

Based on the provided odds ratios, here's the interpretation:

2.1 Continuous Variables

- **loan_amnt**: For every one unit increase in the loan amount, the odds of default increase by **10%**.

- **int_rate**: For every one unit increase in the interest rate, the odds of default increase by **2%**.

- **annual_inc**: For every one unit increase in the annual income, the odds of default remain the same (no change).
- **dti**: For every one unit increase in the debt-to-income ratio, the odds of default decrease by **1%**.
- **revol_bal**: For every one unit increase in the revolving balance, the odds of default decrease by **9%**.
- **revol_util**: For every one unit increase in the revolving utilization ratio, the odds of default decrease by **1%**.
- **total_acc**: For every one unit increase in the total number of accounts, the odds of default increase by **11%**.
- **years_since_first_credit**: For every one unit increase in the number of years since the first credit line, the odds of default increase by **62%**.
- **year_since_issue**: For every one unit increase in the number of years since the issue date, the odds of default decrease by **18%**.

2.2 Categorical Variables

- **pub_rec_flag**: Applicants with public records have **17%** higher odds of default compared to those without public records.
- **open_acc_flag**: Applicants with open accounts have no difference (0%) in odds of default compared to those without open accounts.
- **term_Short**: Applicants with a short-term loan have **1%** lower odds of default compared to those with a long-term loan.

Grade - Compared to grade A: - Grade B: **76%** lower odds of default. - Grade C: **10%** higher odds of default. - Grade D: **29%** higher odds of default. - Grade E: **27%** higher odds of default. - Grade F: **14%** higher odds of default. - Grade G: **8%** lower odds of default.

Employment Length - Compared to employment length of 0-3 years: - 4-6 years: **1%** higher odds of default. - 7-9 years: **10%** lower odds of default. - 10+ years: **8%** lower odds of default. - Unknown: **11%** lower odds of default.

Home Ownership - Compared to having a mortgage: - Other: **63%** higher odds of default. - Own: No change (0%) in odds of default. - Rent: **26%** higher odds of default.

Verification Status - Compared to not verified status: - Source Verified: **36%** higher odds of default. - Verified: **11%** higher odds of default.

Purpose - Compared to car loans: - Credit card: **11%** higher odds of default. - Debt consolidation: **17%** lower odds of default. - Educational: **6%** lower odds of default. - Home improvement: **1%** higher odds of default. - House: No change (0%) in odds of default. - Major purchase: **3%** lower odds of default. - Medical: **4%** lower odds of default. - Moving: **1%** higher odds of default. - Other: **2%** higher odds of default. - Renewable energy: **4%** lower odds of default. - Small business: **1%** higher odds of default. - Vacation: **29%** higher odds of default. - Wedding: **2%** lower odds of default.

Initial List Status - Applicants with a whole loan on the initial list have **5%** lower odds of default compared to those with a fractional loan.

Application Type - Compared to direct pay applications: - Individual: **6%** lower odds of default. - Joint: **14%** lower odds of default.

Month Issue - Compared to April: - August: **4%** lower odds of default.
- December: **8%** lower odds of default. - February: **10%** lower odds of default. - January: **1%** higher odds of default. - July: **3%** lower odds of default. - June: **2%** higher odds of default. - March: **2%** higher odds of default. - May: **3%** lower odds of default. - November: **8%** higher odds of default. - October: **3%** lower odds of default. - September: No change in odds of default.

2.3 Most Influential

- **years_since_first_credit**: A one-unit increase increases the odds of default by **62%**.
- **home_ownership_Other**: Applicants with other home ownership status have **63%** higher odds of default compared to those with a mortgage.
- **grade_B**: Applicants with grade B have **76%** lower odds of default compared to grade A.
- **purpose_Debt consolidation**: Applicants with debt consolidation loans have **17%** lower odds of default compared to car loans.
- **verification_status_Source Verified**: Applicants with source-verified status have **36%** higher odds of default compared to those with not verified status.
- **purpose_Vacation**: Applicants with vacation loans have **29%** higher odds of default compared to car loans.