Project title	**Regression Model Using Logistic Regression**
Module Name	**NICF Principals of Machine Learning (SF)**
Qualification Name	**NICF Diploma in Infocomm Technology (Data)**

Student name	**Assessor name**
TAN DAI JUN	SHANTI SEKHAR

Date issued	**Completion date**	**Submitted on**
28 MARCH, 2022	30 MARCH, 2022	30 MARCH, 2022

Project title	**Regression Model Using Logistic Regression**

Learner declaration
I certify that the work submitted for this assignment is my own and research sources are fully acknowledged. Student signature: DJ Date: 30 MARCH, 2022

Table of Contents

1. Introduction 1

2. Methodology 2

3. Project Execution 3

3.1 Dataset overview 3

3.2 Data pre-processing 8

3.3 Univariate exploratory data analysis 9

3.4 Bivariate exploratory data analysis 13

3.5 Data transformation 18

3.6 Data splitting 20

3.7 Model fitting and evaluation with default regularization solver lbfgs 20

3.8 Model fitting and evaluation using SMOTE for oversampling with default regularization solver lbfgs 25

3.9 Model fitting and evaluation using SMOTE for oversampling with regularization solver saga 28

4. Conclusion 31

1. Introduction

In supervised machine learning, classification is one of the techniques that has been widely implemented to predict the qualitive response. Logistic regression is one of a classification technique in machine learning that uses one or more independent variables to predict the dichotomous dependent variables. In this project, the application of machine learning will be use in banking and finance industry.

DRS bank is facing challenging times as their NPAs (*non-performing assets*) has been on a rise recently and a large part of these are due to the loans given to individual customers (*borrowers*). Chief Risk Officer of the bank decides to put in a scientifically robust framework for approval of loans to individual customers to minimize the risk of loan loans converting into NPAs and initiates a project for the data science team at the bank.

The objective of this project is to identify the criteria to approve loans for an individual customer such that the likelihood of the loan delinquency is minimized. In order to achieve the project goal, we will study and understand the Loan_Delinquent_Dataset, create a classification model to identify potential loan delinquency threat and what are the factors that drive the behaviour of loan delinquency.

Table below describes some of the technical jargons used in banking and finance industry.

Jargon	Description
Transactor	A person who pays his/her due amount balance full and on time.
Revolver	A person who pays the minimum due amount but keeps revolving his balance and does not pay the full amount.
Delinquent	Delinquency means that you are behind on payments, a person who fails to pay even the minimum due amount.
Defaulter	Once you are delinquent for a certain period (*usually 6 months*) your lender will decide you to be in default stage.
Risk analytics	A wide domain in the financial and banking industry, basically analysing the risk of the customer.

2. Methodology

As a methodology of this project, we first extract the "Loan_Delinquency_Dataset" dataset which is in csv format. Since we do not have prior knowledge regarding on the dataset, it is best to implement exploratory data analysis to understand each variable and their respective characteristics with respect to the context of this project. This will greatly help us in deciding the correct approach for imputations so that the dataset is correctly processed and won't affect our analysis and modelling in further stages of the project.

Once the data exploration and imputation are done, we begin with model training and lastly model evaluation. Diagram below describes the methodology of this project conceptually.

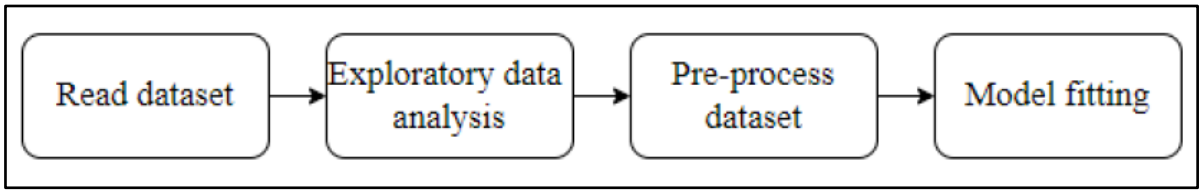

Figure 2.0: Conceptual diagram of methodology of this project

Noted that this entire project will be done using Python in Jupyter Notebook. Table below showcases some of the powerful libraries used in this project.

Library	Descriptions
numpy	House with large collection of mathematical functions to operate arrays.
pandas	Built on top of numpy, offers data structure and operations for manipulating numerical table and time series.
matplotlib	A cross-platform, data visualization and graphical plotting library.
seaborn	Built on top of matplotlib, provides high level interface for drawing attractive and informative statistical graphics.
sklearn	House with large collection of machine learning computation and operation functions.
scipy	House with large collection of scientific and technical computing functinos.

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import seaborn as sns
5  from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
6  from sklearn.metrics import accuracy_score,precision_score,recall_score
7  from sklearn.model_selection import GridSearchCV
8  from sklearn import metrics
9  import scipy.stats as stats
10 from sklearn.linear_model import LogisticRegression
11 import warnings
12 warnings.filterwarnings('ignore')
```

Figure 2.1: Necessary libraries imported for this project

3. Project Execution

With all the necessary libraries imported, we are ready to execute this project. Noted that for every crucial execution, there will be a short summary and explanation throughout this section.

3.1 Dataset overview

In this section, we will read the "Loan_Delinquency_Dataset.csv" dataset using pandas from Python. To prevent tempering with original dataset, we store the extracted dataset to another variable named "loan". We can notice that the dataset contains **11548 observations** and **8 features**.

```
1  df = pd.read_csv("Loan_Delinquent_Dataset.csv")
```

```
1  # copying data to another varaible to avoid any changes to original data
2  loan=df.copy()
```

Figure 3.1.0: Extract dataset and assign to variable "loan"

```
1  loan.shape
```

```
(11548, 8)
```

Figure 3.1.1: Dimension of the dataset

We can eyeball the first and last 5 rows of observations of the dataset.

```
1  loan.head()
```

	ID	isDelinquent	term	gender	purpose	home_ownership	age	FICO
0	1	1	36 months	Female	House	Mortgage	>25	300-500
1	2	0	36 months	Female	House	Rent	20-25	>500
2	3	1	36 months	Female	House	Rent	>25	300-500
3	4	1	36 months	Female	Car	Mortgage	>25	300-500
4	5	1	36 months	Female	House	Rent	>25	300-500

```
1  loan.tail()
```

	ID	isDelinquent	term	gender	purpose	home_ownership	age	FICO
11543	11544	0	60 months	Male	other	Mortgage	>25	300-500
11544	11545	1	36 months	Male	House	Rent	20-25	300-500
11545	11546	0	36 months	Female	Personal	Mortgage	20-25	>500
11546	11547	1	36 months	Female	House	Rent	20-25	300-500
11547	11548	1	36 months	Male	Personal	Mortgage	20-25	300-500

Figure 3.1.2: First and last 5 rows observations of the dataset

Table below describes each feature found in the dataset.

Column	Description
ID	Customer unique identifier
isDelinquent	Value of 1 indicates the customer is delinquent and vice versa for value of 0.
term	Loan term in months.
gender	Gender of the borrower.
age	Age of the borrower.
purpose	Purpose of the loan.
home_ownership	Status of borrower's home.
FICO	FICO score of the borrower.

To further understand more about FICO score, it is a three-digit number based on the information in borrower credit reports. It helps lenders determine how likely a borrower repay their loan. Thus, it dictates how much a person can borrow, how many months they have to repay and the corresponding interest rate.

Based on figure 3.1.2 and data descriptions, we can understand that most of the important features used in the dataset are categorical variable. On the other hand, we can now further understand the dataset by exploring the data type of each feature.

```
1  loan.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11548 entries, 0 to 11547
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   ID              11548 non-null  int64
 1   isDelinquent    11548 non-null  int64
 2   term            11548 non-null  object
 3   gender          11548 non-null  object
 4   purpose         11548 non-null  object
 5   home_ownership  11548 non-null  object
 6   age             11548 non-null  object
 7   FICO            11548 non-null  object
dtypes: int64(2), object(6)
memory usage: 721.9+ KB
```

Figure 3.1.3: Data type of all features in dataset

From figure 3.1.3, we can observe that the whole dataset has costed around 722 kb of hard disc space. Besides, we can notice that only "ID" and "isDelinquent" are numerical variables. On the other hand, we can further study the dataset by checking out all unique values for each of the features.

```
# avoid cell value truncated
pd.set_option('display.max_colwidth', None)
all_values = []
for col in df.columns:
    all_values.append(df[col].unique())

unique_df = pd.DataFrame({'Column':df.columns, 'Unique Values':all_values})
unique_df
```

Figure 3.1.4: Source code to create a data frame that displays all unique values of each feature

	Column	Unique Values
0	ID	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ...]
1	isDelinquent	[1, 0]
2	term	[36 months, 60 months]
3	gender	[Female, Male]
4	purpose	[House, Car, Other, Personal, Wedding, Medical, other]
5	home_ownership	[Mortgage, Rent, Own]
6	age	[>25, 20-25]
7	FICO	[300-500, >500]

Figure 3.1.5: Table of each column and corresponding unique values

From figure 3.1.5, there are several highlights

- Only ID column is actually a numerical variable. The remaining features are all categorical variables that use numerical value for representation.
- There is a duplicate in “purpose” column as the value “other” and “Other” are having the same meaning.

Having knowing the highlights, we can now factorize all the categorical variables.

```
loan["term"] = loan["term"].astype("category")
loan["gender"] = loan["gender"].astype("category")
loan["purpose"] = loan["purpose"].astype("category")
loan["home_ownership"] = loan["home_ownership"].astype("category")
loan["age"] = loan["age"].astype("category")
loan["FICO"] = loan["FICO"].astype("category")
loan["isDelinquent"] = loan["isDelinquent"].astype("category")
```

Figure 3.1.6: Source code to factorize all the categorical variables

Once we are done with the factorization, we can check the data type for all the columns again to ensure the factorization is successfully executed.

```
1 loan.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11548 entries, 0 to 11547
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   ID              11548 non-null  int64
 1   isDelinquent    11548 non-null  int64
 2   term            11548 non-null  category
 3   gender          11548 non-null  category
 4   purpose         11548 non-null  category
 5   home_ownership  11548 non-null  category
 6   age             11548 non-null  category
 7   FICO            11548 non-null  category
dtypes: category(6), int64(2)
memory usage: 249.1 KB
```

Figure 3.1.7: Data types of all columns after factorization

Based on figure 3.1.7, we can clearly observe that the factorization is successful. One main thing to highlight is that the size of the dataset has been greatly reduced from 722kb to 249.1kb. on the other hand, we obtain the summary of all the columns in the dataset.

```
1 loan.describe(include="all")
```

	ID	isDelinquent	term	gender	purpose	home_ownership	age	FICO
count	11548.000000	11548.0	11548	11548	11548	11548	11548	11548
unique	NaN	2.0	2	2	7	3	2	2
top	NaN	1.0	36 months	Male	House	Mortgage	20-25	300-500
freq	NaN	7721.0	10589	6555	6892	5461	5888	6370
mean	5774.500000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
std	3333.764789	NaN	NaN	NaN	NaN	NaN	NaN	NaN
min	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
25%	2887.750000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
50%	5774.500000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
75%	8661.250000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
max	11548.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 3.1.8: Summary of all features in dataset

Based on figure 3.1.8, we can summarize that

- The ID column has the total count that's exactly same as total number of observations of the dataset. Since it does not bring any meaningful insight for the process further down, we will drop this column in later section.
- There are total number of **7721 delinquent borrowers** in this dataset.
- Most of the loan terms in this dataset is having a **36 months loan term**.
- Most of the borrowers are **male**.
- Most of the purpose for loan application is meant for **house purchasing**.
- Most of the customer have **mortgaged** their house.
- Most of the borrowers are in the age group of **20-25 years old**.
- Most of the borrowers have the FICO score between **300 and 500**.

Lastly, we can check if there are any missing value in the dataset.

```
1  loan.isnull().sum()
```

```
ID                0
isDelinquent      0
term              0
gender            0
purpose           0
home_ownership    0
age               0
FICO              0
dtype: int64
```

Figure 3.1.9: Total number of null values in dataset

```
1  loan.isna().sum()
```

```
ID                0
isDelinquent      0
term              0
gender            0
purpose           0
home_ownership    0
age               0
FICO              0
dtype: int64
```

Figure 3.1.10: Total number of not applicable values in dataset

Based on figure 3.1.9 and figure 3.1.10, we can conclude that there are no missing or not applicable values exist in the dataset.

3.2 Data pre-processing

Since we already gained some useful information from the last section, we can process the dataset before the exploratory data analysis. The first execution is to drop the “ID” column as its values are unique and does not bring any meaningful insight for the later section of this project.

```
1  loan.drop(["ID"],axis=1,inplace=True)
```

Figure 3.2.0: Source code to drop “ID” column

We list out all the columns in dataset to ensure we successfully drop “ID” column.

```
1  loan.columns
```

```
Index(['isDelinquent', 'term', 'gender', 'purpose', 'home_ownership', 'age',
       'FICO'],
      dtype='object')
```

Figure 3.2.1: All columns in the dataset

On the other hand, we will impute the duplicates found in “purpose” column. Since the word “other” and “Oher” have the same meaning, we impute the word “other” to “Other”

```
1  loan["purpose"].replace('other',"Other",inplace=True)
```

Figure 3.2.2: Imputation for “purpose” column

Once we are done with the imputation, we recheck all the unique values for each column and conclude the dataset is done with pre-processing.

```
1  # avoid cell value truncated
2  pd.set_option('display.max_colwidth', None)
3  all_values = []
4  for col in loan.columns:
5      all_values.append(loan[col].unique())
6
7  unique_df = pd.DataFrame({'Column':loan.columns, 'Unique Values':all_values})
8  unique_df
```

	Column	Unique Values
0	isDelinquent	[1, 0] Categories (2, int64): [1, 0]
1	term	['36 months', '60 months'] Categories (2, object): ['36 months', '60 months']
2	gender	['Female', 'Male'] Categories (2, object): ['Female', 'Male']
3	purpose	['House', 'Car', 'Other', 'Personal', 'Wedding', 'Medical'] Categories (6, object): ['House', 'Car', 'Other', 'Personal', 'Wedding', 'Medical']
4	home_ownership	['Mortgage', 'Rent', 'Own'] Categories (3, object): ['Mortgage', 'Rent', 'Own']
5	age	['>25', '20-25'] Categories (2, object): ['>25', '20-25']
6	FICO	['300-500', '>500'] Categories (2, object): ['300-500', '>500']

Figure 3.2.3: Unique values for each column

3.3 Univariate exploratory data analysis

Univariate exploratory data analysis can be described as the analysis of one variable. We will try to understand the distribution of each variable by plotting histogram of their respective distribution.

```python
# Function to create barplots that indicate percentage for each category.
sns.set_context('notebook', font_scale = 1.2)
def perc_on_bar(plot, feature):
    '''
    plot
    feature: categorical feature
    the function won't work if a column is passed in hue parameter
    '''
    total = len(feature) # length of the column
    for p in ax.patches:
        percentage = '{:.1f}%'.format(100 * p.get_height()/total) # percentage of each class of the category
        x = p.get_x() + p.get_width() / 2 - 0.05 # width of the plot
        y = p.get_y() + p.get_height()           # hieght of the plot
        ax.annotate(percentage, (x, y + 100), size = 12) # annotate the percantage
    plt.title(f'Distribution of {ax.get_xlabel()}', fontweight = 'bold')
    plt.show() # show the plot
```

Figure 3.3.0: Source code for distribution plot

```python
plt.figure(figsize=(15,6))
ax = sns.countplot(loan["isDelinquent"],palette='spring', edgecolor = 'black')
plt.xticks([0, 1], ['No', 'Yes'])
perc_on_bar(ax,loan["isDelinquent"])
```

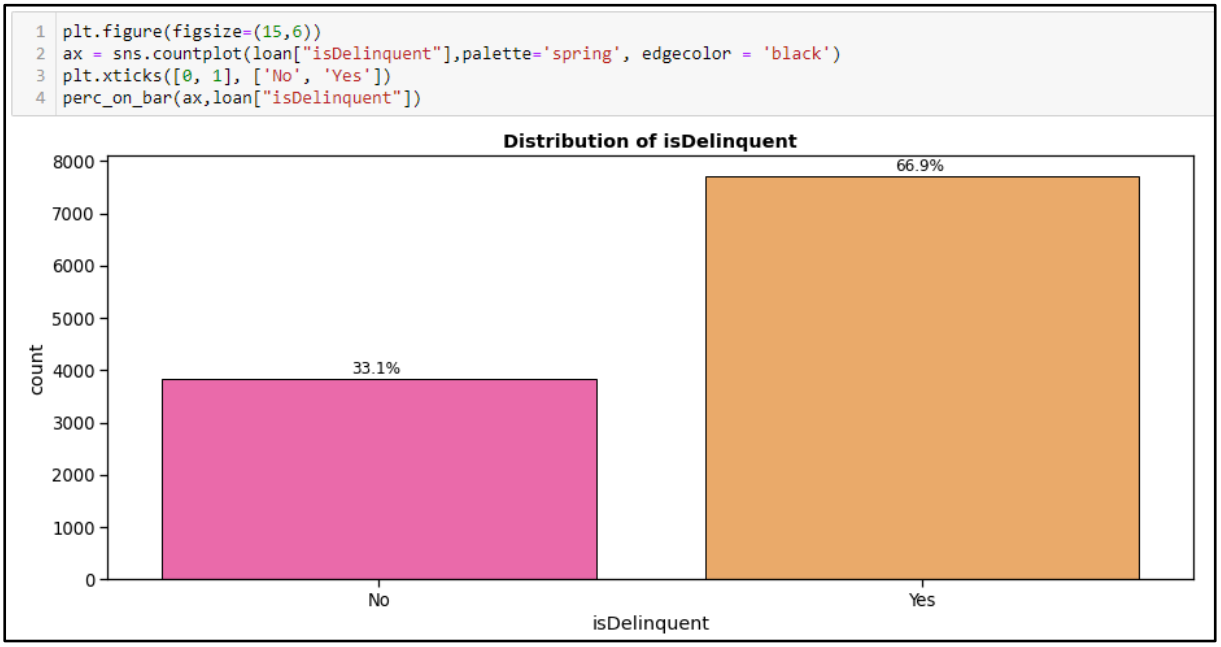

Figure 3.3.1: Distribution plot for “isDelinquent” column

Based on figure 3.3.1, we can observe that more than 50% of the borrowers are delinquent. In fact, there are 66.9% of the borrowers are delinquent and the remaining 33.1% of the borrowers are not delinquent.

```
plt.figure(figsize=(15,5))
ax = sns.countplot(loan["term"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["term"])
```

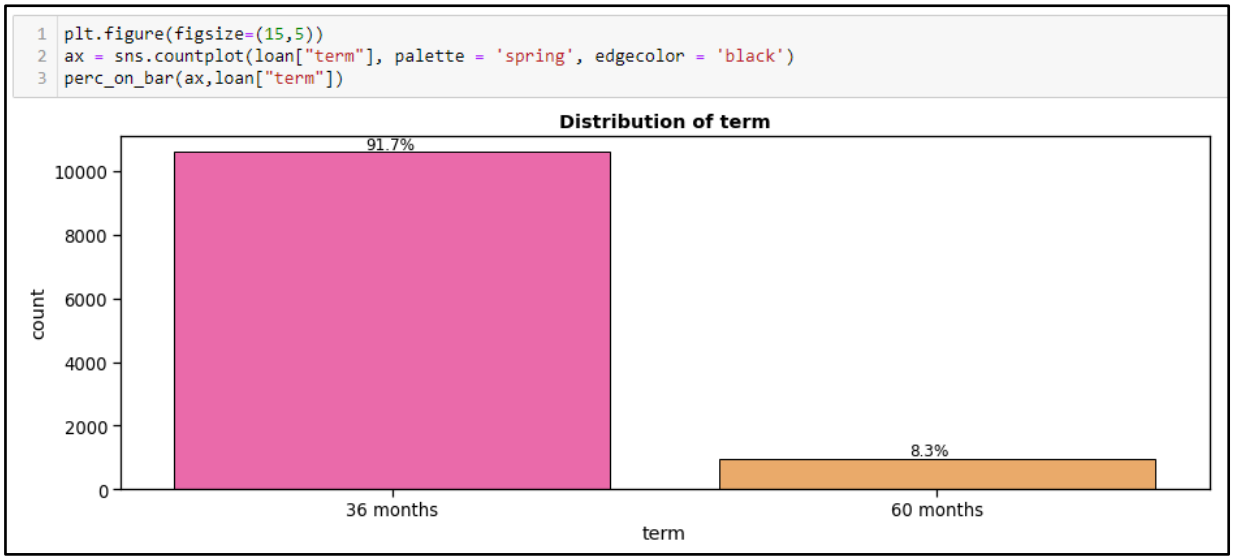

Figure 3.3.2: Distribution plot of "term" column

Based on figure 3.3.2, we can notice that 91.7% of the borrowers are having their loan term stretched to 36 months. Only 8.3% of the loan are in 60 months term.

```
plt.figure(figsize=(15,7))
ax = sns.countplot(loan["gender"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["gender"])
```

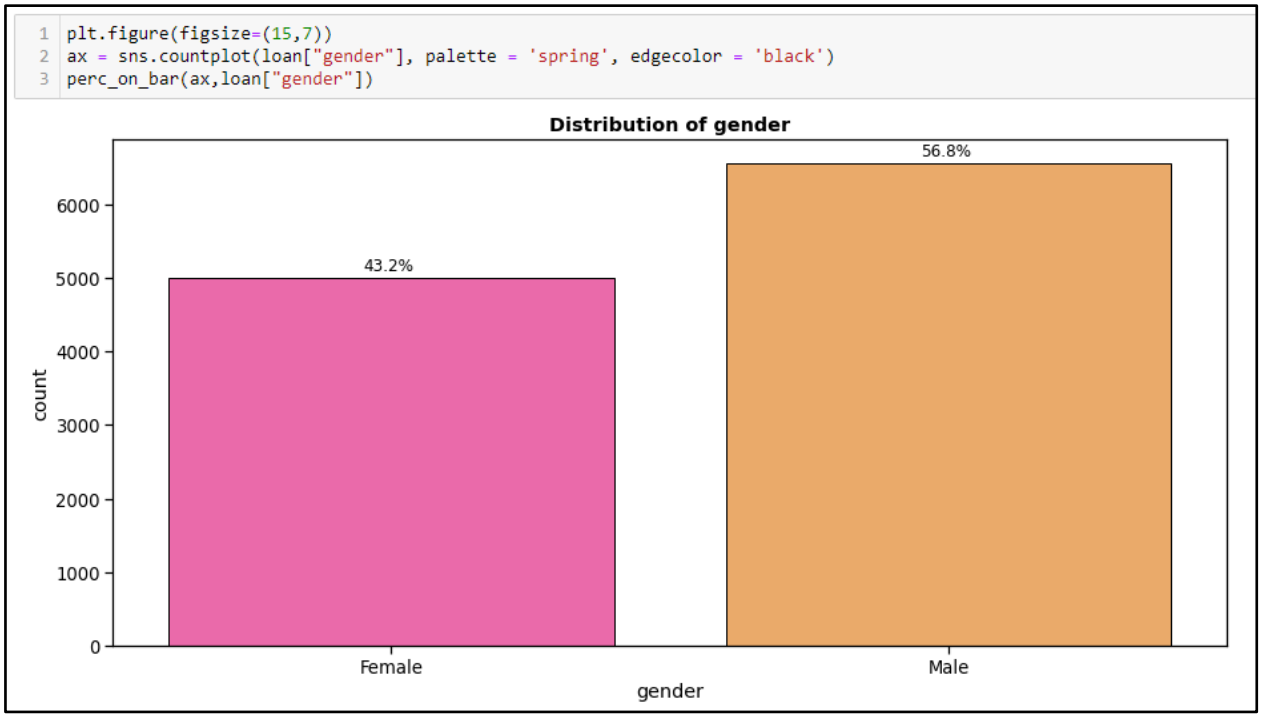

Figure 3.3.3: Distribution plot of "gender" column

Based on figure 3.3.3, we can observe 56.8% of the borrowers are male and 43.2% of the borrowers are female.

```
plt.figure(figsize=(15,7))
ax = sns.countplot(loan["purpose"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["purpose"])
```

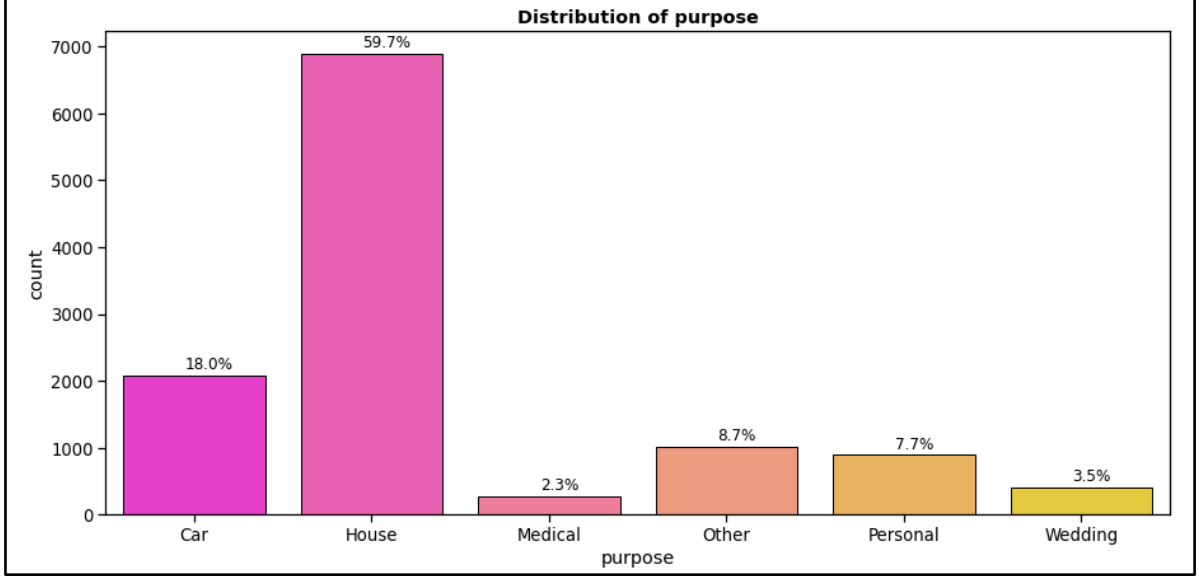

Figure 3.3.4: Distribution plot of "purpose" column

Based on figure 3.3.4, majority of the loan applications are meant for house purchasing, which contributed to 59.7%. Car loan comes in second place which is about 18.0% followed by other purposes which is 8.7%, then personal loan for 7.7%, loan for wedding for about 3.5% and lastly loan for medical purpose which is only 2.3%.

```
plt.figure(figsize=(15,6))
ax = sns.countplot(loan["home_ownership"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["home_ownership"])
```

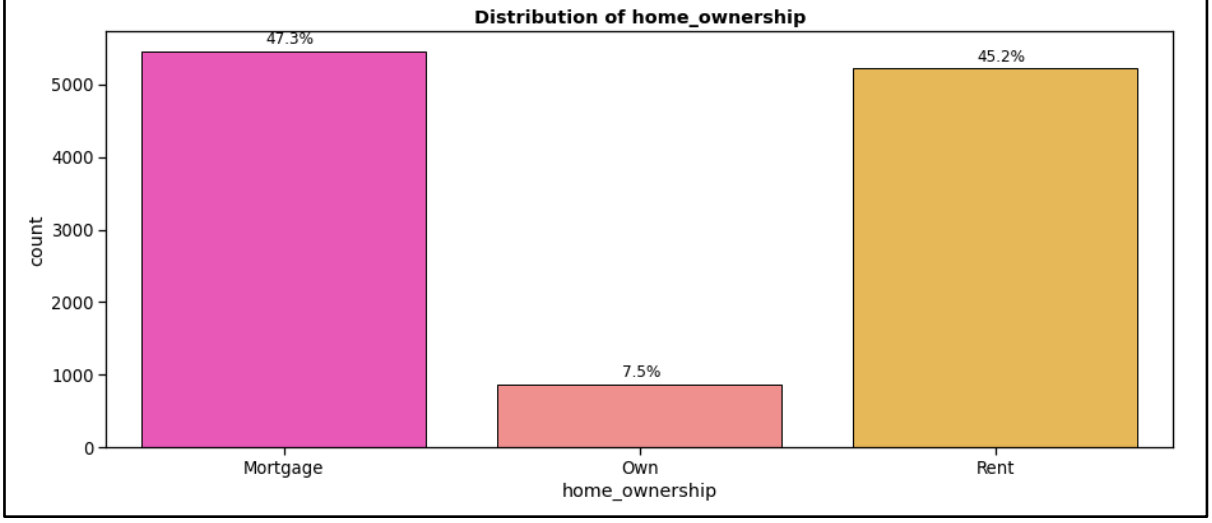

Figure 3.3.5: Distribution of "home_ownership" column

Based on figure 3.3.5, we can notice that 47.3% of the borrowers mortgaged their house, 45.2% of the borrowers are currently renting the house and only 7.5% of the borrowers who really owned the house.

```
plt.figure(figsize=(15,7))
ax = sns.countplot(loan["age"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["age"])
```

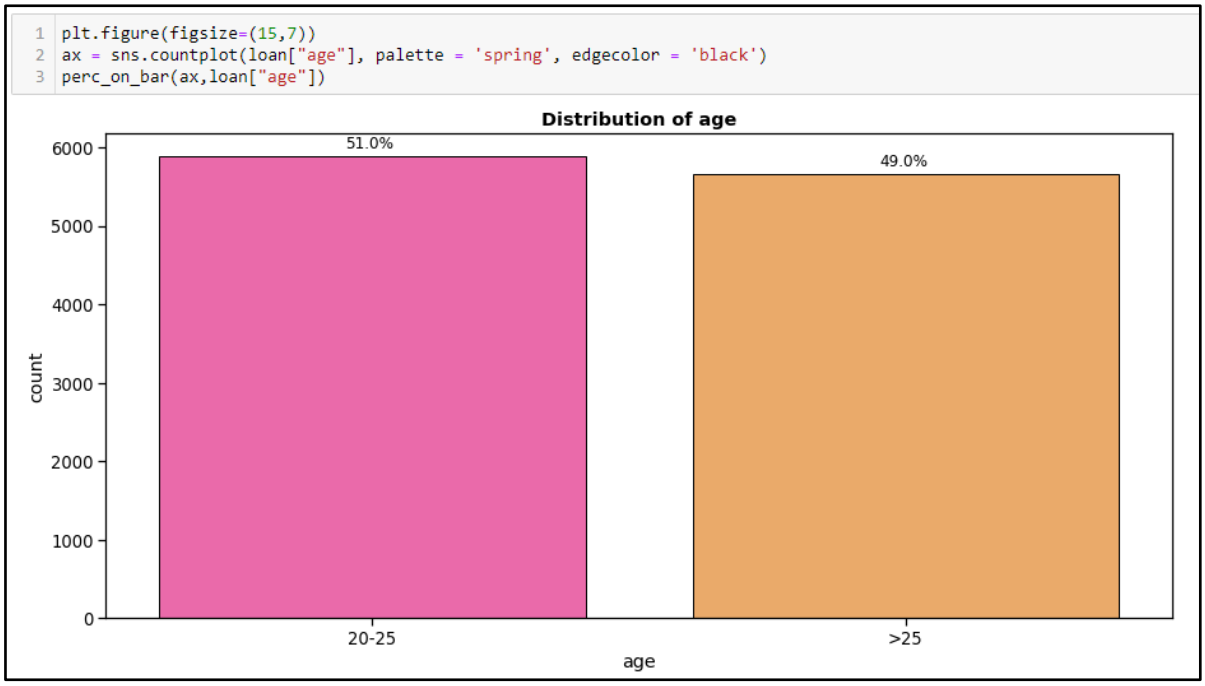

Figure 3.3.6: Distribution of "age" column

Based on figure 3.3.6, we can observe that 51% of the borrowers are in the age group of 20 – 25 years old, the remaining 49.0% are more than 25 years old.

```
plt.figure(figsize=(15,7))
ax = sns.countplot(loan["FICO"], palette = 'spring', edgecolor = 'black')
perc_on_bar(ax,loan["FICO"])
```

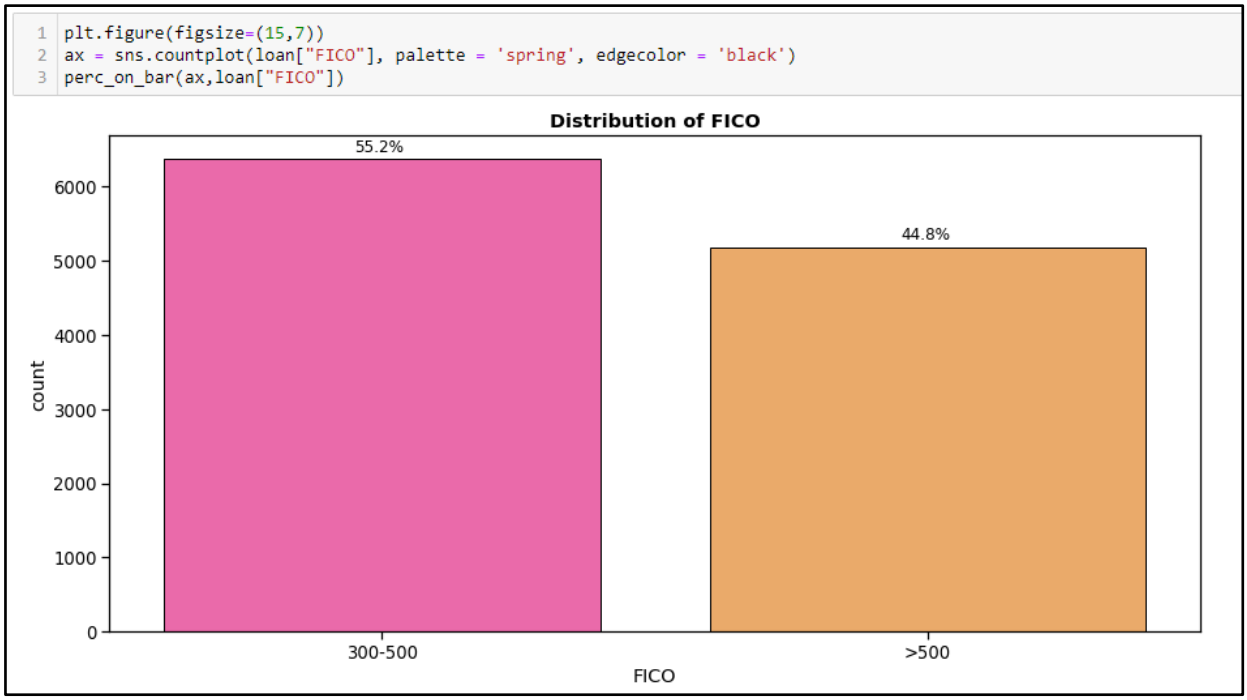

Figure 3.3.7: Distribution of "FICO" column

Based on figure 3.3.7, we can notice that more than 50% of the borrowers do not have FICO score more than 500. In fact, there are 55.2% of the borrowers are having the FICO score between 300 and 500. The remaining 44.8% of the borrowers are having the FICO score that is more than 500.

3.4 Bivariate exploratory data analysis

Bivariate analysis can be understood as exploring the relationship between 2 variables from the dataset. Since logistic regression assumes no multicollinearity among the predictors, can gain benefit from bivariate analysis to check if the dataset violates the assumption of multicollinearity.

Since all the variables in dataset now are categorical, we cannot use correlation matrix to explore the relationship between predictors. However, we can try to understand the relationship of between the dependent variable and each independent variable.

```python
def stacked_plot(x):
    # crosstab
    pal = ["lightblue", "orange"]
    ax= pd.crosstab(x, loan['isDelinquent']).apply(lambda r: r/r.sum()*100, axis=1)
    ax_1 = ax.plot.bar(figsize=(10,5), stacked=True, rot=0, color = pal, edgecolor = 'black')
    display(ax)

    plt.legend(['No', 'Yes'], loc='upper center', bbox_to_anchor=(1.0, 1.0), title="IsDelinquent")

    plt.ylabel('Percent Distribution')

    for rec in ax_1.patches:
        height = rec.get_height()
        ax_1.text(rec.get_x() + rec.get_width() / 2,
                  rec.get_y() + height / 2,
                  "{:.0f}%".format(height),
                  ha='center',
                  va='bottom')
    plt.title(f'Distribution of {ax_1.get_xlabel()} explained with isDelinquent', fontweight = 'bold')
    plt.show()
```

Figure 3.4.0: Source code of distribution plot explained with “isDelinquent”

```python
stacked_plot(loan['term'])
```

isDelinquent	0	1
term		
36 months	29.917839	70.082161
60 months	68.717414	31.282586

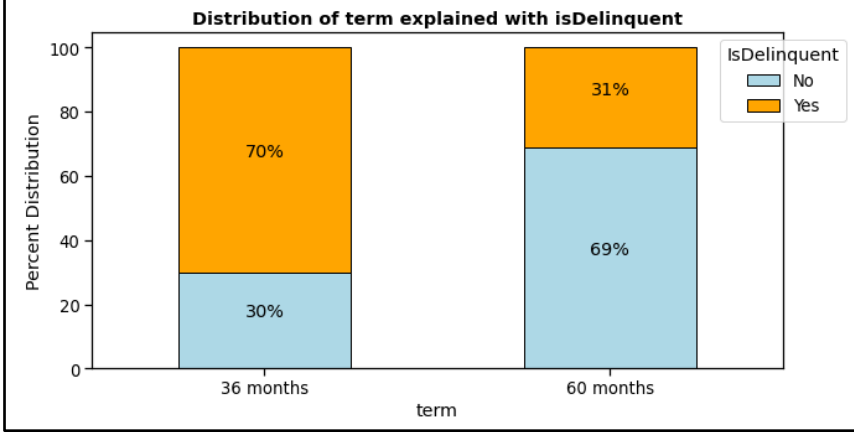

Figure 3.4.1: Distribution of “term” explained with “isDelinquent”

Based on figure 3.4.1, we can observe that for loan term with length of 36 months, 70% of the borrowers are delinquent and 30% of them are not. On the other hand, long term length of 60 months is showing a different trend. 31% of borrowers with loan term of 60 months are delinquent and the remaining 69% of them are not.

```
1 stacked_plot(loan['gender'])
```

isDelinquent	0	1
gender		
Female	37.051873	62.948127
Male	30.160183	69.839817

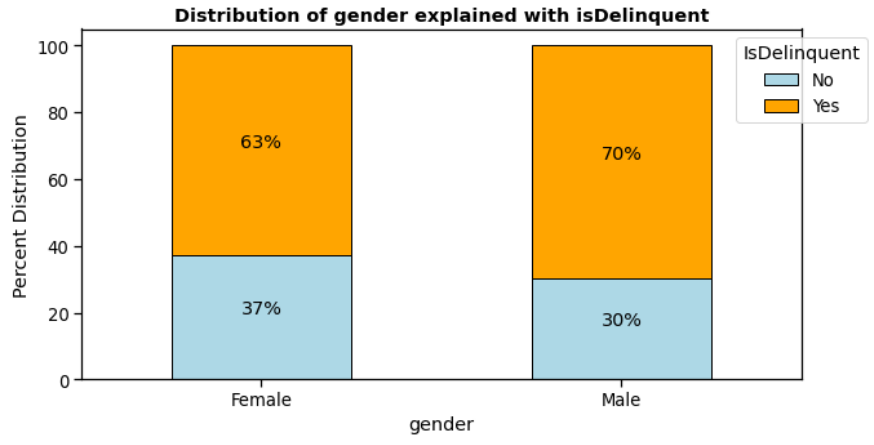

Figure 3.4.2: Distribution of "gender" explained with "isDelinquent"

Based on figure 3.4.2, we can observe that females are less likely to delinquent compared to male. 63% of female borrowers are delinquent while 70% of male borrowers are delinquent.

```
1 stacked_plot(loan['purpose'])
```

isDelinquent	0	1
purpose		
Car	32.596154	67.403846
House	32.965757	67.034243
Medical	40.225564	59.774436
Other	35.346535	64.653465
Personal	30.717489	69.282511
Wedding	34.068627	65.931373

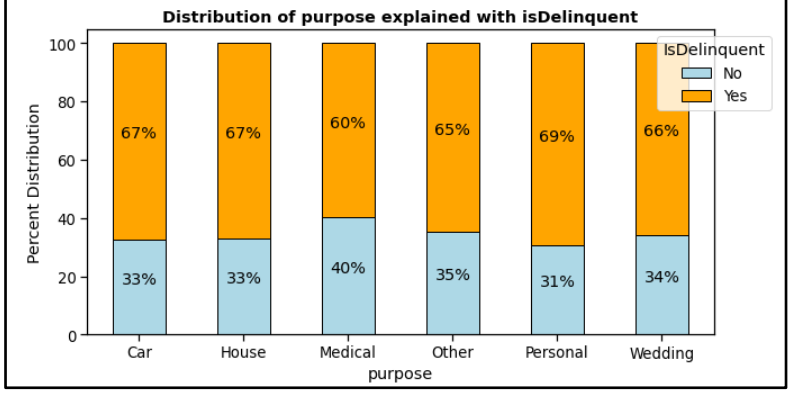

Figure 3.4.3: Distribution of "purpose" explained with "isDelinquent"

Based on figure 3.4.3, we can notice that most loan delinquent borrowers are those who spent on personal while the least loan delinquent borrowers are those who spent on medical purposes.

```
1 stacked_plot(loan['home_ownership'])
```

isDelinquent	0	1
home_ownership		
Mortgage	33.528658	66.471342
Own	29.735936	70.264064
Rent	33.301380	66.698620

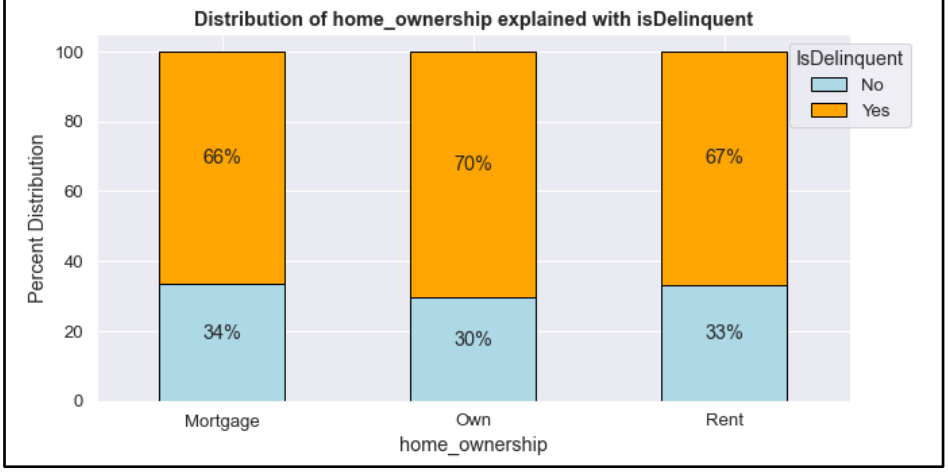

Figure 3.4.4: Distribution of "home_ownership" explained with "isDelinquent"

Based on figure 3.4.4, we can observe that borrowers who have their house mortgaged tends to be less delinquent, which is about 66% of them. On the contrary, borrowers who own their house tend to be delinquent, which is about 70% of them.

```
1 stacked_plot(loan['age'])
```

isDelinquent	0	1
age		
20-25	31.555707	68.444293
>25	34.787986	65.212014

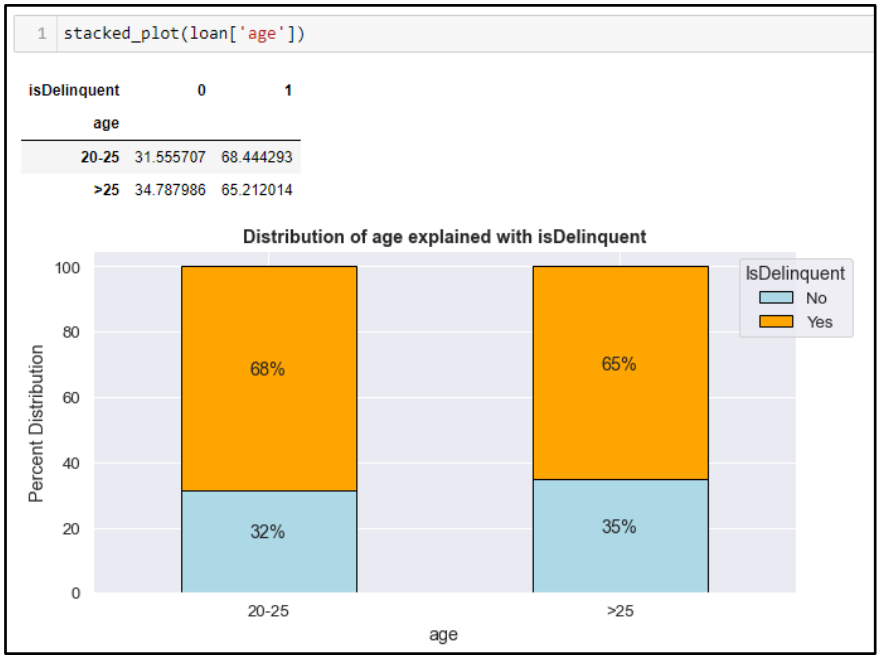

Figure 3.4.5: Distribution of "age" explained with "isDelinquent"

Based on figure 3.4.5, we can notice that age group of 20 – 25 years old are more delinquent, which is 68% compared to age group of borrowers greater than 25 years old, which is 65%.

```
stacked_plot(loan['FICO'])
```

isDelinquent	0	1
FICO		
300-500	14.772370	85.227630
>500	55.735805	44.264195

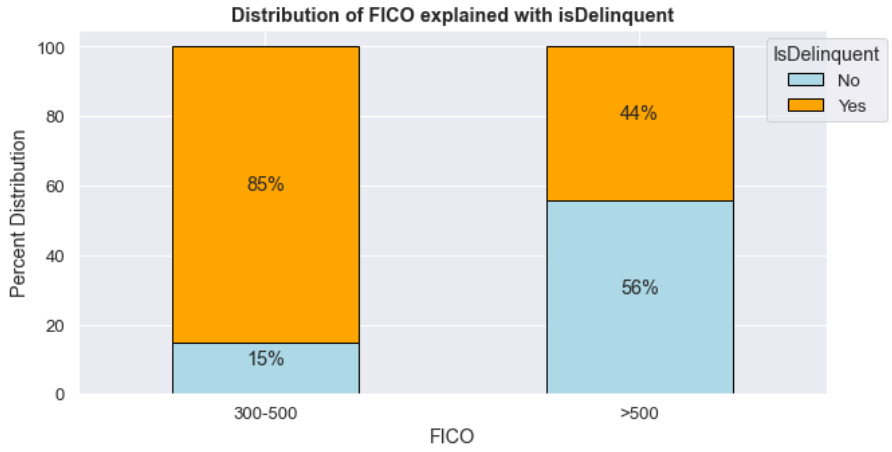

Figure 3.4.6: Distribution of "FICO" explained with "isDelinquent"

Based on figure 3.4.6, we can clearly observe that there's a 41% difference between FICO score group of 300 – 500 and greater than 500. The trend clearly shows that FICO score group of 300 – 500 is more likely to delinquent as there are 85% of delinquent borrowers while FICO score group of greater than 500 has only 44% delinquent borrowers.

Once we are done with exploring relationship of predictors with response, we can further explore the relationship between some of the predictors and FICO score group as we can observe in figure 3.4.5 that FICO score group is dictating whether a person will delinquent or not.

```
def stacked_plot_FICO(x):
    # crosstab
    pal = ["pink", "yellow"]
    ax= pd.crosstab(x, loan['FICO']).apply(lambda r: r/r.sum()*100, axis=1)
    ax_1 = ax.plot.bar(figsize=(10,5), stacked=True, rot=0, color = pal, edgecolor = 'black')
    display(ax)

    plt.legend(['300 - 500', '> 500'], loc='upper center', bbox_to_anchor=(1.0, 1.0), title="FICO score")

    plt.ylabel('Percent Distribution')

    for rec in ax_1.patches:
        height = rec.get_height()
        ax_1.text(rec.get_x() + rec.get_width() / 2,
                  rec.get_y() + height / 2,
                  "{:.0f}%".format(height),
                  ha='center',
                  va='bottom')
    plt.title(f'Distribution of {ax_1.get_xlabel()} explained with FICO score', fontweight = 'bold')
    plt.show()
```

Figure 3.4.7: Source code for distribution plot explained with FICO score group

```
1 stacked_plot_FICO(loan['home_ownership'])
```

FICO	300-500	>500
home_ownership		
Mortgage	52.316426	47.683574
Own	55.109070	44.890930
Rent	58.148006	41.851994

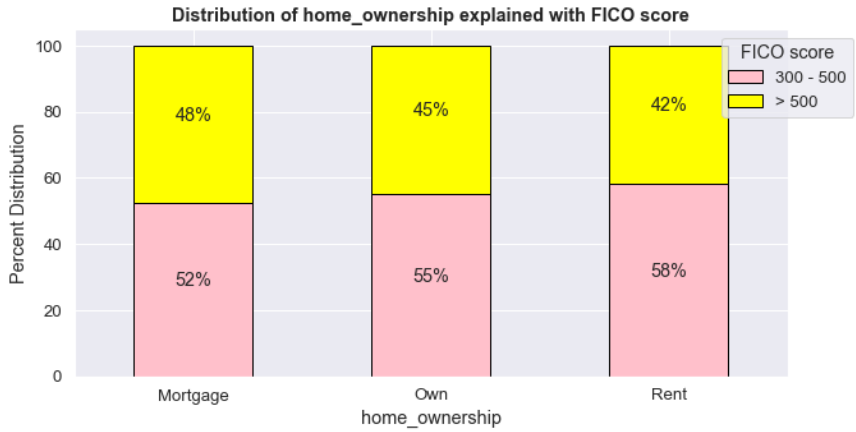

Figure 3.4.8: Distribution plot of "home_ownership" explained with FICO score group

Based on figure 3.4.8, we can observe that borrowers who owned the house has higher FICO score, which are 48% of them compared to borrowers who rent the house which only has 42% of them are having FICO score that is greater than 500.

```
1 stacked_plot_FICO(loan['age'])
```

FICO	300-500	>500
age		
20-25	66.694973	33.305027
>25	43.162544	56.837456

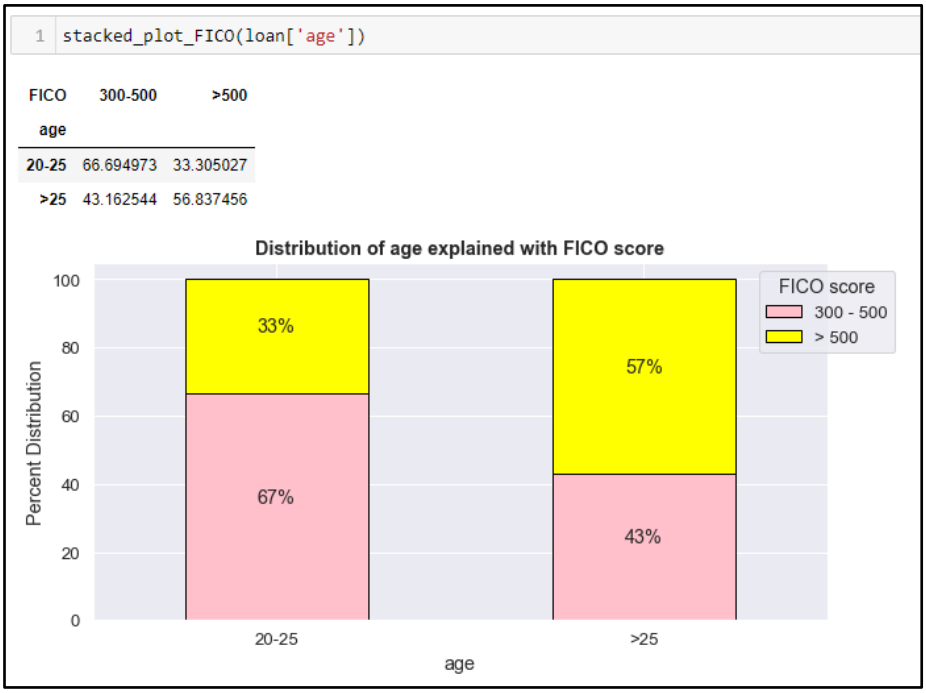

Figure 3.4.9: Distribution plot of "age" explained with FICO score group

Based on figure 3.4.9, we can notice that age group that greater than 25 years old have 57% of them with FICO score that is greater than 500 while age group between 20 and 25 has only 33% of them.

```
1 stacked_plot_FICO(loan['gender'])
```

FICO	300-500	>500
gender		
Female	53.374725	46.625275
Male	56.521739	43.478261

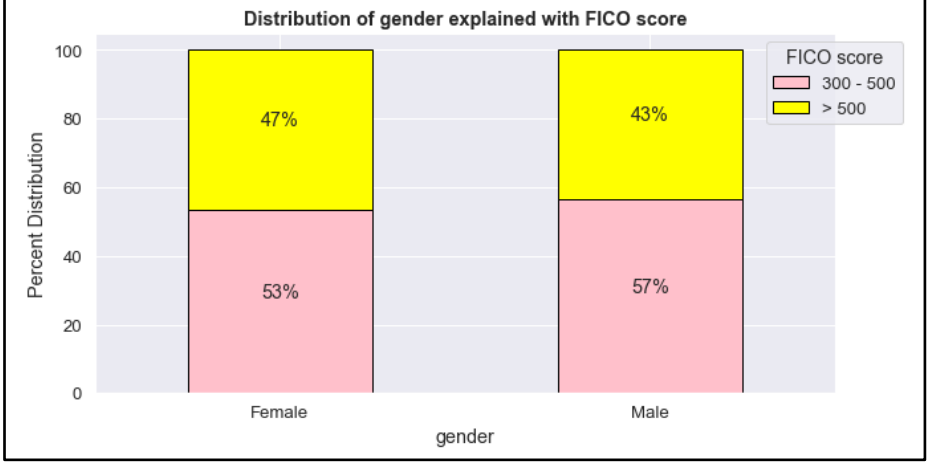

Figure 3.4.10: Distribution plot of "gender" explained with FICO score group

Based on figure 3.4.10, there are 47% of female have FICO score that greater than 500 while 43% of male have FICO score that greater than 500. With here marks the end of our exploratory data analysis, we will proceed to data splitting in the next section.

3.5 Data transformation

Having to know that all of our predictors are categorical value, we will transform all of them to have numerical representation in order to fit the dataset to a machine learning model. One of the most popular approaches to convert categorical data to numerical is by carry out one-hot encoding.

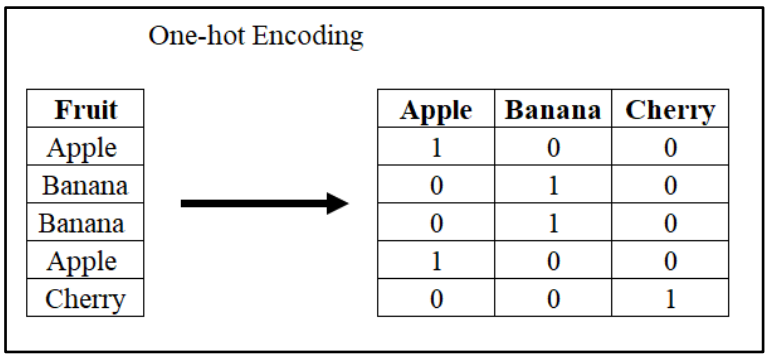

Figure 3.5.0: Conceptual diagram of one-hot encoding

Right before the one-hot encoding, we have to divide the dataset into two major group, one which stores solely predictors and their corresponding value and the other dataset stores only responses. By doing so we can further split them into training and testing dataset in later section.

```
x = loan.drop(["isDelinquent"], axis=1)
y = loan["isDelinquent"]
```

Figure 3.5.1: Splitting predictors and responses into two datasets

```
x.head()
```

	term	gender	purpose	home_ownership	age	FICO
0	36 months	Female	House	Mortgage	>25	300-500
1	36 months	Female	House	Rent	20-25	>500
2	36 months	Female	House	Rent	>25	300-500
3	36 months	Female	Car	Mortgage	>25	300-500
4	36 months	Female	House	Rent	>25	300-500

Figure 3.5.2: First 5 rows of observations of predictors dataset

```
y.head()
```

```
0    1
1    0
2    1
3    1
4    1
Name: isDelinquent, dtype: category
Categories (2, int64): [0, 1]
```

Figure 3.5.3: First 5 rows of observations of responses dataset

```
# encoding the categorical variables
x = pd.get_dummies(x, drop_first=True)
x.head()
```

	term_60 months	gender_Male	purpose_House	purpose_Medical	purpose_Other	purpose_Personal	purpose_Wedding	home_ownership_Own	home_ownership_Rent
0	0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	1
2	0	0	1	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	1

Figure 3.5.4: One-hot encoding apply to predictors dataset

Once we divided the dataset, we apply one-hot encoding to the predictors' dataset.

3.6 Data splitting

Supervised machine learning requires labelled datasets to train the algorithm in order to classify or predict accurately. The contemporary way to train a machine learning model is splitting the dataset into train-test, which become two sets of data.

In this section, the train dataset is used to fit logistic regression model and contrary the remaining dataset, test dataset is used to evaluate the fit of our logistic regression model. In short, the main objective of data splitting Is to estimate the performance of our machine learning model.

Noted that dataset will be split randomly into the ratio of 70:30, which means 70% of the dataset are fed into model training while the remaining 30% are meant for model testing.

```
1 X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 7, stratify = y)
```

Figure 3.6.0: Source code for data splitting

```
1 print(f'There are {np.round(len(X_train)/len(x.index) * 100, 2)}% data in training dataset.')
2 print(f'There are {np.round(len(X_test)/len(x.index) * 100, 2)}% data in testing dataset.')
```

```
There are 69.99% data in training dataset.
There are 30.01% data in testing dataset.
```

Figure 3.6.1: Distribution of dataset after split

3.7 Model fitting and evaluation with default regularization solver lbfgs

Once we are done with data splitting, we are ready to use our training dataset to train our logistic regression model. Noted that in this section, we will be using the default solver in model regularization, lbfgs to create our logistic model. Below shows the general expression for logistic model:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_\rho x_\rho$$

where ρ is the total number of predictors.

```
1 lr = LogisticRegression(random_state=1)
2 lr.fit(X_train,y_train)
```

```
LogisticRegression(random_state=1)
```

Figure 3.7.0: Source code for logistic model fitting

Right before we evaluate the model, we create some useful functions that can help us un getting model's metric scores and plotting the confusion matrix for further evaluation.

```python
##  Function to calculate different metric scores of the model - Accuracy, Recall and Precision
def get_metrics_score(model,train,test,train_y,test_y,flag=True):
    '''
    model : classifier to predict values of X

    '''
    # defining an empty list to store train and test results
    score_list=[]

    pred_train = model.predict(train)
    pred_test = model.predict(test)

    train_acc = model.score(train,train_y)
    test_acc = model.score(test,test_y)

    train_recall = metrics.recall_score(train_y,pred_train)
    test_recall = metrics.recall_score(test_y,pred_test)

    train_precision = metrics.precision_score(train_y,pred_train)
    test_precision = metrics.precision_score(test_y,pred_test)

    score_list.extend((train_acc,test_acc,train_recall,test_recall,train_precision,test_precision))

    # If the flag is set to True then only the following print statements will be dispayed.
    #The default value is set to True.
    if flag == True:
        print("Accuracy on training set : ",model.score(train,train_y))
        print("Accuracy on test set : ",model.score(test,test_y))
        print("Recall on training set : ",metrics.recall_score(train_y,pred_train))
        print("Recall on test set : ",metrics.recall_score(test_y,pred_test))
        print("Precision on training set : ",metrics.precision_score(train_y,pred_train))
        print("Precision on test set : ",metrics.precision_score(test_y,pred_test))

    return score_list # returning the list with train and test scores
```

Figure 3.7.1: Source code to compute model's matric scores

```python
## Function to create confusion matrix
def make_confusion_matrix(model,y_actual,labels=[1, 0]):
    '''
    model : classifier to predict values of X
    y_actual : ground truth

    '''
    y_predict = model.predict(X_test)
    cm=metrics.confusion_matrix( y_actual, y_predict, labels=[0, 1])
    df_cm = pd.DataFrame(cm, index = [i for i in ["Actual - No","Actual - Yes"]],
                         columns = [i for i in ['Predicted - No','Predicted - Yes']])
    group_counts = ["{0:0.0f}".format(value) for value in
                    cm.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in
                         cm.flatten()/np.sum(cm)]
    labels = [f"{v1}\n{v2}" for v1, v2 in
              zip(group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    plt.figure(figsize = (10,7))
    sns.heatmap(df_cm, annot=labels,fmt='')
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title('Confusion matrix\n', fontweight = 'bold')
```

Figure 3.7.2: Source code to plot confusion matrix

With the source code defined, we first apply cross validation to evaluate our trained logistic regression model. Cross validation can be described as a statistical method used to estimate the performance of machine learning models. It is a resampling procedure used to evaluate machine learning models on a limited data sample, the procedure has a single parameter called *k* that refers to the number of groups that a given data sample is to be split into. At such, the procedure is often called k-fold cross-validation. When a specific value of *k* is chosen, it may be used in place of k in the reference to the model, such as *k = 10* becoming 10-fold cross-validation.

K-folds cross validator provides dataset indices to split one dataset into train and validation sets. Once the dataset is split in *k* consecutive stratified folds, each fold is then used once as a validation process to compute for the model metrices while the *k-1* remaining folds form the training set.

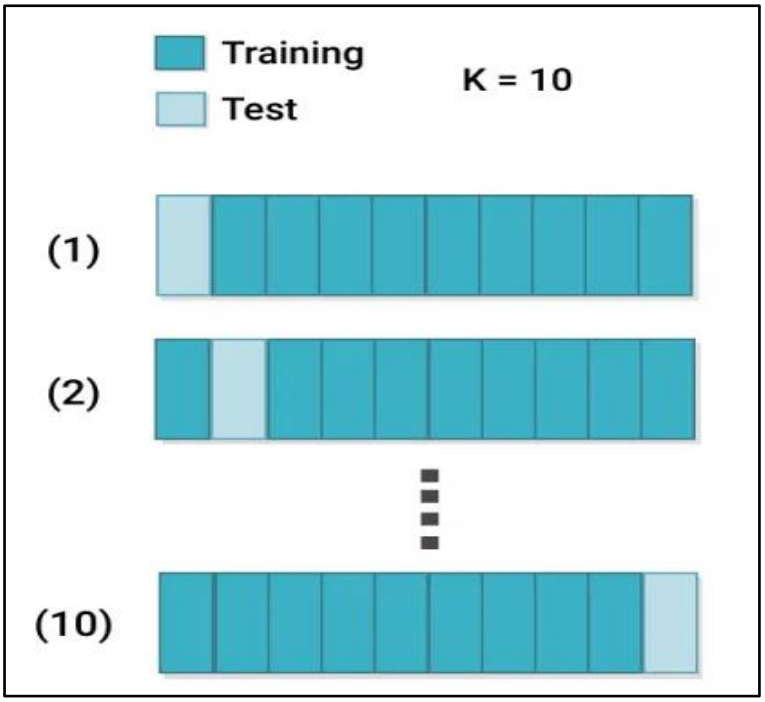

Figure 3.7.3: Conceptual diagram of cross validation

Having to know the evaluation method, it is time for us to chose the suitable metric to evaluate our logistic regression model. We first have to understand that the objective of bank is to maximize profit margin and minimize the potential loss. Currently the bank is facing 2 issues:

- Whenever a bank lends money to customer, they don't return the money.
- If a bank is being too preservative it might losses potential good customer.

It is clear that we can solve this issue by increase the predictability of the model towards the potential customer who will not delinquent, and decrease the rate of false predictions towards the potential customer who might be delinquent.

We must understand that it is impossible to have a 100% correct prediction and a bank would rather risk the opportunity to misclassify a good customer as delinquent than to misclassify a delinquent customer as a good customer. Having that in mind, we can understand the metrices for a classification machine learning model as table below and choose the right one for our model evaluation.

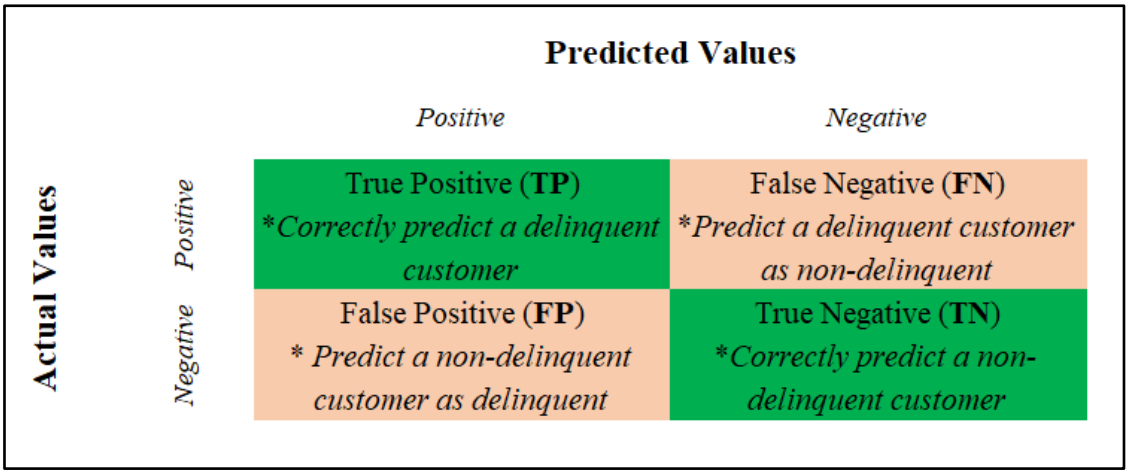

Figure 3.7.4: Confusion matrix in current project context

Measure	General description	Description in current context	Formula
Accuracy	The ratio of correct predictions to total predictions made.	The ratio of correct predictions for delinquent and non-delinquents to total predictions made.	$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$
Recall / Sensitivity	Indicates how many actual positive values the model was able to predict correctly out of total true positive cases.	The ratio of how many correct predictions for delinquent customer to total actual delinquent customer.	$Recall\ (Sensitivity) = \frac{TP}{TP + FN}$
Specificity	Indicates how many actual negative values the model was able to predict correctly out of total predicted negative cases.	The ratio of how many correct predictions for non-delinquent customer to total actual non-delinquent customer.	$Specificity = \frac{TN}{TN + FP}$
Precision	Indicates how many correctly predicted cases actually turn out to be positive out of total predicted positive cases.	The ratio of correct predictions for delinquent customer to total number of true delinquent customers.	$Precision = \frac{TP}{TP + FP}$

Understand that it is better for a bank to misclassify a potential non-delinquent customer as delinquent than misclassify a potential delinquent customer as non-delinquent, we can use recall to evaluate our logistic model. Given that recall is the ratio of how many correct predictions for delinquent customer to total actual delinquent customer, we will wish to ensure the model with the higher the recall score the better.

```
sns.set_style('whitegrid')
```

```
scoring='recall'
kfold=StratifiedKFold(n_splits=5,shuffle=True,random_state=1)     #Setting number of splits equal to 5
cv_result_bfr=cross_val_score(estimator=lr, X=X_train, y=y_train, scoring=scoring, cv=kfold)
#Plotting boxplots for CV scores of model defined above
plt.boxplot(cv_result_bfr)
plt.gca().axes.get_xaxis().set_visible(False)
plt.ylabel('Model score')
plt.title('Box plot of model score after 5 folds corss validation\n', fontweight = 'bold')
plt.show()
```

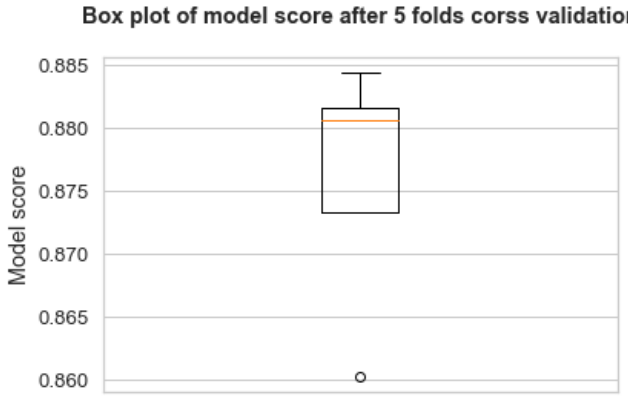

Figure 3.7.5: Cross validation with k = 5 and corresponding model score

Based on figure 3.7.5, we can notice the model has the recall score between 0.874 to 0.881, which is around 87.4% to 88.1%. We can presume that our model well performed in our training dataset. We can now evaluate the performance of our model in testing dataset.

```
#Calculating different metrics
scores_LR = get_metrics_score(lr,X_train,X_test,y_train,y_test)

# creating confusion matrix
make_confusion_matrix(lr,y_test)
```

```
Accuracy on training set :  0.7834962266485216
Accuracy on test set :  0.7948051948051948
Recall on training set :  0.877498149518875
Recall on test set :  0.8813120414328873
Precision on training set :  0.8133790737564323
Precision on test set :  0.8240516545601292
```

Figure 3.7.6: Model performance in testing dataset

Based on figure 3.7.6, we can notice that the recall score of our logistic regression model is indeed within the score obtained from cross-validation. Our model has the recall score of 88.1% in testing dataset. We can further visualize the confusion matrix of our model's performance with testing dataset as figure below.

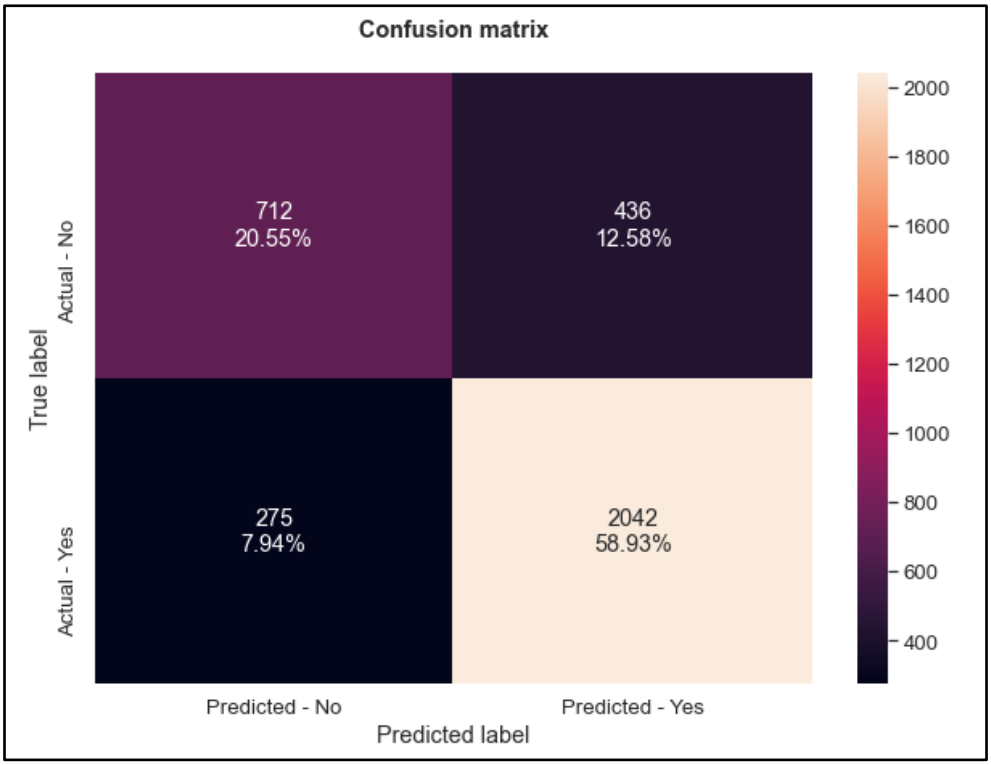

Figure 3.7.7: Confusion matrix of model with default solve in testing dataset

3.8 Model fitting and evaluation using SMOTE for oversampling with default regularization solver lbfgs

In this section, we will try to evaluate the performance of our model under the circumstances of oversampling the training dataset using SMOTE technique. SMOTE can be understand as **Synthetic Minority Oversampling Technique**, it solves a problem with imbalanced classification which there are too few examples of the minority class for a model to effectively learn the decision boundary.

The way SMOTE solve this problem is by oversampling the minority class. Is selects examples/data value that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point long that line and thus achieved oversampling.

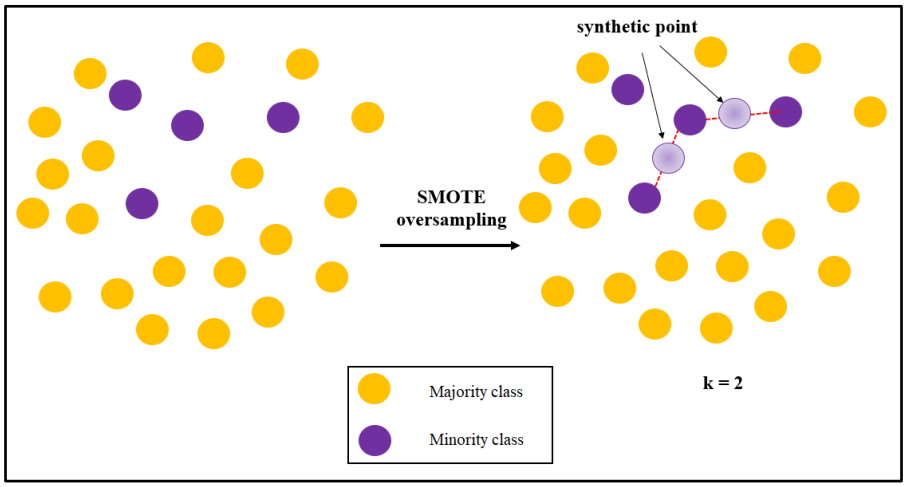

Figure 3.8.0: Conceptual diagram for SMOTE

With that understanding, we will increase the sample size of the minority class in our dataset. Based on figure 3.8.1, we can notice that there are 5404 delinquent customers and 2679 non-delinquent customers, which is about 50.4% difference.

```
print("Before UpSampling, counts of label 'Yes' for delinquent customer: {}".format(sum(y_train==1)))
print("Before UpSampling, counts of label 'No' for delinquent customer: {} \n".format(sum(y_train==0)))
```

```
Before UpSampling, counts of label 'Yes' for delinquent customer: 5404
Before UpSampling, counts of label 'No' for delinquent customer: 2679
```

Figure 3.8.1: Distribution of delinquent customer in dataset before apply SMOTE

```
sm = SMOTE(sampling_strategy = 1 ,k_neighbors = 5, random_state=1)
X_train_over, y_train_over = sm.fit_resample(X_train, y_train)
```

Figure 3.8.2: Source code for SMOTE with k = 5

After the implementation of SMOTE, we can notice that total number of non-delinquent customers in dataset is now having the same value as the delinquent customer.

```
print("After UpSampling, counts of label 'Yes' for delinquent customer: {}".format(sum(y_train_over==1)))
print("After UpSampling, counts of label 'No' for delinquent customer: {} \n".format(sum(y_train_over==0)))


print('After UpSampling, the shape of train_X: {}'.format(X_train_over.shape))
print('After UpSampling, the shape of train_y: {} \n'.format(y_train_over.shape))
```

```
After UpSampling, counts of label 'Yes' for delinquent customer: 5404
After UpSampling, counts of label 'No' for delinquent customer: 5404

After UpSampling, the shape of train_X: (10808, 11)
After UpSampling, the shape of train_y: (10808,)
```

Figure 3.8.3: Distribution of delinquent customer in dataset after apply SMOTE

Once we are done with oversampling using SMOTE, we can now use the processed dataset to train our logistic regression model.

```
log_reg_over = LogisticRegression(random_state = 1)

# Training the basic logistic regression model with training set
log_reg_over.fit(X_train_over,y_train_over)
```

```
LogisticRegression(random_state=1)
```

Figure 3.8.4: Model training using oversampled dataset

With the model trained, again we use cross-validation technique to evaluate the performance of our model in training dataset and compute for its recall score.

```
scoring='recall'
kfold=StratifiedKFold(n_splits=5,shuffle=True,random_state=1)     #Setting number of splits equal to 5
cv_result_over=cross_val_score(estimator=log_reg_over, X=X_train_over, y=y_train_over, scoring=scoring, cv=kfold)
#Plotting boxplots for CV scores of model defined above
plt.boxplot(cv_result_over)
plt.gca().axes.get_xaxis().set_visible(False)
plt.ylabel('Model score')
plt.title('Box plot of model score after 5 folds corss validation\n', fontweight = 'bold')
plt.show()
```

Figure 3.8.5: Cross validation with k = 5 and corresponding model score with oversampled dataset

Based on figure 3.8.5, we can observe that the recall score of our model using oversampled dataset is in the range of 0.69 to 0.72, which is around 69% to 72%. The performance of our model using oversampled dataset is decreased compared to the previous model that uses original dataset.

In order to further confirm this observation, we use testing dataset to further evaluate the performance of our model.

```
#Calculating different metrics
get_metrics_score(log_reg_over,X_train_over,X_test,y_train_over,y_test)

# creating confusion matrix
make_confusion_matrix(log_reg_over,y_test)
```

```
Accuracy on training set :  0.7721132494448557
Accuracy on test set :  0.7523809523809524
Recall on training set :  0.7004071058475203
Recall on test set :  0.7095381959430298
Precision on training set :  0.8176712032836466
Precision on test set :  0.8988518316019682
```

Figure 3.8.6: Model performance using SMOTE dataset

Based on figure 3.8.6, we can notice the recall score of our model has decreased compared to previous model that is not using SMOTE dataset. Our recall score for this model is **70.95%** compared to previous score of **88.13%**. Image below shows the confusion matrix of our logistic regression model trained with SMOTE dataset.

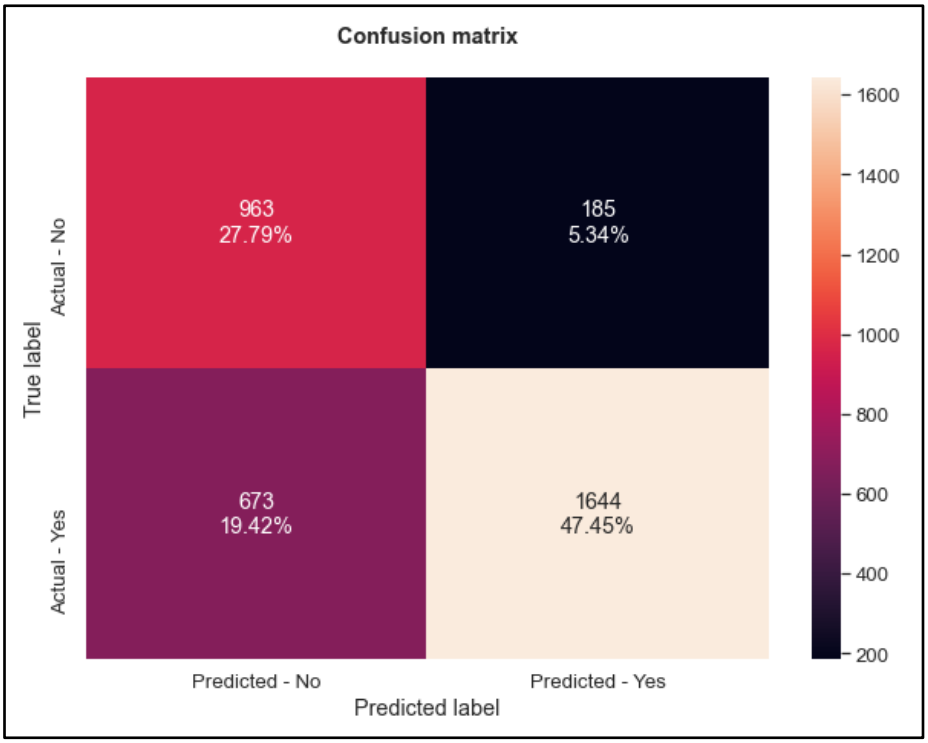

Figure 3.8.7: Confusion matrix of model trained with SMOTE dataset and using default solver

3.9 Model fitting and evaluation using SMOTE for oversampling with regularization solver saga

In this section, we will continue oversample dataset using SMOTE to train the dataset, but regularize the loss function with different solver, saga. Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting and underfitting. By regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it.

During logistic regression modelling process, we can choose different types of solvers to regularize our mode. Noted that the idea of solver is essentially using some of the regularization methods described as below:

Regularization approach	Formula	Short description
L1 regularization	$\min_{w,c} \|w\|_1 + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Shrink less important features or removing them
L2 regularization	$\min_{w,c} \frac{1}{2} w^T w + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Remove the percentage of weight of features, do not remove them
Elastic-Net regularization	$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho\|w\|_1 + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Combination of l1 and l2 by using ρ to control the strength of l1 against l2

By understanding the fundamentals of regularization approaches, it can help us for appropriate solver selection during model fitting as solver is just the combination of different regularization techniques. Table below describes some of the features of solvers in logistic regression.

Solver	Description
lbfgs	Stands for limited-memory BFGS. This solver only calculates an approximation to the Hessian based on the gradient which makes it computationally more effective. On the other hand it's memory usage is limited compared to regular bfgs which causes it to discard earlier gradients and accumulate only fresh gradients as allowed by the memory restriction.
liblinear	More efficient solver with small datasets. Only useful for one-versus-rest problems won't work with multiclass problems unlike other solvers here. Also doesn't work with l2 or none parameter values for penalty.
newton-cg	Solver which calculates Hessian explicitly which can be computationally expensive in high dimensions.
sag	Stands for Stochastic Average Gradient Descent. More efficient solver with large datasets.
saga	Saga is a variant of Sag and it can be used with l1 Regularization. It's a quite time-efficient solver and usually the go-to solver with very large dataset.

Having to understand each type of solver, we will use "saga" due to its time-efficient and works well with large dataset characteristics.

Once we are done with the solver selection, we can now select the value for the inverse of regularization strength, the C parameter. As the name of C parameter indicates, it is the inverse of regularization strength, and in order to minimize or constraints the size of the model coefficients to prevent overfit or underfit, we use small C value.

In this project, we will use three different values of C parameter, which is from 0.1 all the way to 1.0, and let the model to choose the best C parameter.

```
# Choose the type of classifier.
lr_estimator = LogisticRegression(random_state=1,solver='saga')
```

```
# Grid of parameters to choose from
parameters = {'C': np.arange(0.1,1.1,0.1)}
```

```
grid_obj = GridSearchCV(lr_estimator, parameters, scoring='recall')
grid_obj = grid_obj.fit(X_train_over, y_train_over)
```

```
# Set the clf to the best combination of parameters
lr_estimator = grid_obj.best_estimator_
```

```
# Fit the best algorithm to the data.
lr_estimator.fit(X_train_over, y_train_over)
```

```
LogisticRegression(C=0.1, random_state=1, solver='saga')
```

Figure 3.9.0: Source code for model fitting using SMOTE training dataset with sage as regularization solver

Based on figure 3.9.0, we can notice that C parameter value of 0.1 allows the model to generalize well on our SMOTE training dataset.

Once the model training is done, we can finally evaluate the performance of it.

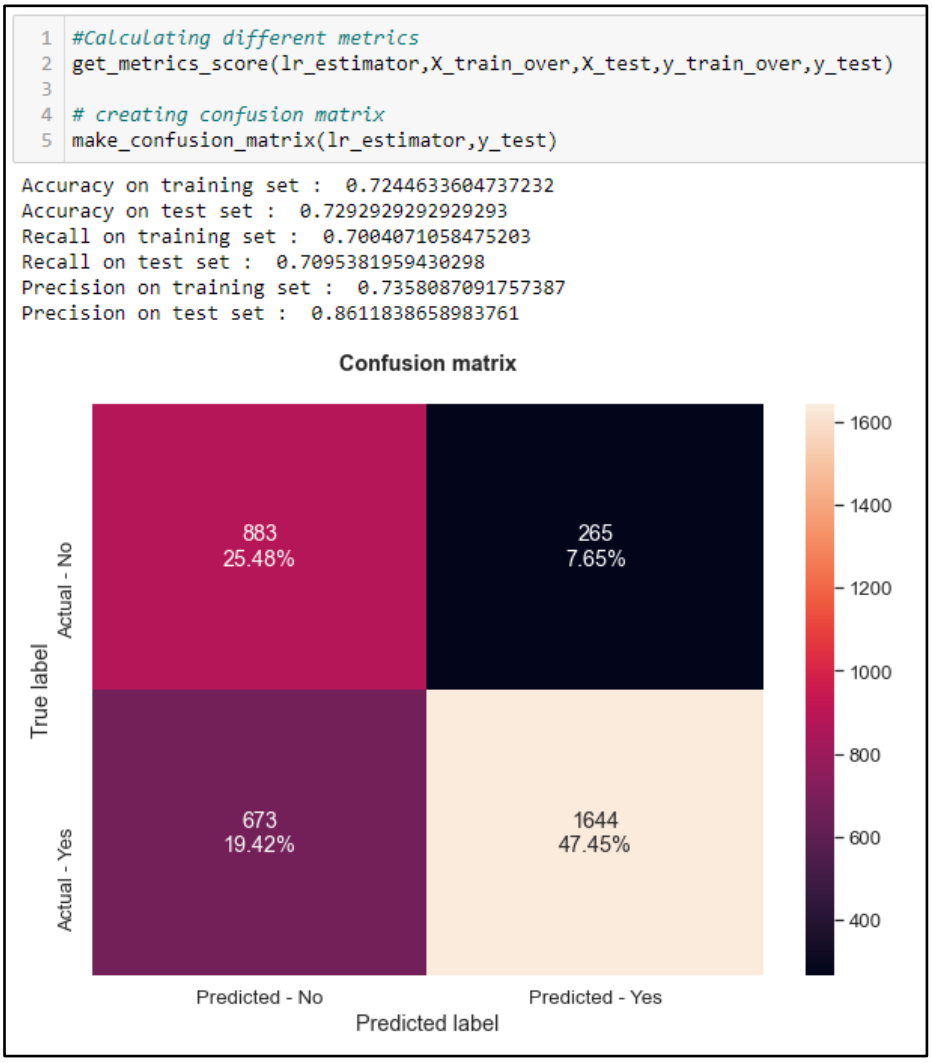

Figure 3.9.1: Performance of model trained with SMOTE dataset with sage as regularization solver on test dataset

Based on figure 3.9.1, we can observe that the recall score of our model on testing dataset is **70.95%** which still is not as good as the model trained without SMOTE dataset and using the default regularization solver, lgfbs. In summary, table below showcases all the models created in this project and their corresponding performance.

Is train dataset applied SMOTE	Type of regularization solver	Recall score on test dataset
No	lbfgs	88.13%
Yes	lbfgs	70.95%
Yes	saga	70.95%

In short, we can conclude that among the three logistic regression model created, model that trained with training dataset that hasn't oversampled by SMOTE with default regularization solver, lbfgs got the highest recall score.

4. Conclusion

As a conclusion, we can presume that FICO score, loan term and gender are important variables in determining if a borrower will get into a delinquent stage. No borrower shall be given a loan if they are applying for a 36-month loan term and have a FICO score in the range 300-500.

On the other hand, female borrower with a FICO score greater than 500 should be our target customers. Besides, in order to approve loan application, bank should focus on these criteria:

- FICO score
- Loan term
- Gender

The ideal customer will be a female customer with FICO score greater than 500.