LoanTap: Logistic Regression

Overview

This project involves a logistic regression case study using real-world financial data from a lending company. The main objective is to predict whether a loan will be **fully paid** or will be **charged off** (i.e., defaulted). This binary classification problem helps financial institutions make more informed and data-driven lending decisions.

Goal of the Study

To build a predictive model using **logistic regression** that accurately classifies loan applicants based on the risk of default. The goal is to:

- Understand which features are significant in predicting loan status
- Improve the decision-making process for loan approvals
- Minimize the financial risk by identifying potentially defaulting applicants

Technologies and Tools Used

- **Python** (primary programming language)
- **Pandas, NumPy** – Data manipulation and preprocessing
- **Matplotlib, Seaborn** – Exploratory Data Analysis (EDA) and visualization
- **Scikit-learn** – Model building, evaluation, data splitting, scaling
- **Statsmodels** – Multicollinearity analysis using VIF
- **SMOTE** – To address class imbalance
- **Jupyter Notebook** – For coding, analysis, and presentation

Methodology Followed

The case study was carried out using a structured data science workflow:

1. **Data Loading & Initial Exploration**
    - Loaded the dataset and examined its structure, data types, and basic statistics
    - Identified missing values and performed initial observations on distributions
2. **Exploratory Data Analysis (EDA)**
    - Analyzed numerical and categorical features
    - Visualized trends, patterns, and relationships with the target variable
    - Identified potential outliers and skewness
3. **Data Cleaning and Preprocessing**
    - Dropped irrelevant features and handled missing values
    - Cleaned textual columns and transformed date-related variables
    - Performed encoding of categorical features using one-hot encoding
4. **Feature Selection and Multicollinearity Check (✅VIF)**
    - Applied **Variance Inflation Factor (VIF)** to check for multicollinearity among independent variables
    - Removed features with high VIF values to ensure model stability and interpretability
5. **Feature Scaling**
    - Scaled continuous variables using **MinMaxScaler** to bring all features to a common scale

6. **Train-Test Split**

- Split the dataset into training and test sets using **stratified sampling** to maintain class balance

7. Handling Class Imbalance

- Applied **SMOTE** (Synthetic Minority Over-sampling Technique) to balance the number of default and non-default cases in the training set

8**. Model Building: Logistic Regression**

- Built a **logistic regression** model using scikit-learn
- Evaluated model performance using:
  - Confusion matrix
  - Accuracy, Precision, Recall, F1-Score
  - **ROC-AUC curve** for overall performance

9. **Insights and Recommendations**

- Identified key factors affecting loan defaults
- Provided business recommendations based on the model outputs and feature importances

1. Loading and Exploring the Dataset

1.1 Importing Necessary Libraries

Code:

```
import pandas as pd
import numpy as np
import seaborn as sns
from scipy import stats
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score, roc_curve, precision_recall_curve
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay
from statsmodels.stats.outliers_influence import variance_inflation_factor
from imblearn.over_sampling import SMOTE
```

1.2 Exploring the Dataset

Code:

```
df = pd.read_csv("Downloads/logistic_regression.csv")
print("Dataset shape:", df.shape)
df.head()
print(df.columns)
df.describe(include='all')
df.info()
```

Output

```
Dataset shape: (396030, 27)
```

	loan_amnt	term	int_rate	installment	grade	sub_grade	emp_title	emp_length	home_ownership	annual_inc	...	open_acc	pub_rec	revol_bal	revol_util
0	10000.0	36 months	11.44	329.48	B	B4	Marketing	10+ years	RENT	117000.0	...	16.0	0.0	36369.0	41.8
1	8000.0	36 months	11.99	265.68	B	B5	Credit analyst	4 years	MORTGAGE	65000.0	...	17.0	0.0	20131.0	53.3
2	15600.0	36 months	10.49	506.97	B	B3	Statistician	< 1 year	RENT	43057.0	...	13.0	0.0	11987.0	92.2
3	7200.0	36 months	6.49	220.65	A	A2	Client Advocate	6 years	RENT	54000.0	...	6.0	0.0	5472.0	21.5
4	24375.0	60 months	17.27	609.33	C	C5	Destiny Management Inc.	9 years	MORTGAGE	55000.0	...	13.0	0.0	24584.0	69.8

```
Index(['loan_amnt', 'term', 'int_rate', 'installment', 'grade', 'sub_grade',
       'emp_title', 'emp_length', 'home_ownership', 'annual_inc',
       'verification_status', 'issue_d', 'loan_status', 'purpose', 'title',
       'dti', 'earliest_cr_line', 'open_acc', 'pub_rec', 'revol_bal',
       'revol_util', 'total_acc', 'initial_list_status', 'application_type',
       'mort_acc', 'pub_rec_bankruptcies', 'address'],
      dtype='object')
```

	loan_amnt	term	int_rate	installment	grade	sub_grade	emp_title	emp_length	home_ownership	annual_inc	...	open_acc	pub
count	396030.000000	396030	396030.000000	396030.000000	396030	396030	373103	377729	396030	3.960300e+05	...	396030.000000	396030.00
unique	NaN	2	NaN	NaN	7	35	173105	11	6	NaN	...	NaN	
top	NaN	36 months	NaN	NaN	B	B3	Teacher	10+ years	MORTGAGE	NaN	...	NaN	
freq	NaN	302005	NaN	NaN	116018	26655	4389	126041	198348	NaN	...	NaN	
mean	14113.888089	NaN	13.639400	431.849698	NaN	NaN	NaN	NaN	NaN	7.420318e+04	...	11.311153	0.178
std	8357.441341	NaN	4.472157	250.727790	NaN	NaN	NaN	NaN	NaN	6.163762e+04	...	5.137649	0.530
min	500.000000	NaN	5.320000	16.080000	NaN	NaN	NaN	NaN	NaN	0.000000e+00	...	0.000000	0.000
25%	8000.000000	NaN	10.490000	250.330000	NaN	NaN	NaN	NaN	NaN	4.500000e+04	...	8.000000	0.000
50%	12000.000000	NaN	13.330000	375.430000	NaN	NaN	NaN	NaN	NaN	6.400000e+04	...	10.000000	0.000
75%	20000.000000	NaN	16.490000	567.300000	NaN	NaN	NaN	NaN	NaN	9.000000e+04	...	14.000000	0.000
max	40000.000000	NaN	30.990000	1533.810000	NaN	NaN	NaN	NaN	NaN	8.706582e+06	...	90.000000	86.000

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 396030 entries, 0 to 396029
Data columns (total 27 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   loan_amnt             396030 non-null  float64
 1   term                  396030 non-null  object 
 2   int_rate              396030 non-null  float64
 3   installment           396030 non-null  float64
 4   grade                 396030 non-null  object 
 5   sub_grade             396030 non-null  object 
 6   emp_title             373103 non-null  object 
 7   emp_length            377729 non-null  object 
 8   home_ownership        396030 non-null  object 
 9   annual_inc            396030 non-null  float64
 10  verification_status   396030 non-null  object 
 11  issue_d               396030 non-null  object 
 12  loan_status           396030 non-null  object 
 13  purpose               396030 non-null  object 
 14  title                 394274 non-null  object 
 15  dti                   396030 non-null  float64
 16  earliest_cr_line      396030 non-null  object 
 17  open_acc              396030 non-null  float64
 18  pub_rec               396030 non-null  float64
 19  revol_bal             396030 non-null  float64
 20  revol_util            395754 non-null  float64
 21  total_acc             396030 non-null  float64
 22  initial_list_status   396030 non-null  object 
 23  application_type      396030 non-null  object 
 24  mort_acc              358235 non-null  float64
 25  pub_rec_bankruptcies  395495 non-null  float64
 26  address               396030 non-null  object 
dtypes: float64(12), object(15)
memory usage: 81.6+ MB
```

Insights:

1. The dataset contains 396030 rows and 27 rows showing a fairly large dataset suitable for statistical Modeling.
2. The datatypes in this dataset are mix of numerical and categorical types require encoding before modeling.
3. No duplicate rows were found, indicating clean transactional records.
4. Columns like id or member_id do not provide predictive power and can be removed before modeling.

1.3 Correlation Analysis

Analyzing the correlation among numerical features to identify potential multicollinearity or redundant variables.

Code:

 # Keep only numeric columns
numeric_data = df.select_dtypes(include=['number'])

plt.figure(figsize=(12, 8))
sns.heatmap(numeric_data.corr(method='spearman'), annot=True, cmap='viridis')
plt.title('Spearman Correlation Heatmap')
plt.show()

Output:

Insights:

To examine the strength and direction of relationships between numerical features, I used the **Spearman correlation heatmap**. This method helps identify both linear and monotonic relationships, making it suitable for skewed financial data. The key insights derived are as follows:

A **very strong correlation (0.97)** was observed between **loan_amnt** and **installment**, indicating multicollinearity. Both features convey almost the same information — higher loan amounts typically result in higher monthly installments. To avoid redundancy and multicollinearity issues in our predictive models, so we dropping **installment**, and only loan_amnt was retained for further analysis.

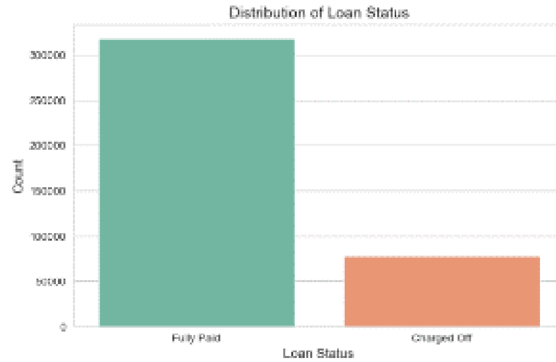
```
df.drop(columns=['installment'], axis=1, inplace=True)
```

1.4 Checking the distributions for outcome labels

Code and Output

```
df.loan_status.value_counts(normalize=True)*100
```

```
loan_status
Fully Paid     80.387092
Charged Off    19.612908
Name: proportion, dtype: float64
```



Insights:

80.39% of the loans are marked as **"Fully Paid"**, indicating a successful repayment.
19.61% of the loans are **"Charged Off"**, meaning these loans were **written off as a loss** due to the borrower's failure to repay.

The data is **imbalanced**, with a significantly higher proportion of fully paid loans. This imbalance should have to addressed during model training, as it can bias the model toward predicting the majority class.

2. Data Exploration

2.1 Loan Amount Analysis by Loan Status
Code:

df.groupby(by='loan_status')['loan_amnt'].describe()

Output:

loan_status	count	mean	std	min	25%	50%	75%	max
Charged Off	77673.0	15126.300967	8505.090557	1000.0	8525.0	14000.0	20000.0	40000.0
Fully Paid	318357.0	13866.878771	8302.319699	500.0	7500.0	12000.0	19225.0	40000.0

Insights:
Average loan amount is higher for **Charged Off** loans (15,126) compared to **Fully Paid** loans (13,867), indicating that **larger loans have a higher risk of default**.

Median loan amount is also higher for Charged Off loans (14,000 vs 12,000).
Both groups share the same **maximum loan cap** (40,000), but the **minimum loan** is lower for Fully Paid loans.
Slightly **higher variation** in Charged Off loans suggests more inconsistency in risky lending.

Conclusion: Larger loan amounts tend to be riskier, suggesting the need for stricter checks on high-value loan applications.

2.2 Home Ownership Distribution:

Code:

df['home_ownership'].value_counts()

Output:

```
home_ownership
MORTGAGE    198348
RENT        159790
OWN          37746
OTHER          112
NONE            31
ANY              3
Name: count, dtype: int64
```

Insights:

The majority of applicants either have a mortgage (198,348) or live in rental properties (159,790).
A smaller portion of applicants own their home outright (37,746).
Categories like OTHER, NONE, and ANY represent very few records, suggesting they may not add significant analytical value on their own.

To handle this, we consolidated the 'NONE' and 'ANY' categories into 'OTHER' to simplify and reduce sparsity in the data:

```
df.loc[(df.home_ownership == 'ANY') | (df.home_ownership == 'NONE'), 'home_ownership'] = 'OTHER'
df.home_ownership.value_counts()
```

2.3 Loan Title Distribution

Code:

Insights:

- The most common reason applicants apply for a loan is **debt consolidation**, with **multiple variations of this title** (e.g., 'Debt consolidation, 'Debt Consolidation', 'debt consolidation', 'Consolidation', 'Consolidation Loan', etc.).
- Other popular purposes include:
    - **Credit card refinancing** (51,487 loans)
    - **Home improvement** (15,264 loans)
    - **Major purchases**, **business**, and **medical expenses**

These top 20 titles account for a **large portion of all loans**, but many of them are **semantically similar**, just written differently (due to case sensitivity or slight wording changes).

Lets handle that by: Converting to lowercase, helps normalize the data, so titles that mean the same are grouped together in analysis.This reduces redundancy and ensures accurate aggregation, counting, and filtering.

```
df['title'] = df.title.str.lower()
```

Output:

```
title
debt consolidation           168108
credit card refinancing       51781
home improvement              17117
other                         12993
consolidation                  5583
major purchase                 4998
debt consolidation loan        3513
business                       3017
medical expenses               2820
credit card consolidation      2638
Name: count, dtype: int64
```

2.4 Date format converting

The columns **issue_d** (loan issue date) and **earliest_cr_line** (date of the borrower's earliest credit line) were originally in **string format**, as seen in the dataset's info summary.
In order to **properly perform time-based operations** (such as calculating credit history length, extracting year or month, sorting chronologically, etc.), it is essential to convert these columns into **datetime format**. Hence, we applied:

```
df['issue_d'] = pd.to_datetime(df['issue_d'])
df['earliest_cr_line'] = pd.to_datetime(df['earliest_cr_line'])
```

3. Exploratory Data Analysis (EDA)

3.1 Loan Status Distribution across Grades and Sub-Grades

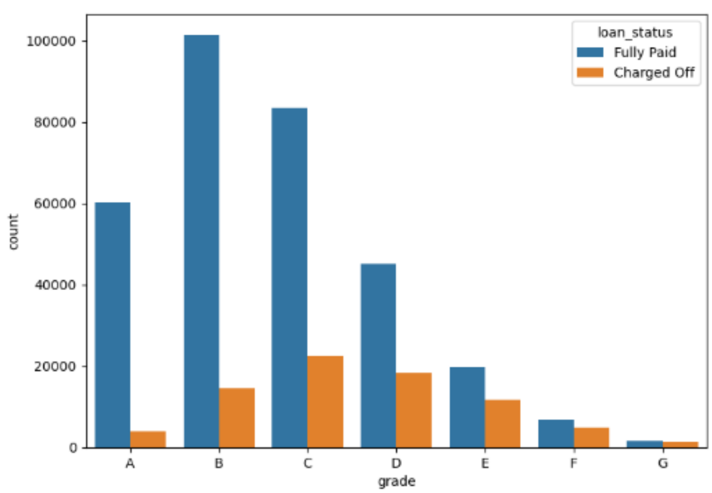
This visualization helps us **understand how loan status (Fully Paid vs Charged Off)** varies across different **credit risk grades** (grade and sub_grade) assigned to borrowers.

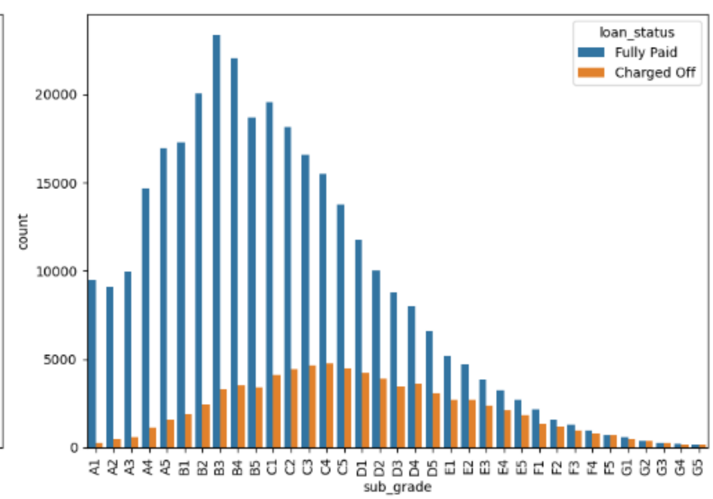
Lending institutions assign **grades (A to G)** and more detailed **sub-grades (like A1, A2… G5)** based on the borrower's creditworthiness. A higher grade generally implies a more reliable borrower.

Code:

```
plt.figure(figsize=(15, 10))

# Plot 1: Grade vs Loan Status
plt.subplot(2, 2, 1)
grade = sorted(df.grade.unique().tolist())
sns.countplot(x='grade', data=df, hue='loan_status', order=grade)

# Plot 2: Sub Grade vs Loan Status
plt.subplot(2, 2, 2)
sub_grade = sorted(df.sub_grade.unique().tolist())
g = sns.countplot(x='sub_grade', data=df, hue='loan_status', order=sub_grade)
g.set_xticklabels(g.get_xticklabels(), rotation=90)

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

Output:





Insights:

Left Plot: Grade vs Loan Status

- Grades range from **A (best credit rating)** to **G (worst)**.
- **Fully Paid loans dominate in higher grades** (A, B, C), indicating that borrowers with good credit grades are more likely to repay loans.
- As we move from **Grade A to Grade G**, the **proportion of Charged Off loans increases**, especially visible in Grades E, F, and G.

- This shows a **strong inverse relationship between loan grade and loan repayment performance**.

Right Plot: Sub-Grade vs Loan Status

- The sub-grade breakdown (e.g., A1–G5) gives a more **granular view of borrower risk**.
- Sub-grades **B2 to B5 and C1 to C5** have **higher loan volumes**, and still mostly show good repayment trends.
- Starting from **sub-grade D2 onwards**, there's a **visible increase in loan defaults**.
- Sub-grades **F3 to G5** have a **notably higher proportion of Charged Off loans**, making them **higher risk categories**.

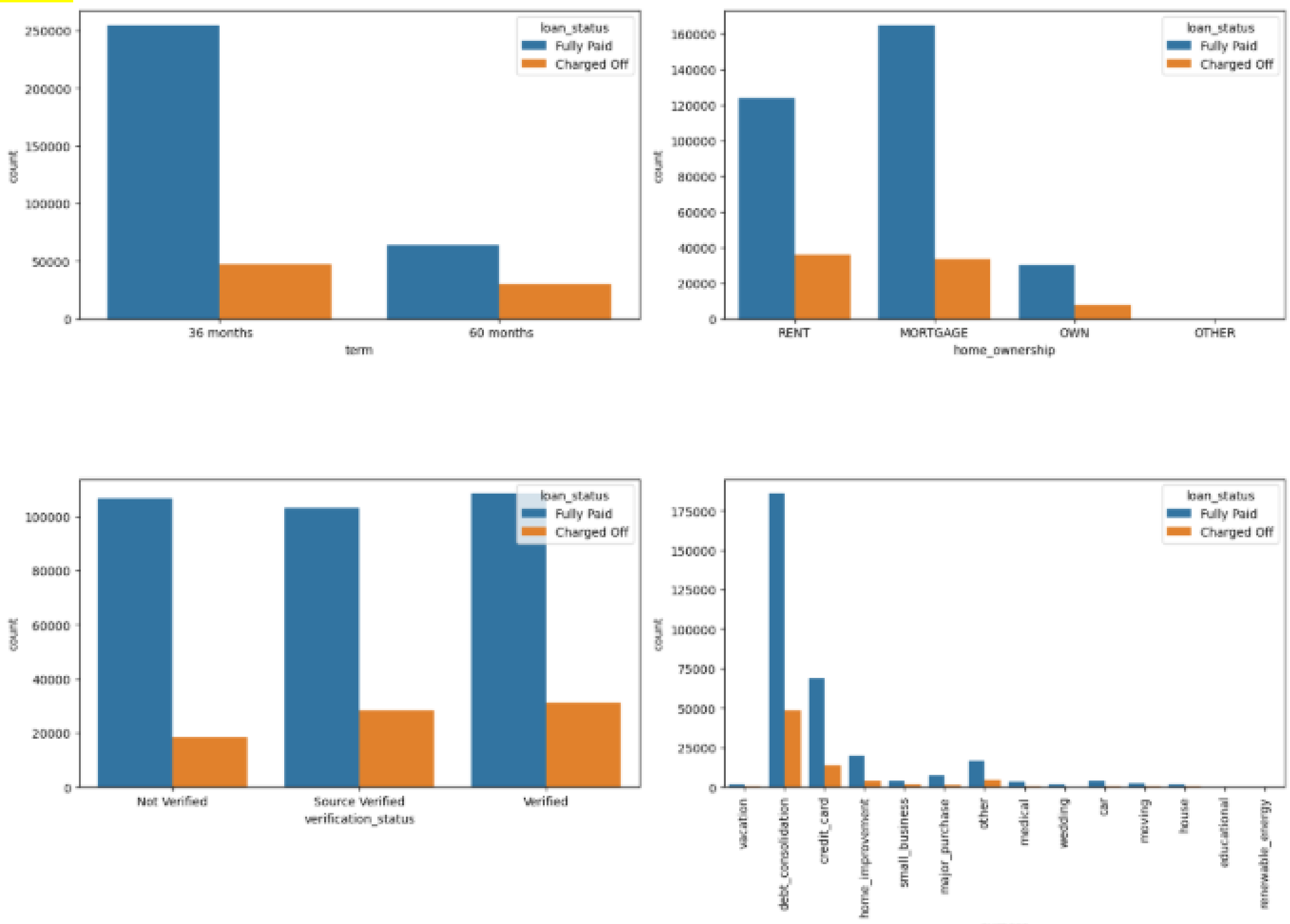
Recommendation

For risk mitigation, it's advisable to be cautious while issuing loans to borrowers in **lower grades or sub-grades beyond D3**.

3.2 Loan Status Distribution Across Different Categorical Features

This set of bar plots helps to **understand how the loan repayment behavior (Fully Paid vs Charged Off)** varies across different borrower-related categorical variables. It enables lenders to **identify patterns and risk indicators** based on term length, home ownership, verification status, and purpose of loan.

Code:



Insights:

Term: 60-month loans have a higher default rate **than 36-month loans.**

Home Ownership: Renters **show a slightly** higher risk of default **than owners or mortgage holders.**

Verification Status: Not Verified **borrowers have the** highest default rate**.**

Purpose: Most loans are for debt consolidation **and** credit cards**.**

Higher defaults **seen in loans for** small business, medical, and renewable energy**.**

Recommendation

This visual analysis helps identify borrower segments with **higher default risk**, enabling lenders to tailor their risk mitigation strategies — such as adjusting interest rates, requiring more documentation, or setting stricter approval conditions based on term, ownership, and purpose.

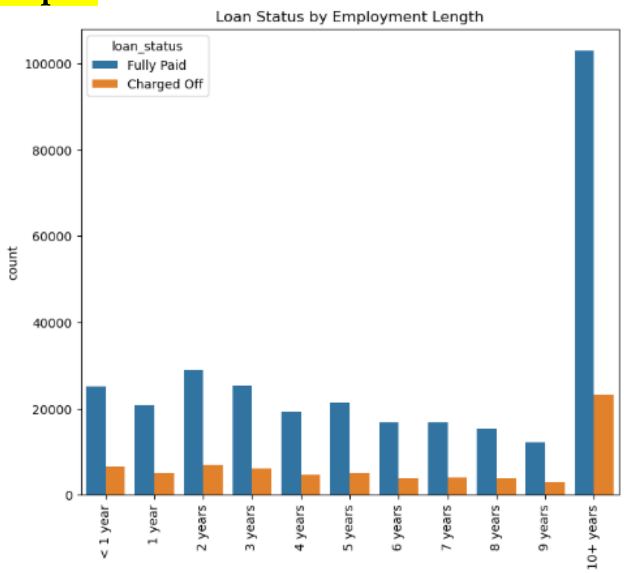
3.3 Analysis of Employment Length and Job Titles with Respect to Loan Status

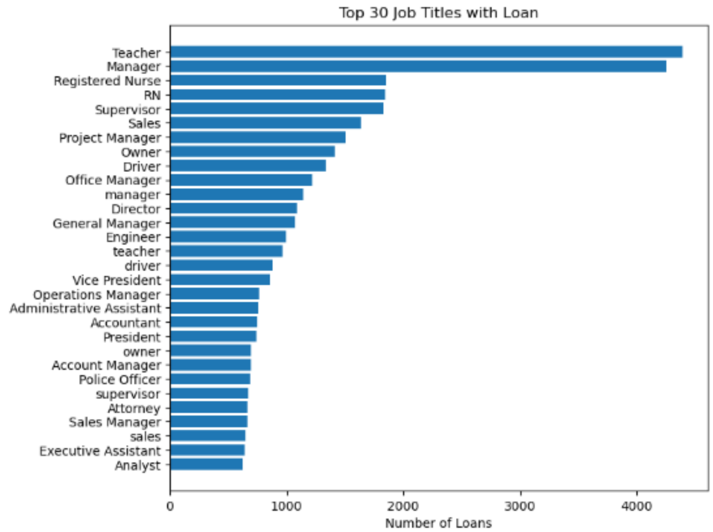
To understand how **employment duration** and **job roles** influence loan outcomes — whether loans are **Fully Paid** or **Charged Off**.

Code:

```
plt.figure(figsize=(15, 12))

# Plot 1: Employment Length vs Loan Status
plt.subplot(2, 2, 1)
order = ['< 1 year', '1 year', '2 years', '3 years', '4 years', '5 years',
         '6 years', '7 years', '8 years', '9 years', '10+ years']
g = sns.countplot(x='emp_length', data=df, hue='loan_status', order=order)
g.set_xticklabels(g.get_xticklabels(), rotation=90)
plt.title('Loan Status by Employment Length')

# Plot 2: Top 30 Job Titles
plt.subplot(2, 2, 2)
top_jobs = df['emp_title'].value_counts().nlargest(30)
plt.barh(top_jobs.index[::-1], top_jobs.values[::-1])  # reverse for top-down display
plt.title("Top 30 Job Titles with Loan")
plt.xlabel("Number of Loans")

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

Output:





Insights:

Left Plot→ Loan Status by Employment Length

- **Most loans** (both fully paid and charged off) come from people with **10+ years of employment**.
- Across all employment durations, the number of **fully paid loans** is consistently **higher** than charged-off ones.
- **Shorter employment (<1 year)** has relatively **fewer loans**, but the default (charged off) rate looks proportionally higher compared to longer tenures.

Right Plot → Top 30 Job Titles with Loan

- **Teacher** and **Manager** are the most common job titles among borrowers.
- Job roles related to **healthcare (e.g., Registered Nurse, RN)** and **management (e.g., Project Manager, Office Manager)** are also frequent.
- These insights help identify **common borrower profiles**.

Recommendations:

Caution with Short-Term Employment

- Applicants with **< 1 year** or **1–3 years** of employment have a **relatively higher default risk**.
- Add **stricter eligibility checks** for these segments (e.g., income stability, credit history).

Target Safe Professions

- Roles like **Teacher**, **Manager**, and **Nurse** appear frequently and have **high repayment potential**.
- Customize **loan products or marketing campaigns** for these job profiles.

4. Feature Engineering

4.1 Creating Binary Flags for Risk Indicators

In this step of feature engineering, we are creating **binary flags** from three numerical features: pub_rec, mort_acc, and pub_rec_bankruptcies. These features originally represent counts—such as the number of public derogatory records or the number of mortgage accounts—but we're transforming them into simple indicators (0 or 1) to signify the **presence or absence** of a potential risk factor. For example, if a person has even one bankruptcy (pub_rec_bankruptcies > 0), the function will return 1; otherwise, it returns 0. This approach simplifies the feature while still preserving the most important information.

Code:

```
def pub_rec(number):
    if number == 0.0:
        return 0
    else:
        return 1 # Whether someone has public derogatory records or not (flag)

def mort_acc(number):
    if number == 0.0:
        return 0
    else:
        return 1


def pub_rec_bankruptcies(number):
    if number == 0.0:
        return 0
    else:
        return 1
```

```
df['pub_rec'] = df.pub_rec.apply(pub_rec)
df['mort_acc'] = df.mort_acc.apply(mort_acc)
df['pub_rec_bankruptcies'] = df.pub_rec_bankruptcies.apply(pub_rec_bankruptcies)
```

4.2 Mapping of target variable

```
# Mapping of target variable -
df['loan_status'] = df.loan_status.map({'Fully Paid':0, 'Charged Off':1})
```

4.3 Null Values

```
df.isnull().sum()
```

```
loan_amnt                   0
term                        0
int_rate                    0
grade                       0
sub_grade                   0
emp_title               22927
emp_length              18301
home_ownership              0
annual_inc                  0
verification_status         0
issue_d                     0
loan_status                 0
purpose                     0
title                    1756
dti                         0
earliest_cr_line            0
open_acc                    0
pub_rec                     0
revol_bal                   0
revol_util                276
total_acc                   0
initial_list_status         0
application_type            0
mort_acc                    0
pub_rec_bankruptcies        0
address                     0
dtype: int64
```

4.4 Droping the null values

For high-cardinality or low-impact features with minimal missing data, it's more practical and cleaner to drop the nulls rather than impute with potentially misleading or non-informative values.

```
# Dropping rows with null values -
df.dropna(inplace=True)
```

```
loan_amnt               0
term                    0
int_rate                0
grade                   0
sub_grade               0
emp_title               0
emp_length              0
home_ownership          0
annual_inc              0
verification_status     0
issue_d                 0
loan_status             0
purpose                 0
title                   0
dti                     0
earliest_cr_line        0
open_acc                0
pub_rec                 0
revol_bal               0
revol_util              0
total_acc               0
initial_list_status     0
application_type        0
mort_acc                0
pub_rec_bankruptcies    0
address                 0
dtype: int64
```

Shape of data after handling null values

```
df.shape
```

```
(371125, 26)
```

5. Outlier Detection & Treatment

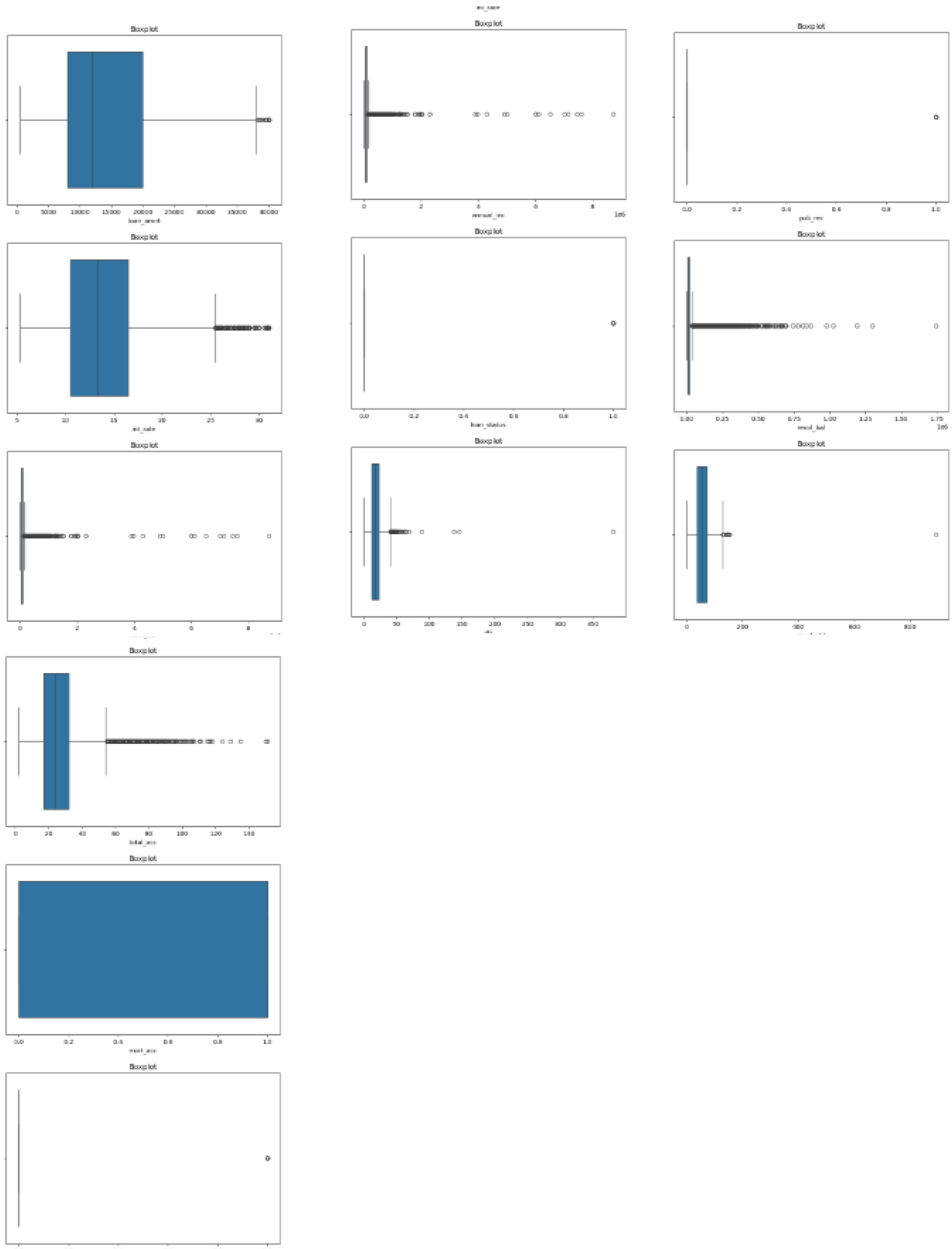
5.1 Identifying Numerical Features for Outlier Detection and Treatment

To perform **Outlier Detection and Treatment**, we first needed to **isolate the numerical columns** in the dataset, because outlier detection techniques only apply to **numeric data** (not text or categorical columns).

Code:

```
numerical_data = df.select_dtypes(include='number')
num_cols = numerical_data.columns
len(num_cols)
```

Output:



Outlier Treatment

To handle outliers in the dataset, we applied the **Z-Score method** across all numerical columns. Here's how it was done:

1. **Iterated through each numerical feature** (selected earlier using select_dtypes).
2. For each column:
   - Calculated the **mean** and **standard deviation**.
   - Defined an **upper limit** as mean + 3 * std and a **lower limit** as mean - 3 * std.
3. **Filtered the dataset** to keep only the rows where the column values lie within this range (i.e., within 3 standard deviations from the mean).
4. This process effectively **removed extreme values** (outliers), which could skew the model's learning and reduce prediction performance.

Why this works:

The 3-sigma rule (or empirical rule) assumes that in a normally distributed dataset, ~99.7% of data falls within 3 standard deviations. So, values beyond this range are highly likely to be anomalies or outliers.

Code:

```
for col in num_cols:
    mean = df[col].mean()
    std = df[col].std()

    upper_limit = mean + 3 * std
    lower_limit = mean - 3 * std

    df = df[(df[col] < upper_limit) & (df[col] > lower_limit)]
df.shape
```

6. Encoding Data into numerical features

```
# Term -
df.term.unique()
```

```
array([' 36 months', ' 60 months'], dtype=object)
```

```
term_values = {' 36 months': 36, ' 60 months': 60}
df['term'] = df.term.map(term_values)
```

```
print(df['term'])
```

```
0         36
1         36
2         36
3         36
4         60
          ..
396025    60
396026    36
396027    36
396028    60
396029    36
Name: term, Length: 338811, dtype: int64
```

```
# Initial List Status -
df['initial_list_status'].unique()
```

```
array(['w', 'f'], dtype=object)
```

```
list_status = {'w': 0, 'f': 1}
df['initial_list_status'] = df.initial_list_status.map(list_status)
```

```
print(df['initial_list_status'])
```

```
0         0
1         1
2         1
3         1
4         1
          ..
396025    0
396026    1
396027    1
396028    1
396029    1
Name: initial_list_status, Length: 338811, dtype: int64
```

```
# Let's fetch ZIP from address and then drop the remaining details -
df['zip_code'] = df.address.apply(lambda x: x[-5:])
```

```
print(df['zip_code'])
```

```
0         22690
1         05113
2         05113
3         00813
4         11650
          ...
396025    30723
396026    05113
396027    70466
396028    29597
396029    48052
Name: zip_code, Length: 338811, dtype: object
```

```
df['zip_code'].value_counts(normalize=True)
```

```
zip_code
70466    0.143942
30723    0.142776
22690    0.142664
48052    0.141415
00813    0.115929
29597    0.115182
05113    0.115138
93700    0.027768
11650    0.027750
86630    0.027437
Name: proportion, dtype: float64
```

Droping the features which we can let go for now

```
df.drop(columns=['issue_d', 'emp_title', 'title', 'sub_grade',
                 'address', 'earliest_cr_line', 'emp_length'],
                 axis=1, inplace=True)
```

One Hot Encoding

```
df.columns = df.columns.str.strip()  # remove leading/trailing spaces
for col in dummies:
    df[col] = df[col].astype(str)

dummies = ['purpose', 'zip_code', 'grade', 'verification_status', 'application_type', 'home_ownership']
df = pd.get_dummies(df, columns=dummies, drop_first=True)
```

Final output

```
df.head()
```

	loan_amnt	term	int_rate	annual_inc	loan_status	dti	open_acc	pub_rec	revol_bal	revol_util	...	grade_E	grade_F	grade_G	verification_status_Source Verified	v
0	10000.0	36	11.44	117000.0	0	26.24	16.0	0	36369.0	41.8	...	0	0	0	0	
1	8000.0	36	11.99	65000.0	0	22.05	17.0	0	20131.0	53.3	...	0	0	0	0	
2	15600.0	36	10.49	43057.0	0	12.79	13.0	0	11987.0	92.2	...	0	0	0	1	
3	7200.0	36	6.49	54000.0	0	2.60	6.0	0	5472.0	21.5	...	0	0	0	0	
4	24375.0	60	17.27	55000.0	1	33.95	13.0	0	24584.0	69.8	...	0	0	0	0	

5 rows × 49 columns

The data is encoded now ,it is ready for model building now

7. Model Building

```
X = df.drop('loan_status', axis=1) # Inplace not equal to True
y = df['loan_status']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,stratify=y, random_state=42)
```

```
print(X_train.shape)
print(X_test.shape)
```

```
(237167, 48)
(101644, 48)
```

```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
logreg = LogisticRegression(max_iter=1000)
logreg.fit(X_train, y_train)
```

```
LogisticRegression
LogisticRegression(max_iter=1000)
```

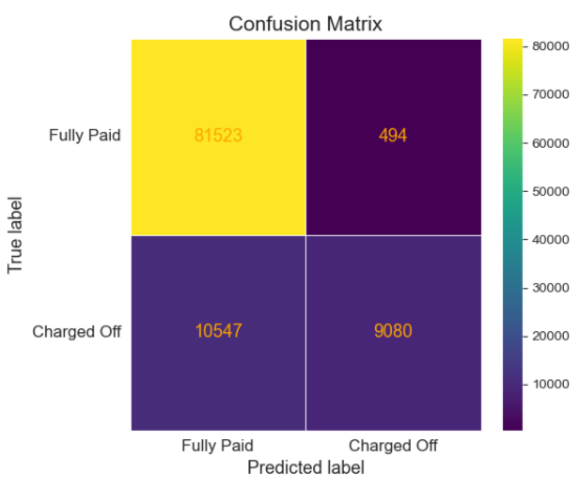
```
y_pred = logreg.predict(X_test)
print('Accuracy of Logistic Regression Classifier on test set: {:.3f}'.format(logreg.score(X_test, y_test)))
```

```
Accuracy of Logistic Regression Classifier on test set: 0.891
```

Accuracy 89.1%

```
confusion_matrix = confusion_matrix(y_test, y_pred)
print(confusion_matrix)
```

```
[[81523   494]
 [10547  9080]]
```




```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.99      0.94     82017
           1       0.95      0.46      0.62     19627

    accuracy                           0.89    101644
   macro avg       0.92      0.73      0.78    101644
weighted avg       0.90      0.89      0.88    101644
```

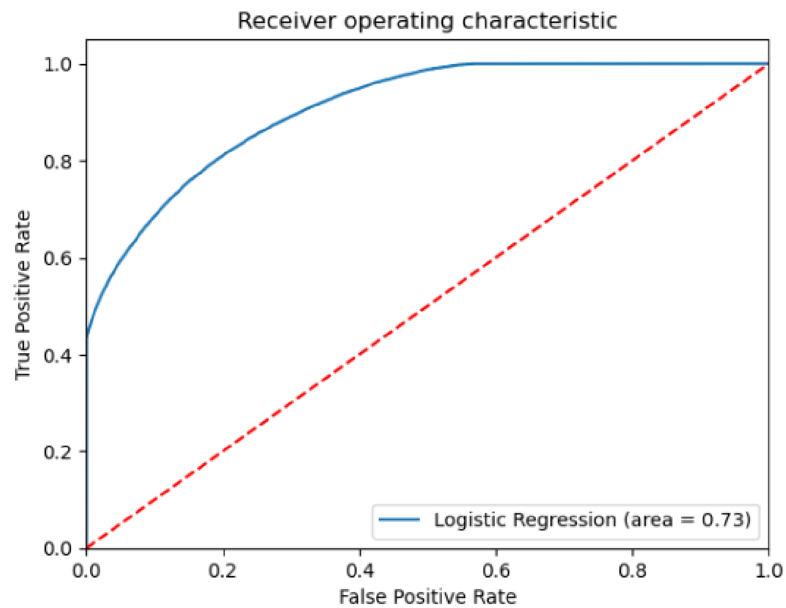
Insights:
What's Good:

- **High accuracy (89.1%)**
- **Very high precision (94.84%)** – when the model says someone is creditworthy, it is correct most of the time.
- This is **important for banks/lenders** – they don't want to approve bad loans.

What to Improve:

- **Low recall (46.28%)** – the model misses a lot of actually creditworthy people (false negatives).
- This could lead to **many good customers being rejected**, which is bad for business.

ROC-Curve:

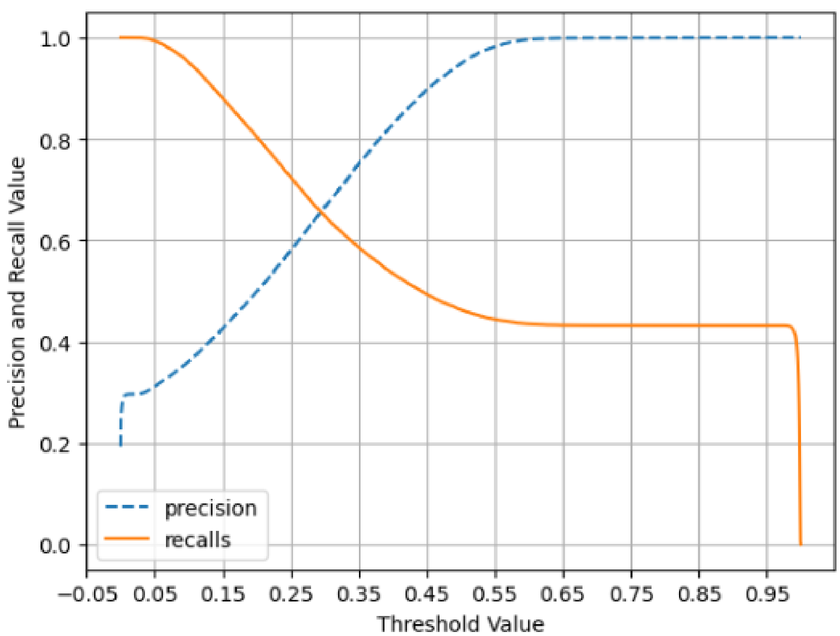


The ROC curve shows that your logistic regression model is moderately effective in distinguishing between the classes. With an AUC of 0.73, it's performing better than chance and gives a good baseline to improve from.

7.2 Precision-Recall Curve Across Thresholds for Binary Classification

```
def precision_recall_curve_plot(y_test, pred_proba_c1):
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)

    threshold_boundary = thresholds.shape[0]
    # plot precision
    plt.plot(thresholds, precisions[0:threshold_boundary], linestyle='--', label='precision')
    # plot recall
    plt.plot(thresholds, recalls[0:threshold_boundary], label='recalls')

    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))

    plt.xlabel('Threshold Value'); plt.ylabel('Precision and Recall Value')
    plt.legend(); plt.grid()
    plt.show()

precision_recall_curve_plot(y_test, logreg.predict_proba(X_test)[:,1])
```

This code defines a function called precision_recall_curve_plot() that visualizes how **precision** and **recall** change with different **classification threshold values** for a binary classifier—specifically, a **logistic regression model** in this case.



7.3 Multicollinearity Check using the Variance Inflation Factor (VIF)

The ROC curve for our logistic regression model indicates a moderate discriminative performance, with an AUC (Area Under the Curve) of 0.73. This suggests the model is able to distinguish between fully paid and charged-off loans better than random guessing, but there is still room for improvement. To enhance the model's predictive ability—particularly focusing on precision and recall—we are carefully analyzing feature multicollinearity using the **Variance Inflation Factor (VIF)**. Features with a VIF value greater than 10 are typically considered highly collinear and can distort the interpretation and performance of the model. Therefore, we are retaining only those features with **VIF less than 10**, ensuring that the model is not affected by redundant or highly correlated predictors. This process helps to simplify the model, reduce overfitting, and improve its generalization capabilities, ultimately aiming to boost both precision (correct identification of charged-off loans) and recall (minimizing false negatives). This step is essential for improving the reliability of predictions in high-stakes financial decision-making.

```python
def calc_vif(X):
    # Calculating the VIF
    vif = pd.DataFrame()
    vif['Feature'] = X.columns
    vif['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    vif['VIF'] = round(vif['VIF'], 2)
    vif = vif.sort_values(by='VIF', ascending = False)
    return vif

calc_vif(X)[:5]
```

	Feature	VIF
43	application_type_INDIVIDUAL	160.66
2	int_rate	123.09
14	purpose_debt_consolidation	50.44
1	term	27.68
13	purpose_credit_card	18.16

Insights:

application_type_INDIVIDUAL exhibited an extremely high VIF score of **160.66**, far exceeding the commonly accepted threshold of 10. This indicates strong multicollinearity with other features, which can adversely impact the stability and accuracy of the model. Therefore, to mitigate multicollinearity and improve overall model performance—particularly with respect to **precision and recall**—I decided to **remove application_type_INDIVIDUAL and all rest features has high multicollinearity** from the feature set before proceeding with model training and evaluation.

```
X.drop(columns=['application_type_INDIVIDUAL'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
2	int_rate	102.89
14	purpose_debt_consolidation	27.72
1	term	24.88
5	open_acc	14.06
9	total_acc	12.29

```
X.drop(columns=['int_rate'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
1	term	23.86
13	purpose_debt_consolidation	22.62
4	open_acc	13.95
8	total_acc	12.29
2	annual_inc	9.47

```
X.drop(columns=['term'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
12	purpose_debt_consolidation	18.99
3	open_acc	13.95
7	total_acc	12.26
1	annual_inc	9.46
6	revol_util	9.34

```
X.drop(columns=['purpose_debt_consolidation'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
3	open_acc	13.35
7	total_acc	12.24
1	annual_inc	8.96
6	revol_util	8.63
2	dti	7.55

```
X.drop(columns=['open_acc'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
1	annual_inc	8.81
5	revol_util	8.26
6	total_acc	8.17
2	dti	6.99
0	loan_amnt	6.88

We got finally the satisfied VIF for features


```
X.drop(columns=['open_acc'], axis=1, inplace=True)
calc_vif(X)[:5]
```

	Feature	VIF
1	annual_inc	8.81
5	revol_util	8.26
6	total_acc	8.17
2	dti	6.99
0	loan_amnt	6.88

7.4 Model Evaluation Using K-Fold Cross-Validation (Post VIF Feature Selection)

After completing the **VIF analysis** and removing highly collinear features, the dataset is **scaled using StandardScaler**, which standardizes the feature values to have a mean of 0 and a standard deviation of 1—crucial for models like Logistic Regression that are sensitive to feature scale.

Following scaling, the model is evaluated using **K-Fold Cross-Validation** with 5 splits (n_splits=5). This technique divides the data into 5 equal parts, trains the model on 4 parts, and tests it on the remaining part, rotating this process across all splits. The **cross_val_score** function is used to compute accuracy in each fold, and the final printed result is the **mean cross-validated accuracy** across all 5 folds.

Code:
X = scaler.fit_transform(X)

kfold = KFold(n_splits=5)
accuracy = np.mean(cross_val_score(logreg, X, y, cv=kfold, scoring='accuracy', n_jobs=-1))
print("Cross Validation accuracy: {:.3f}".format(accuracy))

Output:

```
Cross Validation accuracy: 0.891
```

7.5 Addressing Class Imbalance Using SMOTE (Synthetic Minority Oversampling Technique)

After performing **cross-validation**, which yielded a promising accuracy of **89.1%**, we proceeded to analyze the class distribution in the training data. Despite the high accuracy, further inspection revealed a **class imbalance**, where the number of samples in one class (e.g., 'Fully Paid') significantly outnumbered the other (e.g., 'Charged Off'). This imbalance can cause the model to favor the majority class and perform poorly on the minority class, impacting **recall and precision**, especially for the underrepresented class.

To tackle this issue, we applied **SMOTE (Synthetic Minority Oversampling Technique)**, a technique that synthetically generates new samples for the minority class by interpolating between existing examples. The following line of code:

Code:
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train.ravel())

```
print('After OverSampling, the shape of train_X: {}'.format(X_train_res.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_res.shape))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_res == 1)))
print("After OverSampling, counts of label '0': {}".format(sum(y_train_res == 0)))
```

```
After OverSampling, the shape of train_X: (382742, 48)
After OverSampling, the shape of train_y: (382742,)

After OverSampling, counts of label '1': 191371
After OverSampling, counts of label '0': 191371
```

The final classification report I got

```
# Classification Report
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.95      0.80      0.87     82017
           1       0.49      0.81      0.61     19627

    accuracy                           0.80    101644
   macro avg       0.72      0.80      0.74    101644
weighted avg       0.86      0.80      0.82    101644
```

Model Comparison:

In evaluating the performance of two logistic regression models for classifying individuals as **creditworthy (0)** or **non-creditworthy (1)**, observed the following key differences:

Model 1 Overview:

- **High Precision & Recall for class 0 (creditworthy)** – Performs very well in identifying creditworthy customers (F1-score: 0.94).
- **Poor Recall for class 1 (non-creditworthy)** – Only 46% of the actual non-creditworthy customers are correctly identified (Recall: 0.46).
- **Overall accuracy is high (0.89)**, but heavily biased towards the majority class (class 0).

Model 2 Overview:

- **Balanced Recall across both classes** – Model 2 achieves **Recall of 0.81 for class 1**, significantly better than Model 1 (46%). This means **it can detect far more non-creditworthy customers**, which is critical in credit risk scenarios.
- **F1-score for class 1 improved from 0.62 to 0.61**, showing a more balanced performance even with a trade-off in precision.
- **Lower overall accuracy (0.80)** compared to Model 1, but more **practical for risk management**, since identifying high-risk individuals (class 1) is a priority.

Why Model 2 Is Better for This Use Case:

While **Model 1** has a higher overall accuracy, it performs poorly in identifying **non-creditworthy** individuals (low recall for class 1). In financial and lending scenarios, **failing to detect high-risk applicants can lead to significant financial losses**.

Model 2, despite a lower overall accuracy, has **much higher recall for class 1 (81%)**, which means it catches more potential defaulters. This trade-off is acceptable and even preferable in credit scoring, where **false negatives (undetected non-creditworthy customers)** are far more costly than false positives.

Conclusion:

Model 2 is more suitable for deployment in the credit risk assessment pipeline due to its **better identification of high-risk applicants**, even at the cost of slightly lower accuracy and precision. This shift in performance focus ensures more effective risk management and better decision-making.

Jupyter Notebook Analysis

For a detailed view of the full analysis, including code, visualizations please refer to the complete Jupyter notebook available in the PDF format. The notebook documents each step of the analysis process, from data exploration to the final recommendations.

You can access the Jupyter notebook PDF through the **following link**:

Loantap Analysis