P2P CREDIT RISK ANALYSIS ON LENDING CLUB PLATFORM USING MACHINE LEARNING CLASSIFICATION ALGORITHMS

TABLE OF FIGURE

TABLE OF CONTENTS

I. IDENTIFICATION OF DATA MINING PROBLEMS

1. Reason for choosing the topic

The burgeoning field of peer-to-peer (P2P) lending has significantly altered the landscape of financial services, providing a novel avenue for individuals and small businesses to secure funding outside traditional banking systems. Platforms like Lending Club have democratized access to credit, offering streamlined processes and reduced barriers for both borrowers and lenders. However, this innovation comes with inherent risks, particularly related to the accurate assessment and management of credit risk, which remains a critical challenge for the sustainability of these platforms.

Choosing the topic of credit risk analysis on the Lending Club platform using machine learning classification algorithms is driven by several compelling factors. Firstly, the dynamic and evolving nature of P2P lending demands advanced analytical methods to effectively evaluate borrower risk profiles and predict loan defaults. Traditional credit risk assessment techniques often fall short in addressing the diverse and complex data characteristics inherent in P2P lending, necessitating the application of sophisticated machine learning approaches.

Secondly, the wealth of data available from Lending Club provides an excellent opportunity to explore and implement various machine learning models. By analyzing features such as loan amount, interest rate, employment length, credit score, and payment history, we can gain deeper insights into the factors influencing credit risk. This not only aids in improving the accuracy of risk predictions but also enhances the overall risk management strategies for the platform.

Furthermore, the application of machine learning algorithms like Logistic Regression, Decision Trees, Naive Bayes, Random Forest, Gradient Boosting (XGBoost), and Support Vector Machine in this context offers the potential to significantly improve prediction outcomes. These models can handle the high dimensionality and complexity of the dataset, address class imbalances, and provide interpretable results for practical decision-making.

Finally, this research has substantial implications for both academic and practical applications. Academically, it contributes to the growing body of knowledge on the use of machine learning in financial risk assessment. Practically, it provides valuable insights and

tools for enhancing the operational efficiency and risk management capabilities of P2P lending platforms. By identifying the most effective model for credit risk prediction, this research aims to mitigate risks for lenders and promote the sustainable growth of P2P lending ecosystems.

2. About Dataset

The dataset used in this study is sourced from the Lending Club platform and is publicly available on Kaggle. It contains detailed information on loans issued between 2007 and 2019, with a total of 396,030 rows and 27 columns. This dataset provides comprehensive insights into borrower profiles, loan characteristics, and loan performance.

Dataset source: kaggle.com/datasets/adarshsng/lending-club-loan-data-csv

Dataset description

ID	Attribute	Description	Type
1	loan_amnt	The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.	Integer
2	term	The number of payments on the loan. Values are in months and can be either 36 or 60.	String
3	int_rate	Interest Rate on the loan	Float
4	installment	The monthly payment owed by the borrower if the loan originates.	Float

5	grade	LC assigned loan grade	String
6	sub_grade	LC assigned loan subgrade	String
7	emp_title	The job title supplied by the Borrower when applying for the loan.*	String
8	emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.	Sting
9	home_ownership	The home ownership status provided by the borrower during registration or obtained from the credit report. Our values are: RENT, OWN, MORTGAGE, OTHER	String
10	annual_inc	The self-reported annual income provided by the borrower during registration.	Float
11	verification_status	Indicates if income was verified by LC, not verified, or if the income source was verified	String
12	issue_d	The month which the loan was funded	Date
13	loan_status	Current status of the loan	String

14	purpose	A category provided by the borrower for the loan request.	String
15	title	The loan title provided by the borrower	String
16	dti	A ratio calculated using the borrower’s total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower’s self-reported monthly income.	Float
17	earliest_cr_line	The month the borrower's earliest reported credit line was opened	Date
18	open_acc	The number of open credit lines in the borrower's credit file.	Float
19	pub_rec	Number of derogatory public records	Float
20	revol_bal	Total credit revolving balance	Float
21	revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.	Float
22	total_acc	The total number of credit lines currently in the borrower's credit file	Float

23	initial_list_status	The initial listing status of the loan. Possible values are – W, F	String
24	application_type	Indicates whether the loan is an individual application or a joint application with two co-borrowers	String
25	mort_acc	Number of mortgage accounts.	Float
26	pub_rec_bankruptcies	Number of public record bankruptcies	Float
27	address	The address provided by the borrower in the loan application	String

3. Problem description

The evolution of peer-to-peer (P2P) lending platforms like Lending Club has revolutionized the financial industry by facilitating direct connections between borrowers and lenders. Despite the advantages of increased accessibility and efficiency, the primary challenge remains accurately assessing and managing credit risk. Traditional credit risk assessment methods often fall short in this dynamic environment, making it imperative to employ advanced data mining methodologies to analyze and predict loan default risks effectively.

This project aims to address the issue of credit risk analysis on the Lending Club platform using machine learning classification algorithms. By applying data mining methodologies, we seek to build robust models that can predict the likelihood of loan defaults based on a comprehensive dataset provided by Lending Club. The dataset includes a variety of features such as loan amount, interest rate, employment length, credit score, and payment history, which are crucial for understanding the risk profiles of borrowers.

In the context of this Data Mining course, we will employ a systematic approach to extract meaningful patterns and insights from the data. The methodology involves several key steps:

- Data Collection and Preprocessing: We will begin by collecting and preprocessing the Lending Club dataset to ensure data quality and integrity. This step includes handling missing values, normalizing data, and addressing class imbalances to prepare the data for analysis.
- Feature Selection and Engineering: We will perform feature selection to identify the most relevant variables that influence credit risk. Feature engineering techniques will be applied to create new features that can enhance the predictive power of our models.
- Models Building, Evaluation and Comparison: Various machine learning classification algorithms, including Logistic Regression, Decision Trees, Naive Bayes, Random Forest, Gradient Boosting (XGBoost), and Support Vector Machine, will be employed to build predictive models. We will use cross-validation and other evaluation metrics such as accuracy, precision, recall, F1 score, and AUC-ROC to assess the performance of each model. By evaluating each model's performance across various metrics, we will identify strengths and weaknesses, determining which model provides the best balance between predictive accuracy and interpretability.
- Model Interpretation and Optimization: To ensure the practical applicability of our models, we will interpret the results to identify key factors affecting credit risk. Hyperparameter tuning and model optimization techniques will be used to improve model performance and generalizability.
- Deployment and Visualization: Finally, we will deploy the best-performing model and create visualizations to present our findings. These visualizations will help stakeholders understand the insights derived from the data and make informed decisions regarding risk management.

By integrating data mining methodologies with machine learning classification algorithms, this project aims to provide a comprehensive solution for credit risk analysis

on the Lending Club platform. The outcomes of this research will enhance the platform's ability to manage risk, reduce loan defaults, and improve decision-making processes for both lenders and borrowers. The comparative analysis of models will further ensure that the most effective and efficient model is utilized, providing a robust framework for future risk assessment endeavors.

II. DATA PREPROCESSING PROCESS

1. Exploratory Data Analysis

Overall EDA Goal: Get an understanding for which variables are important, view summary statistics, and visualize the data

Data information & descriptive statistics

```
In [65]: df.info()

<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
```

Figure 1. Dataframe information

```
In [66]: df.describe()
```

Out[66]:

	loan_amnt	int_rate	installment	annual_inc	dti	open_acc	pub_rec	revol_bal	revol_util	total_acc	mort_acc	pub_rec_bankruptcies
count	396030.00	396030.00	396030.00	396030.00	396030.00	396030.00	396030.00	396030.00	395754.00	396030.00	358235.00	395495.00
mean	14113.89	13.64	431.85	74203.18	17.38	11.31	0.18	15844.54	53.79	25.41	1.81	0.12
std	8357.44	4.47	250.73	61637.62	18.02	5.14	0.53	20591.84	24.45	11.89	2.15	0.36
min	500.00	5.32	16.08	0.00	0.00	0.00	0.00	0.00	0.00	2.00	0.00	0.00
25%	8000.00	10.49	250.33	45000.00	11.28	8.00	0.00	6025.00	35.80	17.00	0.00	0.00
50%	12000.00	13.33	375.43	64000.00	16.91	10.00	0.00	11181.00	54.80	24.00	1.00	0.00
75%	20000.00	16.49	567.30	90000.00	22.98	14.00	0.00	19620.00	72.90	32.00	3.00	0.00
max	40000.00	30.99	1533.81	8706582.00	9999.00	90.00	86.00	1743266.00	892.30	151.00	34.00	8.00

Figure 2. Numerical data descriptive statistics

- **loan_status** (Target variable – current status of the loan)

  From the value counts of each status, we can see that this dataset have a little imbalance

  This shows that the majority of loans at Lending Club are fully paid, but there is still a small portion of loans that have been Charged off.

```
In [67]: # Plot value counts with labels of counts
         plt.figure(figsize=(6, 4))
         df['loan_status'].value_counts().plot(kind='bar', title='Loan Status Value Counts')
         df['loan_status'].value_counts()
```

```
Out[67]: loan_status
         Fully Paid     318357
         Charged Off     77673
         Name: count, dtype: int64
```

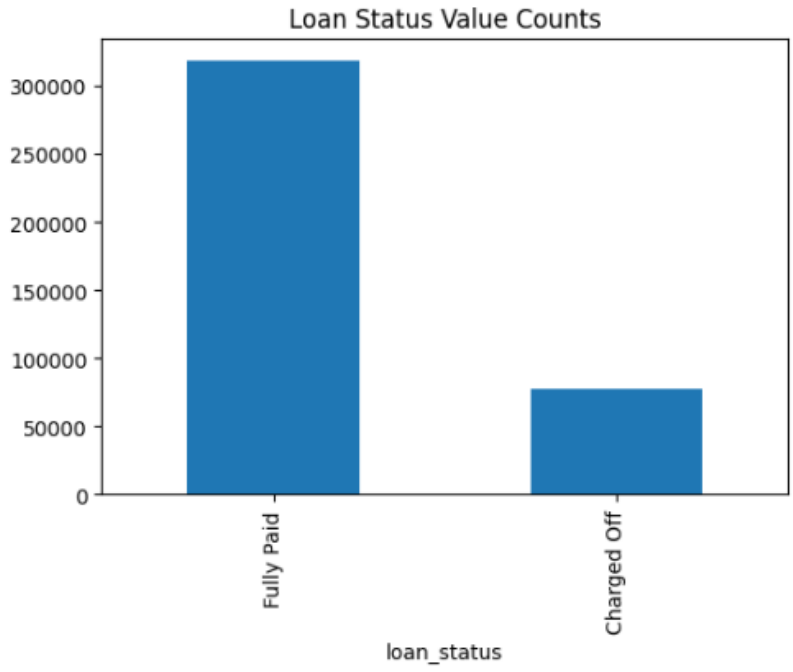

Figure 3. Loan Status values count

- **loan_amnt** & **installment**
    - The correlation between loan_amnt and installment very high, it might indicate that this is the strong relationship (high/low loan amount -> high/low installment)
    - We need to check that, is this relationship make a duplicate information in the data

```
In [68]: # Correlation matrix with numbers
plt.figure(figsize=(10, 4))
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='viridis', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```

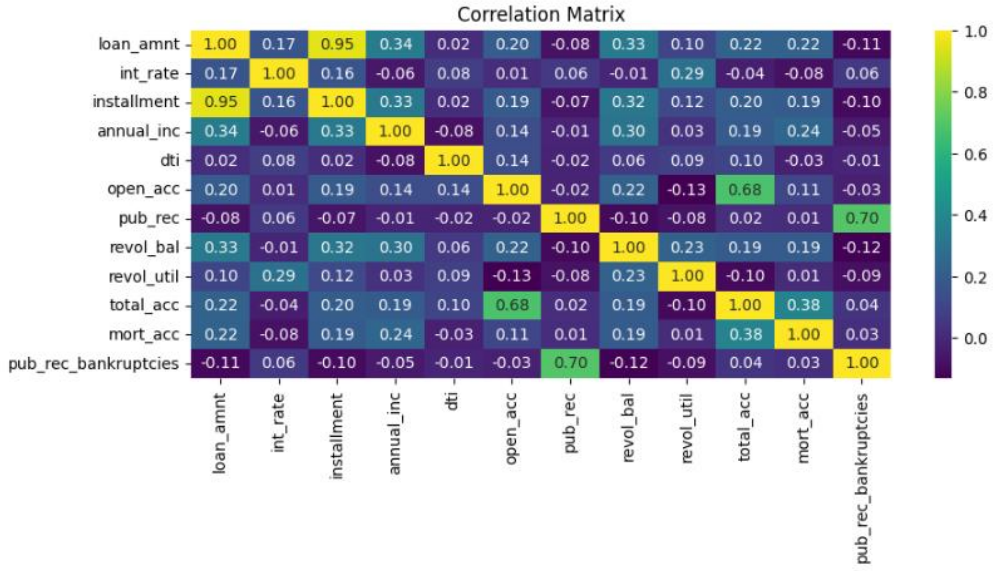

Figure 4. Correlation matrix between loan_amnt & installment

```
In [69]: # Installment and loan amount by status histogram
# Installment by loan status
plt.figure(figsize=(16, 4))
plt.subplot(1, 2, 1)
sns.histplot(data=df, x='installment', hue='loan_status', bins=30, kde=True)
plt.title('Installment by Loan Status')
plt.subplot(1, 2, 2)
sns.histplot(data=df, x='loan_amnt', hue='loan_status', bins=30, kde=True)
plt.title('Loan Amount by Loan Status')
plt.show()
```

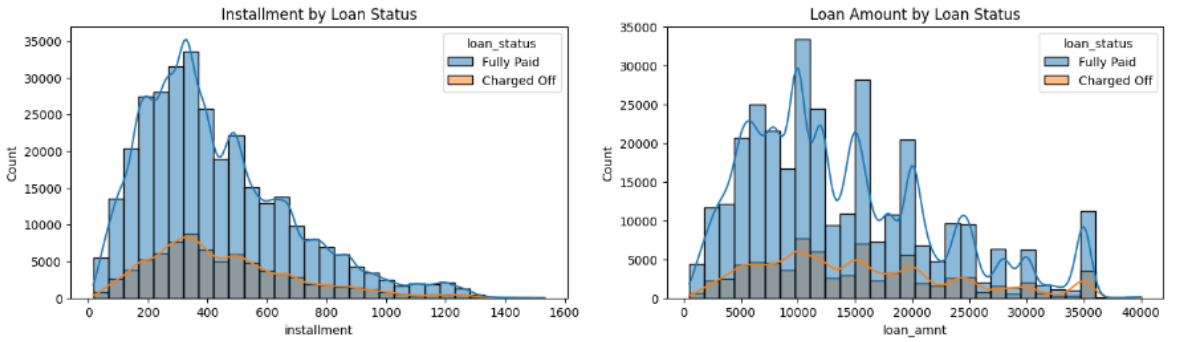

Figure 5. Installment and loan_amnt by status histogram

```
In [70]: # Boxplot of loan amount by loan status # Installment by loan status
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
sns.boxplot(data=df, x='loan_status', y='installment')
plt.title('Installment by Loan Status')
plt.subplot(1, 2, 2)
sns.boxplot(data=df, x='loan_status', y='loan_amnt')
plt.title('Loan Amount by Loan Status')
plt.tight_layout()
plt.show()
```

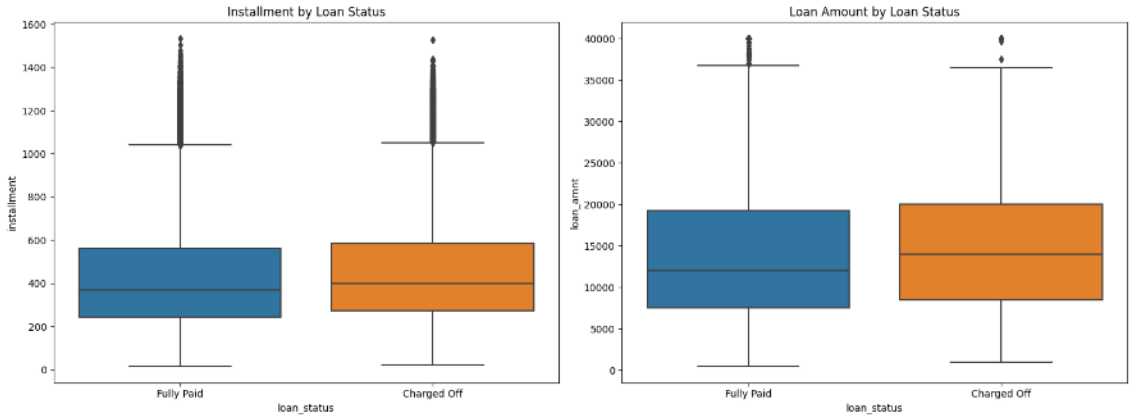

Figure 6. Installment and loan amount by status Boxplot

```
In [71]: df.groupby(by='loan_status')['loan_amnt'].describe()
```

Out[71]:

loan_status	count	mean	std	min	25%	50%	75%	max
Charged Off	77673.00	15126.30	8505.09	1000.00	8525.00	14000.00	20000.00	40000.00
Fully Paid	318357.00	13866.88	8302.32	500.00	7500.00	12000.00	19225.00	40000.00

Figure 7. Loan amount by status descriptive statistics

- From box-plot, histogram and descriptive statistics, it is clear that there is significant difference in loan amount and installment between fully paid and charged off loans.
- We can use these features to predict loan status.

- **grade** & **sub_grade**

```
In [72]: # Loan status by grade and subgrade in 2 plot in 1 frame
plt.figure(figsize=(20, 6))
plt.subplot(1, 2, 1)
sns.countplot(data=df, x='grade', hue='loan_status', order=sorted(df['grade'].unique()))
plt.title('Loan Status by Grade')
plt.subplot(1, 2, 2)
sns.countplot(data=df, x='sub_grade', hue='loan_status', order=sorted(df['sub_grade'].unique()))
plt.title('Loan Status by Sub Grade')
plt.show()
```

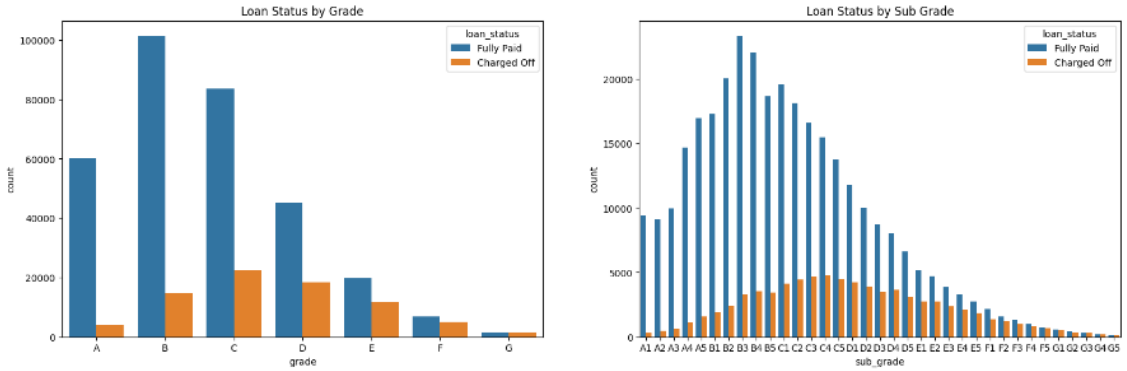

Figure 8. Distribution histogram of grade & subgrade for loan status

- **Term**, **home_ownership**, **verification_status** & **purpose**
  - Grouping all the value with same characteristics and get more explainable data on each attribute

```
In [73]: # Stack bar chart of loan status by home ownership
plt.figure(figsize=(16, 6))
sns.countplot(data=df, x='home_ownership', hue='loan_status')
plt.title('Loan Status by Home Ownership')
plt.show()
```

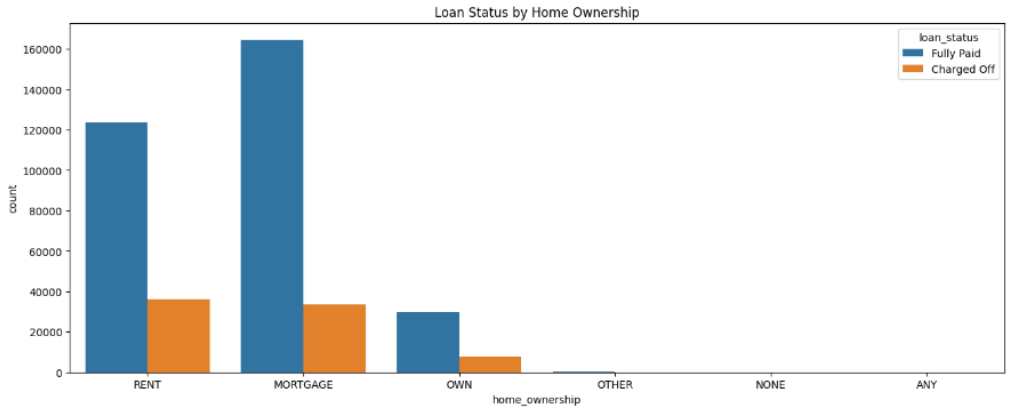

Figure 9. Histogram of Home Ownership by Loan status

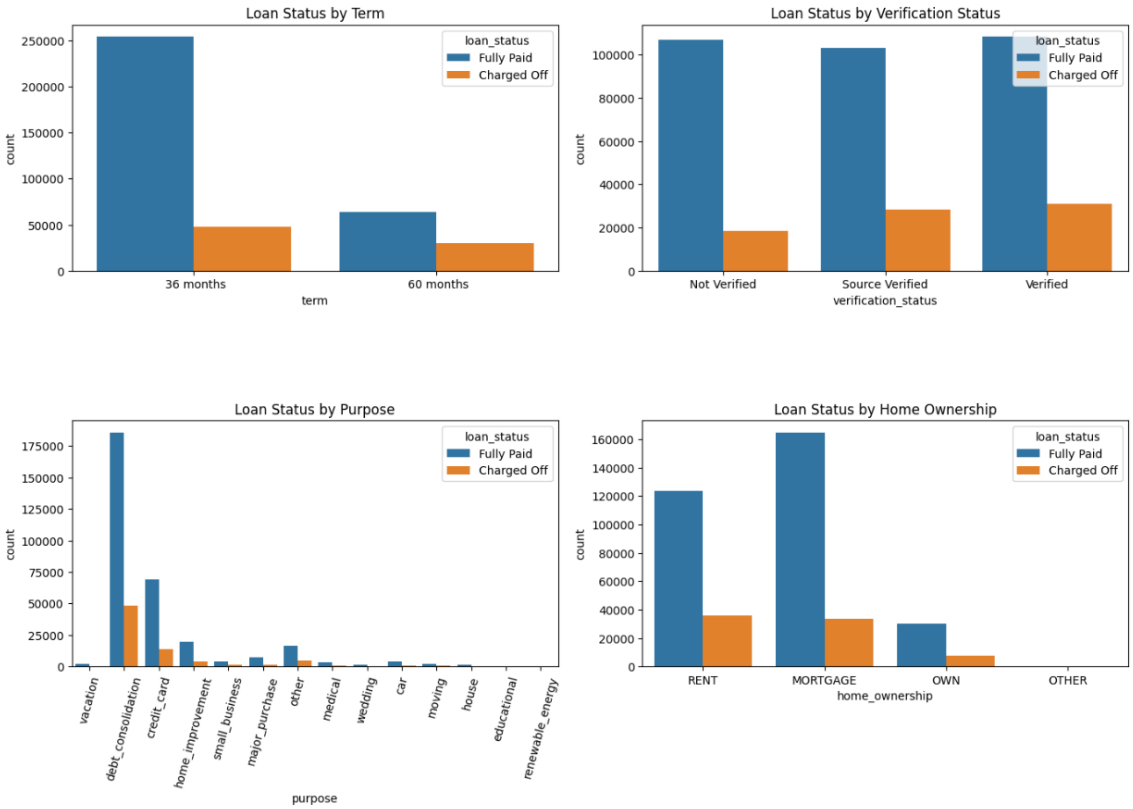

Figure 10. Histogram of term, verification_status, purpose and home_ownership by loan status

- int_rate & annual_inc

```
In [76]: # Plots with loan status by annual income and interest rate
plt.figure(figsize=(16, 6))
plt.subplot(2, 2, 1)
sns.histplot(data=df, x='int_rate', hue='loan_status', bins=30, kde=True)
plt.title('Interest Rate by Loan Status')
plt.subplot(2, 2, 2)
sns.histplot(data=df, x='annual_inc', hue='loan_status', bins=30, kde=True)
plt.title('Annual Income by Loan Status')
# Annual income by loan status (Annual income < 250000)
plt.subplot(2, 2, 3)
sns.histplot(data=df.loc[df['annual_inc'] < 250000], x='annual_inc', hue='loan_status', bins=30, kde=True)
plt.title('Annual Income by Loan Status (Annual Income < 250000)')
plt.subplot(2, 2, 4)
sns.histplot(data=df.loc[df['annual_inc'] < 1000000], x='annual_inc', hue='loan_status', bins=30, kde=True)
plt.title('Annual Income by Loan Status (Annual Income < 1000000)')
plt.tight_layout()
plt.show()
```

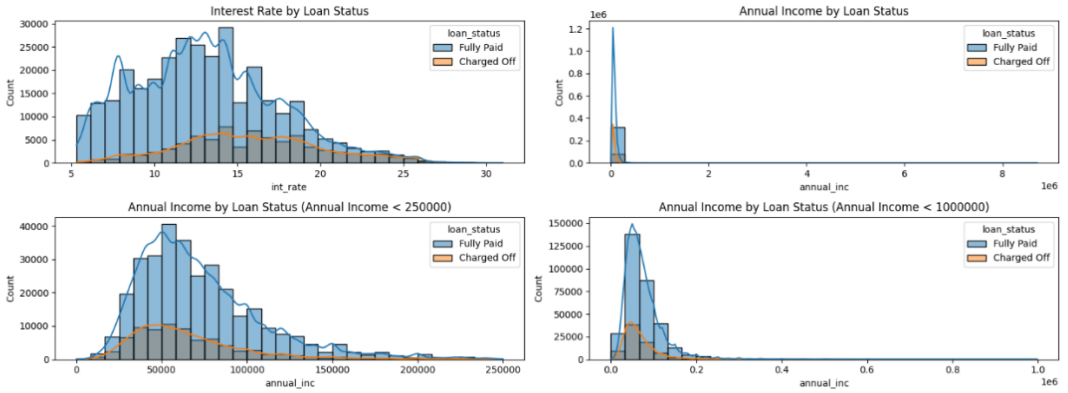

Figure 11. Distribution of continuous data (int_rate & annual_inc) by loan status

- It seems that loans with high intersect rate are more likely to be unpaid.
- Only 75 (less then) borrowers have an annual income more than 1 million, and 4077.

- emp_title & emp_length

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

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',]
plt.title("Loan status by employment length")
g = sns.countplot(x='emp_length', data=df, hue='loan_status', order=order)
g.set_xticklabels(g.get_xticklabels(), rotation=90);

plt.subplot(2, 2, 2)
plt.barh(df.emp_title.value_counts()[:30].index, df.emp_title.value_counts()[:30])
plt.title("The most 30 jobs title afforded a loan")
plt.tight_layout()
```

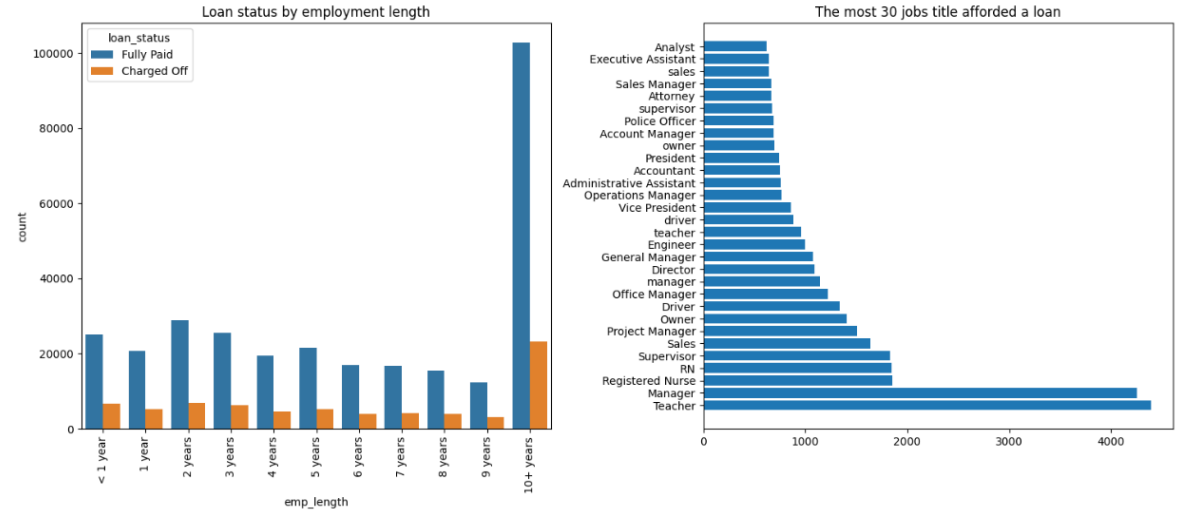

Figure 12. Histogram of emp_length by loan status, popular emp_tile values

- issue_d, earliest_cr_line

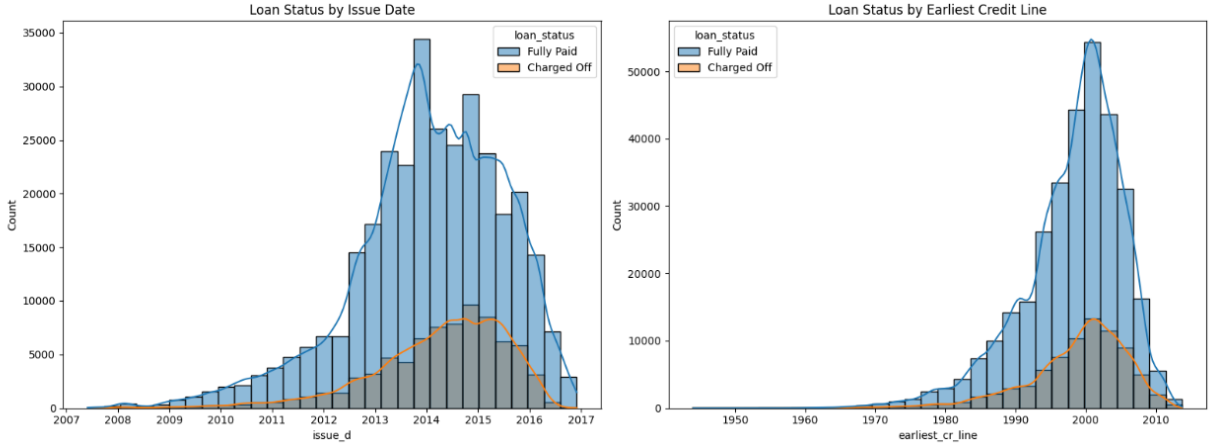

Figure 13. Histogram of Issue_d & earliest_cr_line by loan status

- **title** & **purpose**

```
In [81]: print(df['title'].value_counts())
         print("-"*50)
         print(df['purpose'].value_counts())
```

```
title
Debt consolidation            152472
Credit card refinancing        51487
Home improvement               15264
Other                          12930
Debt Consolidation             11608
                                ...
Graduation/Travel Expenses         1
Daughter's Wedding Bill            1
gotta move                         1
creditcardrefi                     1
Toxic Debt Payoff                  1
Name: count, Length: 48816, dtype: int64
--------------------------------------------------
purpose
debt_consolidation    234507
credit_card            83019
home_improvement       24030
other                  21185
major_purchase          8790
small_business          5701
car                     4697
medical                 4196
moving                  2854
vacation                2452
house                   2201
wedding                 1812
renewable_energy         329
educational              257
Name: count, dtype: int64
```

Figure 14. Value count of title & purpose

- ‘title’ will be removed because we have the purpose column with is generated from it.

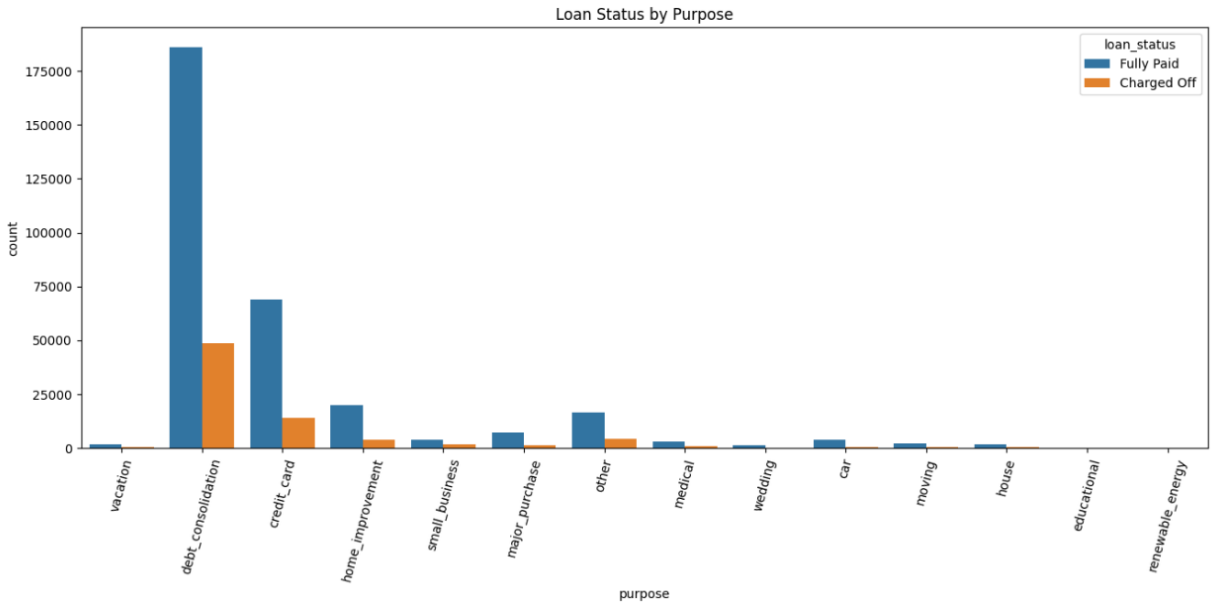

Figure 15. Histogram of Purpose by loan status

- **dti, open_acc, revol_bal, revol_util, & total_acc**

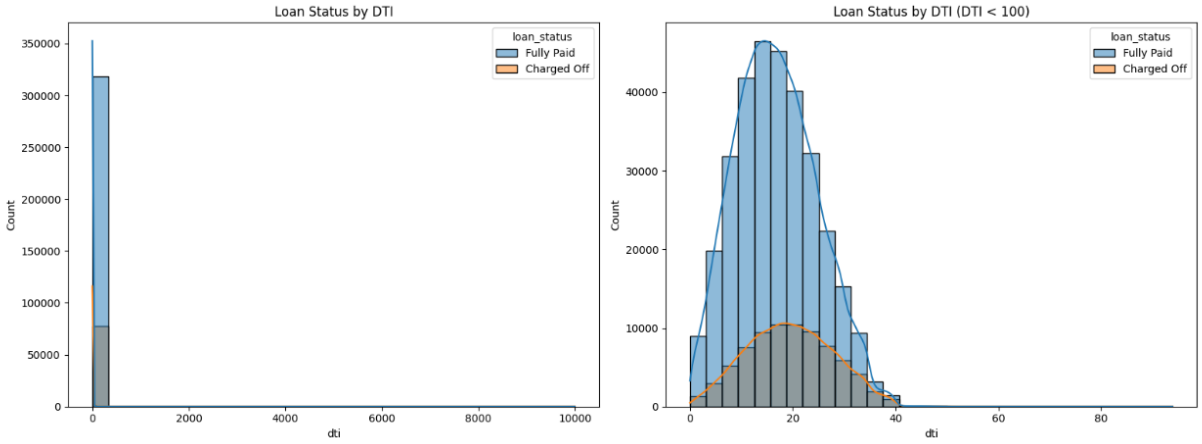

Figure 16. Distribution of Debt-to-Income ratio by loan status

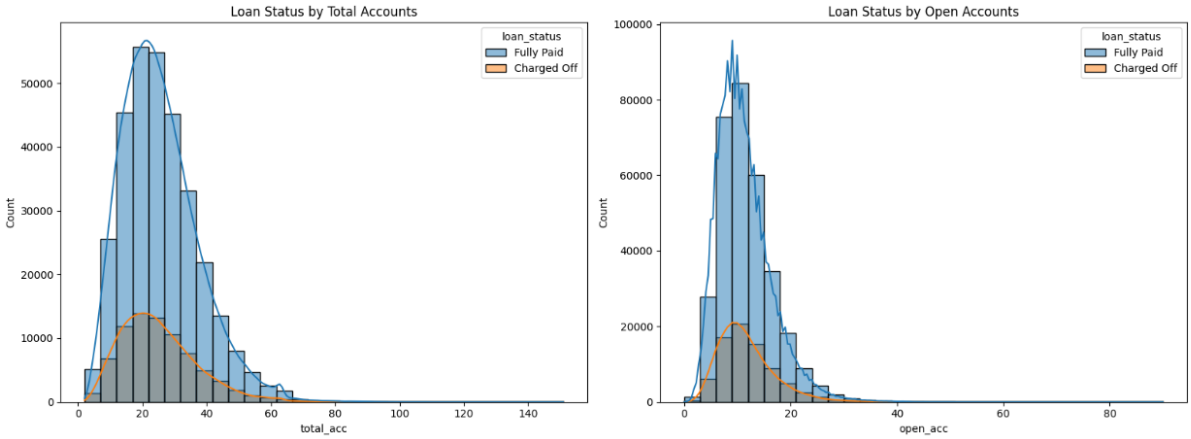

Figure 17. Distribution of Open_acc & total_acc by loan status

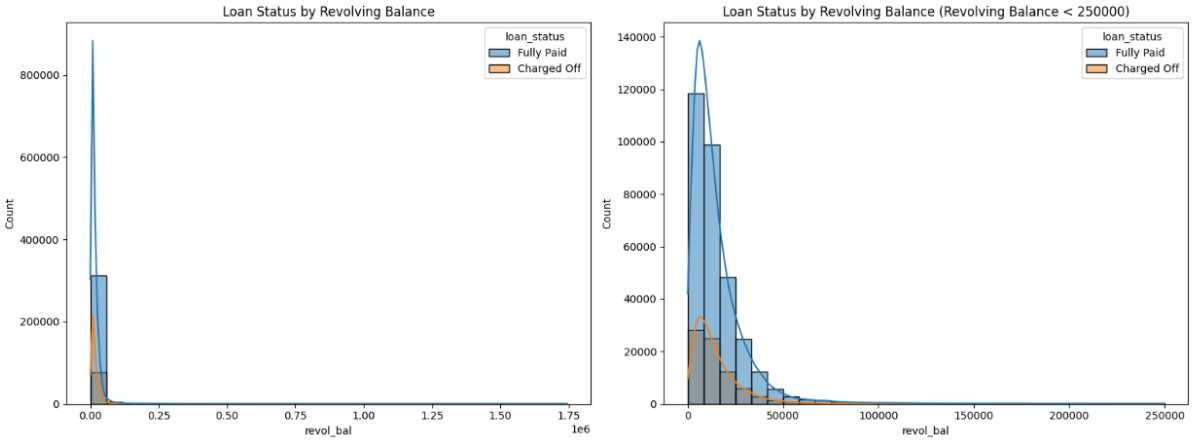

Figure 18. Distribution of Revolving balance by loan status

- Smaller the **dti** the more likely that the loan will not be paid.
- Only 217 borrowers have more than 40 open credit lines.
- Only 266 borrowers have more than 80 credit line in the borrower credit file.

- **pub_rec, initial_list_status, application_type, mort_acc, & pub_rec_bankruptcies**
    - Some features are represented for negative or positive signal of borrower profile, so we just need the flag value (0 & 1) of that feature, some features such as pub_rec (Number of derogatory public records), mort_acc (Number of mortgage accounts) and pub_rec_bankruptcies (Number of public record bankruptcies)
    - We process the value of these feature by the function below.

In [94]:
```
def pub_rec(number):
    if number == 0.0:
        return 0
    else:
        return 1

def mort_acc(number):
    if number == 0.0:
        return 0
    elif number >= 1.0:
        return 1
    else:
        return number

def pub_rec_bankruptcies(number):
    if number == 0.0:
        return 0
    elif number >= 1.0:
        return 1
    else:
        return number

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)
```

Figure 19. Preprcocess fuction for pub_rec, mort_acc & pub_rec_bankruptcies

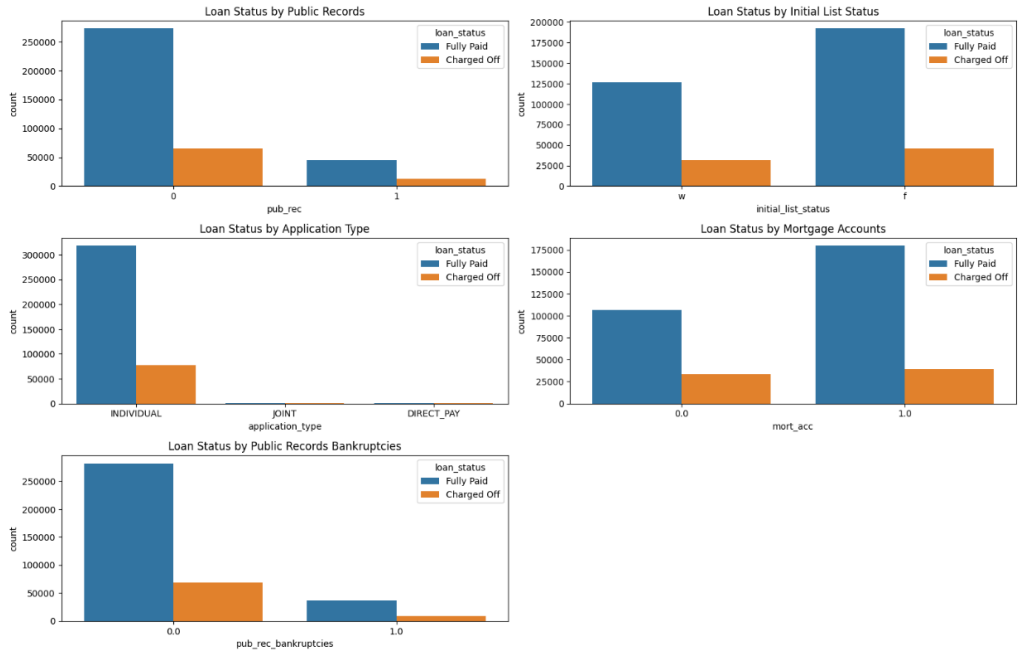

Figure 20. Histogram of pub_rec, initial_list_status, application_type, mort_acc, & pub_rec_bankruptcies by loan status

2. Data type Classification

- ✓ **Categorical**
  - **Ordinal**: grade, sub_grade, emp_length, term
  - **Nominal**: home_ownership, verification_status, purpose, initial_list_status, application_type
- ✓ **Numerical**
  - **Discrete**: loan_amnt, int_rate, installment, annual_inc, dti, revol_bal, revol_util
  - **Continuous**: total_acc, open_acc, pub_rec, pub_rec_bankruptcies, mort_acc
- ✓ **Mix type**: emp_title, address
- ✓ **Date**: issue_d, earliest_cr_line
- ✓ **Target**: loan_status

3. Handle missing data

Checking for percentage of missing data in dataset

```
In [46]: # Missing values
for column in df.columns:
    if df[column].isna().sum() != 0:
        missing = df[column].isna().sum()
        portion = (missing / df.shape[0]) * 100
        print(f"'{column}': number of missing values '{missing}' ==> '{portion:.3f}%'")
```

```
'emp_title': number of missing values '22927' ==> '5.789%'
'emp_length': number of missing values '18301' ==> '4.621%'
'revol_util': number of missing values '276' ==> '0.070%'
'mort_acc': number of missing values '37795' ==> '9.543%'
'pub_rec_bankruptcies': number of missing values '535' ==> '0.135%'
```

Figure 21. Check for missing values

- Dealing with emp_title

`emp_title`

```
In [47]: # Count the number of unique values in emp_title
print(f'The number of unique values in emp_title: {df.emp_title.nunique()}')
```

```
The number of unique values in emp_title: 173105
```

```
In [48]: # Hard to use this column for one-hot encoding, so we will drop it
df.drop('emp_title', axis=1, inplace=True)
```

Figure 22. Drop emp_tile because of high number of unique value in this column

- Dealing with emp_length

```
emp_length
```

```
In [49]: # Count the number of unique values in emp_length
         print(f'The number of unique values in emp_length: {df.emp_length.nunique()}')

         The number of unique values in emp_length: 11
```

Figure 23. Unique value count in emp_length column

```
for year in df.emp_length.unique():
    print(f"{year} in this position:")
    print(f"{df[df.emp_length == year].loan_status.value_counts(normalize=True)}")
    print('=======================================')
```

```
10+ years in this position:
loan_status
Fully Paid     0.82
Charged Off    0.18
Name: proportion, dtype: float64
=======================================
4 years in this position:
loan_status
Fully Paid     0.81
Charged Off    0.19
Name: proportion, dtype: float64
=======================================
< 1 year in this position:
loan_status
Fully Paid     0.79
Charged Off    0.21
Name: proportion, dtype: float64
=======================================
6 years in this position:
loan_status
Fully Paid     0.81
Charged Off    0.19
Name: proportion, dtype: float64
=======================================
9 years in this position:
loan_status
Fully Paid     0.80
Charged Off    0.20
Name: proportion, dtype: float64
=======================================
2 years in this position:
loan_status
Fully Paid     0.81
Charged Off    0.19
Name: proportion, dtype: float64
=======================================
3 years in this position:
loan_status
Fully Paid     0.80
Charged Off    0.20
Name: proportion, dtype: float64
=======================================
8 years in this position:
loan_status
Fully Paid     0.80
Charged Off    0.20
Name: proportion, dtype: float64
=======================================
7 years in this position:
loan_status
Fully Paid     0.81
Charged Off    0.19
Name: proportion, dtype: float64
=======================================
5 years in this position:
loan_status
Fully Paid     0.81
Charged Off    0.19
Name: proportion, dtype: float64
=======================================
1 year in this position:
loan_status
Fully Paid     0.80
Charged Off    0.20
Name: proportion, dtype: float64
=======================================
nan in this position:
Series([], Name: proportion, dtype: float6
=======================================
```

Figure 24. Percent of Value count of loan status by each value in emp_length column

- All the value in emp_length column have the same value count ratio of loan status (~80-Fully Paid / 20-Charged off) so it is extremely similar across all employment lengths. So we are going to drop the emp_length column.

- Dealing with mort_acc

```
mort_acc
```

```
print(df.mort_acc.value_counts())
print(df.mort_acc.isna().sum())
```

```
mort_acc
1.00    218458
0.00    139777
Name: count, dtype: int64
37795
```

Figure 25. Check value count of mort_acc include Nas

```
mort_acc_corr = df.corr(numeric_only=True)['mort_acc'].sort_values()
# plot the mort_acc_corr
plt.figure(figsize=(8, 4))
mort_acc_corr.drop('mort_acc').plot(kind='bar')
plt.title('Correlation of mort_acc with other variables')
plt.show()
```

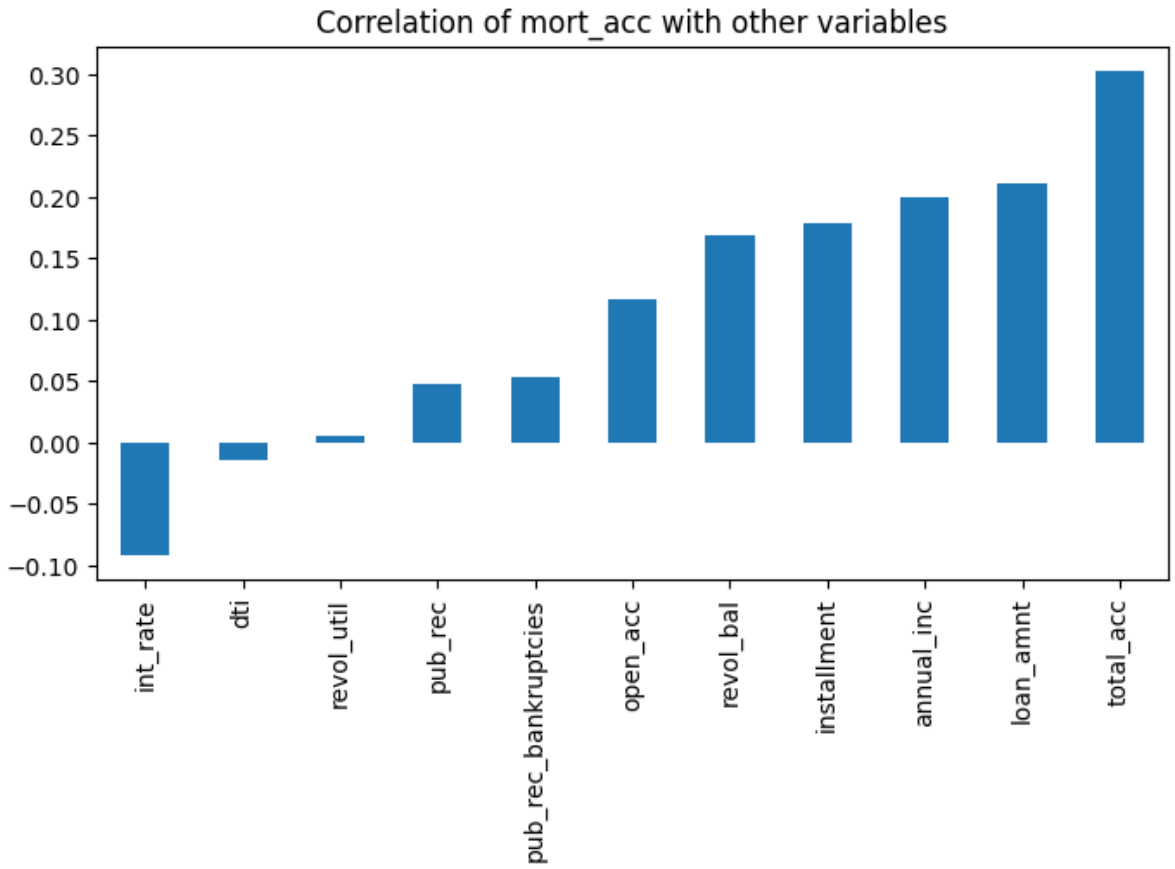

Figure 26. Checking correlation of mort_acc with other variables

- Looks like the total_acc feature correlates with the mort_acc , this makes sense! Let's try this fillna() approach. We will group the dataframe by the total_acc and calculate the mean value for the mort_acc per total_acc entry.

```
total_acc_avg = df.groupby(by='total_acc')
total_acc_avg.mort_acc.mean()
```

Out[54]:
```
total_acc
2.00      0.00
3.00      0.05
4.00      0.06
5.00      0.09
6.00      0.12
          ...
124.00    1.00
129.00    1.00
135.00    1.00
150.00    1.00
151.00    0.00
Name: mort_acc, Length: 118, dtype: float64
```

In [55]:
```
def fill_mort_acc(total_acc, mort_acc):
    if np.isnan(mort_acc):
        return total_acc_avg['mort_acc'].mean()[total_acc].round()
    else:
        return mort_acc

df['mort_acc'] = df.apply(lambda x: fill_mort_acc(x['total_acc'], x['mort_acc']), axis=1)
```

In [56]:
```
print(df.mort_acc.value_counts())
print(df.mort_acc.isna().sum())
```

```
mort_acc
1.00    242794
0.00    153236
Name: count, dtype: int64
0
```

Figure 27. Filling NAs values with mentioned method

- Dealing with revol_util & pub_rec_bankruptcies

In [57]:
```
# Missing values
for column in df.columns:
    if df[column].isna().sum() != 0:
        missing = df[column].isna().sum()
        portion = (missing / df.shape[0]) * 100
        print(f"'{column}': number of missing values '{missing}' ==> '{portion:.3f}%'")
```

```
'revol_util': number of missing values '276' ==> '0.070%'
'pub_rec_bankruptcies': number of missing values '535' ==> '0.135%'
```

In [58]:
```
# These two features have missing data points, but they account for less than 0.5% of the total data. So we are going to remove
the rows that are missing those values in those columns with dropna().
df.dropna(inplace=True)
df.shape
```

Out[58]:
```
(395219, 24)
```

Figure 28. The percentage of missing value for revol_util & pub_rec_bankruptcies is very low (<0.5% of total), so we gonna remove rows with NAs value of these columns

4. Handle Categorical variables

- List all the Categorical data

```
In [59]: print([column for column in df.columns if df[column].dtype == object])

['term', 'grade', 'sub_grade', 'home_ownership', 'verification_status', 'issue_d', 'loan_status', 'purpose', 'earliest_cr_lin
e', 'initial_list_status', 'application_type', 'address']
```

Figure 29. List of all Categorical data

- Dealing with term

```
In [60]: df.term.value_counts()

Out[60]: term
         36 months    301247
         60 months     93972
         Name: count, dtype: int64

In [61]: term_dict = {' 36 months': 36, ' 60 months': 60}
         df['term'] = df['term'].map(term_dict)
         df.term.value_counts()

Out[61]: term
         36    301247
         60     93972
         Name: count, dtype: int64
```

Figure 30. Mapping the value with numeric value

- Dealing with grade & sub_grade

```
In [62]: # grade is part of sub_grade, so we are going to drop the grade column
         df.drop('grade', axis=1, inplace=True)
```

Figure 31. Drop grade column because sub_grade already have the grade prefix

```
In [63]: df.sub_grade.value_counts()

Out[63]: sub_grade
B3    26611      E1     7906
B4    25558      E2     7410
C1    23609      E3     6196
C2    22541      E4     5354
B2    22457      E5     4561
B5    22046      F1     3530
C3    21178      F2     2756
C4    20232      F3     2277
B1    19140      F4     1782
A5    18500      F5     1395
C5    18215      G1     1057
D1    15947      G2      752
A4    15763      G3      552
D2    13916      G4      371
D3    12196      G5      313
D4    11625      Name: count, dtype: int64
A3    10537
A1     9717
D5     9680
A2     9539
```

Figure 32. Value count for sub_grade

```
In [64]: # Ordinal encoding for sub_grade
sub_grade_dict = {'A1': 1, 'A2': 2, 'A3': 3, 'A4': 4, 'A5': 5,
                  'B1': 6, 'B2': 7, 'B3': 8, 'B4': 9, 'B5': 10,
                  'C1': 11, 'C2': 12, 'C3': 13, 'C4': 14, 'C5': 15,
                  'D1': 16, 'D2': 17, 'D3': 18, 'D4': 19, 'D5': 20,
                  'E1': 21, 'E2': 22, 'E3': 23, 'E4': 24, 'E5': 25,
                  'F1': 26, 'F2': 27, 'F3': 28, 'F4': 29, 'F5': 30,
                  'G1': 31, 'G2': 32, 'G3': 33, 'G4': 34, 'G5': 35}
df['sub_grade'] = df['sub_grade'].map(sub_grade_dict)
df.sub_grade.value_counts().sort_index()

Out[64]: sub_grade
1     9717      21     7906
2     9539      22     7410
3    10537      23     6196
4    15763      24     5354
5    18500      25     4561
6    19140      26     3530
7    22457      27     2756
8    26611      28     2277
9    25558      29     1782
10   22046      30     1395
11   23609      31     1057
12   22541      32      752
13   21178      33      552
14   20232      34      371
15   18215      35      313
16   15947      Name: count, dtype: int64
17   13916
18   12196
19   11625
20    9680
```

Figure 33. Mapping the value with ordinal numeric value

- Dealing with verification_status, purpose, initial_list_status, application_type, home_ownership
    - The number of unique value in each of these column is suitable for dummy encoding.

```
dummies = ['verification_status', 'purpose', 'initial_list_status',
           'application_type', 'home_ownership']
print(f'The number of unique values in each column: {[(column, df[column].nunique()) for column in dummies]}')
```

```
The number of unique values in each column: [('verification_status', 3), ('purpose', 14), ('initial_list_status', 2), ('applica
tion_type', 3), ('home_ownership', 4)]
```

Figure 34. Value count of these columns

```
In [66]: df = pd.get_dummies(df, columns=dummies, drop_first=True)
         df.columns

Out[66]: Index(['loan_amnt', 'term', 'int_rate', 'installment', 'sub_grade',
                'annual_inc', 'issue_d', 'loan_status', 'dti', 'earliest_cr_line',
                'open_acc', 'pub_rec', 'revol_bal', 'revol_util', 'total_acc',
                'mort_acc', 'pub_rec_bankruptcies', 'address',
                'verification_status_Source Verified', 'verification_status_Verified',
                'purpose_credit_card', 'purpose_debt_consolidation',
                'purpose_educational', 'purpose_home_improvement', 'purpose_house',
                'purpose_major_purchase', 'purpose_medical', 'purpose_moving',
                'purpose_other', 'purpose_renewable_energy', 'purpose_small_business',
                'purpose_vacation', 'purpose_wedding', 'initial_list_status_w',
                'application_type_INDIVIDUAL', 'application_type_JOINT',
                'home_ownership_OTHER', 'home_ownership_OWN', 'home_ownership_RENT'],
               dtype='object')
```

Figure 35. Dummy encoding for categorical variables

5. Feature Engineering (Extraction)

- We are going to feature engineer a zip code column from the address in the data set. Create a column called 'zip_code' that extracts the zip code from the address column.

```
df.address.head()
```

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

Figure 36. Address value pattern

```
In [68]: df['zip_code'] = df['address'].apply(lambda x: x[-5:])
         df.zip_code.value_counts()

Out[68]: zip_code
         70466    56880
         22690    56413
         30723    56402
         48052    55811
         00813    45725
         29597    45393
         05113    45300
         11650    11210
         93700    11126
         86630    10959
         Name: count, dtype: int64
```

Figure 37. Extract last 5 index of address column as zip_code value

```
In [69]: df = pd.get_dummies(df, columns=['zip_code'], drop_first=True)
         df.drop('address', axis=1, inplace=True)
         df.columns

Out[69]: Index(['loan_amnt', 'term', 'int_rate', 'installment', 'sub_grade',
                'annual_inc', 'issue_d', 'loan_status', 'dti', 'earliest_cr_line',
                'open_acc', 'pub_rec', 'revol_bal', 'revol_util', 'total_acc',
                'mort_acc', 'pub_rec_bankruptcies',
                'verification_status_Source Verified', 'verification_status_Verified',
                'purpose_credit_card', 'purpose_debt_consolidation',
                'purpose_educational', 'purpose_home_improvement', 'purpose_house',
                'purpose_major_purchase', 'purpose_medical', 'purpose_moving',
                'purpose_other', 'purpose_renewable_energy', 'purpose_small_business',
                'purpose_vacation', 'purpose_wedding', 'initial_list_status_w',
                'application_type_INDIVIDUAL', 'application_type_JOINT',
                'home_ownership_OTHER', 'home_ownership_OWN', 'home_ownership_RENT',
                'zip_code_05113', 'zip_code_11650', 'zip_code_22690', 'zip_code_29597',
                'zip_code_30723', 'zip_code_48052', 'zip_code_70466', 'zip_code_86630',
                'zip_code_93700'],
               dtype='object')
```

Figure 38. Dummy encoding for new zip_code column and also drop addres column

6. Handle date data

- Issue_d is data leakage, so we are going to drop it, because it is data that wouldn't be available when we are using the model.

```
df.drop('issue_d', axis=1, inplace=True)
```

Figure 39. Drop issue_d

- This appears to be a historical time stamp feature. Extract the year from this feature using a .apply() function, then convert it to a numeric feature.

```
df['earliest_cr_line'] = pd.to_datetime(df['earliest_cr_line'])
df['earliest_cr_year'] = df['earliest_cr_line'].apply(lambda x: x.year)
df.drop('earliest_cr_line', axis=1, inplace=True)
df.earliest_cr_year.value_counts()
```

```
earliest_cr_year
2000    29302
2001    29031
1999    26444
2002    25849
2003    23623
        ...
1951        3
1950        3
1953        2
1944        1
1948        1
Name: count, Length: 65, dtype: int64
```

Figure 40. Deal with earliest_cr_line by get year from this value and cast as integer

7. Process Target variable

```
loan_status
```

```
df['loan_status'].value_counts()
```

```
loan_status
Fully Paid     317696
Charged Off     77523
Name: count, dtype: int64
```

```
# Check the data types of the columns
dict_loan_status = {'Fully Paid': 1, 'Charged Off': 0}
df['loan_status'] = df['loan_status'].map(dict_loan_status)
df['loan_status'].value_counts()
```

```
loan_status
1    317696
0     77523
Name: count, dtype: int64
```

Figure 41. Mapping the value with numeric value

8. Remove duplicate record

```
In [77]: print(f"Data shape before removing duplicate rows: {df.shape}")

         # Remove Duplicate Rows
         df.drop_duplicates(inplace=True)

         # Remove nan values
         df.dropna(inplace=True)

         print(f"Data shape after removing duplicate rows: {df.shape}")

         Data shape before removing duplicate rows: (395219, 46)
         Data shape after removing duplicate rows: (395219, 46)
```

Figure 42. Before & after removing duplicate rows

III. IMPLEMENTATION OF DATA MINING ALGORITHMS ON DATASET

1. Normalizing data

Normalizing data is a crucial preprocessing step in building machine learning models, as it ensures that features are on a similar scale, which enhances model performance and convergence speed. By adjusting the numerical range of the data, normalization mitigates the impact of features with larger scales, preventing them from disproportionately influencing the model's learning process. Techniques such as min-max scaling, which rescales the data to a fixed range, typically [0, 1], and z-score normalization, which standardizes data to have a mean of 0 and a standard deviation of 1, are commonly used. This step is particularly important for algorithms that rely on distance metrics, such as k-nearest neighbors and support vector machines, ensuring that each feature contributes equally to the computation of distances and gradients.

Min-max normalization is one of the most common ways to normalize data. For every feature, the minimum value of that feature gets transformed into a 0, the maximum value gets transformed into a 1, and every other value gets transformed into a decimal between 0 and 1.

In [4]:
```
# MinMaxScaler data
Input_features = df.drop('loan_status', axis=1).columns.values
X = df[Input_features]
y = df['loan_status']

scaler = MinMaxScaler()
scaler.fit(X[Input_features])
X[Input_features] = scaler.transform(X)
X.head()
```

Out[4]:

	loan_amnt	term	int_rate	installment	sub_grade	annual_inc	dti	open_acc	pub_rec	revol_bal	revol_util	total_acc	mort_acc	pub_rec_bankruptcie
0	0.240506	0.0	0.238411	0.206493	0.235294	0.013438	0.002624	0.168539	0.0	0.020863	0.046845	0.154362	0.0	0.(
1	0.189873	0.0	0.259836	0.164456	0.264706	0.007466	0.002205	0.179775	0.0	0.011548	0.059733	0.167785	1.0	0.(
2	0.382278	0.0	0.201402	0.323437	0.205882	0.004945	0.001279	0.134831	0.0	0.006876	0.103328	0.161074	0.0	0.(
3	0.169620	0.0	0.045578	0.134787	0.029412	0.006202	0.000260	0.056180	0.0	0.003139	0.024095	0.073826	0.0	0.(
4	0.604430	1.0	0.465524	0.390880	0.411765	0.006317	0.003395	0.134831	0.0	0.014102	0.078225	0.275168	1.0	0.(

Figure 43. Normalizing Data using Min-Max Scaler method

2. Split data

Split the data set into two pieces — a training set and a testing set. This consists of random sampling without replacement about 80 percent of the rows (you can vary this) and putting them into your training set. The remaining 20 percent is put into your test set. Note that the colors in “Features” and “Target” indicate where their data will go (“X_train,” “X_test,” “y_train,” “y_test”) for a particular train test split.

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(316175, 45) (79044, 45) (316175,) (79044,)
```

Figure 44. Split the data into training and testing sets

3. Model Selection Reason

The selection of machine learning classification algorithms such as Logistic Regression, Decision Trees, Naive Bayes, Random Forest, Gradient Boosting (XGBoost), and Support Vector Machine for analyzing credit risk on the Lending Club platform was driven by several key considerations. Firstly, each algorithm offers unique strengths that are applicable to different aspects of the credit risk prediction problem. For instance, Random Forest and Gradient Boosting are robust against complex data structures often found in Lending Club's loan data, while Logistic Regression and Naive Bayes are well-suited for datasets with high interpretability needs.

Secondly, the nature of Lending Club data necessitates algorithms that can handle large and heterogeneous datasets effectively. Algorithms like Support Vector Machine and XGBoost are adept at managing imbalanced data distributions commonly seen in credit risk scenarios, allowing for nuanced adjustments in the importance of different loan samples.

Furthermore, the computational efficiency of these algorithms was a crucial factor. While Decision Trees are intuitive and easy to interpret, algorithms like Logistic Regression and Support Vector Machine offer faster training times and high accuracy, which is essential when dealing with large volumes of data from Lending Club.

Additionally, the ability to evaluate and improve models played a significant role in their selection. By comparing the performance of each algorithm, we aimed to identify the most influential factors affecting credit risk and enhance risk management practices on the Lending Club platform. This analysis not only aids in understanding the predictive capabilities of each model but also guides the recommendation of optimized models for future credit risk prediction endeavors.

In conclusion, the choice of these specific machine learning algorithms was guided by their suitability for the credit risk analysis task on the Lending Club platform and their capability to leverage and analyze the unique characteristics of its loan data effectively.

4. Using Logistic Regression algorithm

4.1. Introduction

- Logistic regression is a statistical method used for modeling the relationship between a binary dependent variable and one or more independent variables (1 or 0, True or False, Yes or No, etc). It's commonly used for classification tasks where the dependent variable represents a binary outcome.

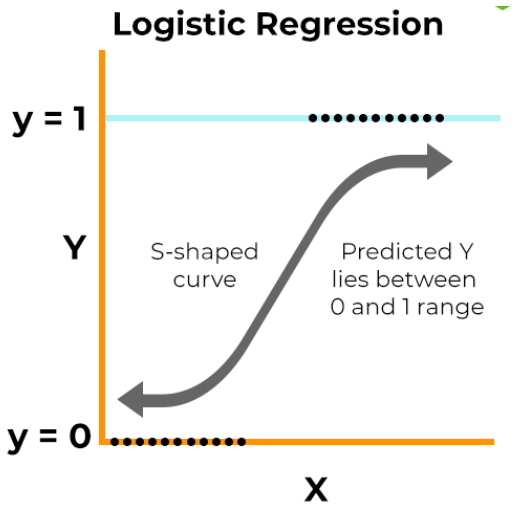

Figure 45. Presenting Logistic Model

4.2. Implement the algorithm

Logistic Regression

In [23]:
```
# Logistic Regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

# Train the model

start_log_reg = time.time()

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

time_log_reg = time.time() - start_log_reg

y_test_pred = log_reg.predict(X_test)

# Evaluate the model
print('Logistic Regression')
print('Time:', time_log_reg)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=log_reg.classes_)
disp.plot()
```

```
Logistic Regression
Time: 3.2966020107269287
              precision    recall  f1-score   support

           0       0.92      0.47      0.62     15421
           1       0.88      0.99      0.93     63623

    accuracy                           0.89     79044
   macro avg       0.90      0.73      0.78     79044
weighted avg       0.89      0.89      0.87     79044
```

Figure 46. Logistic Regression Model building and Model testing report

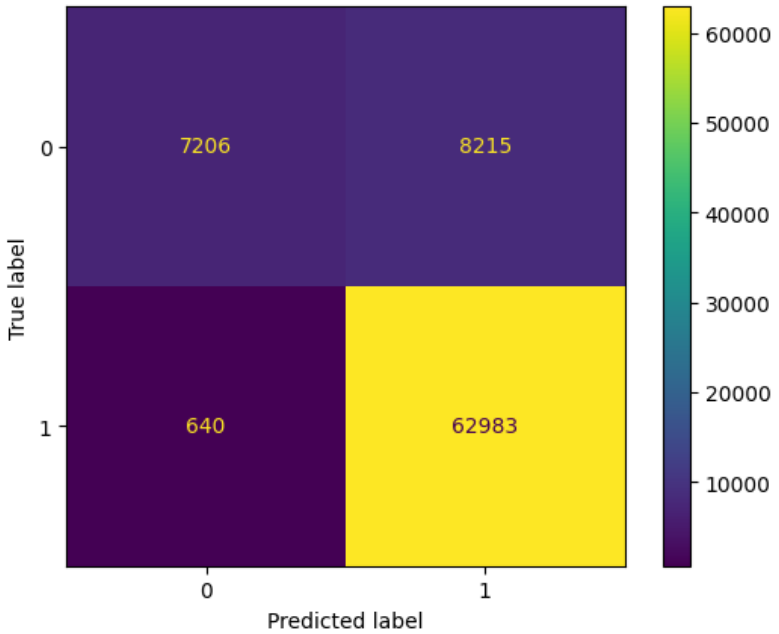

Figure 47. Confusion matrix of Logistic Regression model result

- Execution time: 3.2966 (s)
- Accuracy: 89%

5. Using Decision Tree algorithm

5.1. Introduction

- Decision Tree is a decision-making tool that uses a flowchart-like tree structure or is a model of decisions and all of their possible results. It works for both continuous as well as categorical output variables.
- Decision tree regression observes features of an object and trains a model in the structure of a tree to predict data in the future to produce meaningful continuous output. Continuous output means that the output/result is not discrete, i.e., it is not represented just by a discrete, known set of numbers or values

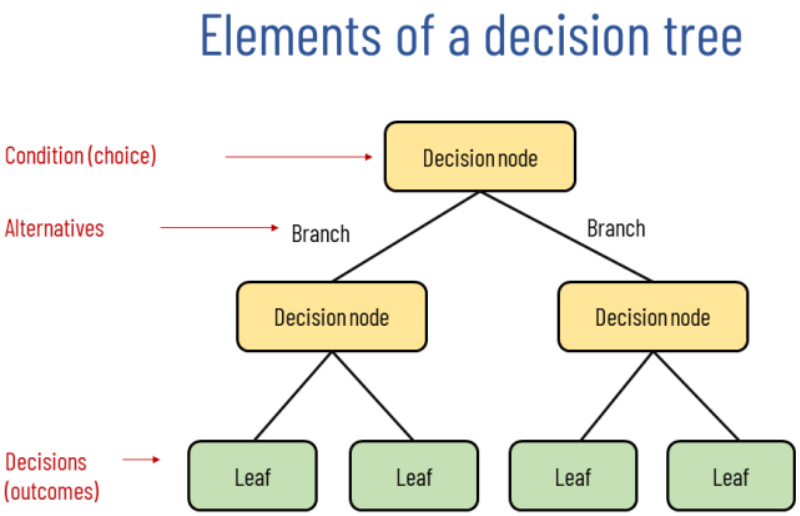

Figure 48. Presenting decision tree

5.2. Implement the algorithm

Decision Trees

In [18]:
```
# Decision Tree
from sklearn.tree import DecisionTreeClassifier

# Train the model

start_dt = time.time()

dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

time_dt = time.time() - start_dt

y_test_pred = dt.predict(X_test)

# Evaluate the model
print('Decision Tree')
print('Time:', time_dt)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=dt.classes_)
disp.plot()
```

```
Decision Tree
Time: 5.919177055358887
              precision    recall  f1-score   support

           0       0.56      0.59      0.58     15421
           1       0.90      0.89      0.89     63623

    accuracy                           0.83     79044
   macro avg       0.73      0.74      0.74     79044
weighted avg       0.83      0.83      0.83     79044
```

Figure 49. Decision Tree Model building and Model testing report

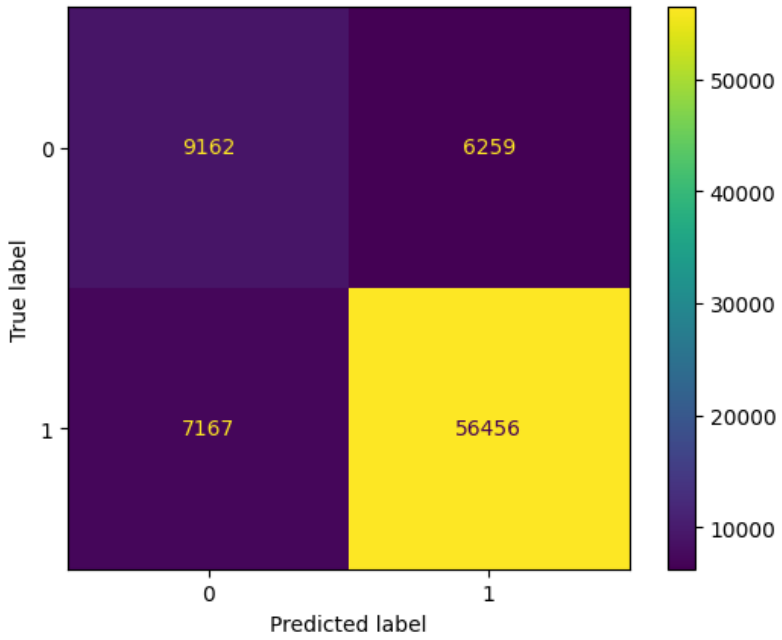

Figure 50. Confusion matrix of Decision tree model result

- Execution time: 5.9192 (s)
- Accuracy: 83%

6. Using Random Forest algorithm

6.1. Introduction

- A random forest is a machine learning technique that's used to solve regression and classification problems. It utilizes ensemble learning, which is a technique that combines many classifiers to provide solutions to complex problems.
- A random forest algorithm consists of many decision trees. The random forest algorithm establishes the outcome based on the predictions of the decision trees. It predicts by taking the average or mean of the output from various trees. Increasing the number of trees increases the precision of the outcome

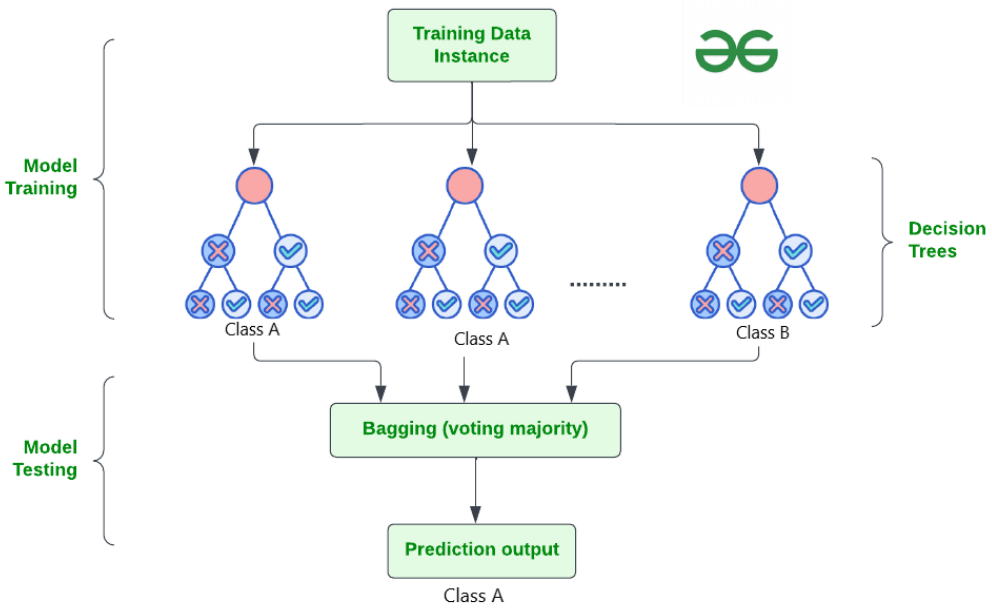

Figure 51. Presenting Random forest

6.2. Implement the algorithm

Random Forest

In [19]:
```
# Random Forest
from sklearn.ensemble import RandomForestClassifier

# Train the model

start_rf = time.time()

rf = RandomForestClassifier(n_jobs=-1)
rf.fit(X_train, y_train)

time_rf = time.time() - start_rf

y_test_pred = rf.predict(X_test)

# Evaluate the model
print('Random Forest')
print('Time:', time_rf)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf.classes_)
disp.plot()
```

```
Random Forest
Time: 15.99890422821045
              precision    recall  f1-score   support

           0       0.95      0.46      0.62     15421
           1       0.88      0.99      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.92      0.73      0.78     79044
weighted avg       0.90      0.89      0.87     79044
```

Figure 52. Random Forest Model building and Model testing report

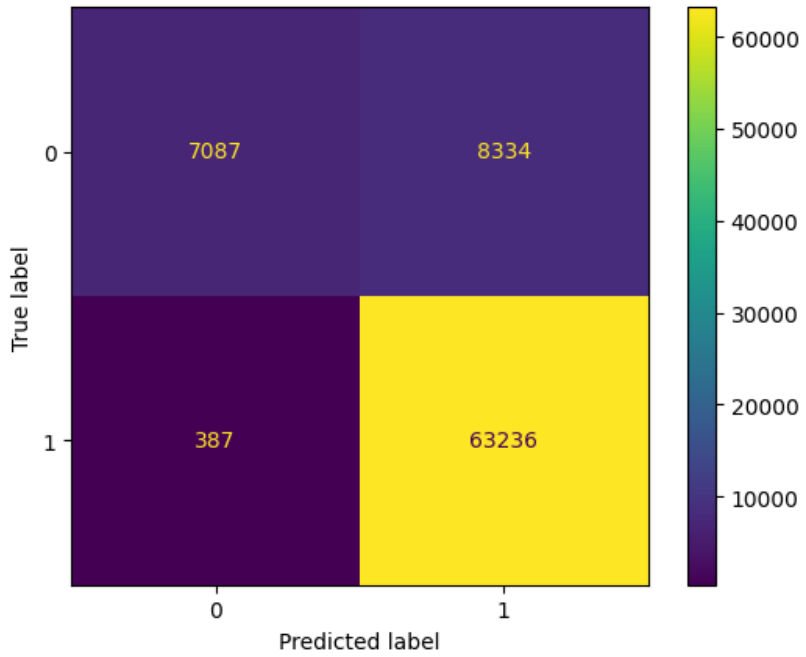

Figure 53. Confusion matrix of Random Forest model result

- Execution time: 15.9989 (s)
- Accuracy: 89%

7. Using Naive Bayes

7.1. Introduction

- It is a classification technique based on Bayes' Theorem with an independence assumption among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.
- The Naïve Bayes classifier is a popular supervised machine learning algorithm used for classification tasks such as text classification. It belongs to the family of generative learning algorithms, which means that it models the distribution of inputs for a given class or category. This approach assumes that the features of the input data are conditionally independent given the class, allowing the algorithm to make predictions quickly and accurately.

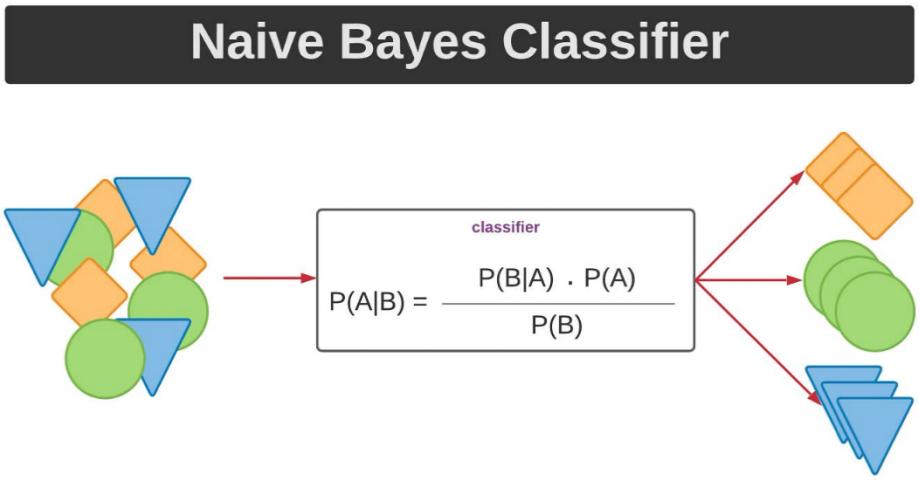

Figure 54. Presenting Naive Bayes Classifier

7.2. Implement the algorithm

Naive Bayes

In [16]:
```
# Naive Bayes
from sklearn.naive_bayes import GaussianNB

# Train the model

start_nb = time.time()

nb = GaussianNB()
nb.fit(X_train, y_train)

time_nb = time.time() - start_nb

y_test_pred = nb.predict(X_test)

# Evaluate the model
print('Naive Bayes')
print('Time:', time_nb)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=nb.classes_)
disp.plot()
```

```
Naive Bayes
Time: 0.30797672271728516
              precision    recall  f1-score   support

           0       0.88      0.45      0.59     15421
           1       0.88      0.99      0.93     63623

    accuracy                           0.88     79044
   macro avg       0.88      0.72      0.76     79044
weighted avg       0.88      0.88      0.86     79044
```

Figure 55. Naïve Bayes Model building and Model testing report

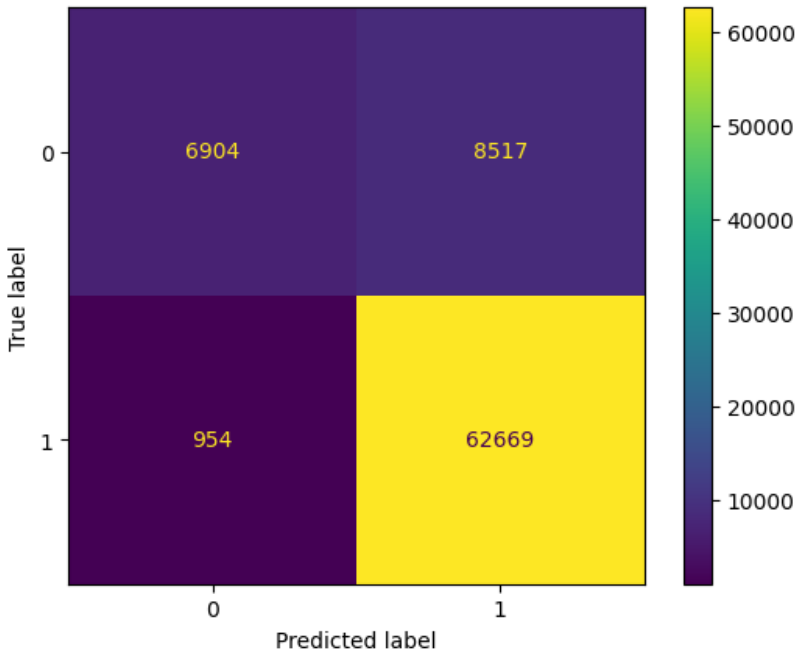

Figure 56. Confusion matrix of Naïve Bayes model result

- Execution time: 0.308 (s)
- Accuracy: 88%

8. Using Extreme Gradient Boosting (XGBoost)

8.1. Introduction

- XGBoost, short for Extreme Gradient Boosting, is a powerful and scalable machine learning algorithm that has gained immense popularity in recent years. It is an implementation of gradient boosting machines designed for speed and performance. XGBoost is particularly known for its efficiency in handling large-scale datasets and its ability to harness the power of parallel and distributed computing. This makes it a top choice for many data scientists and machine learning practitioners who are looking to build robust and accurate predictive models.
- One of the key strengths of XGBoost lies in its flexibility and capability to be tuned to achieve optimal performance. It supports various objective functions, including regression, classification, and ranking, and offers extensive hyperparameter tuning options. Additionally, XGBoost integrates well with many data science pipelines, providing interfaces for several programming languages such as Python, R, and Julia. Its comprehensive documentation and active community further contribute to its widespread adoption and continuous improvement.

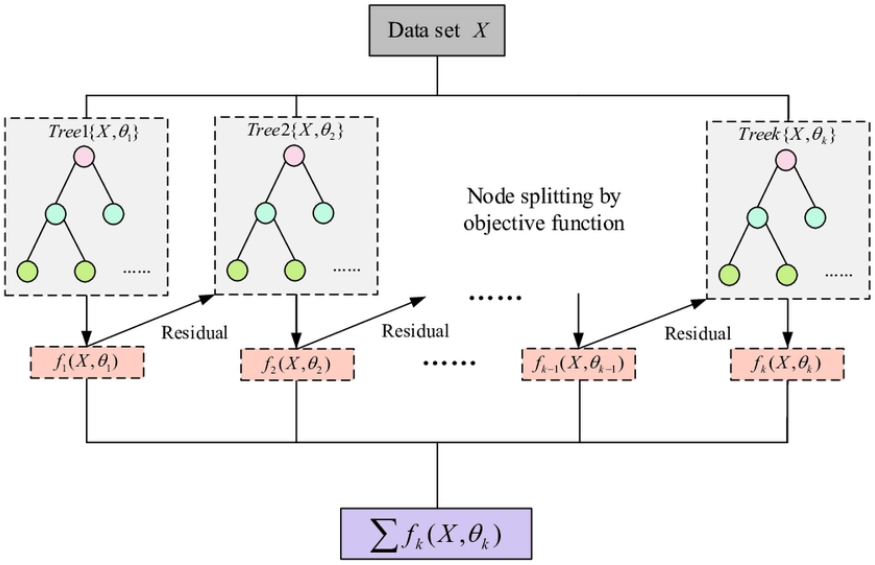

Figure 57. Presenting XGBoost

8.2. Implement the algorithm

XGBoost

In [20]:
```
# XGBoost
from xgboost import XGBClassifier

# Train the model

start_xgb = time.time()

xgb = XGBClassifier()
xgb.fit(X_train, y_train)

time_xgb = time.time() - start_xgb

y_test_pred = xgb.predict(X_test)

# Evaluate the model
print('XGBoost')
print('Time:', time_xgb)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=xgb.classes_)
disp.plot()
```

```
XGBoost
Time: 1.3323287963867188
              precision    recall  f1-score   support

           0       0.91      0.48      0.63     15421
           1       0.89      0.99      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.90      0.74      0.78     79044
weighted avg       0.89      0.89      0.88     79044
```

Figure 58. XGBoost Model building and Model testing report

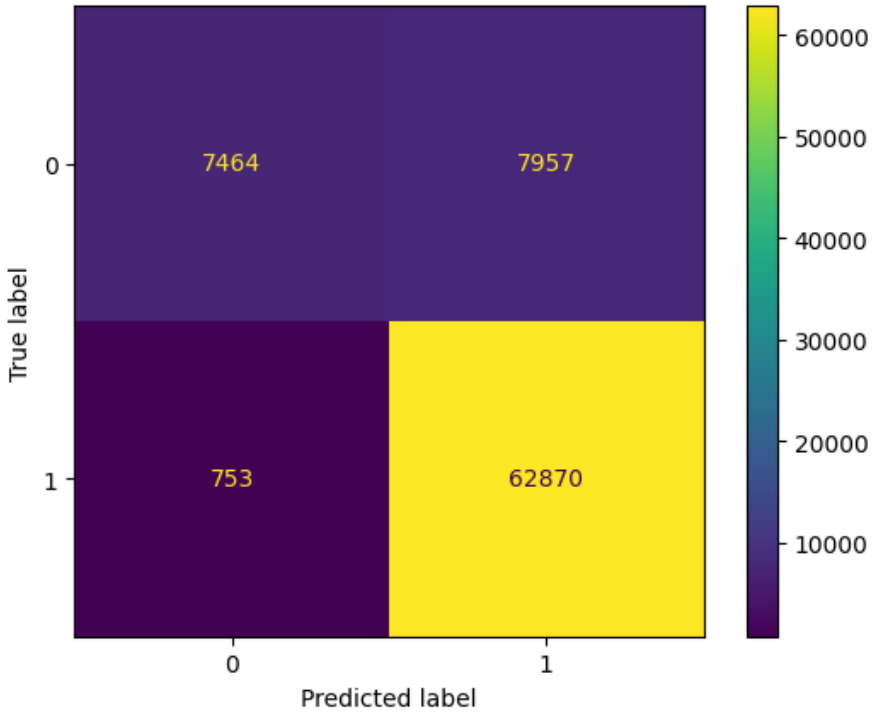

Figure 59. Confusion matrix of XGBoost model result

- Execution time: 1.3323 (s)

- Accuracy: 89%

9. Support Vector Machine

9.1. Introduction

- Support Vector Machine(SVM) is a supervised machine learning algorithm used for both classification and regression. Though we say regression problems as well it's best suited for classification. SVMs can handle highly non-linear data using an amazing technique called kernel trick. It implicitly maps the input vectors to higher dimensional (adds more dimensions) feature spaces by the transformation which rearranges the dataset in such a way that it is linearly solvable.
- SVR or Support Vector Regression uses the same principle of Support Vector Machines. In other words, the approach of using SVMs to solve regression problems is called Support Vector Regression or SVR. SVR is used to predict discrete values. The basic idea behind SVR is to find the best line. In SVR, the best fit line is the hyperplane that has the maximum number of points.

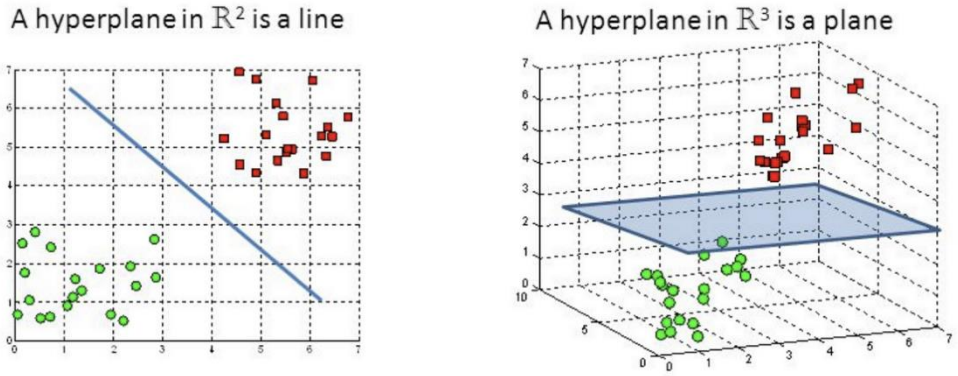

Figure 60. Presenting Support Vector Machine

9.2. Implement the algorithm

Support Vector Machine

In [17]:
```
# Support Vector Machine
from sklearn.svm import LinearSVC

# Train the model

start_svc = time.time()

svc = LinearSVC()
svc.fit(X_train, y_train)

time_svc = time.time() - start_svc

y_test_pred = svc.predict(X_test)

# Evaluate the model
print('Support Vector Machine')
print('Time:', time_svc)
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=svc.classes_)
disp.plot()
```

```
Support Vector Machine
Time: 14.334577083587646
              precision    recall  f1-score   support

           0       0.96      0.45      0.61     15421
           1       0.88      1.00      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.92      0.72      0.77     79044
weighted avg       0.90      0.89      0.87     79044
```

Figure 61. SVM Model building and Model testing report

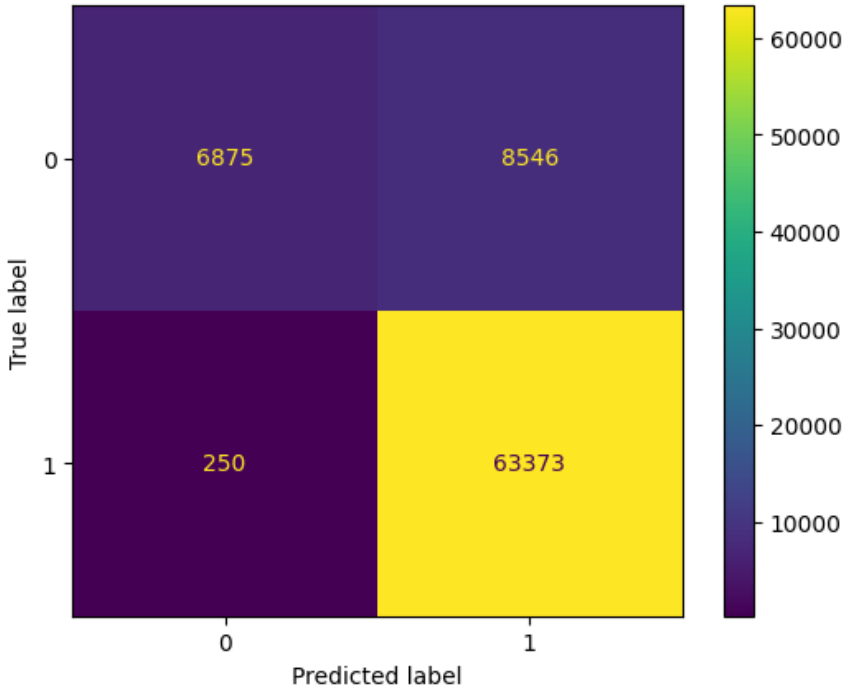

Figure 62. Confusion matrix of SVM model result

- Execution time: 14.3346 (s)
- Accuracy: 89%

10. Model Hyperparameters tuning

10.1. Logistic Regression

In [7]:
```
# GridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}
grid_log_reg = GridSearchCV(LogisticRegression(max_iter=1000), param_grid, verbose=3)
grid_log_reg.fit(X_train, y_train)

print("Best parameters: ", grid_log_reg.best_params_)
print("Best estimator: ", grid_log_reg.best_estimator_)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
[CV 1/5] END ..................C=0.1, penalty=l1;, score=nan total time=   0.0s
[CV 2/5] END ..................C=0.1, penalty=l1;, score=nan total time=   0.0s
[CV 3/5] END ..................C=0.1, penalty=l1;, score=nan total time=   0.0s
[CV 4/5] END ..................C=0.1, penalty=l1;, score=nan total time=   0.0s
[CV 5/5] END ..................C=0.1, penalty=l1;, score=nan total time=   0.0s
[CV 1/5] END .................C=0.1, penalty=l2;, score=0.887 total time=   4.3s
[CV 2/5] END .................C=0.1, penalty=l2;, score=0.887 total time=   3.6s
[CV 3/5] END .................C=0.1, penalty=l2;, score=0.888 total time=   4.5s
[CV 4/5] END .................C=0.1, penalty=l2;, score=0.890 total time=   3.6s
[CV 5/5] END .................C=0.1, penalty=l2;, score=0.888 total time=   3.7s
[CV 1/5] END ..................C=1, penalty=l1;, score=nan total time=   0.0s
[CV 2/5] END ..................C=1, penalty=l1;, score=nan total time=   0.0s
[CV 3/5] END ..................C=1, penalty=l1;, score=nan total time=   0.0s
[CV 4/5] END ..................C=1, penalty=l1;, score=nan total time=   0.0s
```

Figure 63. Logistic Regression - Initial param grid and searching for best estimator

```
Best parameters:  {'C': 100, 'penalty': 'l2'}
Best estimator:  LogisticRegression(C=100, max_iter=1000)
```

In [8]:
```
# Evaluate the model after GridSearchCV
y_test_pred = grid_log_reg.predict(X_test)
print('Logistic Regression after GridSearchCV')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_log_reg.classes_)
disp.plot()
```

```
Logistic Regression after GridSearchCV
              precision    recall  f1-score   support

           0       0.91      0.47      0.62     15421
           1       0.89      0.99      0.93     63623

    accuracy                           0.89     79044
   macro avg       0.90      0.73      0.78     79044
weighted avg       0.89      0.89      0.87     79044
```

Out[8]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x19f4c790a30>
```

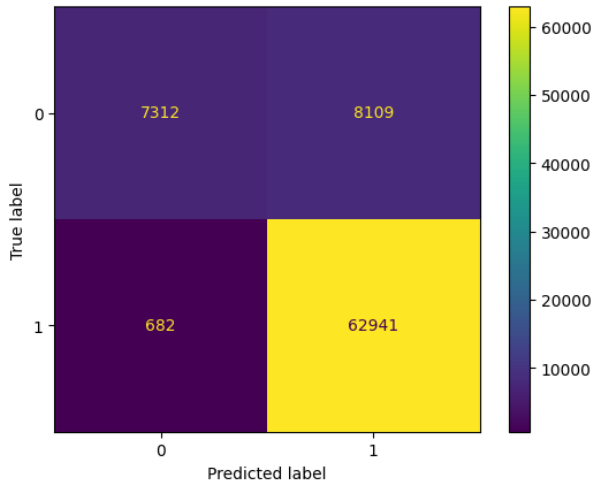

Figure 64. Logistic Regression model classification report on best estimator

10.2. Naïve Bayes

In [10]:
```
# GridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid = {
    'var_smoothing': np.logspace(0,-9, num=100)
}
grid_NB = GridSearchCV(GaussianNB(), param_grid, verbose=3)
grid_NB.fit(X_train, y_train)

print("Best parameters: ", grid_NB.best_params_)
print("Best estimator: ", grid_NB.best_estimator_)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
[CV 1/5] END .................var_smoothing=1.0;, score=0.816 total time=   0.2s
[CV 2/5] END .................var_smoothing=1.0;, score=0.815 total time=   0.2s
[CV 3/5] END .................var_smoothing=1.0;, score=0.815 total time=   0.2s
[CV 4/5] END .................var_smoothing=1.0;, score=0.816 total time=   0.2s
[CV 5/5] END .................var_smoothing=1.0;, score=0.815 total time=   0.2s
[CV 1/5] END ..var_smoothing=0.8111308307896871;, score=0.832 total time=   0.2s
[CV 2/5] END ..var_smoothing=0.8111308307896871;, score=0.831 total time=   0.2s
[CV 3/5] END ..var_smoothing=0.8111308307896871;, score=0.832 total time=   0.2s
[CV 4/5] END ..var_smoothing=0.8111308307896871;, score=0.833 total time=   0.2s
[CV 5/5] END ..var_smoothing=0.8111308307896871;, score=0.831 total time=   0.2s
[CV 1/5] END ...var_smoothing=0.657933224657568;, score=0.855 total time=   0.2s
[CV 2/5] END ...var_smoothing=0.657933224657568;, score=0.853 total time=   0.2s
[CV 3/5] END ...var_smoothing=0.657933224657568;, score=0.856 total time=   0.2s
[CV 4/5] END ...var_smoothing=0.657933224657568;, score=0.855 total time=   0.2s
[CV 5/5] END ...var_smoothing=0.657933224657568;, score=0.854 total time=   0.2s
[CV 1/5] END ...var_smoothing=0.533669923120631;, score=0.879 total time=   0.2s
```

Figure 65. Naïve Bayes - Initial param grid and searching for best estimator

```
Best parameters:  {'var_smoothing': 0.3511191734215131}
Best estimator:  GaussianNB(var_smoothing=0.3511191734215131)
```

In [11]:
```
# Evaluate the model àfter GridSearchCV
y_test_pred = grid_NB.best_estimator_.predict(X_test)
print('Naive Bayes')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_NB.best_estimator_.classes_)
disp.plot()
```

```
Naive Bayes
              precision    recall  f1-score   support

           0       0.98      0.44      0.60     15421
           1       0.88      1.00      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.93      0.72      0.77     79044
weighted avg       0.90      0.89      0.87     79044
```

Out[11]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x19f4cf65850>
```

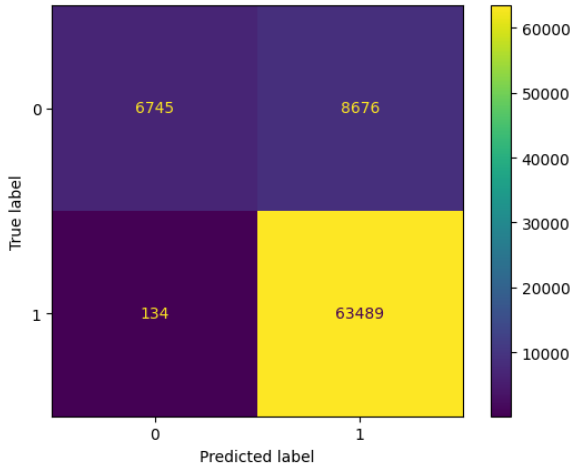

Figure 66. Naïve Bayes model classification report on best estimator

10.3. Support Vector Machine

In [13]:
```
# GridSearchCV for LinearSVC
param_grid = {'C': [0.1, 1, 10, 100, 1000], 'penalty': ['l1', 'l2'], 'loss': ['hinge', 'squared_hinge']}
grid_svc = GridSearchCV(LinearSVC(), param_grid, verbose=3)
grid_svc.fit(X_train, y_train)

print("Best parameters: ", grid_svc.best_params_)
print("Best estimator: ", grid_svc.best_estimator_)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[CV 1/5] END .......C=0.1, loss=hinge, penalty=l1;, score=nan total time=   0.0s
[CV 2/5] END .......C=0.1, loss=hinge, penalty=l1;, score=nan total time=   0.0s
[CV 3/5] END .......C=0.1, loss=hinge, penalty=l1;, score=nan total time=   0.0s
[CV 4/5] END .......C=0.1, loss=hinge, penalty=l1;, score=nan total time=   0.0s
[CV 5/5] END .......C=0.1, loss=hinge, penalty=l1;, score=nan total time=   0.0s
[CV 1/5] END .....C=0.1, loss=hinge, penalty=l2;, score=0.887 total time=  40.3s
[CV 2/5] END .....C=0.1, loss=hinge, penalty=l2;, score=0.887 total time=  39.8s
```

Figure 67. SVM - Initial param grid and searching for best estimator

```
Best parameters:  {'C': 10, 'loss': 'squared_hinge', 'penalty': 'l2'}
Best estimator:  LinearSVC(C=10)
```

In [14]:
```
# Evaluate the model after GridSearchCV
y_test_pred = grid_svc.predict(X_test)
print('Support Vector Machine after GridSearchCV')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_svc.classes_)
disp.plot()
```

```
Support Vector Machine after GridSearchCV
              precision    recall  f1-score   support

           0       0.96      0.45      0.61     15421
           1       0.88      1.00      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.92      0.72      0.77     79044
weighted avg       0.90      0.89      0.87     79044
```

Out[14]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x19f138a61f0>
```

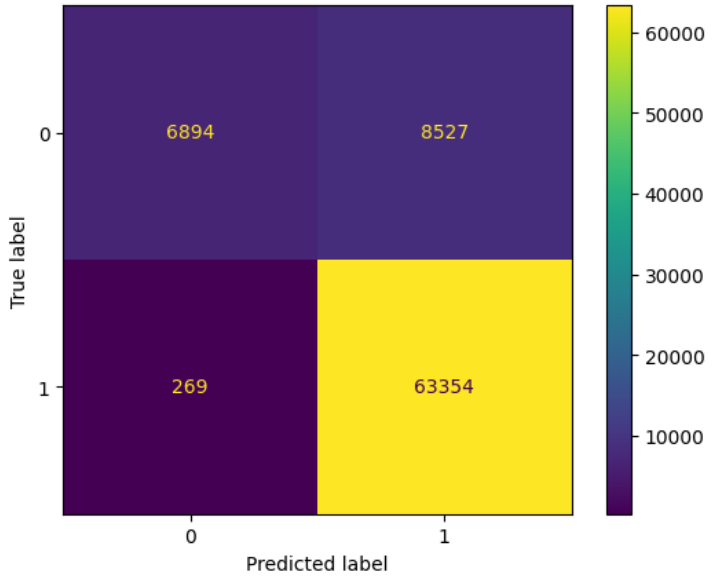

Figure 68. SVM model classification report on best estimator

10.4. Decision Tree

In [12]:
```
# # GridSearchCV for Decision Tree
param_grid = {
    'max_depth': [2, 3, 5, 10, 20],
    'criterion': ['gini', 'entropy'],
    'max_features': ['auto', 'sqrt', 'log2']
}
grid_dt = GridSearchCV(DecisionTreeClassifier(), param_grid, verbose=3)
grid_dt.fit(X_train, y_train)

print("Best parameters: ", grid_dt.best_params_)
print("Best estimator: ", grid_dt.best_estimator_)
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[CV 1/5] END criterion=gini, max_depth=2, max_features=auto;, score=nan total time=   0.0s
[CV 2/5] END criterion=gini, max_depth=2, max_features=auto;, score=nan total time=   0.0s
[CV 3/5] END criterion=gini, max_depth=2, max_features=auto;, score=nan total time=   0.0s
[CV 4/5] END criterion=gini, max_depth=2, max_features=auto;, score=nan total time=   0.0s
[CV 5/5] END criterion=gini, max_depth=2, max_features=auto;, score=nan total time=   0.0s
[CV 1/5] END criterion=gini, max_depth=2, max_features=sqrt;, score=0.823 total time=   0.2s
[CV 2/5] END criterion=gini, max_depth=2, max_features=sqrt;, score=0.815 total time=   0.1s
[CV 3/5] END criterion=gini, max_depth=2, max_features=sqrt;, score=0.804 total time=   0.1s
[CV 4/5] END criterion=gini, max_depth=2, max_features=sqrt;, score=0.804 total time=   0.1s
[CV 5/5] END criterion=gini, max_depth=2, max_features=sqrt;, score=0.831 total time=   0.1s
[CV 1/5] END criterion=gini, max_depth=2, max_features=log2;, score=0.860 total time=   0.1s
[CV 2/5] END criterion=gini, max_depth=2, max_features=log2;, score=0.804 total time=   0.1s
[CV 3/5] END criterion=gini, max_depth=2, max_features=log2;, score=0.804 total time=   0.1s
[CV 4/5] END criterion=gini, max_depth=2, max_features=log2;, score=0.832 total time=   0.1s
[CV 5/5] END criterion=gini, max_depth=2, max_features=log2;, score=0.821 total time=   0.1s
[CV 1/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
```

Figure 69. Decision Tree - Initial param grid and searching for best estimator

```
Best parameters:  {'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt'}
Best estimator:  DecisionTreeClassifier(criterion='entropy', max_depth=10, max_features='sqrt')
```

In [13]:
```
# Evaluate the model after GridSearchCV
y_test_pred = grid_dt.predict(X_test)
print('Decision Tree after GridSearchCV')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_dt.classes_)
disp.plot()
```

```
Decision Tree after GridSearchCV
              precision    recall  f1-score   support

           0       0.86      0.29      0.43     15421
           1       0.85      0.99      0.92     63623

    accuracy                           0.85     79044
   macro avg       0.86      0.64      0.67     79044
weighted avg       0.85      0.85      0.82     79044
```

Out[13]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2be8e5025b0>
```

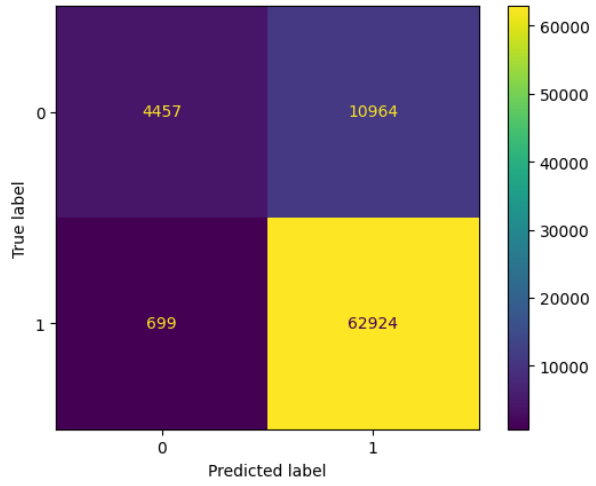

Figure 70. Decision Tree model classification report on best estimator

10.5. Random Forest

In [9]:
```
# GridSearchCV for Random Forest
param_grid = {
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [3, 4, 5],
    'criterion' :['gini', 'entropy']
}


grid_rf = GridSearchCV(RandomForestClassifier(), param_grid, verbose=3)
grid_rf.fit(X_train, y_train)

print("Best parameters: ", grid_rf.best_params_)
print("Best estimator: ", grid_rf.best_estimator_)
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[CV 1/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
[CV 2/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
[CV 3/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
[CV 4/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
[CV 5/5] END criterion=gini, max_depth=3, max_features=auto;, score=nan total time=   0.0s
[CV 1/5] END criterion=gini, max_depth=3, max_features=sqrt;, score=0.818 total time=  10.1s
[CV 2/5] END criterion=gini, max_depth=3, max_features=sqrt;, score=0.821 total time=  10.2s
[CV 3/5] END criterion=gini, max_depth=3, max_features=sqrt;, score=0.826 total time=  10.5s
[CV 4/5] END criterion=gini, max_depth=3, max_features=sqrt;, score=0.827 total time=   9.3s
[CV 5/5] END criterion=gini, max_depth=3, max_features=sqrt;, score=0.822 total time=   9.2s
[CV 1/5] END criterion=gini, max_depth=3, max_features=log2;, score=0.808 total time=   8.3s
[CV 2/5] END criterion=gini, max_depth=3, max_features=log2;, score=0.812 total time=   8.1s
[CV 3/5] END criterion=gini, max_depth=3, max_features=log2;, score=0.821 total time=   8.4s
[CV 4/5] END criterion=gini, max_depth=3, max_features=log2;, score=0.810 total time=   8.0s
[CV 5/5] END criterion=gini, max_depth=3, max_features=log2;, score=0.804 total time=   8.4s
```

Figure 71. Random Forest - Initial param grid and searching for best estimator

```
Best parameters:  {'criterion': 'entropy', 'max_depth': 5, 'max_features': 'sqrt'}
Best estimator:  RandomForestClassifier(criterion='entropy', max_depth=5)
```

In [10]:
```
# Evaluate the model after GridSearchCV
y_test_pred = grid_rf.predict(X_test)
print('Random Forest after GridSearchCV')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_rf.classes_)
disp.plot()
```

```
Random Forest after GridSearchCV
              precision    recall  f1-score   support

           0       1.00      0.40      0.58     15421
           1       0.87      1.00      0.93     63623

    accuracy                           0.88     79044
   macro avg       0.94      0.70      0.75     79044
weighted avg       0.90      0.88      0.86     79044
```

Out[10]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2be8f0a4a30>
```

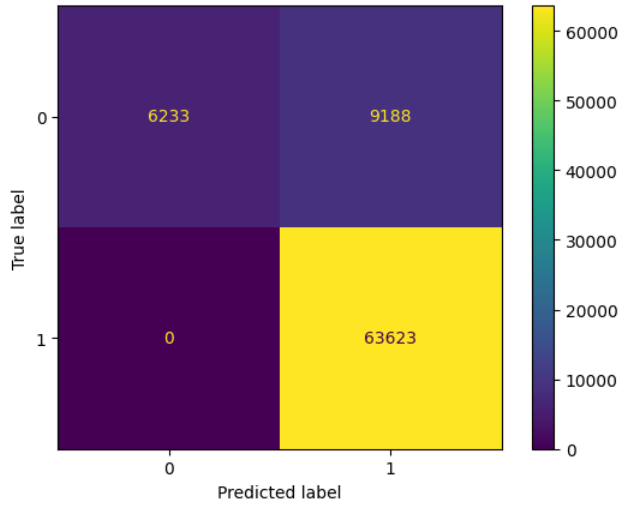

Figure 72. Random Forest model classification report on best estimator

10.6. XGBoost

In [9]:
```
# GridSearchCV for XGBoost
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.05, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}

grid_xgb = GridSearchCV(XGBClassifier(), param_grid, verbose=3)
grid_xgb.fit(X_train, y_train)

print("Best parameters: ", grid_xgb.best_params_)
print("Best estimator: ", grid_xgb.best_estimator_)
```

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits
[CV 1/5] END learning_rate=0.05, max_depth=3, n_estimators=100;, score=0.887 total time=   0.7s
[CV 2/5] END learning_rate=0.05, max_depth=3, n_estimators=100;, score=0.887 total time=   0.7s
[CV 3/5] END learning_rate=0.05, max_depth=3, n_estimators=100;, score=0.888 total time=   0.7s
[CV 4/5] END learning_rate=0.05, max_depth=3, n_estimators=100;, score=0.889 total time=   0.7s
[CV 5/5] END learning_rate=0.05, max_depth=3, n_estimators=100;, score=0.888 total time=   0.7s
[CV 1/5] END learning_rate=0.05, max_depth=3, n_estimators=200;, score=0.887 total time=   1.2s
[CV 2/5] END learning_rate=0.05, max_depth=3, n_estimators=200;, score=0.887 total time=   1.3s
[CV 3/5] END learning_rate=0.05, max_depth=3, n_estimators=200;, score=0.888 total time=   1.6s
[CV 4/5] END learning_rate=0.05, max_depth=3, n_estimators=200;, score=0.889 total time=   3.2s
[CV 5/5] END learning_rate=0.05, max_depth=3, n_estimators=200;, score=0.888 total time=   3.0s
[CV 1/5] END learning_rate=0.05, max_depth=3, n_estimators=300;, score=0.888 total time=   3.0s
[CV 2/5] END learning_rate=0.05, max_depth=3, n_estimators=300;, score=0.888 total time=   2.6s
[CV 3/5] END learning_rate=0.05, max_depth=3, n_estimators=300;, score=0.889 total time=   2.1s
[CV 4/5] END learning_rate=0.05, max_depth=3, n_estimators=300;, score=0.890 total time=   1.9s
```

Figure 73. XGBoost - Initial param grid and searching for best estimator

```
Best parameters:  {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 300}
Best estimator:  XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=5, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=300, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

In [10]:
```
# Evaluate the model after GridSearchCV
y_test_pred = grid_xgb.predict(X_test)
print('XGBoost after GridSearchCV')
print(classification_report(y_test, y_test_pred))
cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid_xgb.classes_)
disp.plot()
```

```
XGBoost after GridSearchCV
              precision    recall  f1-score   support

           0       0.93      0.48      0.63     15421
           1       0.89      0.99      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.91      0.73      0.78     79044
weighted avg       0.90      0.89      0.88     79044
```

Out[10]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1e000e98df0>
```

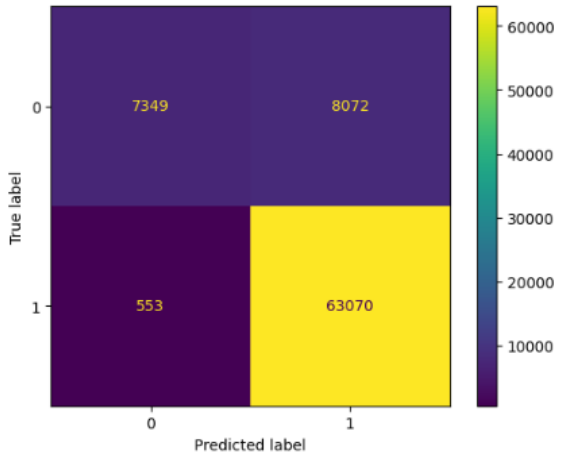

Figure 74. XGBoost model classification report on best estimator

IV. ALGORITHMS EVALUATION

1. Evaluation Metrics For Classification

With:

TP (True Positives): Instances correctly predicted as positive.
TN (True Negatives): Instances correctly predicted as negative.
FP (False Positives): Instances incorrectly predicted as positive.
FN (False Negatives): Instances incorrectly predicted as negative.

- **Accuracy:** measures the proportion of correctly classified instances (both true positives and true negatives) among the total number of instances.

$$Accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

Significance	Accuracy is a straightforward metric that indicates overall model performance. It's easy to interpret and understand because it gives a single measure of correct predictions relative to all predictions.
Reason for choosing	Its simplicity in measuring overall correctness and ease of interpretation, making it ideal for balanced class distributions

- **Recall:** measures the proportion of actual positive instances that are correctly identified by the classifier.

$$Recall = \frac{TP}{TP + FN}$$

Significance	Recall is important when the cost of missing a positive (false negative) is high. It indicates how well the model identifies positive instances from all actual positives.
Reason for choosing	It's crucial in applications where identifying all positives is more important than correctly labeling negatives.

- **Precision:** measures the proportion of instances predicted as positive that are actually positive.

$$Precision = \frac{TP}{TP + FP}$$

Significance	Precision is useful when the cost of false positives is high. It quantifies how many of the predicted positive instances are actually positive.
Reason for choosing	It helps in scenarios where we want to be confident that the predicted positives are indeed correct, even if some positives are missed.

- **F1-Score:** is the harmonic mean of precision and recall, providing a single metric that balances both measures.

$$F1 - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Significance	F1-Score considers both false positives and false negatives, making it a balanced measure of model performance. It's particularly useful when the class distribution is imbalanced.
Reason for choosing	It takes into account the type of errors — false positive and false negative – and not just the number of predictions that were incorrect.

- **Excution time:** refers to the time taken by the classifier to make predictions on a dataset or individual instances.

Significance	Execution time is critical for real-time applications or when dealing with large datasets. It affects the practical usability and scalability of the model.
Reason for choosing	It directly impacts the practical efficiency, and cost-effectiveness of deploying a classification model.

2. Present Results

```
In [26]: # Plot all Classification Reports
from sklearn.metrics import classification_report

models = [log_reg, nb, svc, dt, rf, xgb]
model_names = ['Logistic Regression', 'Naive Bayes', 'Support Vector Machine', 'Decision Tree', 'Random Forest', 'XGBoost']
reports = [classification_report(y_test, model.predict(X_test)) for model in models]

for model_name, report in zip(model_names, reports):
    print(model_name)
    print(report)
```

```
Logistic Regression
              precision    recall  f1-score   support

           0       0.92      0.47      0.62     15421
           1       0.88      0.99      0.93     63623

    accuracy                           0.89     79044
   macro avg       0.90      0.73      0.78     79044
weighted avg       0.89      0.89      0.87     79044

Decision Tree
              precision    recall  f1-score   support

           0       0.56      0.59      0.58     15421
           1       0.90      0.89      0.89     63623

    accuracy                           0.83     79044
   macro avg       0.73      0.74      0.74     79044
weighted avg       0.83      0.83      0.83     79044

Naive Bayes
              precision    recall  f1-score   support

           0       0.88      0.45      0.59     15421
           1       0.88      0.99      0.93     63623

    accuracy                           0.88     79044
   macro avg       0.88      0.72      0.76     79044
weighted avg       0.88      0.88      0.86     79044

Random Forest
              precision    recall  f1-score   support

           0       0.95      0.46      0.62     15421
           1       0.88      0.99      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.92      0.73      0.78     79044
weighted avg       0.90      0.89      0.87     79044

Support Vector Machine
              precision    recall  f1-score   support

           0       0.96      0.45      0.61     15421
           1       0.88      1.00      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.92      0.72      0.77     79044
weighted avg       0.90      0.89      0.87     79044

XGBoost
              precision    recall  f1-score   support

           0       0.91      0.48      0.63     15421
           1       0.89      0.99      0.94     63623

    accuracy                           0.89     79044
   macro avg       0.90      0.74      0.78     79044
weighted avg       0.89      0.89      0.88     79044
```

Figure 75. Overall Models Classification reports

```
# Plot all Confusion Matrices in one figure
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

models = [log_reg, nb, svc, dt, rf, xgb]
model_names = ['Logistic Regression', 'Naive Bayes', 'Support Vector Machine', 'Decision Tree', 'Random Forest', 'XGBoost']
cms = [confusion_matrix(y_test, model.predict(X_test)) for model in models]

fig, axs = plt.subplots(2, 3, figsize=(20, 10))
for ax, model_name, cm in zip(axs.flatten(), model_names, cms):
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf.classes_)
    disp.plot(ax=ax)
    ax.set_title(model_name)
```

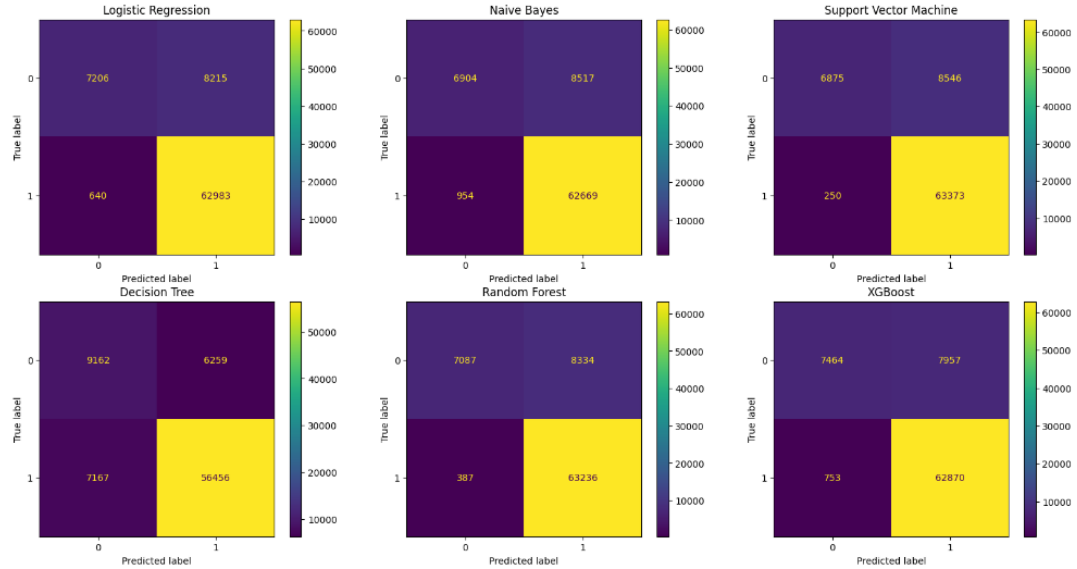

Figure 76. Overall Models Confusion Matrix

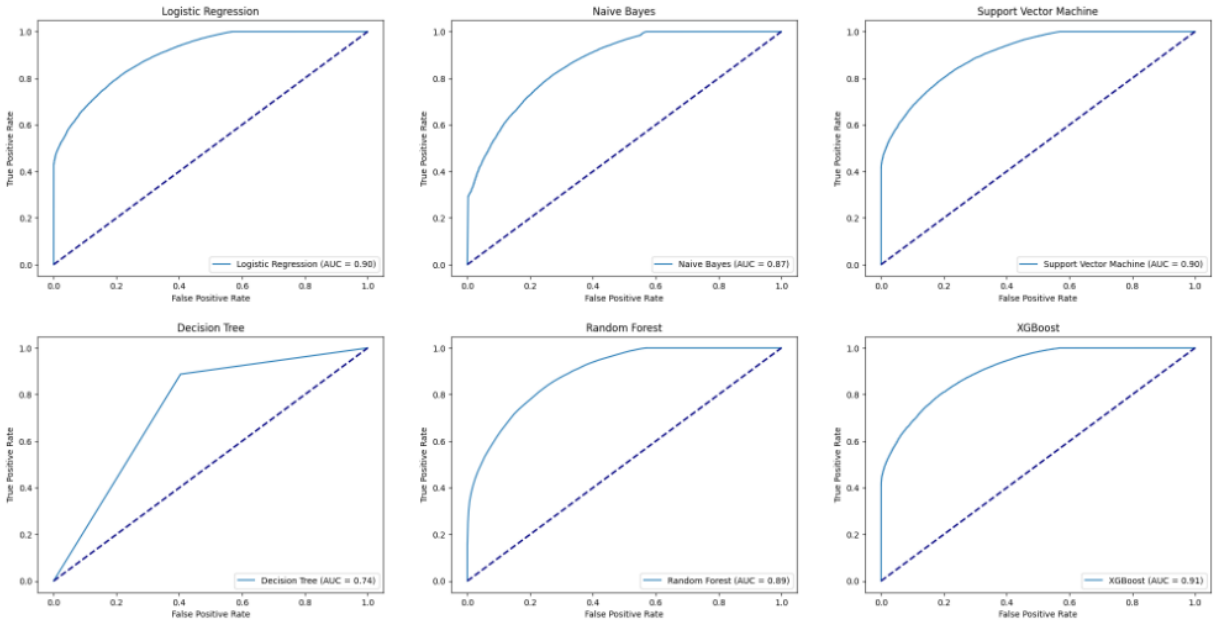

Figure 77. Overall Models ROC/AUC Curves

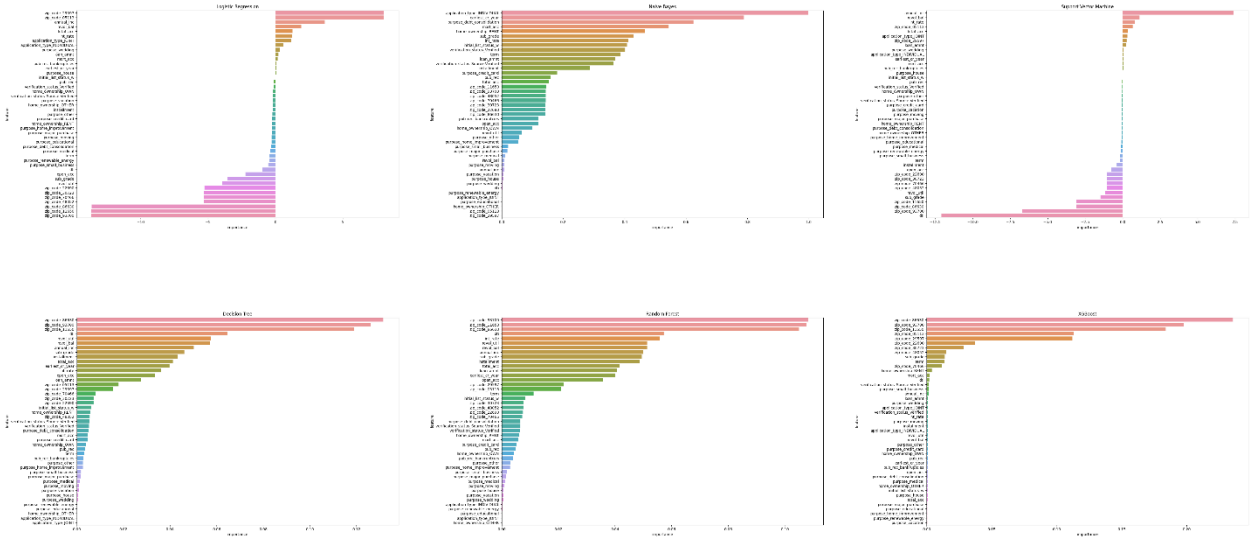

Figure 78. Overall Models Feature Importance (Clear details in attached code)

3. Compare and evaluate

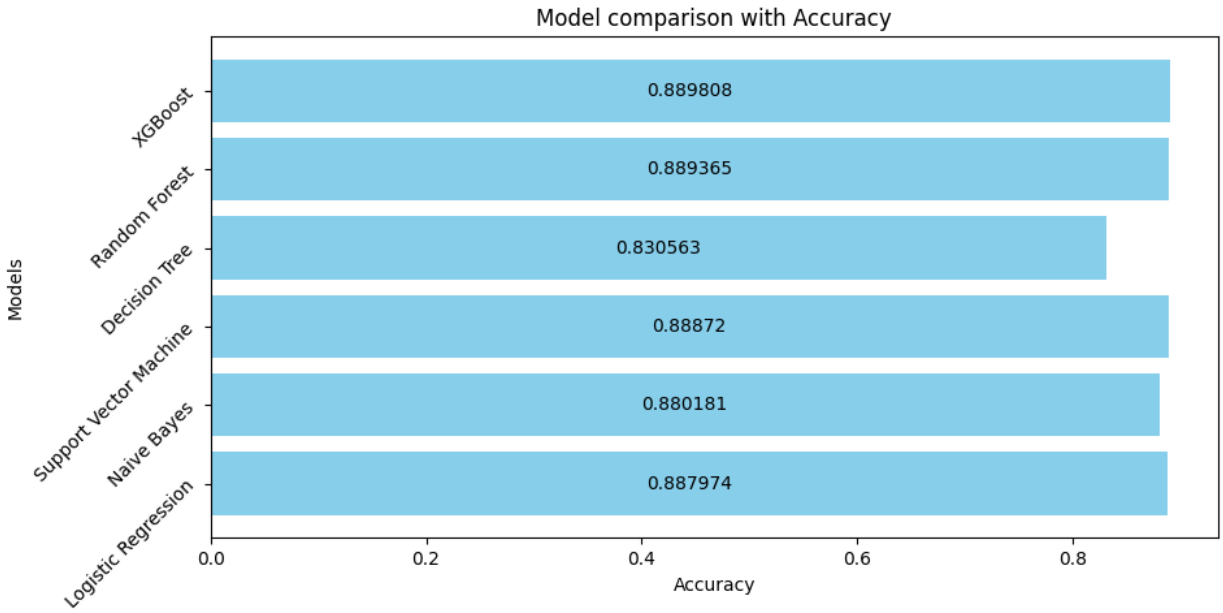

Figure 79. Models Comparison with Accuracy

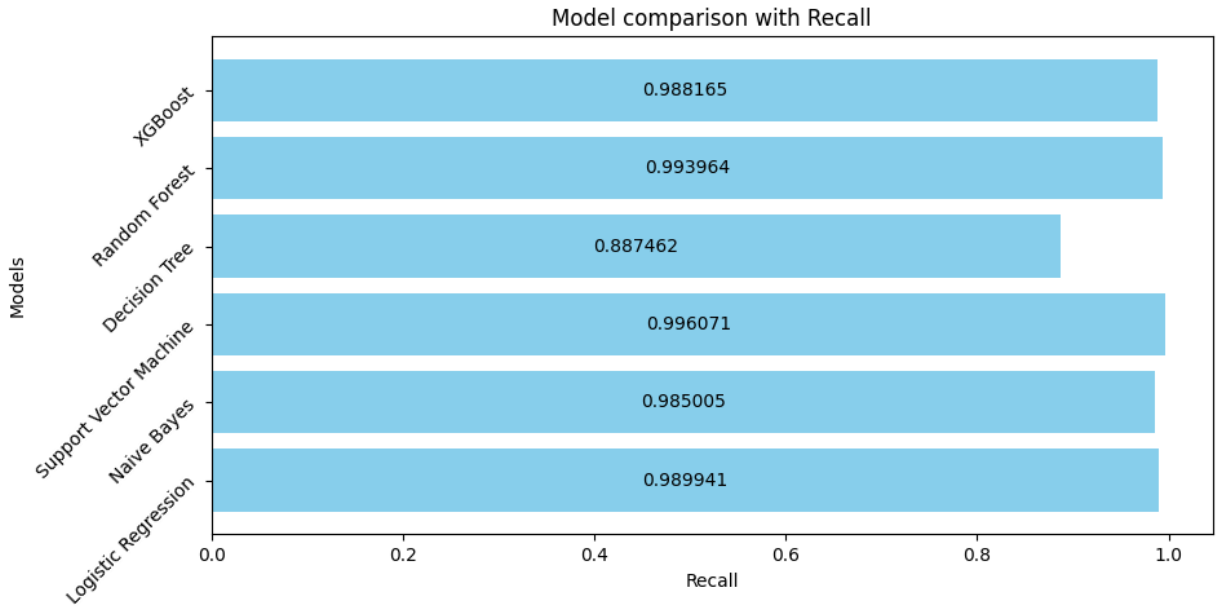

Figure 80. Models Comparison with Recall

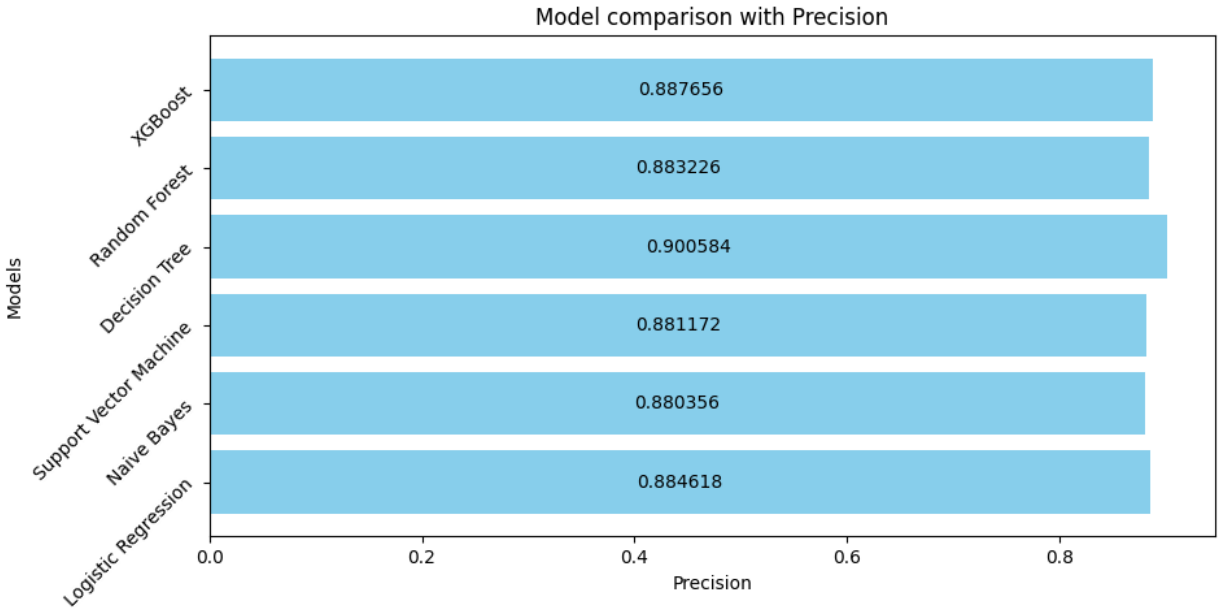

Figure 81. Models Comparison with Precision

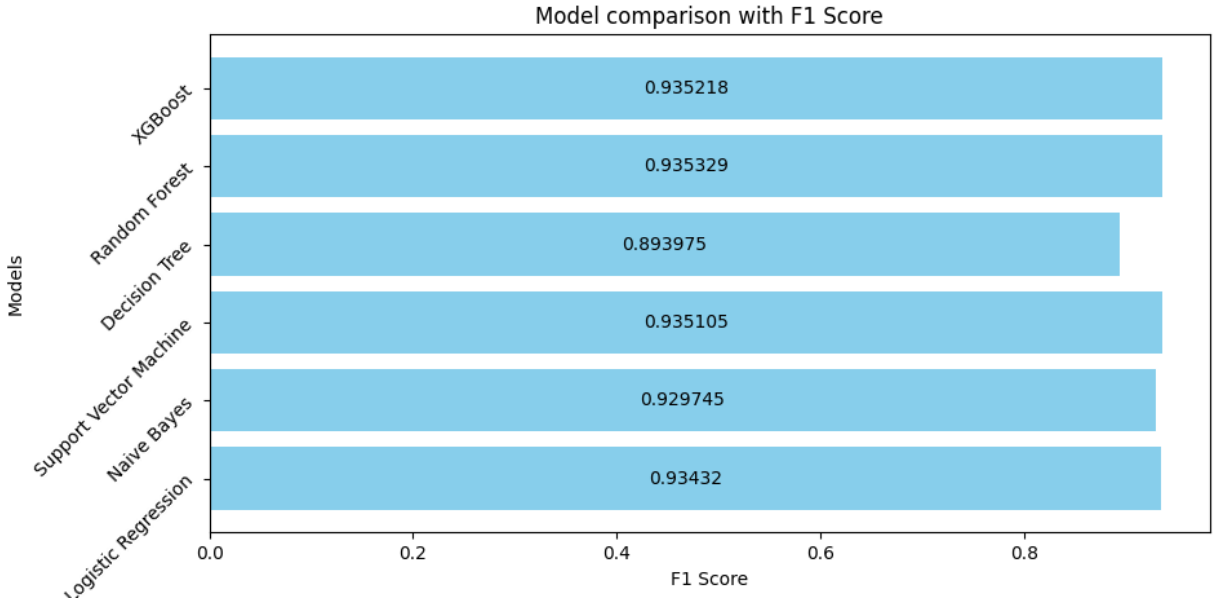

Figure 82. Models Comparison with F1-Score

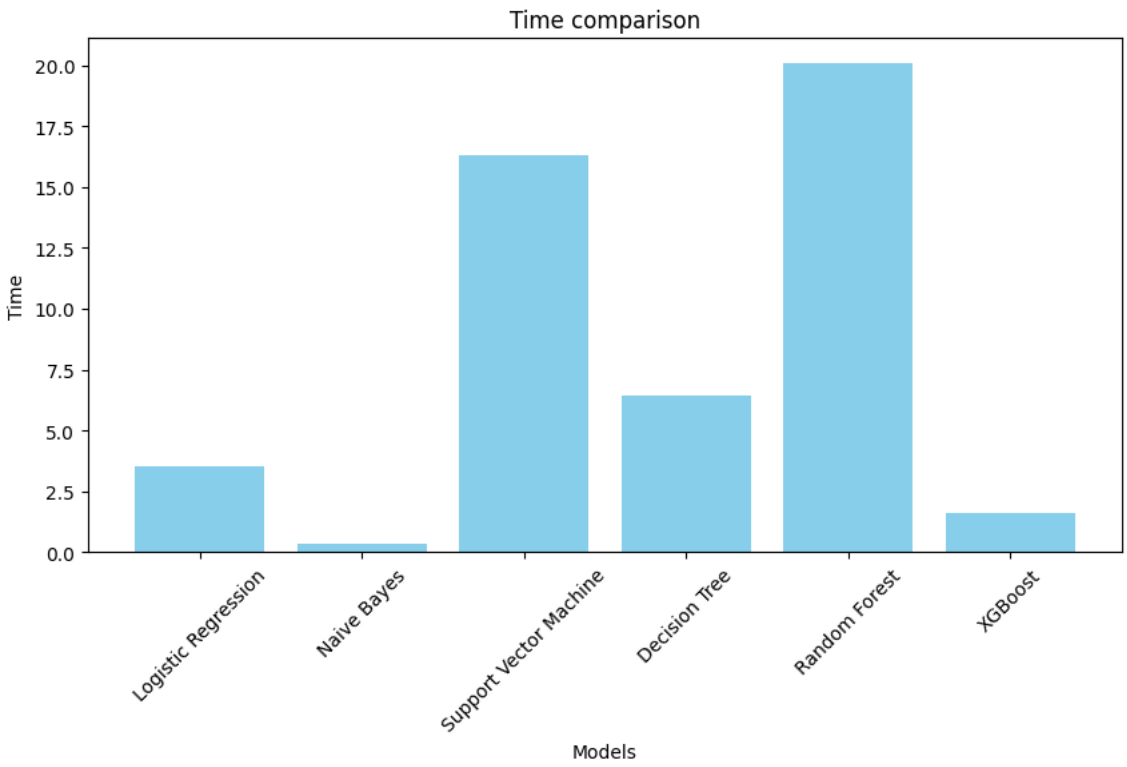

Figure 83. Models Comparison with Execution Time

Model Comparisons

To evaluate the performance of different machine learning algorithms on the loan prediction dataset, several models were trained and tested. The models compared include:

- ✓ Logistic Regression
- ✓ Naive Bayes (GaussianNB)
- ✓ Support Vector Machine (LinearSVC)
- ✓ Decision Tree Classifier
- ✓ Random Forest Classifier
- ✓ XGBoost Classifier

The models were assessed based on their accuracy, recall, precision, F1-score, and execution time. Here are the key findings from the model comparison:

Model	Accuracy	Recall	Precision	F1-Score	Execution time
Logistic Regression	0.887974	0.989941	0.884618	0.93432	~4s
Naïve Bayes	0.880181	0.985005	0.880356	0.929745	~1s
SVM	0.88872	0.996071	0.881172	0.935105	~16.5s
Decision Tree	0.830563	0.887462	0.900584	0.893975	~6.5s
Random Forest	0.889365	0.993964	0.883226	0.935329	~20s
XGBoost	0.889808	0.988165	0.887656	0.935218	~2s

Conclusion

Based on the comparative analysis, the following conclusions can be drawn:

1. **XGBoost Classifier** and **Random Forest Classifier** emerged as the top-performing models. Both achieved high accuracy and balanced precision, recall, and F1-scores. XGBoost, with its shorter execution time, is highly suitable for deployment where computational efficiency is critical.
2. **Support Vector Machine** also provided competitive results with high accuracy and recall, making it a good choice for scenarios prioritizing true positive rates. However, its longer execution time may be a consideration for large datasets.
3. **Logistic Regression** offered decent performance across all metrics and is a viable option when model simplicity and interpretability are important.
4. **Naive Bayes** demonstrated good performance with the shortest execution time, making it suitable for real-time applications where speed is a priority, despite its slightly lower accuracy.

5. **Decision Tree** showed lower accuracy and was prone to overfitting, which limits its effectiveness on unseen data. However, it remains valuable for its interpretability.

In conclusion, the **XGBoost Classifier** is recommended for deployment due to its high accuracy, robust performance across different metrics, and efficient execution time. Future work could involve further hyperparameter tuning and exploring advanced ensemble methods to enhance model performance even further.

4. Baseline model comparison with cross validation

Baseline model comparison with cross-validation is a crucial step to evaluate and select the best-performing model. Initially, simple baseline models like linear regression or decision trees are chosen to establish performance benchmarks. Using k-fold cross-validation, the dataset is divided into k subsets, with each model trained and validated k times, ensuring a robust assessment of performance. Key metrics such as accuracy, precision, recall, or mean squared error are computed during this process, providing a comprehensive evaluation. By comparing these metrics across baseline models, one can identify the model that best generalizes to unseen data, setting a reliable standard for more complex model development.

Baseline Model Comparison

In [21]:
```
# Baseline Model Comparison
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
# Import the models
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

models = [
    LogisticRegression(),
    GaussianNB(),
    LinearSVC(),
    DecisionTreeClassifier(),
    RandomForestClassifier(n_jobs=-1),
    XGBClassifier()
]
```

In [22]:
```
from sklearn.model_selection import StratifiedKFold
# Cross validation score
from sklearn.model_selection import cross_val_score

def generate_baseline_report(models, X, y, cv=5, metric='accuracy'):
    # Define k-fold cross validation
    skf = StratifiedKFold(n_splits=cv, random_state=42, shuffle=True)
    entries = []
    for model in models:
        model_name = model.__class__.__name__
        scores = cross_val_score(model, X, y, scoring=metric, cv=skf)
        for fold_idx, score in enumerate(scores):
            entries.append((model_name, fold_idx, score))

    df = pd.DataFrame(entries, columns=['model_name', 'fold_idx', metric])


    return df

baseline_report = generate_baseline_report(models, X, y)
baseline_report
```

Figure 84. Initial Baseline models

	model_name	fold_idx	accuracy
0	LogisticRegression	0	0.887695
1	LogisticRegression	1	0.887860
2	LogisticRegression	2	0.887367
3	LogisticRegression	3	0.887000
4	LogisticRegression	4	0.889048
5	GaussianNB	0	0.879877
6	GaussianNB	1	0.878776
7	GaussianNB	2	0.879460
8	GaussianNB	3	0.879725
9	GaussianNB	4	0.882861
10	LinearSVC	0	0.887898
11	LinearSVC	1	0.887733
12	LinearSVC	2	0.887253
13	LinearSVC	3	0.887721
14	LinearSVC	4	0.889744
15	DecisionTreeClassifier	0	0.829816
16	DecisionTreeClassifier	1	0.830120
17	DecisionTreeClassifier	2	0.828349
18	DecisionTreeClassifier	3	0.831448
19	DecisionTreeClassifier	4	0.830978
20	RandomForestClassifier	0	0.888277
21	RandomForestClassifier	1	0.888505
22	RandomForestClassifier	2	0.887582
23	RandomForestClassifier	3	0.888581
24	RandomForestClassifier	4	0.890756
25	XGBClassifier	0	0.889707
26	XGBClassifier	1	0.889543
27	XGBClassifier	2	0.887658
28	XGBClassifier	3	0.888391
29	XGBClassifier	4	0.890958

Figure 85. Model Cross valuation result

In [23]:
```
# Summary report
models_comp = baseline_report.groupby('model_name').agg({
    'accuracy': ['mean', 'std']
}).sort_values(('accuracy', 'mean'), ascending=False)

models_comp = models_comp.reset_index()
models_comp.columns = ['model_name', 'mean_accuracy', 'std_accuracy']
models_comp
```

Out[23]:

	model_name	mean_accuracy	std_accuracy
0	XGBClassifier	0.889251	0.001273
1	RandomForestClassifier	0.888740	0.001194
2	LinearSVC	0.888070	0.000966
3	LogisticRegression	0.887794	0.000775
4	GaussianNB	0.880140	0.001579
5	DecisionTreeClassifier	0.830142	0.001197

Figure 86. Summary report on cross validation

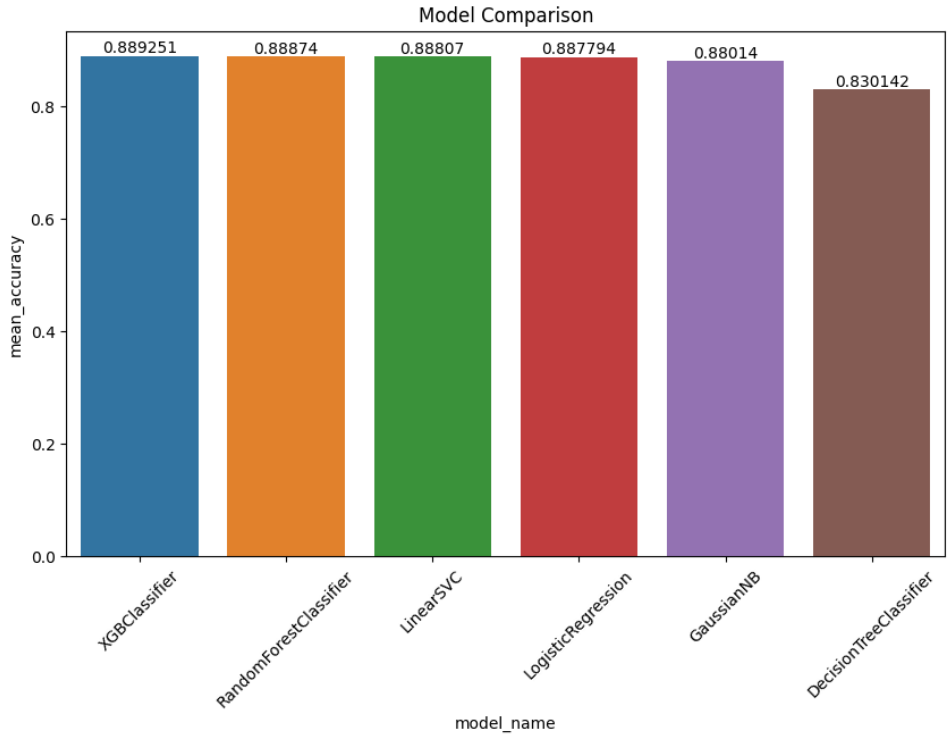

Figure 87. Baseline Models comparison

Conclusion

Given the comprehensive evaluation and consistent results across various metrics and validation techniques, the **XGBoost Classifier** stands out as the most effective model for the loan prediction problem. Its high accuracy, balanced performance, and efficiency make it the optimal choice for deployment in real-world scenarios.

Future work could focus on:

- Further fine-tuning the hyperparameters of the XGBoost model to potentially enhance its performance.
- Exploring more advanced ensemble methods or hybrid models that could combine the strengths of different algorithms.
- Implementing feature engineering techniques to extract more meaningful features from the dataset, potentially boosting the model's accuracy and robustness.

In summary, the XGBoost Classifier's superior performance, as demonstrated through baseline comparisons and cross-validation, makes it the recommended model for accurately predicting loan defaults in this study.

V. APPLICATION PROGRAM UTILIZING RESULTS

We have developed a simple application using Streamlit to utilize the models trained on the Lending Club loan dataset. This application allows users to select the appropriate model and input loan information to predict whether the loan will be repaid or not.

The application provides an intuitive user interface, enabling users to input necessary information such as loan amount, interest rate, loan term, and personal details. It also calculates the monthly installment base on the term and loan amount, debt-to-income (dti) ratio. After entering the information, users can choose a prediction model from a list of trained models, including classification algorithms like Logistic Regression, Random Forest, and Gradient Boosting.

The prediction results are displayed instantly, giving users an overview of the loan's repayment likelihood. This application not only supports individual users but can also be useful for investors or financial institutions in assessing the credit risk of loans.

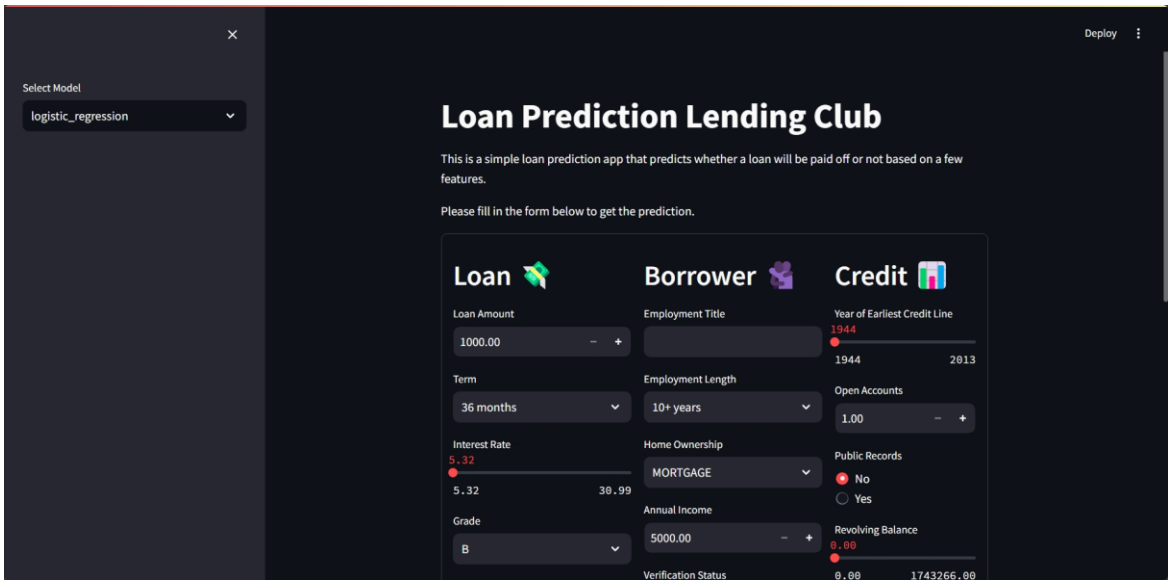

Figure 88. Main page of Loan Prediction application

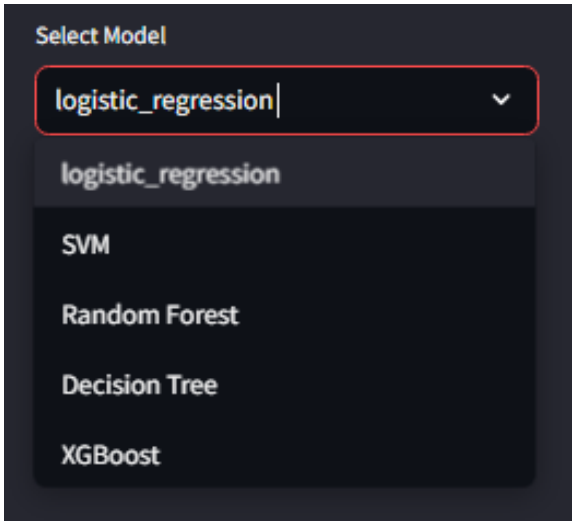

Figure 89. Deploy all model that have tested before

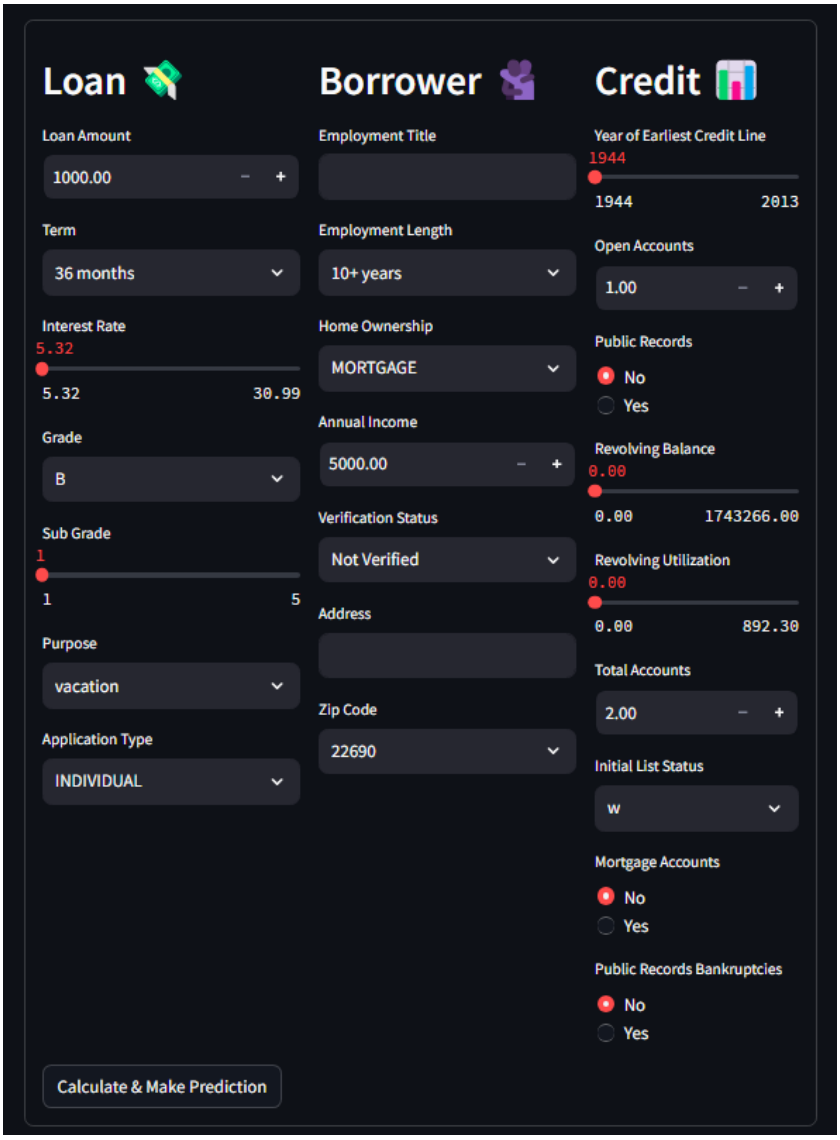

Figure 90. Loan application form include Loan, Borrower & Credit Characteristics

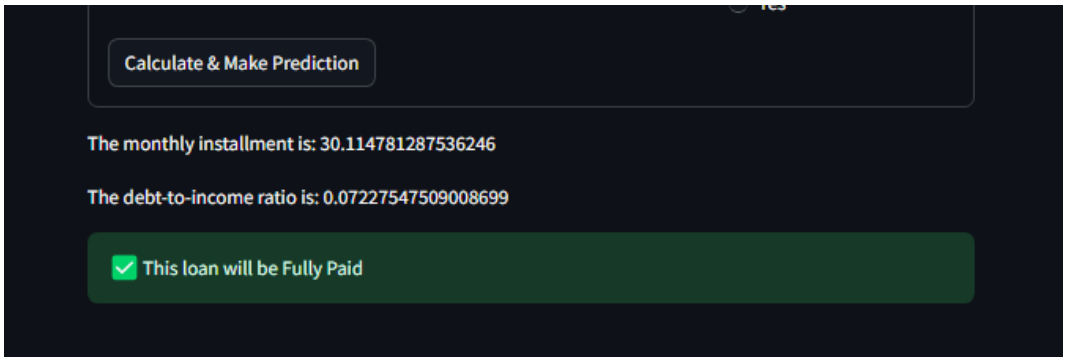

Figure 91. Calculation & Model Prediction result

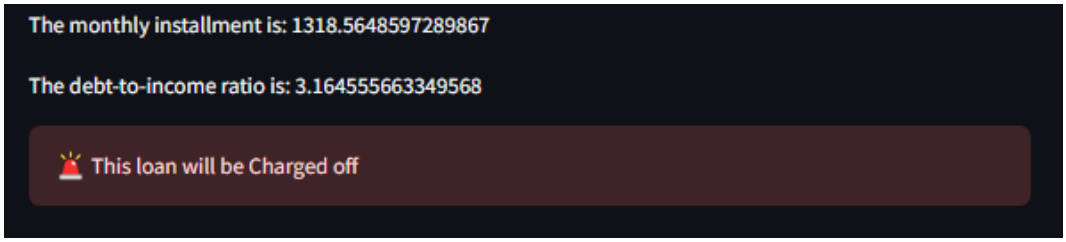

Figure 92. Negative result of loan application

VI. CONCLUSION

1. Advantages

- **Enhanced Risk Prediction**: The application of machine learning classification algorithms on Lending Club data significantly enhances the accuracy of credit risk predictions. Advanced models like Random Forest, Gradient Boosting (XGBoost), and Support Vector Machines can effectively capture complex patterns and relationships within the data, leading to more reliable assessments of borrower risk profiles and potential loan defaults.
- **Data-Driven Decision Making**: By leveraging the wealth of data available from Lending Club, this approach facilitates data-driven decision-making processes. Insights derived from analyzing features such as loan amount, interest rate, employment length, credit score, and payment history provide valuable guidance for both lenders and platform operators, improving the overall efficiency and effectiveness of risk management strategies.
- **Scalability and Adaptability**: Machine learning models offer scalability and adaptability, enabling the system to handle large volumes of data and continuously improve as more data becomes available. This adaptability is crucial in the dynamic environment of P2P lending, where borrower behavior and market conditions can change rapidly.

2. Limitations

- **Data Quality and Completeness**: The accuracy and reliability of machine learning models are heavily dependent on the quality and completeness of the dataset. Missing

or inaccurate data can lead to biased or incorrect predictions. In the context of Lending Club, ensuring the integrity of data across different periods and borrower profiles remains a challenge.

- **Model Interpretability**: While advanced machine learning models like XGBoost and Support Vector Machines offer high predictive power, they often lack interpretability. This can pose difficulties in understanding the underlying factors driving the predictions, making it harder for stakeholders to trust and act on the results without additional validation.
- **Regulatory and Ethical Considerations**: The use of machine learning in credit risk assessment must navigate regulatory and ethical considerations, including issues of fairness, transparency, and bias. Ensuring that models do not disproportionately disadvantage certain groups and comply with financial regulations is critical for the sustainable deployment of these technologies in P2P lending.

3. Development Orientation

- **Improving Data Quality**: Future work should focus on improving the quality and completeness of the dataset. This can be achieved through better data collection practices, rigorous data cleaning processes, and leveraging external data sources to fill gaps. Enhanced data quality will lead to more accurate and reliable predictions.
- **Model Interpretability and Transparency**: Developing methods to improve the interpretability and transparency of machine learning models is essential. Techniques such as SHAP (SHapley Additive exPlanations) values or LIME (Local Interpretable Model-agnostic Explanations) can help stakeholders understand model predictions better, fostering trust and facilitating informed decision-making.
- **Addressing Ethical and Regulatory Challenges**: Ensuring compliance with regulatory standards and addressing ethical concerns is vital for the responsible use of machine learning in credit risk assessment. Ongoing research should focus on developing fair and unbiased models, incorporating ethical considerations into the algorithm design, and adhering to relevant financial regulations to protect both borrowers and lenders.

REFERENCES

[1] Lending club loan data. (2021, June 17). Kaggle. https://www.kaggle.com/datasets/adarshsng/lending-club-loan-data-csv

[2] Li, P., & Han, G. (2018). LendingClub loan default and profitability prediction. CS229: *Computer Science.*

[3] Bhagat, A. (2018). *Predicting Loan Defaults using Machine Learning Techniques* (Doctoral dissertation, California State University, Northridge).

[4] *Prediction of the Borrowers' Payback to the Loan with Lending Club Data*. (2020, September 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9384612

[5] Yang, R. (2024). Machine Learning-Based Loan Default Prediction in Peer-to-Peer Lending. *Highlights in Science, Engineering and Technology*, 94, 310-318.

[6] Reddy, S., & Gopalaraman, K. (2016, November 4). *PEER TO PEER LENDING, DEFAULT PREDICTION-EVIDENCE FROM LENDING CLUB*. Open Access Journals. https://www.icommercecentral.com/open-access/peer-to-peer-lending-default-predictionevidence-from-lending-club.php?aid=81766

[7] *Data Mining: Concepts and techniques.* (n.d.). ScienceDirect. https://www.sciencedirect.com/book/9780123814791/data-mining-concepts-and-techniques

[8] Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785-794).