UNIVERSITI TUNKU ABDUL RAHMAN

Lee Kong Chian Faculty of Engineering and Science

Academic Year 2023 /2024

UECS3483 Data Mining

Group Assignment

No.	Name	Id	Course	Practical Group
1.	Lee Boon Hao	2106860	SE	P2
2.	Christopher Chu Chen Fong	2300100	SE	P2
3.	Kou Zi Hong	1702952	ET	P1

Table of Contents

1.0 Introduction

This assignment studies a loan application dataset and make a classification model based on different model (K-Nearest Neighbors, Logistic Regression, Support Vector Machine, Decision Trees, Random Forest Classifier, Naïve Bayes, Neural Network). Before making a robust classification model, we implement the exploratory data analysis (EDA) on our dataset, clean and preprocessing the data in our dataset. Later, we train model based on different classification method and evaluate the model performance for each different classification model. Next, we choose the model that has the best performance in terms of accuracy, precision, recall, f1-score, as well as AUC score. Then, the best model will be used to predict the loan application on different data. Our assignment will be done by using the Python programming language in Google Colab.

The following link shows our assignment in Google Colab: https://colab.research.google.com/drive/1FH_9fjiyHxbOVCDXF5vwB11lSjtc53ds?authuser=1#scrollTo=S7y6nI4TjxAl

2.0 Preparation

During the preparation phase, there are three steps that need to be done, which include data exploration, data splitting, and data preprocessing. Some of the tasks that need to be done in this stage include exploratory data analysis on the dataset, splitting the data into three sets (training, testing, and validation set), identifying and treating missing value, identifying and removing duplicate rows, identifying and removing outliers, and identifying imbalanced classes.

2.1 Data Exploration

To kickstart our project, we first import the necessary libraries that we need to use for our project.

```
# import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from scipy import stats
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from scipy.stats import uniform
from imblearn.over_sampling import SMOTE
from sklearn.neural_network import MLPClassifier
```

The libraries that we have used in this assignment include:

1. pandas: pandas is a powerful data manipulation library in Python. It provides data structures and functions necessary to manipulate and analyze structured data. It is widely used for data cleaning, and analysis.
2. matplotlib.pyplot: Matplotlib is a plotting library in Python that produces publication-quality figures in a variety of formats. ‘pyplot’ is a module in Matplotlib that provides a

MATLAB-like interface for plotting. It is commonly used for creating visualizations such as line plots, scatter plots, histograms, and other diagrams.

3. numpy: NumPy is a fundamental package for scientific computing with Python. It provides support for large, multi-dimensional arrays and matrices. Besides that, it also provides a collection of mathematical functions to operate on these arrays efficiently.
4. seaborn: Seaborn is a statistical data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn simplifies the process of creating complex visualizations and supports various types of plots like violin plots, pair plots, and others.
5. scipy: SciPy is a library used for scientific and technical computing. The 'stats' module within SciPy provides a wide range of statistical functions and distributions for probability calculations, hypothesis testing, and statistical analysis.
6. sklearn: Scikit-learn is a popular machine learning library in Python. It provides simple and efficient tools for data mining and data analysis. Scikit-learn includes various algorithms for classification, regression, clustering, dimensionality reduction, and model selection.
7. imblearn.over_sampling: Imbalanced-learn (imblearn) is a library specifically designed to address imbalanced datasets in machine learning. SMOTE (Synthetic Minority Over-sampling Technique) is a technique used to solve the class imbalance problem by oversampling the minority class.

```
# read the data from the excel sheet
data = pd.read_csv('((GAssign) BankLoanApproval.csv')
```

Next, we read the data from a CSV file, which is our dataset. We have used the read_csv() function from the pandas library to read data from a CSV file. This function reads the contents of the CSV file and stores it into a pandas DataFrame, which is a two-dimensional labeled data structure with columns of potentially different types.

```
# print the first 5 rows of the data to briefly know the structure for each column
data.head()
```

Since we want to get a brief and quick overview of the structure of our dataset, which include the column names and the data type stored in each column, we have used the ‘data.head()’ function. The ‘data.head()’ function allows us to read the first 5 rows of the dataset. It shows the values of each column for those rows.

	LoanID	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner	Default
0	IA35XVH6ZO	28	140466	163781	652	94	2	9.08	48	0.23	High School	Unemployed	Married	No	No	Education	No	0
1	Y8UETC3LSG	28	149227	139759	375	56	3	5.84	36	0.80	PhD	Full-time	Divorced	No	No	Education	Yes	1
2	RM6QSRHIYP	41	23265	63527	829	87	4	9.73	60	0.45	Master's	Full-time	Divorced	Yes	No	Auto	Yes	0
3	GX5YQOGROM	53	117550	95744	395	112	4	3.58	24	0.73	High School	Unemployed	Single	No	No	Auto	Yes	0
4	X0BVPZLDC0	57	139699	88143	635	112	4	5.63	48	0.20	Master's	Part-time	Divorced	No	No	Home	No	0

The figure above shows the result from the ‘data.head()’ function. We can see that the column for this dataset include LoanID, age, income, loan amount, credit score, months employed, number of credit lines, interest rate, loan term, DTI ratio, education status, employment type, marital status, has mortgage, has dependents, loan purpose, has cosigner, and default. The value in the default column is either 0 or 1. 0 means that the loan is approved for the respective loan application, while 1 means that the loan has been rejected for the respective loan application.

```
# print this to know the number of rows and columns in the data as well as the data type for each column
data.info()
```

After that, we used the ‘data.info()’ method so that we can view the concise summary of our dataset. The ‘info()’ method in the pandas library provides values such as the number of entries, the number of non-null values, and the datatype of each column. It is a useful way to get an overview of the dataset’s structure and the types of data it contains. Furthermore, this information also helps us to understand the size and structure of our dataset.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 255327 entries, 0 to 255326
Data columns (total 18 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   LoanID           255327 non-null  object
 1   Age              255327 non-null  int64
 2   Income           255327 non-null  int64
 3   LoanAmount       255327 non-null  int64
 4   CreditScore      255327 non-null  int64
 5   MonthsEmployed   255327 non-null  int64
 6   NumCreditLines   255327 non-null  int64
 7   InterestRate     255327 non-null  float64
 8   LoanTerm         255327 non-null  int64
 9   DTIRatio         255327 non-null  float64
 10  Education        255327 non-null  object
 11  EmploymentType   255327 non-null  object
 12  MaritalStatus    255327 non-null  object
 13  HasMortgage      255327 non-null  object
 14  HasDependents    255327 non-null  object
 15  LoanPurpose      255327 non-null  object
 16  HasCoSigner      255327 non-null  object
 17  Default          255327 non-null  int64
dtypes: float64(2), int64(8), object(8)
memory usage: 35.1+ MB
```

The figure above shows the result obtained from the 'data.info()' method. We can see that our dataset consists of 255,327 rows and 18 columns. Besides that, we can see that out of all 18 columns, the data type for 2 columns (interest rate and DTI ratio) is 'float64', which suggests that the data for these two columns contains floating-point numbers. Meanwhile, there are 8 columns that have the data type of 'int64', while other columns have the data type of 'object'. Data type of 'int64' indicates that the data value in these columns are integer, while data type of 'object' suggests that the data type is string, in which it is categorical variables or mixed types variable. The memory usage for the dataset is approximately about 35.1MB.

```
# print this to know the descriptive statistics for each variable except for categorical variable
data.describe()
```

Next, we used the ‘describe()’ method in pandas to generate the descriptive statistics for numerical columns (10 columns) in the dataset. It provides summary statistics that include count, mean, standard deviation, minimum, quartiles, and maximum values.

	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Default
count	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000	255327.000000
mean	43.498059	82500.225585	127579.236559	574.266125	59.542516	2.501036	13.492848	36.025896	0.500222	0.116118
std	14.990304	38963.150663	70841.308245	158.904496	34.643129	1.117021	6.636456	16.969297	0.230917	0.320367
min	18.000000	15000.000000	5000.000000	300.000000	0.000000	1.000000	2.000000	12.000000	0.100000	0.000000
25%	31.000000	48826.000000	66156.000000	437.000000	30.000000	2.000000	7.770000	24.000000	0.300000	0.000000
50%	43.000000	82467.000000	127557.000000	574.000000	60.000000	2.000000	13.460000	36.000000	0.500000	0.000000
75%	56.000000	116219.000000	188986.500000	712.000000	90.000000	3.000000	19.250000	48.000000	0.700000	0.000000
max	69.000000	149999.000000	249999.000000	849.000000	119.000000	4.000000	25.000000	60.000000	0.900000	1.000000

The figure above illustrates the descriptive statistics for numerical data in our dataset. The result consists of 8 rows and 10 columns. The 10 columns is the column in our dataset that has numerical value, while the 8 rows consist of:

1. count: Number of non-null values in each column.
2. mean: Average value of each column.
3. std: Standard deviation, a measure of the dispersion or spread of the values.
4. min: Minimum value observed in each column.
5. 25%: First quartile (25th percentile), where 25% of the data falls below this value.
6. 50%: Second quartile (median), where 50% of the data falls below this value.
7. 75%: Third quartile (75th percentile), where 75% of the data falls below this value.
8. max: Maximum value observed in each column.

```
# print this to know the descriptive satistics for other variables
data.describe(include=['O'])
```

Next, we used the ‘describe()’ method with the ‘include=['O']’ parameter to generate descriptive statistics for categorical columns in the dataset. This will provide summary statistics for variables such as counts, unique values, the most frequent value, and its frequency.

	LoanID	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner
count	255327	255327	255327	255327	255327	255327	255327	255327
unique	255327	4	4	3	2	2	5	2
top	IA35XVH6ZO	Bachelor's	Part-time	Married	Yes	Yes	Business	Yes
freq	1	64360	64156	85295	127664	127735	51296	127690

The figure above shows the result from the ‘data.describe(include=[‘O’])’ method. We can see that the result consists of 4 rows and 8 columns. The 4 rows consist of:

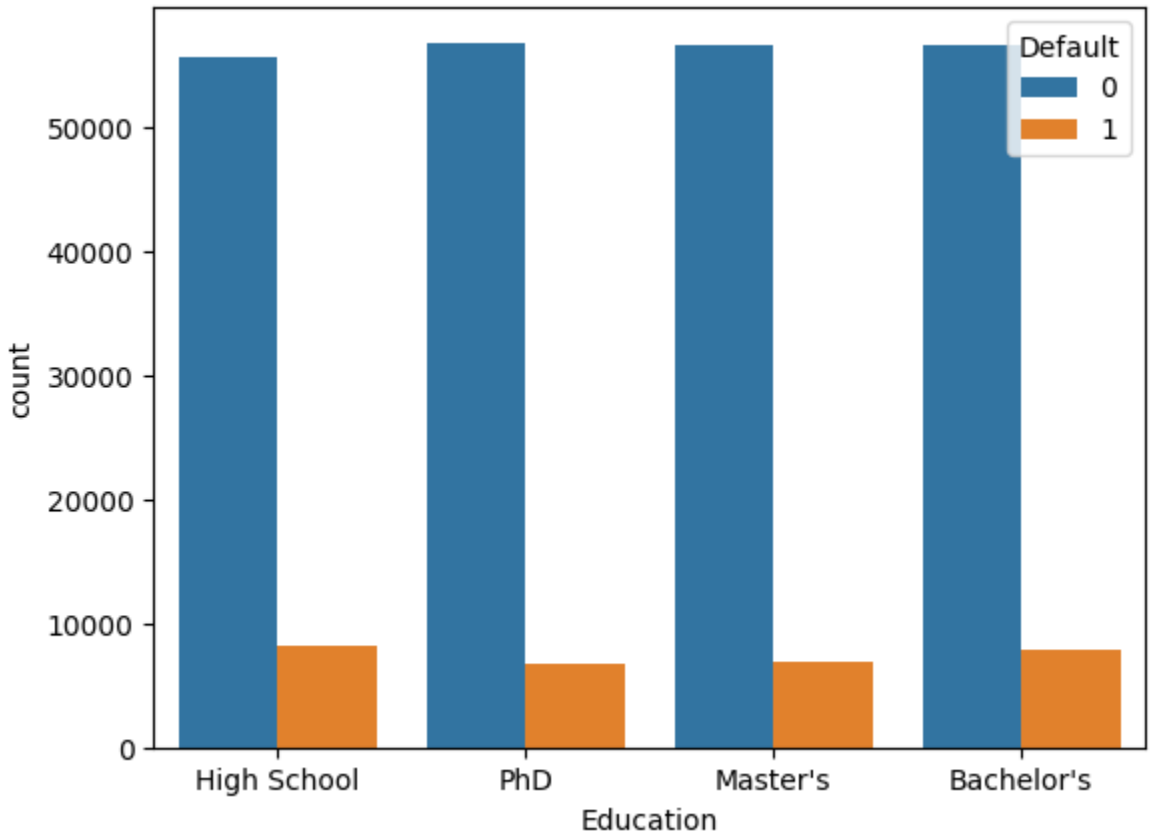
1. count: number of not null values in the column
2. unique: number of unique values in the column.
3. top: most frequent value in the column
4. freq: number of the most frequent value in the column.

From the descriptive statistics provided for the categorical variables in the dataset, we can draw several conclusions:

1. Since each entry in the dataset has a unique loan ID, that means that this column may need to be dropped later since this column cannot provide valuable insight to make predictions for the loan application.
2. Majority of applicants have a Bachelor's degree, followed by other education levels. There are four unique education levels present in the dataset.
3. Most of the applicants are part-time employees. There are four unique employment types present in the dataset.
4. The most common marital status among applicants is Married, followed by other statuses. There are three unique marital statuses present in the dataset.
5. About half of the applicants have a mortgage, while the other half do not. Similarly, about half of the applicants have dependents, while the other half do not. Also, about half of the applicants have a cosigner, while the other half do not. ‘HasMortgage’, ‘HasCoSigner’, and ‘HasDependents’ variables have two unique values indicating either yes or no.
6. The most common loan purpose is Business, followed by other purposes. There are five unique loan purposes present in the dataset.

```
# print the countplot for the education column against the default column
sns.countplot(data, x='Education', hue='Default')
```

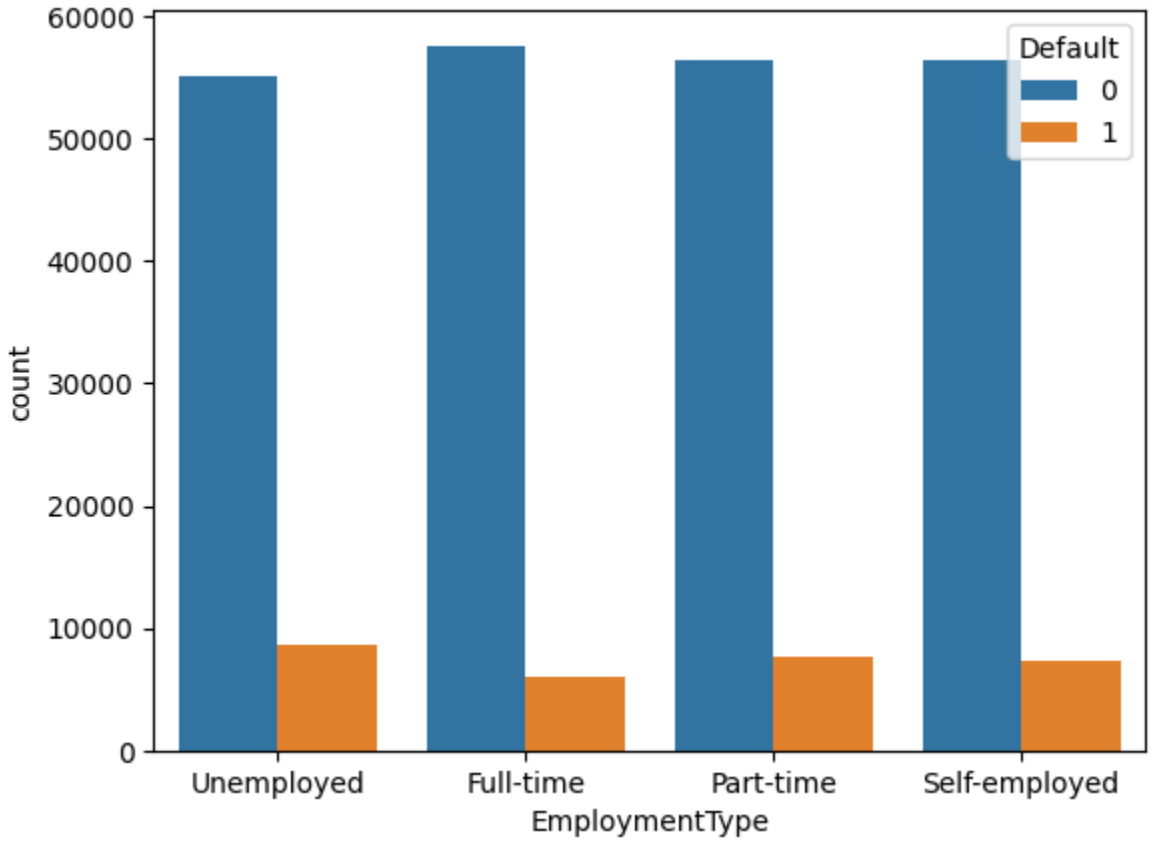
Next, since we have identified that the ‘Default’ column is our target variable column, we have generated a countplot from the Seaborn library (sns) to find the relationship between ‘Education’ column and ‘Default’ column.



The diagram above illustrates the countplot for the ‘Education’ column against the ‘Default’ column. We can see that the ‘Education’ column in our dataset includes values such as ‘High School’, ‘PhD’, ‘Master’s’, and ‘Bachelor’s’. The number of loan applications rejected for applicants that have ‘High School’ education status is slightly higher than the number of loan applications rejected for applicants that have other education status. This may be due to the fact that the applicants with lower education may have less stable or lower incomes, which could make it harder for them to meet repayment obligations.

```
# print the countplot for the employment column against the default column
sns.countplot(data, x='EmploymentType', hue='Default')
```

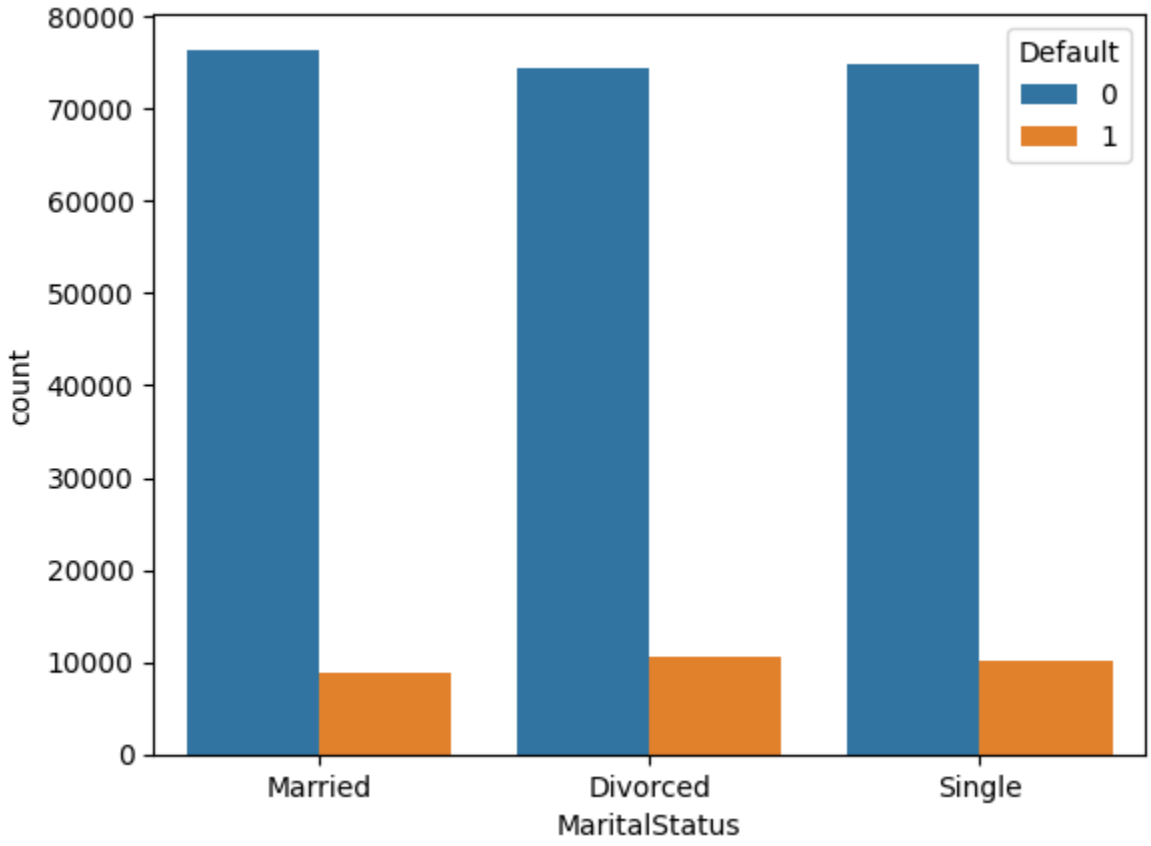
Then, we will print the countplot to illustrate the relationship between the ‘EmploymentType’ column and the ‘Default’ column.



The figure above illustrates the countplot for the ‘EmploymentType’ column against the ‘Default’ column. We can see that the ‘EmploymentType’ column consists of values such as: ‘Unemployed’, ‘Full-time’, ‘Part-time’, and ‘Self-employed’. We can also see that the applicants that have the employment status of ‘Unemployed’ and ‘Part-time’ will have a higher percentage of their loan applications get rejected. This may be due to the fact that these individuals lack of steady income stream to repay the loan.

```
# print the countplot for the marital status column against the default column
sns.countplot(data, x='MaritalStatus', hue='Default')
```

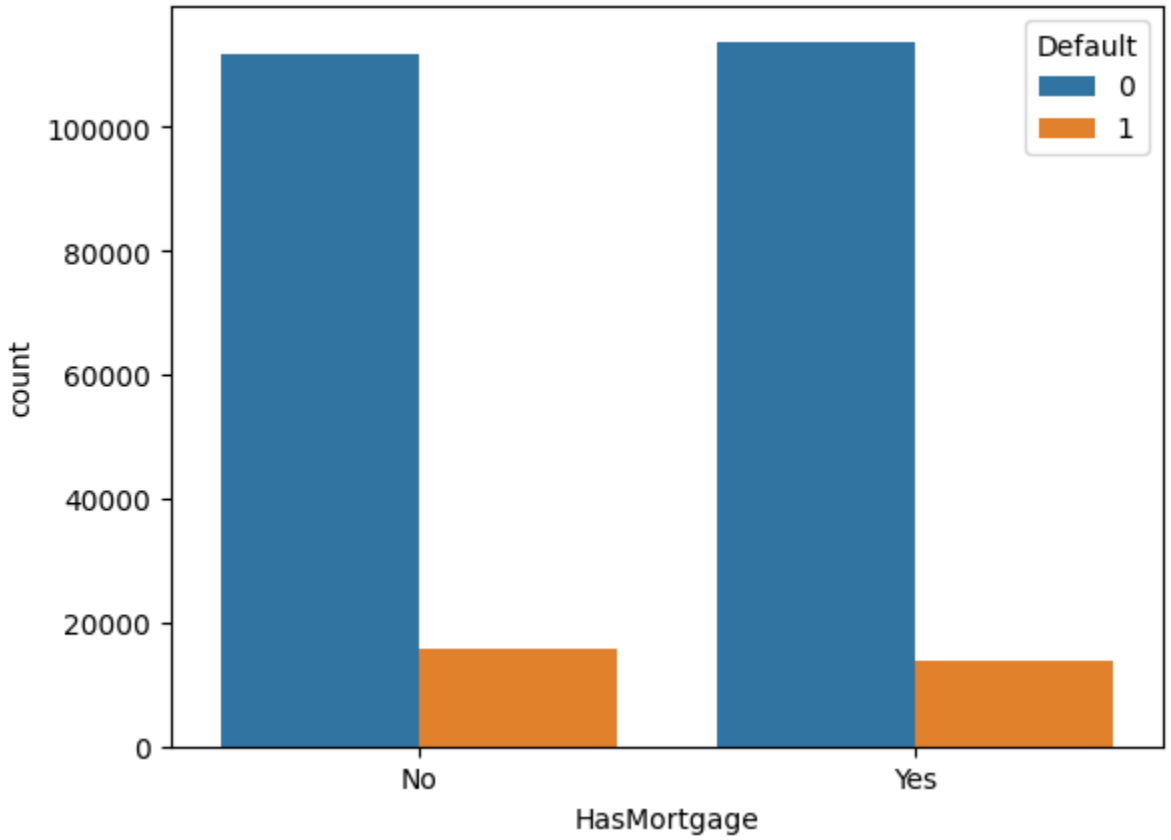
After that, we will print the countplot to illustrate the relationship between the ‘MaritalStatus’ column and the ‘Default’ column.



The figure above illustrates the countplot for the ‘MaritalStatus’ column against the ‘Default’ column. We can see that the ‘MaritalStatus’ column consists of values such as: ‘Married’, ‘Divorced’, and ‘Single’. We can see that applicants that have married will have a slightly higher percentage of loans getting accepted. This is because married couples tend to have two sources of income, which can thus increase their overall household income and improve their ability to repay loans.

```
# print the countplot for the HasMortgage column against the default column
sns.countplot(data, x='HasMortgage', hue='Default')
```

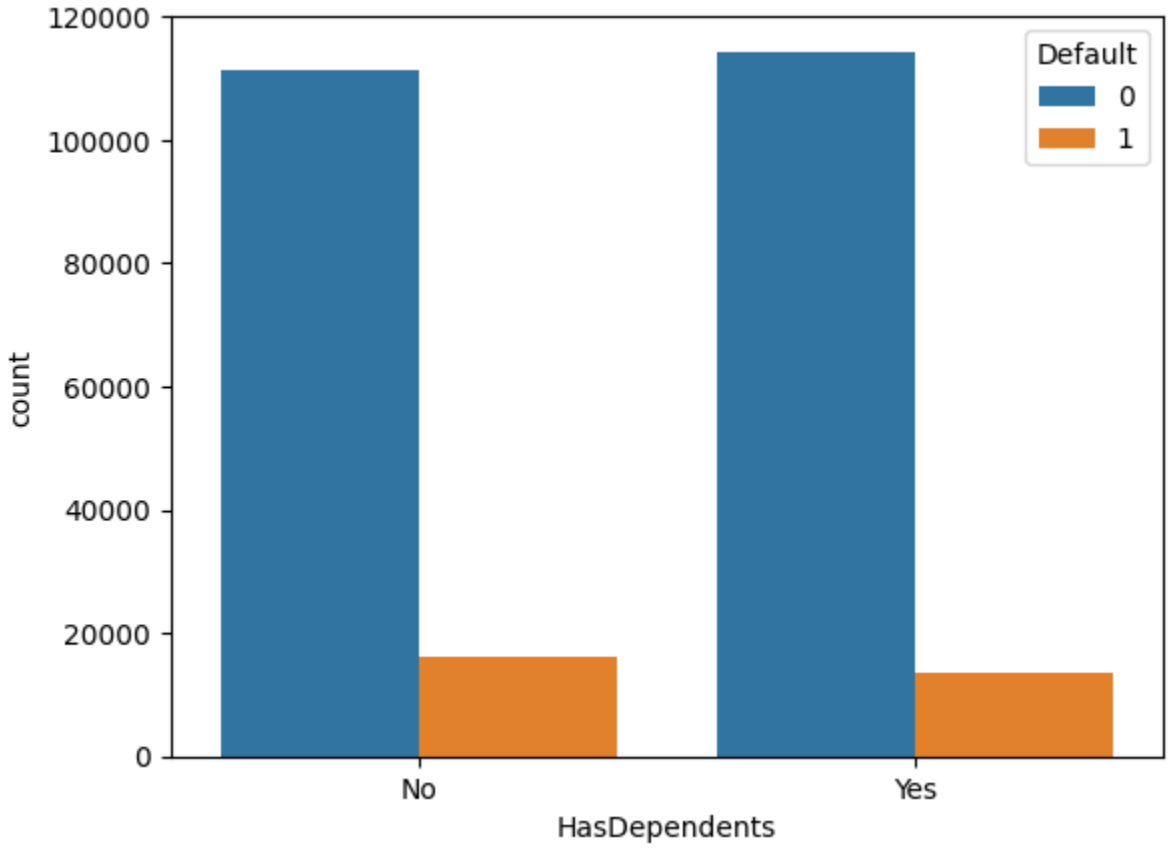
After that, we will print the countplot to illustrate the relationship between the ‘HasMortgage’ column and the ‘Default’ column.



The figure above illustrates the countplot for the ‘HasMortgage’ column against the ‘Default’ column. We can see that the ‘HasMortgage’ column consists of values such as: ‘No’ and ‘Yes’. We can see that applicants that have no mortgage will have a slightly higher percentage of loans getting rejected. This is because individuals who have never had a mortgage may have limited or no credit history associated with large or long-term loans. Lenders rely on credit history to assess an applicant's creditworthiness and ability to manage debt responsibly. Without a mortgage or similar long-term loan on their credit report, applicants may appear riskier to lenders.

```
# print the countplot for the HasDependents column against the default column
sns.countplot(data, x='HasDependents', hue='Default')
```

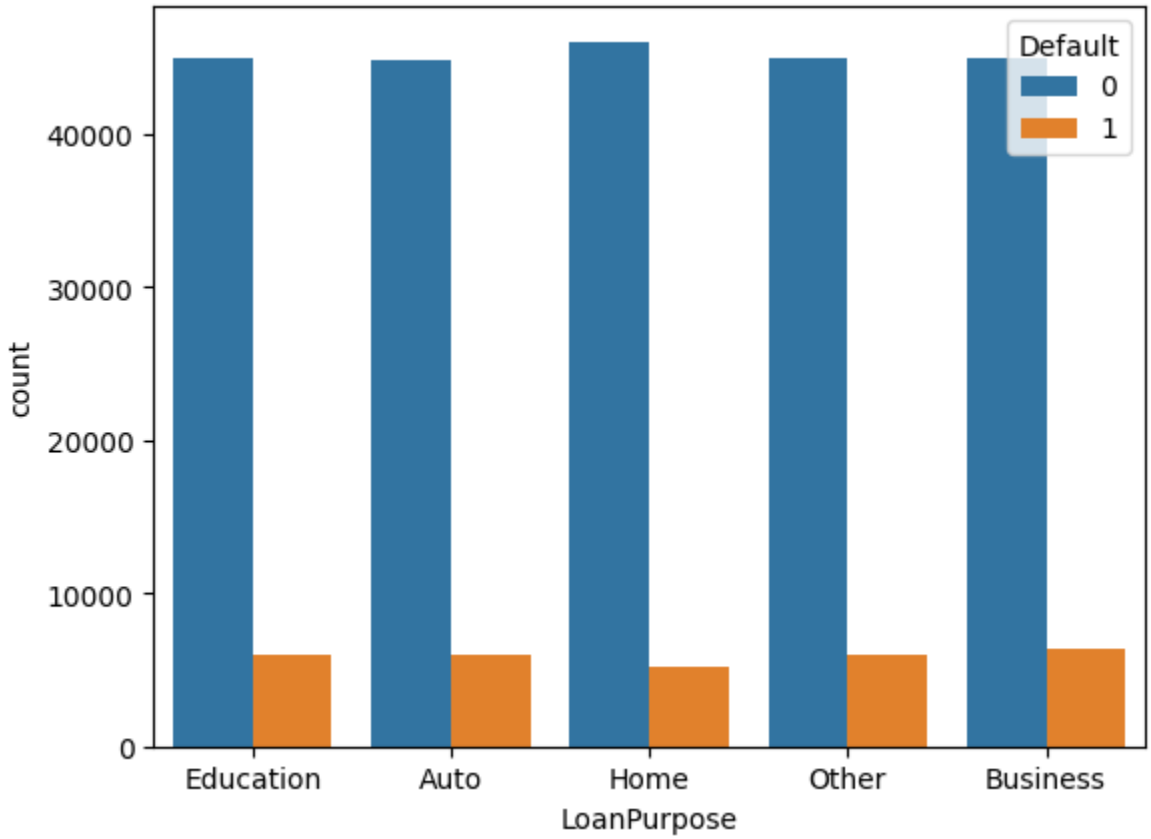
Next, we will print the countplot to illustrate the relationship between the ‘HasDependents’ column and the ‘Default’ column.



The figure above illustrates the countplot for the 'HasDependents' column against the 'Default' column. We can see that the 'HasDependents' column consists of values such as: 'No' and 'Yes'. We can see that applicants that have no dependents will have a slightly higher percentage of loans getting rejected. This is because lenders may treat individuals with dependents to have more stable financial obligations. Dependents, such as children or elderly parents, can represent significant financial responsibilities, including expenses for housing, education, healthcare, and other essential needs. Applicants without dependents may be perceived as having fewer financial commitments, which can raise concerns about their ability to manage and prioritize loan repayments.

```
# print the countplot for the LoanPurpose column against the default column
sns.countplot(data, x='LoanPurpose', hue='Default')
```

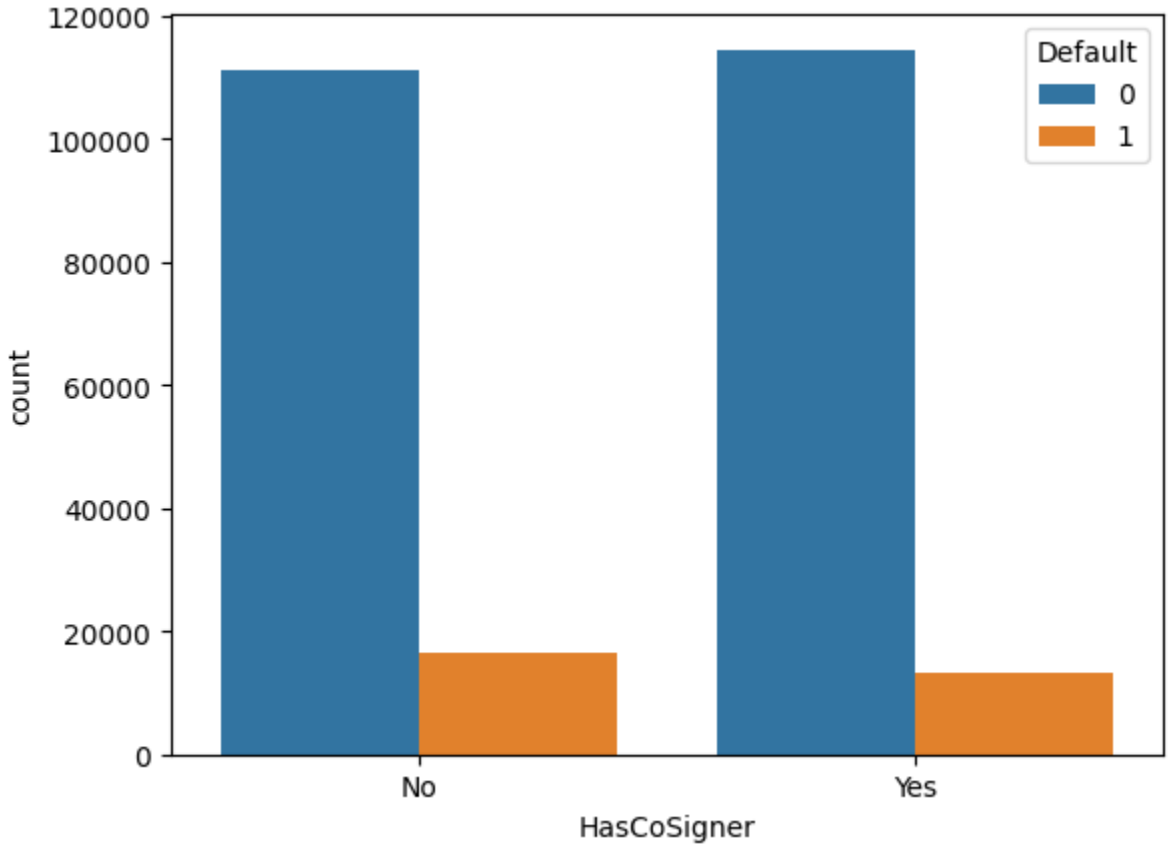
Next, we will print the countplot to illustrate the relationship between the 'LoanPurpose' column and the 'Default' column.



The figure above illustrates the countplot for the 'LoanPurpose' column against the 'Default' column. We can see that the 'LoanPurpose' column consists of values such as: 'Education', 'Auto', 'Home', 'Other', and 'Business'. From this figure, we can see that there is no loan purpose that will have a higher percentage of loan applications getting accepted or rejected. This may be due to the fact that the applicants with different loan purposes have similar levels of financial stability and creditworthiness. Lenders or banks normally will evaluate the loan application based on the applicant's ability to repay. Thus, it is difficult to use loan purposes to evaluate whether the loan application will be accepted or rejected.

```
# print the countplot for the HasCoSigner column against the default column
sns.countplot(data, x='HasCoSigner', hue='Default')
```

Then, we will print the countplot to illustrate the relationship between the ‘HasCoSigner’ column and the ‘Default’ column.

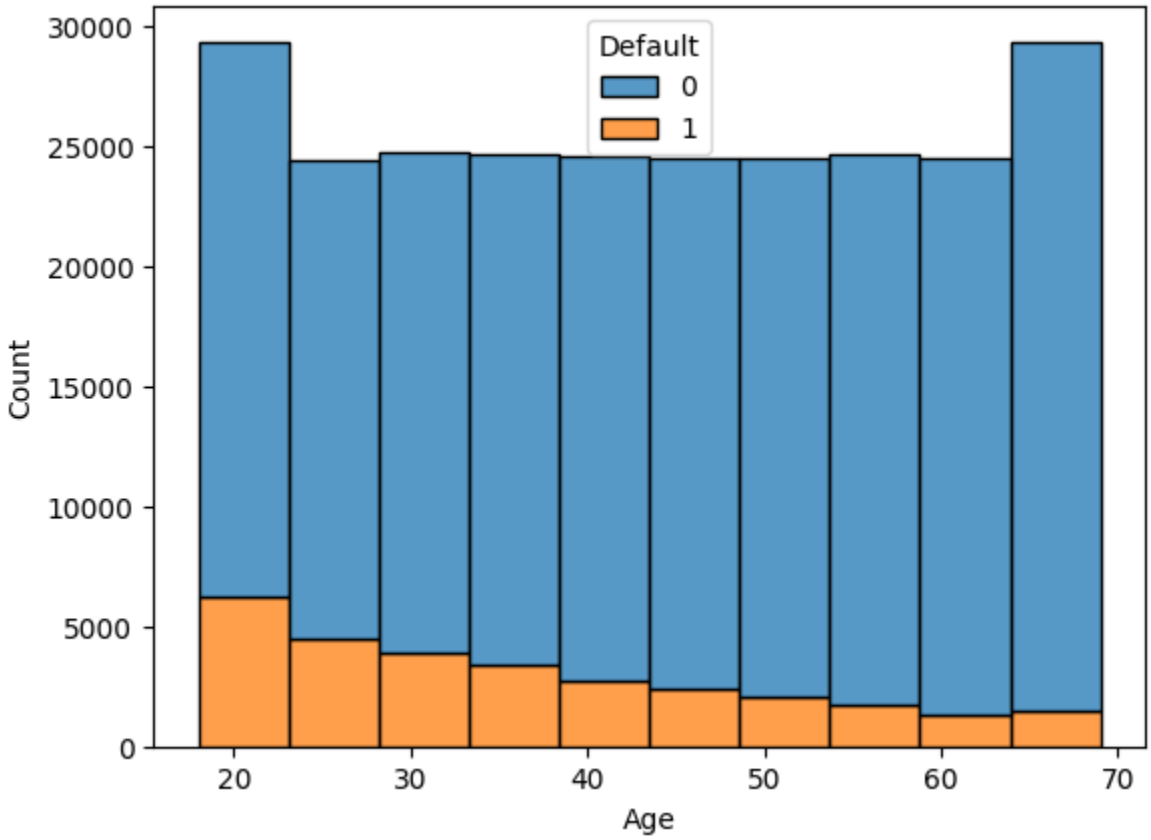


The figure above illustrates the countplot for the ‘HasCoSigner’ column against the ‘Default’ column. We can see that the ‘HasCoSigner’ column consists of values such as: ‘No’, and ‘Yes’. We can see that applicants that have no cosigner will have a slightly higher percentage of loans getting rejected. This is because lenders or banks assess an applicant's ability to repay the loan based on their income. If the primary applicant's income is insufficient or unstable, lenders may view them as a higher risk for default. A cosigner with a stable income can reassure lenders or banks that the loan will be repaid even if the primary applicant faces financial difficulties.

```
# print the histplot for age column against the default column
sns.histplot(data, x='Age', hue='Default', bins=10, multiple="stack")
```

The next diagram that we will plot is the diagram to describe the relationship between the ‘Age’ column and the ‘Default’ column. ‘bins=10’ in our code means that there will be 10 bars in this

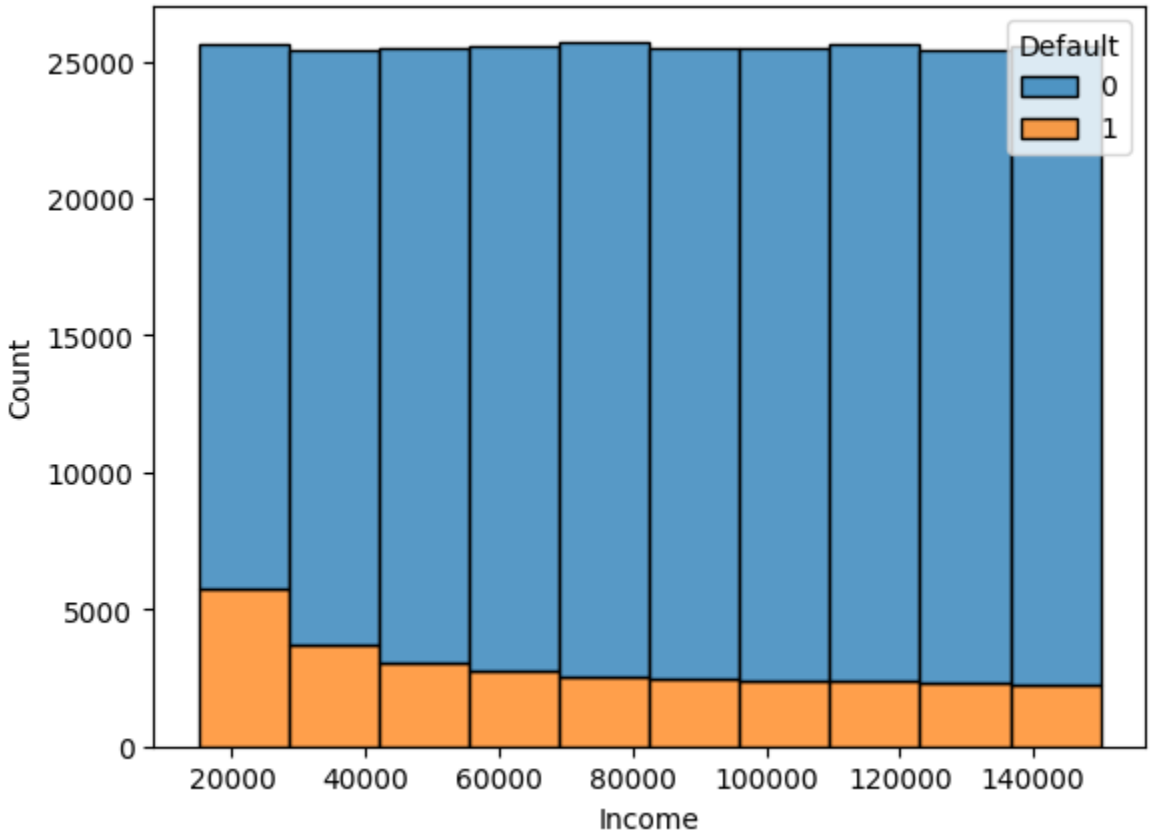
observation. Meanwhile, ‘multiple=stack’ means that the histogram will be stacked vertically, one on top of the other, for each category of the hue variable.



The figure above illustrates the histogram for the ‘Age’ column against the ‘Default’ column. We can see that as the age becomes higher, the number of loan applications getting rejected has decreased. This is due to the fact that older applicants typically have longer credit histories, which lenders can use to assess their creditworthiness. A longer credit history provides more data points for lenders to evaluate, potentially showing a pattern of responsible financial behavior. Moreover, older individuals may have higher incomes due to career progression, accumulated savings, or retirement benefits. A higher income can make it easier for individuals to qualify for loans and demonstrate their ability to repay them.

```
# print the histplot for income column against the default column
sns.histplot(data, x='Income', hue='Default', bins=10, multiple="stack")
```

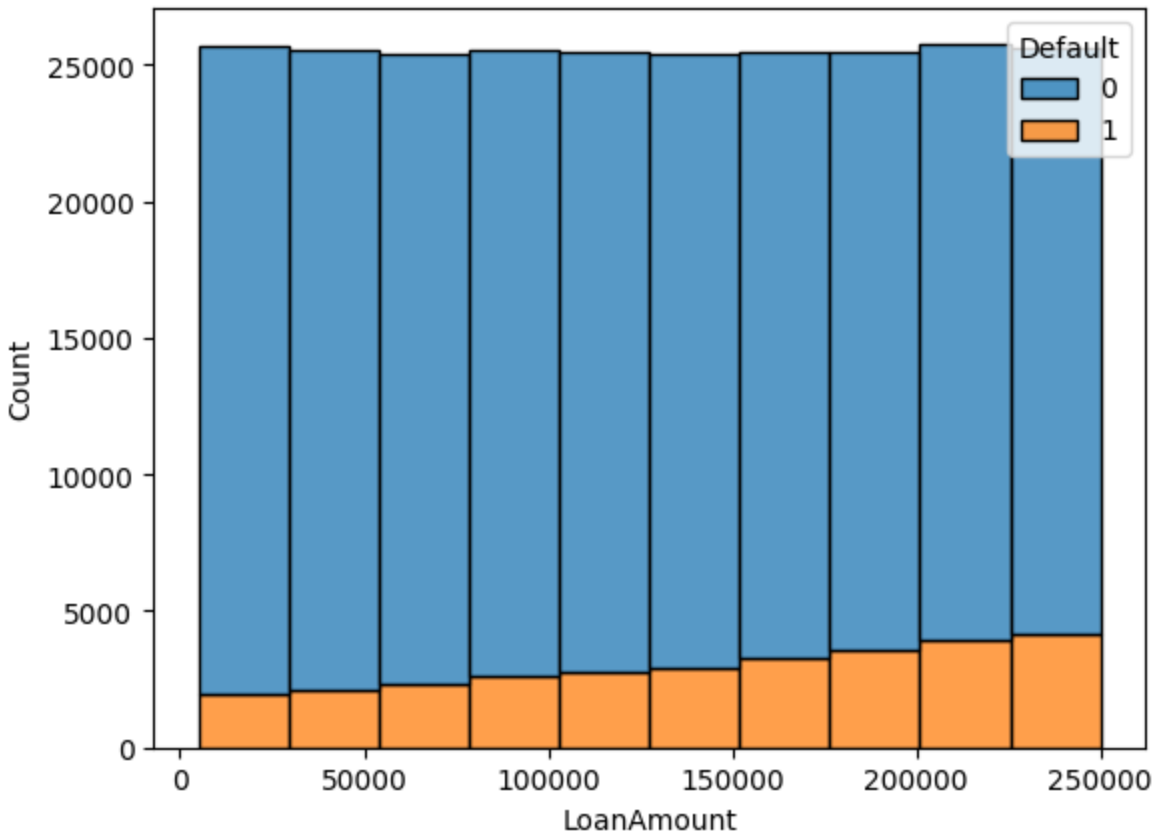
Next, we print out the histogram to describe the relationship between the ‘Income’ column and the ‘Default’ column.



The figure above illustrates the histogram for the ‘Income’ column against the ‘Default’ column. We can see that the percentage of loan applications getting rejected is higher when the income is lower. This may be due to the fact that lenders assess an individual's ability to repay a loan based on their income. If an applicant's income is lower, it may not be sufficient to cover the loan payments along with their existing financial commitments. Lenders are less likely to approve loans for individuals who do not demonstrate the capacity to repay the debt.

```
# print the histplot for loan amount column against the default column
sns.histplot(data, x='LoanAmount', hue='Default', bins=10, multiple="stack")
```

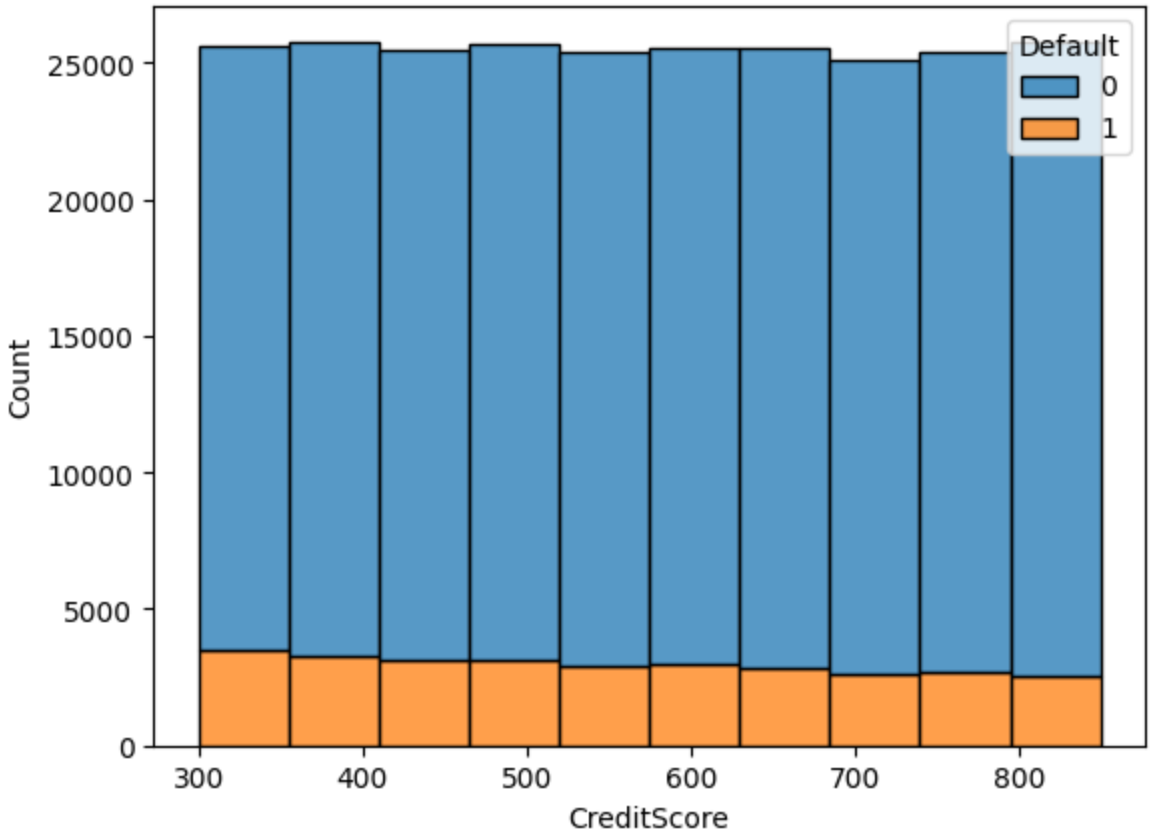
After that, we print out the histogram to describe the relationship between the ‘LoanAmount’ column and the ‘Default’ column.



The figure above illustrates the histogram for the ‘LoanAmount’ column against the ‘Default’ column. We can see as the loan amount becomes higher, the percentage of loan applications getting rejected becomes higher. This is because larger loan amounts represent a higher risk for lenders because they involve larger sums of money and longer repayment periods. Lenders carefully evaluate the risk associated with each loan application, considering factors such as the borrower's creditworthiness, income stability, and debt-to-income ratio. Higher loan amounts may require stricter criteria for approval to mitigate the increased risk.

```
# print the histplot for credit score column against the default column
sns.histplot(data, x='CreditScore', hue='Default', bins=10, multiple="stack")
```

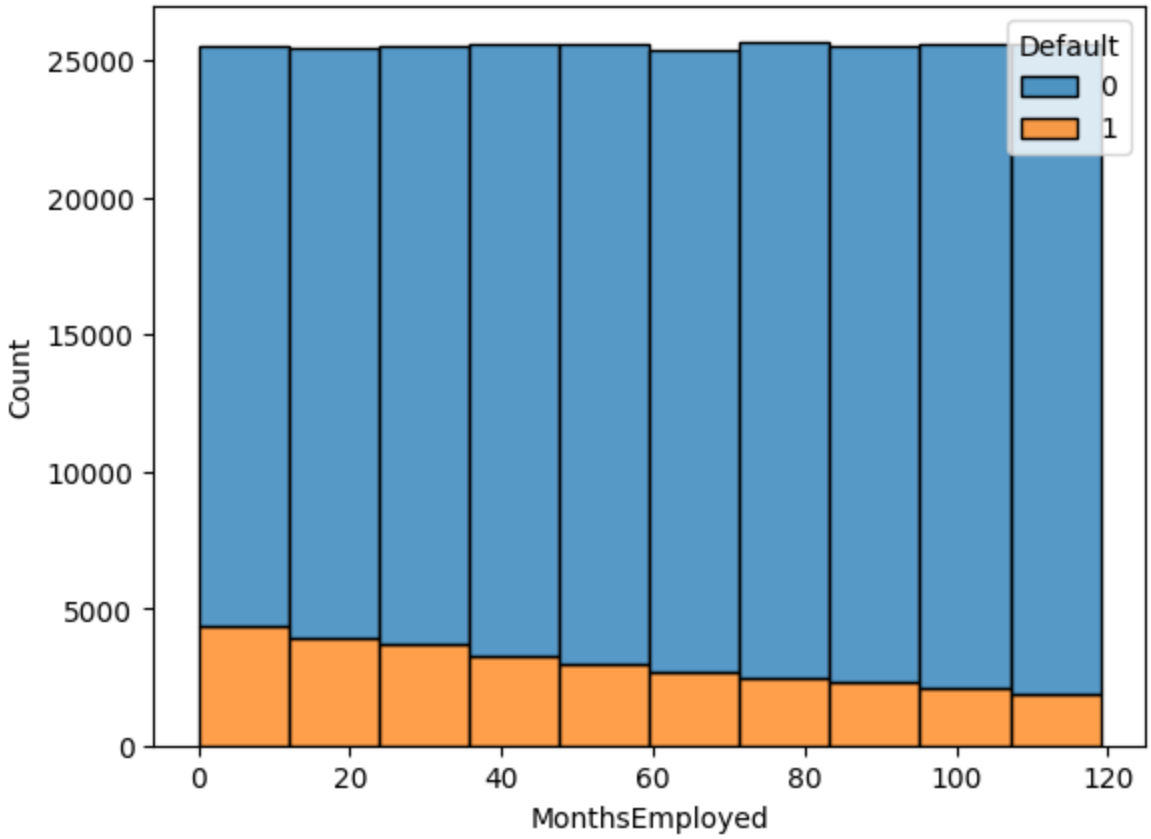
Later, we print out the histogram to describe the relationship between the ‘CreditScore’ column and the ‘Default’ column.



The figure above illustrates the histogram for the ‘CreditScore’ column against the ‘Default’ column. We can see that the relationship between the ‘CreditScore’ column and the ‘Default’ column is quite unclear. This may be due to the fact that although credit score is an important factor in the loan approval process, lenders typically consider multiple criteria, including income, employment history, debt-to-income ratio, and loan amount. A high credit score does not guarantee loan approval if other aspects of the applicant's financial profile are unfavorable, and conversely, a lower credit score may not always result in rejection if other factors are strong.

```
# print the histplot for months employed column against the default column
sns.histplot(data, x='MonthsEmployed', hue='Default', bins=10, multiple="stack")
```

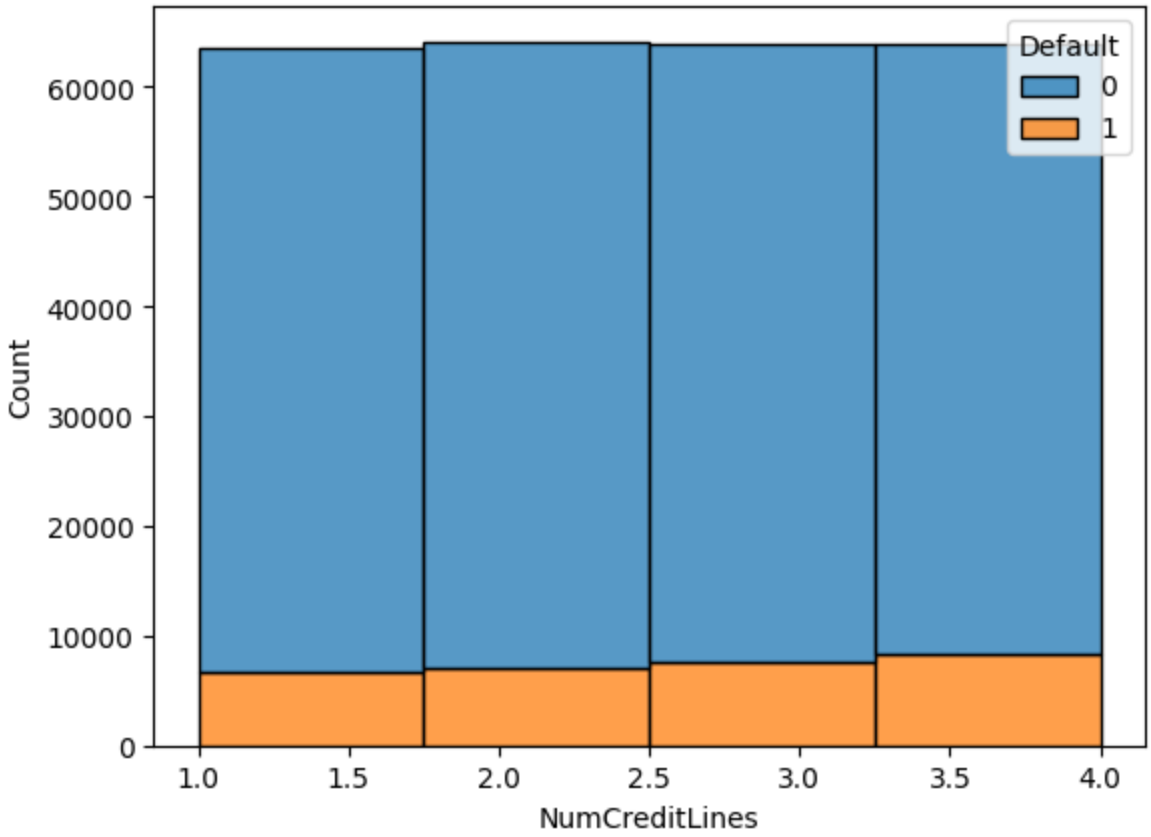
After that, we print out the histogram to describe the relationship between the ‘MonthsEmployed’ column and the ‘Default’ column.



The figure above illustrates the histogram for the 'MonthsEmployed' column against the 'Default' column. We can see that the applicants that have higher employment tenure will be more likely to have their loan applications accepted. This is because longer employment tenure suggests a steady source of income for the borrower. Lenders often view applicants with stable employment as less risky because they are more likely to have a reliable income stream to support loan repayments. Applicants with shorter job histories may face higher rejection rates due to concerns about income volatility or job insecurity.

```
# print the histplot for NumCreditLine column against the default column
sns.histplot(data, x='NumCreditLines', hue='Default', bins=4, multiple="stack")
```

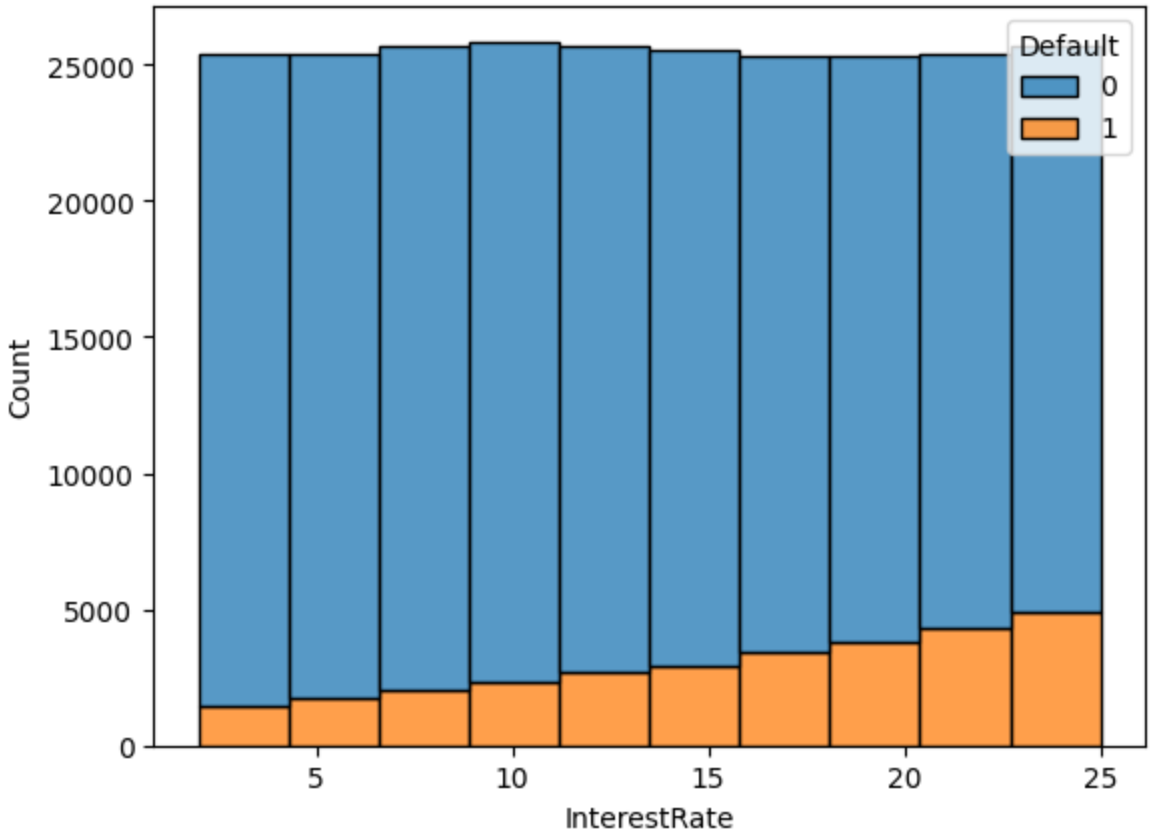
Then, we print out the histogram to describe the relationship between the 'NumCreditLines' column and the 'Default' column.



The figure above illustrates the histogram for the 'NumCreditLines' column against the 'Default' column. We can see that as the number of credit lines become higher, the percentage of loan applications getting rejected becomes higher. This may be due to the fact that having multiple credit lines means the applicant has access to more credit. If the applicant has already borrowed a significant amount across various credit lines, lenders may be concerned about their ability to take on additional debt without becoming overextended. High levels of existing debt can strain the applicant's finances and increase the risk of default, leading to higher rejection rates for new loan applications.

```
# print the histplot for interest rate column against the default column
sns.histplot(data, x='InterestRate', hue='Default', bins=10, multiple="stack")
```

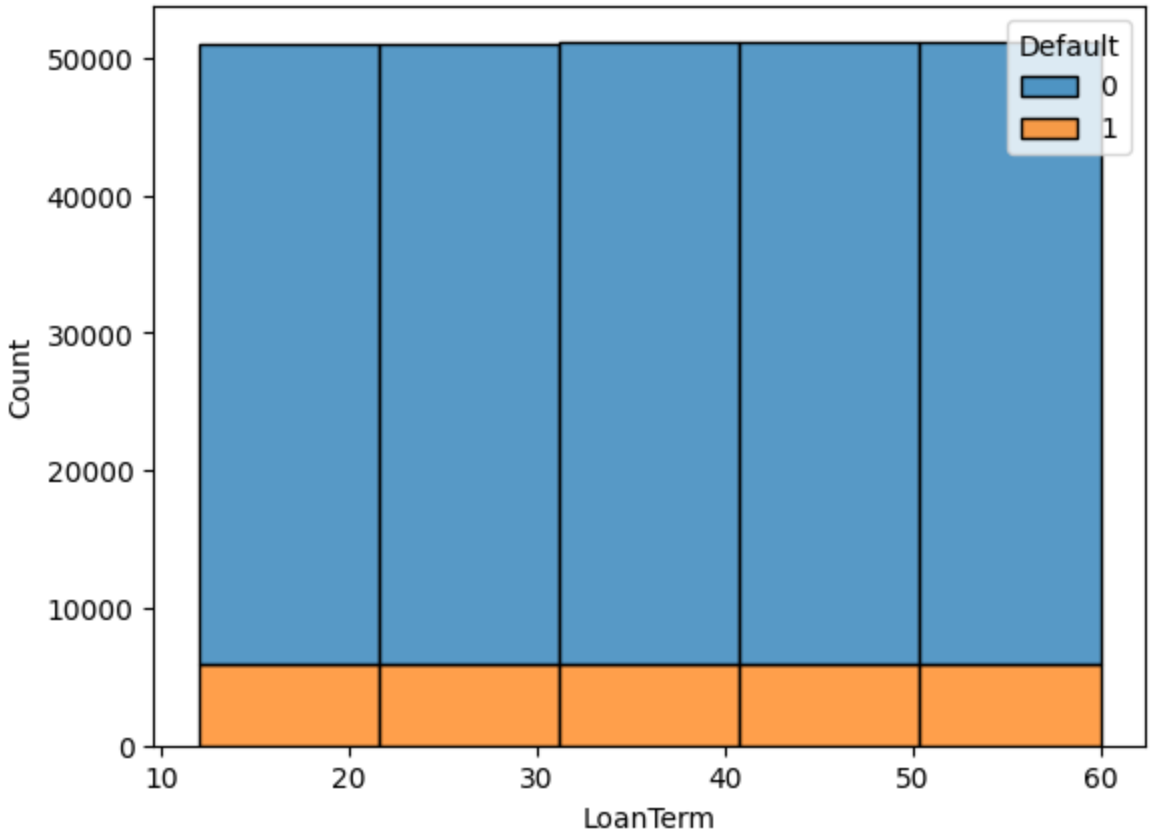
Next, we print out the histogram to describe the relationship between the 'InterestRate' column and the 'Default' column.



The figure above illustrates the histogram for the ‘InterestRate’ column against the ‘Default’ column. We can see that as the interest rate becomes higher, the percentage of loan applications getting rejected becomes higher. This may be due to the fact that higher interest rates result in higher monthly loan payments for applicants. Lenders assess applicants' ability to afford these payments based on their income and existing financial obligations. If the higher payments strain the applicants’ budget or exceed their debt-to-income ratio threshold, lenders may reject the loan application due to concerns about repayment capacity.

```
# print the histplot for loan term column against the default column
sns.histplot(data, x='LoanTerm', hue='Default', bins=5, multiple="stack")
```

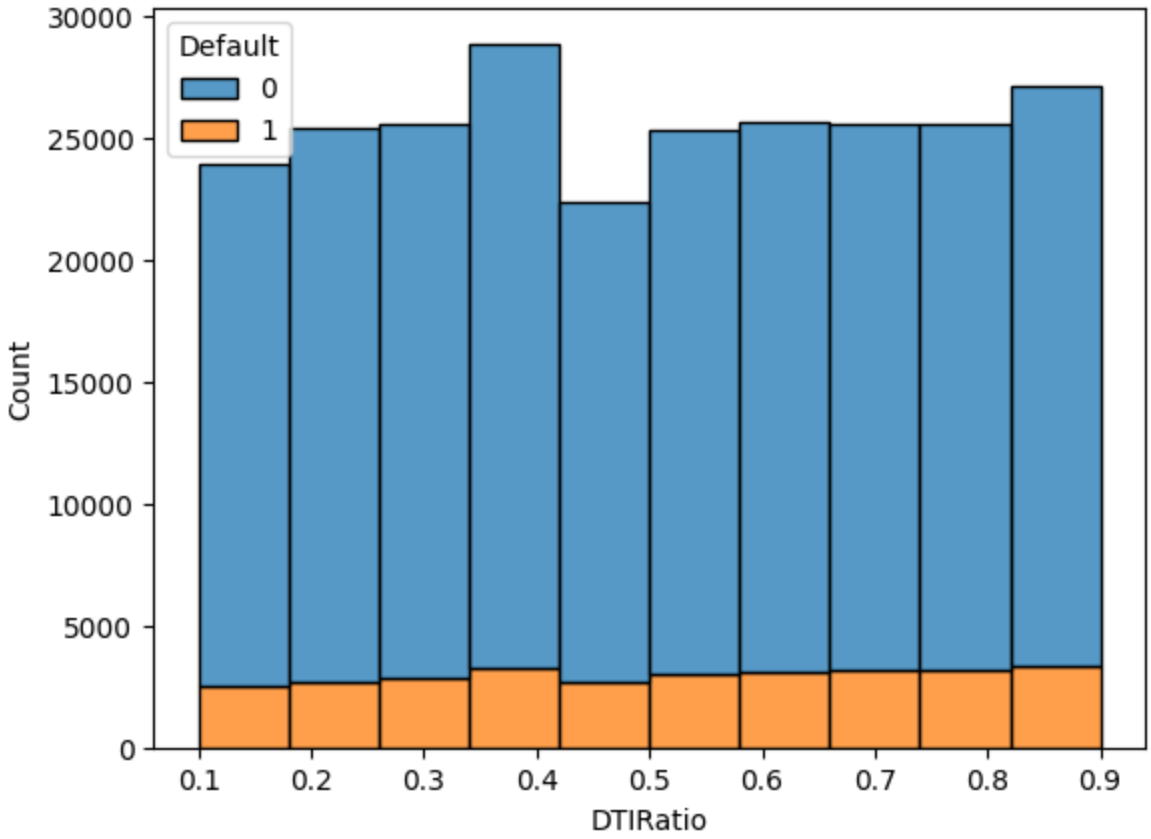
After that, we print out the histogram to describe the relationship between the ‘LoanTerm’ column and the ‘Default’ column.



The figure above illustrates the histogram for the ‘LoanTerm’ column against the ‘Default’ column. We can see that the relationship between the ‘LoanTerm’ column and the ‘Default’ column is unclear. This is because some applicants prefer shorter loan terms because they want to pay off their debt quickly and minimize interest costs. Others may opt for longer loan terms to reduce their monthly payments and improve cash flow. The diversity of applicant preferences can lead to variations in loan approval rates across different loan terms.

```
# print the histplot for DTIRatio column against the default column
sns.histplot(data, x='DTIRatio', hue='Default', bins=10, multiple="stack")
```

Then, we print out the histogram to describe the relationship between the ‘DTIRatio’ column and the ‘Default’ column.



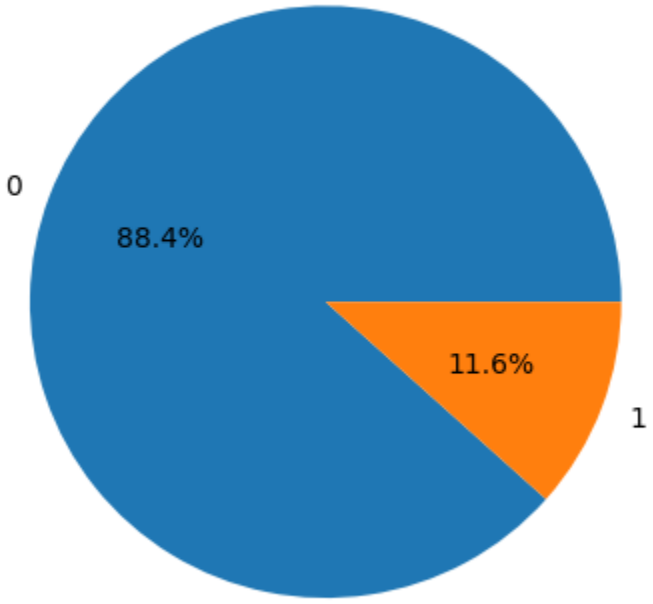
The figure above illustrates the histogram for the 'DTIRatio' column against the 'Default' column. The meaning of 'DTIRatio' is the debt-to-income ratio. We can see that the relationship between the debt-to-income ratio and the percentage of loan application approval is unclear. This is because applicants with different DTI ratios may have diverse financial backgrounds, employment histories, credit scores, and debt obligations. While a high DTI ratio may be a red flag for some lenders, others may approve loans for applicants with higher DTI ratios if they have strong credit scores, stable income, or other compensating factors. The relationship between DTI ratio and loan approval rates can depend on the overall financial health and risk profile of the applicant.

```
# print a pie chart for the Default column to see whether the class is imbalanced or not
temp = data['Default'].value_counts()
plt.pie(temp.values, labels=temp.index, autopct='%1.1f%%')
plt.show()
```

Then, we will need to identify whether the target variable class is balanced or not. Firstly, we calculate the value counts for the 'Default' column, which is the target variable column in our

dataset. The value_counts() function from pandas library returns a series containing the numbers of unique values in descending order. Here, the temp variable contains the unique values in the ‘Default’ column, as well as their respective counts.

Then, we create a pie chart, by using the plt.pie() function. The first argument is the data values for the wedges of the pie chart, which are the counts of each unique value in the 'Default' column (temp.values). The second argument provides labels for each wedge, which are the unique values themselves (temp.index). The autopct parameter specifies the format of the percentages displayed on each wedge. Lastly, we use plt.show() to display the pie chart.



The figure above illustrates the pie chart for the ‘Default’ column in the dataset. From this figure, we can see that 88.4% of the data in the ‘Default’ column has the value of 0, which means that their loan application has been accepted. Meanwhile, 11.6% of the loan application has been rejected. Thus, it suggests that the ‘Default’ column class is very imbalanced. So, we need to use some sampling techniques to tackle the class imbalance problem.

```
# Define feature vector and target variable (class)
X = data.drop(['Default'], axis=1)
y = data['Default']
```

Before we tackle the class imbalance problem, we will need to preprocess the data. Firstly, we need to define the independent and dependent variables. The dependent variable is the 'Default' column, while other variables belong to the independent variable. So, in this code, we have dropped the 'Default' column from the X variable, and set the 'Default' column and its value to the Y variable.

```
# deleting of unneeded columns since this columns will not help us in making a predictive model
X.drop(['LoanID'], axis=1, inplace=True)
```

Next, we need to drop the unnecessary variable in the X variable. In this code, we have removed the 'LoanID' variable from the dataset. This is because the 'LoanID' variable is just use to differentiate different loan applications and this variable will not help in making predictions.

```
# Transforming the qualitatives variables into quantitatives for training data
X['Education'] = X['Education'].map({"Bachelor's": 0, "High School": 1, "Master's": 2, "PhD": 3} ).astype(int)
X['EmploymentType'] = X['EmploymentType'].map( {'Full-time': 0, 'Part-time': 1,'Unemployed':2, 'Self-employed': 3} ).astype(int)
X['MaritalStatus'] = X['MaritalStatus'].map( {'Divorced': 0, 'Married': 1, 'Single': 2}).astype(int)
X['HasMortgage'] = X['HasMortgage'].map( {'No': 0, 'Yes': 1}).astype(int)
X['HasDependents'] = X['HasDependents'].map( {'No': 0, 'Yes': 1}).astype(int)
X['LoanPurpose'] = X['LoanPurpose'].map( {'Auto': 0, 'Business': 1, 'Education': 2, 'Home': 3, 'Other': 4}).astype(int)
X['HasCoSigner'] = X['HasCoSigner'].map( {'No': 0, 'Yes': 1}).astype(int)
X.head()
```

Then, we need to transform the value in the categorical variable into the numerical variable. This step is important because we need to use the categorical variable as numeric input when we train a model. The categorical columns that we need to transform include 'Education', 'EmploymentType', 'MaritalStatus', 'HasMortgage', 'HasDependents', 'LoanPurpose', and 'HasCoSigner' column. After we have transformed the above categorical columns, we need to show the first 5 rows of the dataset to check whether these categorical variables values have been transformed to numeric values.

	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner
0	28	140466	163781	652	94	2	9.08	48	0.23	1	2	1	0	0	2	0
1	28	149227	139759	375	56	3	5.84	36	0.80	3	0	0	0	0	2	1
2	41	23265	63527	829	87	4	9.73	60	0.45	2	0	0	1	0	0	1
3	53	117550	95744	395	112	4	3.58	24	0.73	1	2	2	0	0	0	1
4	57	139699	88143	635	112	4	5.63	48	0.20	2	1	0	0	0	3	0

The figure above shows the result after transforming the categorical variables into numeric values. We can see that the values in the ‘Education’ column change to 1,2 or 3 instead of ‘Bachelor’s’, ‘Master’s’, or ‘PhD’.

```
# Set up the oversampling method
oversampler = SMOTE()
X_new, y_new = oversampler.fit_resample(X, y)

X.shape, X_new.shape
```

After that, we use the sampling techniques to handle the imbalance class issues. The method that we have used for oversampling is SMOTE (Synthetic Minority Over-sampling Technique). This method will create synthetic data samples based on the minority class to balance the class distribution. Firstly, we initialize a SMOTE oversampler. Next, we will apply the fit_resample() method in the oversampler on our X and y variable to create new synthetic data. The resulting ‘X_new’ and ‘y_new’ contain the oversampled data. Lastly, we will show the shapes of the original and oversampled feature matrices, which helps us to detect the effect of oversampling.

```
((255327, 16), (451358, 16))
```

The figure above shows the value X.shape and X_new.shape. We can see that the original data contains 255327 rows. After applying the oversampling method, the number of data increases to 451358 rows.

2.2 Data Splitting

```
#Split data into separate training, test, and validation set
X_train, X_temp, y_train, y_temp = train_test_split(X_new, y_new, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

Then, we go to the data splitting process in which we split the data into 3 different sets, that is training set, test set, and validation set by using the train_test_split function from the scikit-learn library. So, the data for training the machine learning model(X_train and y_train) is 60% of the original data. Then the validation set that is used to optimize model parameters like using hyperparameter tuning.(X_val and y_val) consists of about 20% of the original data. The test set(X_test and y_test) also uses 20% of the original data to evaluate the model performance.

```
#Check the shape of X_train and X_test
print("train sample size: ", X_train.shape)
print("test sample size: ", X_test.shape)
print("validation sample size: ", X_val.shape)
```

This code will display the shape of the training data (X_train), the shape of the test data (X_test), and the shape of the validation data (X_val).

```
train sample size:  (270814, 16)
test sample size:  (90272, 16)
validation sample size:  (90272, 16)
```

This result shows the shape of each data set and the result is showing the data sets has been successfully split into the designated size.

2.3 Data Preprocessing

```
# Looking for null values in training data
print("Null values:\n", X_train.isnull().sum())
```

After that, we go to the data preprocessing phase in which we first check for null values or missing data in the training data (X_train) by using the .isnull() method which is a pandas function specifically designed to identify missing data. Then this code will display the result of the .sum() method that is applied to the boolean DataFrame that will be treating True (missing values) as 1 and False (valid values) as 0.

```
Null values:
 Age                0
Income              0
LoanAmount          0
CreditScore         0
MonthsEmployed      0
NumCreditLines      0
InterestRate        0
LoanTerm            0
DTIRatio            0
Education           0
EmploymentType      0
MaritalStatus       0
HasMortgage         0
HasDependents       0
LoanPurpose         0
HasCoSigner         0
dtype: int64
```

This result shows the training data did not have any missing values or null values.

```
# Check duplicate rows in training data
duplicate_rows = X_train[X_train.duplicated()]
print("Number of duplicate rows: ", duplicate_rows.shape)
```

This code checks for duplicate rows in our training data (X_train) using the .duplicated() method in pandas. This method identifies rows that are entirely identical to another row within the

dataset and considers all the columns. It will return a boolean series for each row. True indicates a duplicate row, and false indicates a unique row. Then the number of duplicate rows that is found in the training data is printed out with its columns.

```
Number of duplicate rows:  (0, 16)
```

This result shows that the training data did not contain any duplicate data.

```
# Removing outliers using z-score
z = np.abs(stats.zscore(X_train))

df3 = X_train[(z<3).all(axis=1)]
df3.shape
```

This code is to removes outliers from the dataset using the z-score method. The first line "z = np.abs(stats.zscore(X_train))" is to calculates the z-scores of the value in the training data. In "df3 = X_train[(z < 3).all(axis=1)]" , z < 3 is a condition that check the z-scores that less then 3, 3 is a threshold for outlier detection using z-scores. It creates a new DataFrame,df3 that contains the rows that passed the outlier check. The shape of df3 will also be displayed.

```
(270814, 16)
```

This result shows the non-outliers rows are a total of 270814 rows.

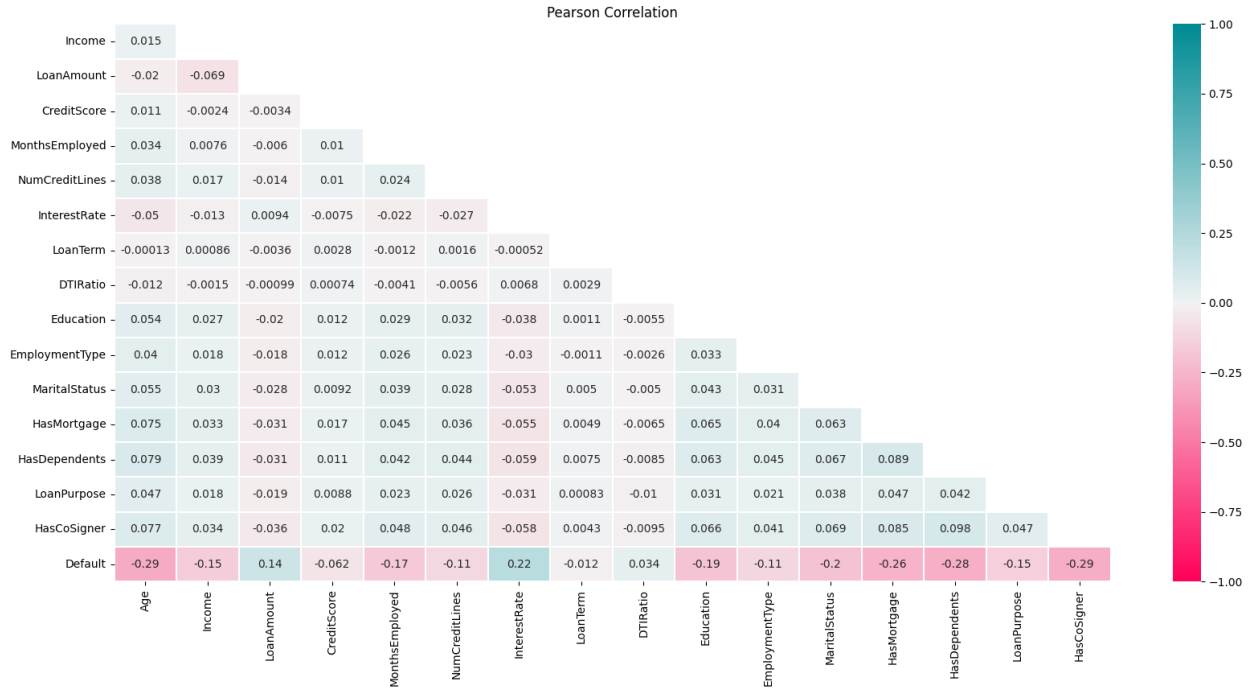
```
df3['Default'] = y_train

# Finding the correlation between variables
pearsonCorr = df3.corr(method='pearson')

# Create mask for both correlation matrices
# Pearson corr masking
# Generating mask for upper triangle
maskP = np.triu(np.ones_like(pearsonCorr,dtype=bool))

# Adjust mask and correlation
maskP = maskP[1:,:-1]
pCorr = pearsonCorr.iloc[1:,:-1].copy()
# Setting up a diverging palette
cmap = sns.diverging_palette(0, 200, 150, 50, as_cmap=True)
fig = plt.subplots(figsize=(20,9))
sns.heatmap(pCorr, vmin=-1,vmax=1, cmap = cmap, annot=True, linewidth=0.3, mask=maskP)
plt.title("Pearson Correlation")
```

The code snippet creates a heatmap that illustrates the relationship between each independent and dependent variable. It first adds a new column named 'Default' to the dataset and populates it with the values from the variable y_train. The reason we need to apply this step is because we need to add the dependent variable values into the independent variable so that we can illustrate the heatmap. Next, we need to calculate the Pearson correlation matrix for the dataset using the corr() method with the method parameter set to 'pearson'. Then, we create a mask to hide the upper triangle of the correlation matrix to avoid redundant information. Later, we generate a heatmap of the Pearson correlation matrix using seaborn library and display it with annotated correlation values and a diverging color palette. This visualization helps to understand the pairwise relationships between variables in the df3 by displaying correlation coefficients. Positive correlations are indicated by lighter shades, negative correlations by darker shades, and no correlation by a neutral color. The annotations on the heatmap provide the actual correlation coefficients.



The figure above shows the result after running the code to generate the heatmap. From this figure, we can see that each variable is not correlated. Besides that, only several independent variables have small positive or negative correlation with the dependent variable which is the 'Default' variable. Since all of the correlation coefficient values are less than 0.3, it means that the data in the independent variable is not highly correlated with the dependent variable. So, we do not need to remove the variable from our dataset.

```
# Looking for null values in testing data
print("Null values:\n", X_test.isnull().sum())
```

Then, we check for null values or missing data in the testing data (X_test) by using the .isnull() method which is a pandas function specifically designed to identify missing data. Then this code will display the result of the .sum() method that is applied to the boolean in which the missing data will be treating True (missing values) as 1 and False (valid values) as 0.

```
Null values:
 Age                0
Income              0
LoanAmount          0
CreditScore         0
MonthsEmployed      0
NumCreditLines      0
InterestRate        0
LoanTerm            0
DTIRatio            0
Education           0
EmploymentType      0
MaritalStatus       0
HasMortgage         0
HasDependents       0
LoanPurpose         0
HasCoSigner         0
dtype: int64
```

The figure above shows the result of missing data in the testing dataset. The result shows that there is no data missing in the testing dataset.

```
# Check duplicate rows in testing data
duplicate_rows = X_test[X_test.duplicated()]
print("Number of duplicate rows: ", duplicate_rows.shape)
```

This code checks for duplicate rows in our testing data (X_test) using the .duplicated() method in pandas. This method identifies rows that are entirely identical to another row within the dataset and considers all the columns. It will return a boolean series for each row. True indicates a duplicate row, and false indicates a unique row. Then the number of duplicate rows that is found in the testing data is printed out with its columns.

```
Number of duplicate rows:  (0, 16)
```

The figure above shows the result of duplicate rows in the testing dataset. The result shows that there is no duplicate row in the testing dataset.

```
# Looking for null values in validation data
print("Null values:\n", X_val.isnull().sum())
```

Then, we check for null values or missing data in the validation data (X_val) by using the .isnull() method which is a pandas function specifically designed to identify missing data. Then this code will display the result of the .sum() method that is applied to the boolean in which the missing data will be treating True (missing values) as 1 and False (valid values) as 0.

```
Null values:
 Age                0
Income              0
LoanAmount          0
CreditScore         0
MonthsEmployed      0
NumCreditLines      0
InterestRate        0
LoanTerm            0
DTIRatio            0
Education           0
EmploymentType      0
MaritalStatus       0
HasMortgage         0
HasDependents       0
LoanPurpose         0
HasCoSigner         0
dtype: int64
```

The figure above shows the result of missing data in the validation dataset. The result shows that there is no data missing in the validation dataset.

```
# Check duplicate rows in validation data
duplicate_rows = X_val[X_val.duplicated()]
print("Number of duplicate rows: ", duplicate_rows.shape)
```

This code checks for duplicate rows in our validation data (X_val) using the .duplicated() method in pandas. This method identifies rows that are entirely identical to another row within the dataset and considers all the columns. It will return a boolean series for each row. True indicates a duplicate row, and false indicates a unique row. Then the number of duplicate rows that is found in the validation data is printed out with its columns.

```
Number of duplicate rows:  (0, 16)
```

The figure above shows the result of duplicate rows in the validation dataset. The result shows that there is no duplicate row in the validation dataset.

3.0 Creation

```
# Scale the features using StandardScaler
scaler = StandardScaler()
X_train_new = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
X_val = scaler.transform(X_val)
```

Next, we step into the next stage of the machine learning project which is the creation stage. In this stage, we build and train a model based on different algorithms, which include Logistic Regression, Decision Trees, Random Forest Classifier, Naive Bayes, Neural Network, K-Nearest Neighbor (KNN), and Support Vendor Machine. Then, we evaluate each model based on their accuracy, precision, recall, f1-score, and auc score on the testing and validation data. We choose a model that has the highest accuracy, precision, recall, f1-score, and auc score and store it to the next stage.

Before we start building a model, we will need to standardize our dataset. The code above shows the steps that we do to standardize our data. This step is important because unstandardized data may lead to biased results. So, we use the fit_transform() function from the sklearn.preprocessing library to standardize our dataset on the training, testing, and validation data.

3.1 Logistic Regression

```
logReg = LogisticRegression(random_state=0, solver='liblinear')
logReg.fit(X_train_new, y_train)

y_pred_logReg = logReg.predict(X_test)
y_pred_logReg_val = logReg.predict(X_val)
```

The first model that we build is the model based on the Logistic Regression. The parameters that we have set in here are the random_state and solver. The random_state at here sets the random

seed for reproducibility while the solver specifies the algorithm to use for optimization. In this situation, we use the 'liblinear' as our algorithm for the optimization.

```
# Classification Report for LR with validation data
print("\nLogistic Regression Evaluation:\n")
print(classification_report(y_val, y_pred_logReg_val))

# Confusion Matrix for LR with validation data
print("\nLogistic Regression confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_logReg_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Logistic Regression based on the validation data.

```
Logistic Regression Evaluation:

              precision    recall  f1-score   support

           0       0.79      0.77      0.78     45184
           1       0.78      0.80      0.79     45088

    accuracy                           0.78     90272
   macro avg       0.78      0.78      0.78     90272
weighted avg       0.78      0.78      0.78     90272


Logistic Regression confusion matrix:

[[34905 10279]
 [ 9198 35890]]
```

The figure above shows the result of the Logistic Regression based on the validation data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 80%, and 79% respectively. The confusion matrix based on class 1 is TP consists of 35890 predictions, FP consists of 10279 predictions, TN consists of 34905 predictions, and FN consists of 9198 predictions.

```
# Classification Report for LR with test data
print("\nLogistic Regression Evaluation:\n")
print(classification_report(y_test, y_pred_logReg))

# Confusion Matrix for LR with test data
print("\nLogistic Regression confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_logReg)
print(cm)
```

After that, we print out the classification report and confusion matrix for Logistic Regression based on the testing data.

Logistic Regression Evaluation:

	precision	recall	f1-score	support
0	0.79	0.77	0.78	45180
1	0.78	0.79	0.78	45092
accuracy			0.78	90272
macro avg	0.78	0.78	0.78	90272
weighted avg	0.78	0.78	0.78	90272

Logistic Regression confusion matrix:

```
[[34811 10369]
 [ 9295 35797]]
```

The figure above shows the result of the Logistic Regression based on the testing data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 79%, and 78% respectively. The confusion matrix based on class 1 is TP consists of 35797 predictions, FP consists of 10369 predictions, TN consists of 34811 predictions, and FN consists of 9295 predictions.

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'penalty': ['l1', 'l2']}
grid_search = GridSearchCV(LogisticRegression(random_state=0, solver='liblinear'), param_grid, cv=5)
grid_search.fit(X_train_new, y_train)

print("Best Parameters: ", grid_search.best_params_)
```

Then, we tune the hyperparameters of the Logistic Regression model by using the GridSearchCV method. The grid of hyperparameters that will be defined are the regularization parameter C and

the penalty type penalty. Next, we use GridSearchCV which will perform an exhaustive search over the hyperparameter grid defined by param_grid using cross-validation. cv=5 specifies 5-fold cross-validation. Then, we fit the GridSearchCV object to the training data (X_train_new and y_train). This step searches for the best combination of hyperparameters that yields the highest cross-validated accuracy. Lastly, we print out the best combination of hyperparameters found by the GridSearchCV object.

```
Best Parameters:  {'C': 1, 'penalty': 'l2'}
```

The figure above shows that the best combination of hyperparameters are regularization type C with the value of 1 and penalty type with the value of 12.

```
y_pred_logReg_hyper = grid_search.predict(X_test)
y_pred_logReg_hyper_val = grid_search.predict(X_val)
```

Then, we evaluate the model that is being tuned with validation and testing data.

```
# Classification Report for LR with validation data
print("\nLogistic Regression Evaluation:\n")
print(classification_report(y_val, y_pred_logReg_hyper_val))

# Confusion Matrix for LR with validation data
print("\nLogistic Regression confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_logReg_hyper_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Logistic Regression after tuning the hyperparameters based on the validation data.

```
Logistic Regression Evaluation:

              precision    recall  f1-score   support

           0       0.79      0.77      0.78     45184
           1       0.78      0.80      0.79     45088

    accuracy                           0.78     90272
   macro avg       0.78      0.78      0.78     90272
weighted avg       0.78      0.78      0.78     90272


Logistic Regression confusion matrix:

[[34905 10279]
 [ 9198 35890]]
```

The figure above shows the result of the Logistic Regression after tuning the hyperparameters based on the validation data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 80%, and 79% respectively. The confusion matrix based on class 1 is TP consists of 35890 predictions, FP consists of 10279 predictions, TN consists of 34905 predictions, and FN consists of 9198 predictions.

```
# Classification Report for LR with test data
print("\nLogistic Regression Evaluation:\n")
print(classification_report(y_test, y_pred_logReg_hyper))

# Confusion Matrix for LR with test data
print("\nLogistic Regression confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_logReg_hyper)
print(cm)
```

Next, we print out the classification report and confusion matrix for Logistic Regression after tuning the hyperparameters based on the testing data.

```
Logistic Regression Evaluation:
```

	precision	recall	f1-score	support
0	0.79	0.77	0.78	45180
1	0.78	0.79	0.78	45092
accuracy			0.78	90272
macro avg	0.78	0.78	0.78	90272
weighted avg	0.78	0.78	0.78	90272

```
Logistic Regression confusion matrix:

[[34811 10369]
 [ 9295 35797]]
```

The figure above shows the result of the Logistic Regression after tuning the hyperparameters based on the testing data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 79%, and 78% respectively. The confusion matrix based on class 1 is TP consists of 35797 predictions, FP consists of 10369 predictions, TN consists of 34811 predictions, and FN consists of 9295 predictions.

3.2 Decision Trees

```
dt = DecisionTreeClassifier(criterion='entropy', max_depth=5)
dt = dt.fit(X_train_new, y_train)
y_pred_dt = dt.predict(X_test)
y_pred_dt_val = dt.predict(X_val)
```

Next, we build a model based on the Decision Trees. The parameters that we have set in here are the criterion which is the function to measure the quality of a split, and 'max_depth' which is the maximum depth of the tree. We have set the 'criterion' to 'entropy' and 'max_depth' to 5. Then, we fit this model into our training dataset before evaluating it using the testing and validation dataset.

```
# Classification Report for Decision Trees with validation data
print("\nDecision Trees Evaluation:\n")
print(classification_report(y_val, y_pred_dt_val))

# Confusion Matrix for Decision Trees with validation data
print("\nDecision Trees confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_dt_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Decision Trees based on the validation data.

```
Decision Trees Evaluation:

              precision    recall  f1-score   support

           0       0.76      0.72      0.74     45184
           1       0.73      0.77      0.75     45088

    accuracy                           0.74     90272
   macro avg       0.74      0.74      0.74     90272
weighted avg       0.74      0.74      0.74     90272


Decision Trees confusion matrix:

[[32457 12727]
 [10443 34645]]
```

The figure above shows the result of the Decision Trees based on the validation data. We can see that the accuracy is 74%, while the precision, recall, and f1-score based on class 1 is 73%, 77%, and 75% respectively. The confusion matrix based on class 1 is TP consists of 34645 predictions, FP consists of 12727 predictions, TN consists of 32457 predictions, and FN consists of 10443 predictions.

```
# Classification Report for Decision Trees with testing data
print("\nDecision Trees Evaluation:\n")
print(classification_report(y_test, y_pred_dt))

# Confusion Matrix for Decision Trees with validation data
print("\nDecision Trees confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_dt)
print(cm)
```

Next, we print out the classification report and confusion matrix for Decision Trees based on the testing data.

```
Decision Trees Evaluation:

              precision    recall  f1-score   support

           0       0.75      0.72      0.74     45180
           1       0.73      0.76      0.75     45092

    accuracy                           0.74     90272
   macro avg       0.74      0.74      0.74     90272
weighted avg       0.74      0.74      0.74     90272


Decision Trees confusion matrix:

[[32524 12656]
 [10597 34495]]
```

The figure above shows the result of the Decision Trees based on the testing data. We can see that the accuracy is 74%, while the precision, recall, and f1-score based on class 1 is 73%, 76%, and 75% respectively. The confusion matrix based on class 1 is TP consists of 34495 predictions, FP consists of 12656 predictions, TN consists of 32524 predictions, and FN consists of 10597 predictions.

```
# Define the parameter grid to search
param_grid = {
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize DecisionTreeClassifier
dt = DecisionTreeClassifier()

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=dt, param_grid=param_grid, cv=5)

# Fit the GridSearchCV to the training data
grid_search.fit(X_train_new, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Retrieve the best model
best_dt = grid_search.best_estimator_
```

After that, we try to tune the hyperparameters of the Decision Trees model with the method of GridSearchCV. The parameters that we have chosen to tune are the 'min_samples_split' which is the minimum number of samples required to split an internal node and the 'min_samples_leaf' which is the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

Next, we use GridSearchCV which will perform an exhaustive search over the hyperparameter grid defined by param_grid using cross-validation. cv=5 specifies 5-fold cross-validation. Then, we fit the GridSearchCV object to the training data (X_train_new and y_train). This step searches for the best combination of hyperparameters that yields the highest cross-validated accuracy. Lastly, we print out the best combination of hyperparameters found by the GridSearchCV object and retrieve the best model based on the best parameters which will be evaluated with the testing and validation dataset.

```
Best Parameters: {'min_samples_leaf': 4, 'min_samples_split': 5}
```

The figure above shows the best parameters received from the two hyperparameters: the ‘min_samples_leaf’ is 4, and the ‘min_samples_split’ is 5.

```
y_pred_dt_hyper = best_dt.predict(X_test)
y_pred_dt_hyper_val = best_dt.predict(X_val)
```

After that, we evaluate the best model retrieved from the above code with our testing and validation dataset.

```
# Classification Report for Decision Trees with validation data
print("\nDecision Trees Evaluation:\n")
print(classification_report(y_val, y_pred_dt_hyper_val))

# Confusion Matrix for Decision Trees with validation data
print("\nDecision Trees confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_dt_hyper_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Decision Trees after tuning the hyperparameters based on the validation data.

```
Decision Trees Evaluation:

              precision    recall  f1-score   support

           0       0.81      0.83      0.82     45184
           1       0.82      0.80      0.81     45088

    accuracy                           0.82     90272
   macro avg       0.82      0.82      0.82     90272
weighted avg       0.82      0.82      0.82     90272


Decision Trees confusion matrix:

[[37432  7752]
 [ 8875 36213]]
```

The figure above shows the result of the Decision Trees after tuning the hyperparameters based on the validation data. We can see that the accuracy is 82%, while the precision, recall, and f1-score based on class 1 is 82%, 80%, and 81% respectively. The confusion matrix based on class 1 is TP consists of 36213 predictions, FP consists of 7752 predictions, TN consists of 37432 predictions, and FN consists of 8875 predictions.

```
# Classification Report for Decision Trees with validation data
print("\nDecision Trees Evaluation:\n")
print(classification_report(y_test, y_pred_dt_hyper))

# Confusion Matrix for Decision Trees with validation data
print("\nDecision Trees confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_dt_hyper)
print(cm)
```

Next, we print out the classification report and confusion matrix for Decision Trees after tuning the hyperparameters based on the testing data.

```
Decision Trees Evaluation:

              precision    recall  f1-score   support

           0       0.80      0.83      0.82     45180
           1       0.82      0.80      0.81     45092

    accuracy                           0.81     90272
   macro avg       0.81      0.81      0.81     90272
weighted avg       0.81      0.81      0.81     90272


Decision Trees confusion matrix:

[[37304  7876]
 [ 9043 36049]]
```

The figure above shows the result of the Decision Trees after tuning the hyperparameters based on the testing data. We can see that the accuracy is 81%, while the precision, recall, and f1-score based on class 1 is 82%, 80%, and 81% respectively. The confusion matrix based on class 1 is TP consists of 36049 predictions, FP consists of 7876 predictions, TN consists of 37304 predictions, and FN consists of 9043 predictions.

3.3 Random Forest Classifier

```
rf = RandomForestClassifier()
rf.fit(X_train_new,y_train)
y_pred_rf = rf.predict(X_test)
y_pred_rf_val = rf.predict(X_val)
```

The third model is Random Forest Classifier which is using the RandomForestClassifier class from scikit-learn. The LogisticRegression class is used to create a logistic regression model with a fixed random seed for reproducibility, while the RandomForestClassifier class is used to create a random forest classifier with default parameters.

```
# Classification Report for Random Forest Classifier with validation data
print("\nRandom Forest Classifier Evaluation:\n")
print(classification_report(y_val, y_pred_rf_val))

# Confusion Matrix for Random Forest Classifier with validation data
print("\nRandom Forest Classifier confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_rf_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Random Forest Classifier based on the validation data.

```
Random Forest Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.86      0.90      0.88     45184
           1       0.89      0.85      0.87     45088

    accuracy                           0.87     90272
   macro avg       0.87      0.87      0.87     90272
weighted avg       0.87      0.87      0.87     90272


Random Forest Classifier confusion matrix:

[[40656  4528]
 [ 6858 38230]]
```

The figure above shows the result of the Random Forest Classifier based on the validation data. We can see that the accuracy is 87%, while the precision, recall, and f1-score based on class 1 is 89%, 85%, and 87% respectively. The confusion matrix based on class 1 is TP consists of 38230 predictions, FP consists of 4528 predictions, TN consists of 40656 predictions, and FN consists of 6858 predictions.

```
# Classification Report for Random Forest Classifier with testing data
print("\nRandom Forest Classifier Evaluation:\n")
print(classification_report(y_test, y_pred_rf))

# Confusion Matrix for Random Forest Classifier with testing data
print("\nRandom Forest Classifier confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_rf)
print(cm)
```

Next, we print out the classification report and confusion matrix for Random Forest Classifier based on the testing data.

Random Forest Classifier Evaluation:

	precision	recall	f1-score	support
0	0.85	0.90	0.88	45180
1	0.89	0.85	0.87	45092
accuracy			0.87	90272
macro avg	0.87	0.87	0.87	90272
weighted avg	0.87	0.87	0.87	90272

Random Forest Classifier confusion matrix:

```
[[40642  4538]
 [ 6960 38132]]
```

The figure above shows the result of the Random Forest Classifier based on the testing data. We can see that the accuracy is 87%, while the precision, recall, and f1-score based on class 1 is 89%, 85%, and 87% respectively. The confusion matrix based on class 1 is TP consists of 38132 predictions, FP consists of 4538 predictions, TN consists of 40642 predictions, and FN consists of 6960 predictions.

```
rf=RandomForestClassifier()
grid_space={
              'min_samples_leaf':[1,2,3],
              'min_samples_split':[1,2,3]
           }

grid = GridSearchCV(rf,param_grid=grid_space,cv=3,scoring='accuracy')
model_grid = grid.fit(X_train_new, y_train)

# Get the best parameters
best_params = model_grid.best_params_
print("Best Parameters:", best_params)

# Retrieve the best model
best_rf = model_grid.best_estimator_
```

This code fine-tunes a Random Forest Classifier by searching for the best hyperparameters using GridSearchCV. It defines a grid of possible values for two key parameters: min_samples_leaf which is minimum samples per leaf node and min_samples_split which is minimum samples required for a split. GridSearchCV then trains and evaluates the model with various combinations from this grid using 3-fold cross-validation. Finally, the code outputs the best hyperparameter set that maximizes accuracy and extracts the corresponding best-performing Random Forest model.

```
Best Parameters: {'min_samples_leaf': 1, 'min_samples_split': 2}
```

The figure above shows the best parameters received from the two hyperparameters: the ‘min_samples_leaf’ is 1, and the ‘min_samples_split’ is 2.

```
y_pred_rf_hyper = best_rf.predict(X_test)
y_pred_rf_hyper_val = best_rf.predict(X_val)
```

After that, we evaluate the best model retrieved from the above code with our testing and validation dataset.

```
# Classification Report for Random Forest Classifier with validation data
print("\nRandom Forest Classifier Evaluation:\n")
print(classification_report(y_val, y_pred_rf_hyper_val))

# Confusion Matrix for Random Forest Classifier with validation data
print("\nRandom Forest Classifier confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_rf_hyper_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Random Forest Classifier after tuning the hyperparameters based on the validation data.

```
Random Forest Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.85      0.90      0.88     45184
           1       0.89      0.85      0.87     45088

    accuracy                           0.87     90272
   macro avg       0.87      0.87      0.87     90272
weighted avg       0.87      0.87      0.87     90272


Random Forest Classifier confusion matrix:

[[40594  4590]
 [ 6900 38188]]
```

The figure above shows the result of the Random Forest Classifier after tuning the hyperparameters based on the validation data. We can see that the accuracy is 87%, while the precision, recall, and f1-score based on class 1 is 89%, 85%, and 87% respectively. The

confusion matrix based on class 1 is TP consists of 38188 predictions, FP consists of 4590 predictions, TN consists of 40594 predictions, and FN consists of 6900 predictions.

```
# Classification Report for Random Forest Classifier with testing data
print("\nRandom Forest Classifier Evaluation:\n")
print(classification_report(y_test, y_pred_rf_hyper))

# Confusion Matrix for Random Forest Classifier with testing data
print("\nRandom Forest Classifier confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_rf_hyper)
print(cm)
```

Next, we print out the classification report and confusion matrix for Random Forest Classifier after tuning the hyperparameters based on the testing data.

```
Random Forest Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.85      0.90      0.88     45180
           1       0.89      0.85      0.87     45092

    accuracy                           0.87     90272
   macro avg       0.87      0.87      0.87     90272
weighted avg       0.87      0.87      0.87     90272


Random Forest Classifier confusion matrix:

[[40622  4558]
 [ 6921 38171]]
```

The figure above shows the result of the Random Forest Classifier after tuning the hyperparameters based on the testing data. We can see that the accuracy is 87%, while the precision, recall, and f1-score based on class 1 is 89%, 85%, and 87% respectively. The confusion matrix based on class 1 is TP consists of 38171 predictions, FP consists of 4558 predictions, TN consists of 40622 predictions, and FN consists of 6921 predictions.

3.4 Naive Bayes

```
nb = GaussianNB()
nb.fit(X_train_new,y_train)

y_pred_nb = nb.predict(X_test)
y_pred_nb_val = nb.predict(X_val)
```

The next model that we build is based on the Naive Bayes model. We have chosen the Gaussian Naive Bayes classifier. The reason why we choose the Gaussian Naive Bayes is because this model is computationally efficient and straightforward to implement. Besides that, it is particularly suitable for large datasets where computational resources are limited or when computational speed is crucial. Besides that, Gaussian Bayes is based on Gaussian, or normal distribution. It can significantly speeds up the search under some non-strict conditions.

```
# Classification Report for Naive Bayes with validation data
print("\nNaive Bayes Classifier Evaluation:\n")
print(classification_report(y_val, y_pred_nb_val))

# Confusion Matrix for Naive Bayes with validation data
print("\nNaive Bayes confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_nb_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Naive Bayes based on the validation data.

```
Naive Bayes Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.81      0.82      0.82     45184
           1       0.82      0.81      0.81     45088

    accuracy                           0.82     90272
   macro avg       0.82      0.82      0.82     90272
weighted avg       0.82      0.82      0.82     90272


Naive Bayes confusion matrix:

[[37243  7941]
 [ 8670 36418]]
```

The figure above shows the result of the Naive Bayes based on the validation data. We can see that the accuracy is 82%, while the precision, recall, and f1-score based on class 1 is 82%, 81%, and 81% respectively. The confusion matrix based on class 1 is TP consists of 36418 predictions, FP consists of 7941 predictions, TN consists of 37243 predictions, and FN consists of 8670 predictions.

```
# Classification Report for Naive Bayes with testing data
print("\nNaive Bayes Classifier Evaluation:\n")
print(classification_report(y_test, y_pred_nb))

# Confusion Matrix for Naive Bayes with testing data
print("\nNaive Bayes confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_nb)
print(cm)
```

Next, we print out the classification report and confusion matrix for Naive Bayes based on the testing data.

```
Naive Bayes Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.81      0.82      0.82     45180
           1       0.82      0.80      0.81     45092

    accuracy                           0.81     90272
   macro avg       0.81      0.81      0.81     90272
weighted avg       0.81      0.81      0.81     90272


Naive Bayes confusion matrix:

[[37141  8039]
 [ 8820 36272]]
```

The figure above shows the result of the Naïve Bayes based on the validation data. We can see that the accuracy is 81%, while the precision, recall, and f1-score based on class 1 is 82%, 80%, and 81% respectively. The confusion matrix based on class 1 is TP consists of 36272 predictions,

FP consists of 8039 predictions, TN consists of 37141 predictions, and FN consists of 8820 predictions.

```
nb = GaussianNB()

params_NB = {'var_smoothing': np.logspace(0,-9, num=100)}
gs_NB = GridSearchCV(estimator=nb,
                     param_grid=params_NB,
                     cv=10,   # use any cross validation technique
                     verbose=1,
                     scoring='accuracy')
gs_NB.fit(X_train_new, y_train)

gs_NB.best_params_
```

```
Fitting 10 folds for each of 100 candidates, totalling 1000 fits
{'var_smoothing': 0.0012328467394420659}
```

After that, we tune the Gaussian Naive Bayes model that we have created by using the GridSearchCV method. Firstly, we set the parameters grid for the grid search. In this case, it's var_smoothing, which is a smoothing parameter used in the Gaussian Naive Bayes model. It's using a logarithmic space from 1 to 1e-9 with 100 values. Next, Instantiate a GridSearchCV object. It takes the Naive Bayes classifier, parameter grid, number of folds for cross-validation, verbosity level, and the scoring metric. Then, we fit this new model to our existing training dataset and print out the best parameters from this model. The result above shows the best smoothing parameter for the Gaussian Naive Bayes model.

```
y_pred_nb_hyper = gs_NB.predict(X_test)
y_pred_nb_hyper_val = gs_NB.predict(X_val)
```

Next, we evaluate this new model with the validation and testing dataset.

```
# Classification Report for Naive Bayes with validation data
print("\nNaive Bayes Classifier Evaluation:\n")
print(classification_report(y_val, y_pred_nb_hyper_val))

# Confusion Matrix for Naive Bayes with validation data
print("\nNaive Bayes confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_nb_hyper_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Naive Bayes after tuning the hyperparameters based on the validation data.

```
Naive Bayes Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.81      0.82      0.82     45184
           1       0.82      0.81      0.81     45088

    accuracy                           0.82     90272
   macro avg       0.82      0.82      0.82     90272
weighted avg       0.82      0.82      0.82     90272


Naive Bayes confusion matrix:

[[37243  7941]
 [ 8670 36418]]
```

The figure above shows the result of the Naïve Bayes after tuning the hyperparameters based on the validation data. We can see that the accuracy is 82%, while the precision, recall, and f1-score based on class 1 is 82%, 81%, and 81% respectively. The confusion matrix based on class 1 is

TP consists of 36418 predictions, FP consists of 7941 predictions, TN consists of 37243 predictions, and FN consists of 8670 predictions.

```
# Classification Report for Naive Bayes with testing data
print("\nNaive Bayes Classifier Evaluation:\n")
print(classification_report(y_test, y_pred_nb_hyper))

# Confusion Matrix for Naive Bayes with testing data
print("\nNaive Bayes confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_nb_hyper)
print(cm)
```

Next, we print out the classification report and confusion matrix for Naive Bayes after tuning the hyperparameters based on the testing data.

```
Naive Bayes Classifier Evaluation:

              precision    recall  f1-score   support

           0       0.81      0.82      0.82     45180
           1       0.82      0.80      0.81     45092

    accuracy                           0.81     90272
   macro avg       0.81      0.81      0.81     90272
weighted avg       0.81      0.81      0.81     90272


Naive Bayes confusion matrix:

[[37141  8039]
 [ 8820 36272]]
```

The figure above shows the result of the Naïve Bayes after tuning the hyperparameters based on the testing data. We can see that the accuracy is 81%, while the precision, recall, and f1-score

based on class 1 is 82%, 80%, and 81% respectively. The confusion matrix based on class 1 is TP consists of 36272 predictions, FP consists of 8039 predictions, TN consists of 37141 predictions, and FN consists of 8820 predictions.

3.5 Neural Network

```
mlp = MLPClassifier()
mlp.fit(X_train_new,y_train)

y_pred_mlp = mlp.predict(X_test)
y_pred_mlp_val = mlp.predict(X_val)
```

The fifth model that we have created is the Neural Network. Neural Network was created by using the MLPClassifier which is the default Neural Network model in sklearn.neural_network library.

```
# Classification Report for Neural Network with validation data
print("\nNeural Network Evaluation:\n")
print(classification_report(y_val, y_pred_mlp_val))

# Confusion Matrix for Neural Network with validation data
print("\nNeural Network confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_mlp_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Neural Network based on the validation data.

Neural Network Evaluation:

	precision	recall	f1-score	support
0	0.83	0.83	0.83	45184
1	0.83	0.83	0.83	45088
accuracy			0.83	90272
macro avg	0.83	0.83	0.83	90272
weighted avg	0.83	0.83	0.83	90272

Neural Network confusion matrix:

```
[[37380  7804]
 [ 7829 37259]]
```

The figure above shows the result of the Neural Network based on the validation data. We can see that the accuracy is 83%, while the precision, recall, and f1-score based on class 1 is 83%, 83%, and 83% respectively. The confusion matrix based on class 1 is TP consists of 37259 predictions, FP consists of 7804 predictions, TN consists of 37380 predictions, and FN consists of 7829 predictions.

```
# Classification Report for Neural Network with testing data
print("\nNeural Network Evaluation:\n")
print(classification_report(y_test, y_pred_mlp))

# Confusion Matrix for Neural Network with testing data
print("\nNeural Network confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_mlp)
print(cm)
```

Next, we print out the classification report and confusion matrix for Neural Network based on the testing data.

```
Neural Network Evaluation:

              precision    recall  f1-score   support

           0       0.83      0.83      0.83     45180
           1       0.83      0.83      0.83     45092

    accuracy                           0.83     90272
   macro avg       0.83      0.83      0.83     90272
weighted avg       0.83      0.83      0.83     90272


Neural Network confusion matrix:

[[37372  7808]
 [ 7878 37214]]
```

The figure above shows the result of the Neural Network based on the testing data. We can see that the accuracy is 83%, while the precision, recall, and f1-score based on class 1 is 83%, 83%, and 83% respectively. The confusion matrix based on class 1 is TP consists of 37214 predictions, FP consists of 7808 predictions, TN consists of 37372 predictions, and FN consists of 7878 predictions.

```
mlp = MLPClassifier()
tuned_parameters= {'hidden_layer_sizes': range(1,200,10),
                   'activation': ['tanh','logistic','relu'], 'alpha':[0.0001,0.001,0.01,0.1,1,10],
                   'max_iter': range(50,200,50)}

model_mlp= RandomizedSearchCV(mlp, tuned_parameters,cv=3,scoring='accuracy',n_iter=3,n_jobs= -1,random_state=5)
model_mlp.fit(X_train_new, y_train)

model_mlp.best_params_
```

After evaluating the default Neural Network model, we tune the hyperparameters by using RandomizedSearchCV method. The parameters that we have chosen to tune include the 'hidden_layer_sizes' which is the number of hidden layers in the network, 'activation' which is the activation function for the hidden layer, 'alpha' which is the L2 penalty or regularization term parameter and 'max_iter' which is the maximum number of iterations. Then, we apply the RandomizedSearchCV which include the parameters such as model used, parameters tuned, cross-validation splitting strategy used (cv), strategy to evaluate the performance of the cross-validated model on the test set (scoring), number of parameter settings that are sampled. (n_iter), number of jobs to run in parallel (n_jobs), and random_state. The RandomizedSearchCV will randomly select parameters and test it with the model to find the best solution. After finding the optimal solution, we fit the model with our training dataset, and print the best parameters used for the model.

```
{'max_iter': 150, 'hidden_layer_sizes': 81, 'alpha': 0.1, 'activation': 'tanh'}
```

The figure above shows the best parameters retrieved from the Neural Network model that has been tuned.

```
y_pred_mlp_hyper = model_mlp.predict(X_test)
y_pred_mlp_hyper_val = model_mlp.predict(X_val)
```

Then, we evaluate the model based on the testing and validation dataset.

```
# Classification Report for Neural Network with validation data
print("\nNeural Network Evaluation:\n")
print(classification_report(y_val, y_pred_mlp_hyper_val))

# Confusion Matrix for Neural Network with validation data
print("\nNeural Network confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_mlp_hyper_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Neural Network after tuning the hyperparameters based on the validation data.

```
Neural Network Evaluation:

              precision    recall  f1-score   support

           0       0.82      0.84      0.83     45184
           1       0.83      0.82      0.83     45088

    accuracy                           0.83     90272
   macro avg       0.83      0.83      0.83     90272
weighted avg       0.83      0.83      0.83     90272


Neural Network confusion matrix:

[[37784  7400]
 [ 8215 36873]]
```

The figure above shows the result of the Neural Network after tuning the hyperparameters based on the validation data. We can see that the accuracy is 83%, while the precision, recall, and f1-score based on class 1 is 83%, 82%, and 83% respectively. The confusion matrix based on class 1 is TP consists of 36873 predictions, FP consists of 7400 predictions, TN consists of 37784 predictions, and FN consists of 8215 predictions.

```
# Classification Report for Neural Network with testing data
print("\nNeural Network Evaluation:\n")
print(classification_report(y_test, y_pred_mlp_hyper))

# Confusion Matrix for Neural Network with testing data
print("\nNeural Network confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_mlp_hyper)
print(cm)
```

Next, we print out the classification report and confusion matrix for Neural Network after tuning the hyperparameters based on the testing data.

```
Neural Network Evaluation:

              precision    recall  f1-score   support

           0       0.82      0.84      0.83     45180
           1       0.83      0.82      0.82     45092

    accuracy                           0.83     90272
   macro avg       0.83      0.83      0.83     90272
weighted avg       0.83      0.83      0.83     90272


Neural Network confusion matrix:

[[37766  7414]
 [ 8290 36802]]
```

The figure above shows the result of the Neural Network after tuning the hyperparameters based on the testing data. We can see that the accuracy is 83%, while the precision, recall, and f1-score based on class 1 is 83%, 82%, and 82% respectively. The confusion matrix based on class 1 is

TP consists of 36802 predictions, FP consists of 7414 predictions, TN consists of 37766 predictions, and FN consists of 8290 predictions.

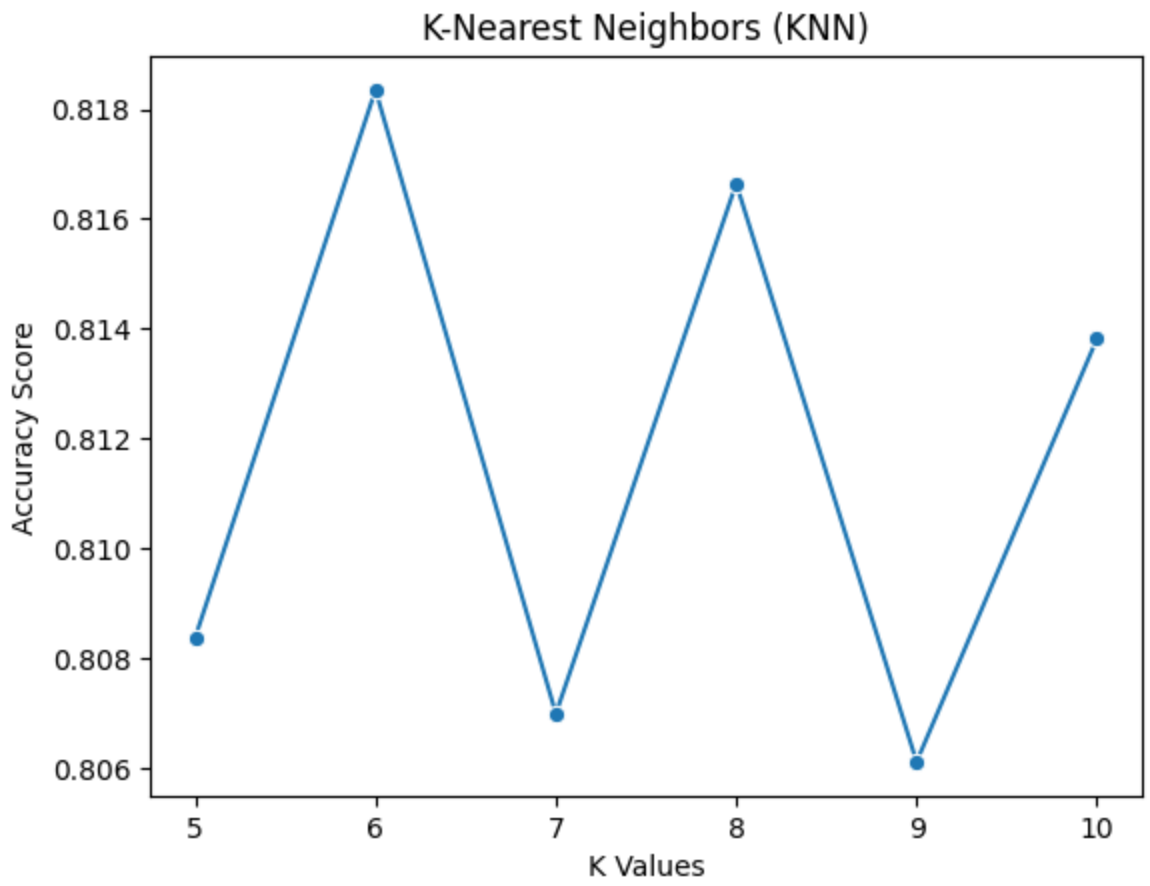
3.6 K-Nearest Neighbour (KNN)

```
# find the accuracy and errors rate for k starts from 5 to 11
k_values = [i for i in range (5,11)]
scores = []
error_rate = []
for k in k_values:
 knn = KNeighborsClassifier(n_neighbors=k)
 knn.fit(X_train_new, y_train)
 y_pred = knn.predict(X_test)
 scores.append(knn.score(X_test, y_test))
 error_rate.append(np.mean(y_pred!=y_test))
```

The sixth model that we build is based on the K-Nearest Neighbors. The code snippet above finds the accuracy rate and errors rate for k starts from 5 to 11 for our training dataset.

```
# Plot the accuracy score for each k-values
sns.lineplot(x = k_values, y = scores, marker = 'o')
plt.title("K-Nearest Neighbors (KNN)")
plt.xlabel("K Values")
plt.ylabel("Accuracy Score")
plt.show()
```

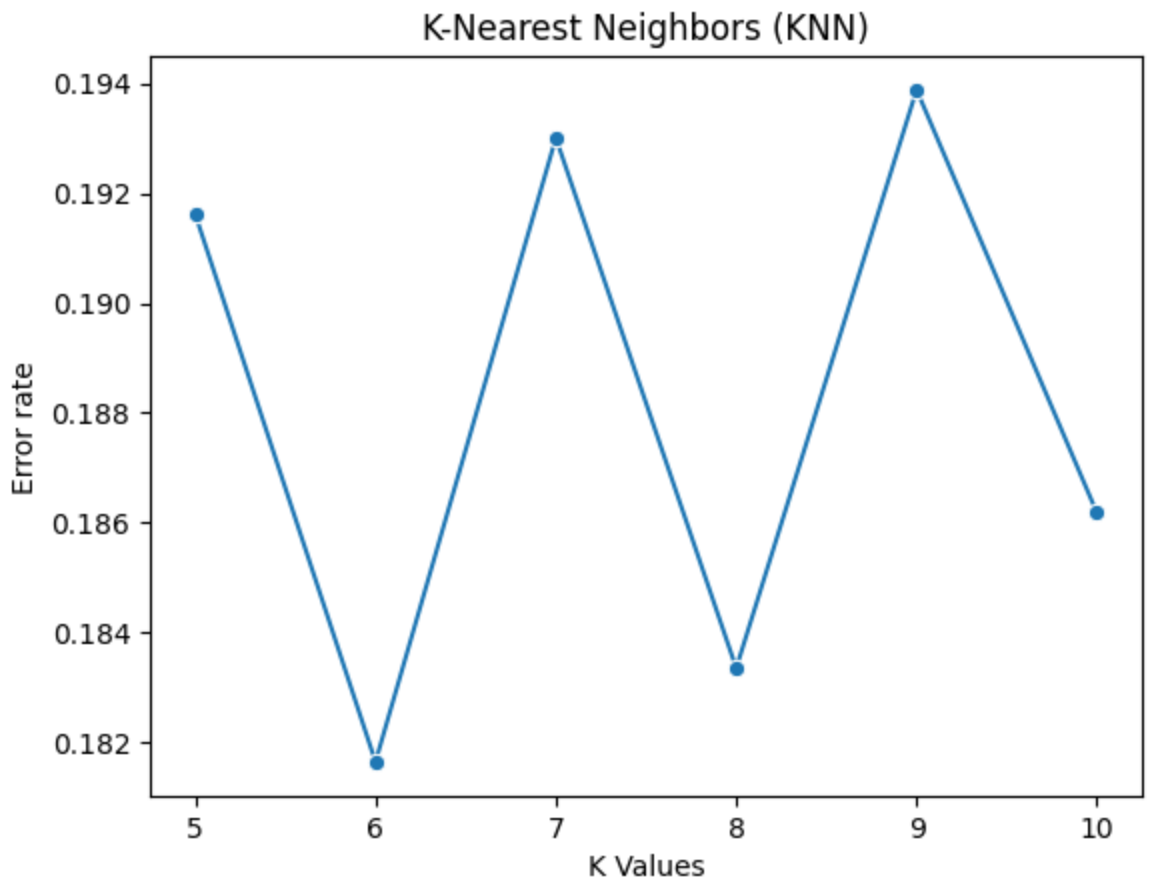
Then, we plot the accuracy score by using the line plot from the seaborn library.



The figure above shows the line plot for the accuracy scores of the K-Nearest Neighbors (KNN) model. We can see that the accuracy score when k equals to 6 is the highest, while the accuracy score when k equals to 9 is the lowest.

```
# Plot the error rate for each k-values
sns.lineplot(x = k_values, y = error_rate, marker = 'o')
plt.title("K-Nearest Neighbors (KNN)")
plt.xlabel("K Values")
plt.ylabel("Error rate")
plt.show()
```

Then, we use the line plot to plot from the seaborn library to plot the error rate for each of the k-values.



The figure above shows the line plot based on the error rate of the k-values. We can see that when k-6, the error rate is the lowest, and when k-9, the error rate is the highest.

```
#Get the best k and train kNN
best_index = np.argmax(scores)
best_k = k_values[best_index]
print("\nk = ", best_k)
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(X_train_new, y_train)
y_pred_knn = knn.predict(X_test)
y_pred_knn_val = knn.predict(X_val)
```

After that, we print out the best k-value and use this value as a parameter to train our training dataset and evaluate it using the validation and testing dataset.

```
k =  6
```

From this figure, we can see that the best k-value is 6.

```
# Classification Report for K-Nearest Neighbour with validation data
print("\nK-Nearest Neighbour Evaluation:\n")
print(classification_report(y_val, y_pred_knn_val))

# Confusion Matrix for K-Nearest Neighbour with validation data
print("\nK-Nearest Neighbour confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_knn_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for K-Nearest Neighbour based on the validation data.

```
K-Nearest Neighbour Evaluation:

              precision    recall  f1-score   support

           0       0.85      0.78      0.81     45184
           1       0.79      0.87      0.83     45088

    accuracy                           0.82     90272
   macro avg       0.82      0.82      0.82     90272
weighted avg       0.82      0.82      0.82     90272


K-Nearest Neighbour confusion matrix:

[[35020 10164]
 [ 6029 39059]]
```

The figure above shows the result of the K-Nearest Neighbour based on the validation data. We can see that the accuracy is 82%, while the precision, recall, and f1-score based on class 1 is 79%, 87%, and 83% respectively. The confusion matrix based on class 1 is TP consists of 39059 predictions, FP consists of 10164 predictions, TN consists of 35020 predictions, and FN consists of 6029 predictions.

```
# Classification Report for K-Nearest Neighbour with testing data
print("\nK-Nearest Neighbour Evaluation:\n")
print(classification_report(y_test, y_pred_knn))

# Confusion Matrix for K-Nearest Neighbour with testing data
print("\nK-Nearest Neighbour confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_knn)
print(cm)
```

Next, we print out the classification report and confusion matrix for K-Nearest Neighbour based on the testing data.

K-Nearest Neighbour Evaluation:

	precision	recall	f1-score	support
0	0.85	0.78	0.81	45180
1	0.79	0.86	0.83	45092
accuracy			0.82	90272
macro avg	0.82	0.82	0.82	90272
weighted avg	0.82	0.82	0.82	90272

K-Nearest Neighbour confusion matrix:

```
[[35078 10102]
 [ 6261 38831]]
```

The figure above shows the result of the K-Nearest Neighbour based on the testing data. We can see that the accuracy is 82%, while the precision, recall, and f1-score based on class 1 is 79%, 86%, and 83% respectively. The confusion matrix based on class 1 is TP consists of 38831 predictions, FP consists of 10102 predictions, TN consists of 35078 predictions, and FN consists of 6261 predictions.

The reason why we do not tune the hyperparameter for the K-Nearest Neighbour model is because the time needed to run the K-Nearest Neighbors model without tuning the parameters is around 30 to 40 minutes. So, we think that the time to run the K-Nearest Neighbour will be longer than 30 to 40 minutes. We think this is not practical for the loan application in the bank because the model should be able to train with our training dataset quickly. If the time is too long, that means that the loan application will take a very long time to predict whether it should be approved or not.

3.7 Support Vector Machine (SVM)

```
svm = CalibratedClassifierCV(LinearSVC())
svm.fit(X_train_new, y_train)
y_pred_svm = svm.predict(X_test)
y_pred_svm_val = svm.predict(X_val)
```

The last model is built based on the Support Vector Machine (SVM) model with probability calibration. 'CalibratedClassifierCV' helps in calibrating the probabilities predicted by the SVM model. The SVM classifier object was created using the LinearSVC class from scikit-learn.

```
# Classification Report for Support Vector Machine with validation data
print("\nSupport Vector Machine Evaluation:\n")
print(classification_report(y_val, y_pred_svm_val))

# Confusion Matrix for Support Vector Machine with validation data
print("\nSupport Vector Machine confusion matrix:\n")
cm = confusion_matrix(y_val, y_pred_svm_val)
print(cm)
```

Next, we print out the classification report and confusion matrix for Support Vector Machine based on the validation data.

```
Support Vector Machine Evaluation:

              precision    recall  f1-score   support

           0       0.79      0.77      0.78     45184
           1       0.78      0.80      0.79     45088

    accuracy                           0.78     90272
   macro avg       0.78      0.78      0.78     90272
weighted avg       0.78      0.78      0.78     90272


Support Vector Machine confusion matrix:

[[34914 10270]
 [ 9236 35852]]
```

The figure above shows the result of the Support Vector Machine based on the validation data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 80%, and 79% respectively. The confusion matrix based on class 1 is TP consists of 35852 predictions, FP consists of 10270 predictions, TN consists of 34914 predictions, and FN consists of 9236 predictions.

```
# Classification Report for Support Vector Machine with testing data
print("\nSupport Vector Machine Evaluation:\n")
print(classification_report(y_test, y_pred_svm))

# Confusion Matrix for Support Vector Machine with testing data
print("\nSupport Vector Machine confusion matrix:\n")
cm = confusion_matrix(y_test, y_pred_svm)
print(cm)
```

Next, we print out the classification report and confusion matrix for Support Vector Machine based on the testing data.

```
Support Vector Machine Evaluation:

              precision    recall  f1-score   support

           0       0.79      0.77      0.78     45180
           1       0.78      0.80      0.79     45092

    accuracy                           0.78     90272
   macro avg       0.78      0.78      0.78     90272
weighted avg       0.78      0.78      0.78     90272


Support Vector Machine confusion matrix:

[[34811 10369]
 [ 9215 35877]]
```

The figure above shows the result of the Support Vector Machine based on the data. We can see that the accuracy is 78%, while the precision, recall, and f1-score based on class 1 is 78%, 80%, and 79% respectively. The confusion matrix based on class 1 is TP consists of 35877 predictions, FP consists of 10369 predictions, TN consists of 34811 predictions, and FN consists of 9215 predictions.

3.8 Comparison and Selection

```
auc_score_logReg = roc_auc_score(y_test, logReg.predict_proba(X_test)[:, 1])
auc_score_dt = roc_auc_score(y_test, grid_search.predict_proba(X_test)[:, 1])
auc_score_rf = roc_auc_score(y_test, model_grid.predict_proba(X_test)[:, 1])
auc_score_nb = roc_auc_score(y_test, gs_NB.predict_proba(X_test)[:, 1])
auc_score_mlp = roc_auc_score(y_test, model_mlp.predict_proba(X_test)[:, 1])
auc_score_knn = roc_auc_score(y_test, knn.predict_proba(X_test)[:, 1])
auc_score_svm = roc_auc_score(y_test, svm.predict_proba(X_test)[:, 1])
print("AUC score for LR: ", auc_score_logReg)
print("AUC score for Decision Trees: ", auc_score_dt)
print("AUC score for Random Forest Classifier: ", auc_score_rf)
print("AUC score for Naive Bayes: ", auc_score_nb)
print("AUC score for Neural Network: ", auc_score_mlp)
print("AUC score for K-Nearest Neighbour: ", auc_score_knn)
print("AUC score for Support Vector Machine: ", auc_score_svm)
```

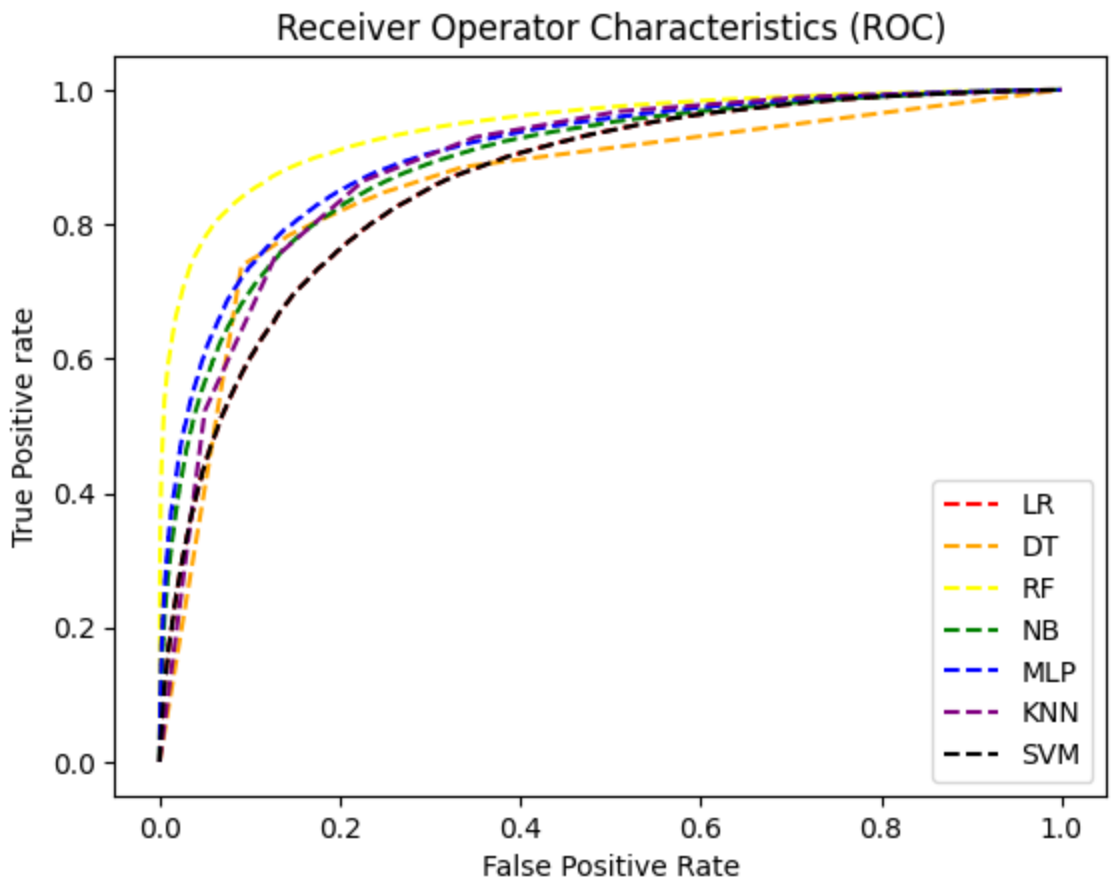
The next step that we do is to compare each model's performance based on the area under ROC curve (AUC) using the predicted probabilities on the test data. After we have found AUC score for each of the models, we will print out the result.

```
AUC score for LR:  0.8615074053314129
AUC score for Decision Trees:  0.8610499896979102
AUC score for Random Forest Classifier:  0.9433116038168506
AUC score for Naive Bayes:  0.8918466705047694
AUC score for Neural Network:  0.9044224822621261
AUC score for K-Nearest Neighbour:  0.8890793153219739
AUC score for Support Vector Machine:  0.8615083055616716
```

The figure above shows the result for the AUC score for each of the models. We can see that the result for the Random Forest Classifier model is the highest, which is 0.9433, followed by the Neural Network model, and Naive Bayes model respectively.

```
fpr_LR, tpr_LR, thresholds_LR = roc_curve(y_test, logReg.predict_proba(X_test)[:,1])
fpr_DT, tpr_DT, thresholds_DT = roc_curve(y_test, grid_search.predict_proba(X_test)[:,1])
fpr_RF, tpr_RF, thresholds_RF = roc_curve(y_test, model_grid.predict_proba(X_test)[:,1])
fpr_NB, tpr_NB, thresholds_NB = roc_curve(y_test, gs_NB.predict_proba(X_test)[:,1])
fpr_MLP, tpr_MLP, thresholds_MLP = roc_curve(y_test, model_mlp.predict_proba(X_test)[:,1])
fpr_KNN, tpr_KNN, thresholds_KNN = roc_curve(y_test, knn.predict_proba(X_test)[:,1])
fpr_SVM, tpr_SVM, thresholds_SVM = roc_curve(y_test, svm.predict_proba(X_test)[:,1])
plt.plot(fpr_LR, tpr_LR, linestyle = "--", color = "red", label = "LR")
plt.plot(fpr_DT, tpr_DT, linestyle = "--", color = "orange", label = "DT")
plt.plot(fpr_RF, tpr_RF, linestyle = "--", color = "yellow", label = "RF")
plt.plot(fpr_NB, tpr_NB, linestyle = "--", color = "green", label = "NB")
plt.plot(fpr_MLP, tpr_MLP, linestyle = "--", color = "blue", label = "MLP")
plt.plot(fpr_KNN, tpr_KNN, linestyle = "--", color = "purple", label = "KNN")
plt.plot(fpr_SVM, tpr_SVM, linestyle = "--", color = "black", label = "SVM")
plt.title('Receiver Operator Characteristics (ROC)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc = 'best')
plt.savefig('ROC', dpi = 300)
plt.show()
```

Next, we will plot the ROC curve between the False Positive Rate and True Positive Rate for the seven different models. Each model will be plotted using a different color.



The figure above shows the ROC curve between False Positive Rate (FPR) and True Positive Rate (TPR). The result shows that the model with the yellow line, which is the Random Forest Classifier model, has the largest area under the curve.

Model	Type	Accuracy	Precision	Recall	F1-score
Logistic Regression	Validation	0.78	0.78	0.80	0.79
	Testing	0.78	0.78	0.79	0.78
Logistic Regression (Tuned)	Validation	0.78	0.78	0.80	0.79
	Testing	0.78	0.78	0.79	0.78
Decision Trees	Validation	0.74	0.73	0.77	0.75
	Testing	0.74	0.73	0.76	0.75
Decision Trees (Tuned)	Validation	0.82	0.82	0.80	0.81
	Testing	0.81	0.82	0.80	0.81
Random Forest	Validation	0.87	0.89	0.85	0.87

	Testing	0.87	0.89	0.85	0.87
Random Forest (Tuned)	Validation	0.87	0.89	0.85	0.87
	Testing	0.87	0.89	0.85	0.87
Naïve Bayes	Validation	0.82	0.82	0.81	0.81
	Testing	0.81	0.82	0.80	0.81
Naïve Bayes (Tuned)	Validation	0.82	0.82	0.81	0.81
	Testing	0.81	0.82	0.80	0.81
Neural Network	Validation	0.83	0.83	0.83	0.83
	Testing	0.83	0.83	0.83	0.83
Neural Network (Tuned)	Validation	0.83	0.83	0.82	0.83
	Testing	0.83	0.83	0.82	0.83
K-Nearest Neighbors	Validation	0.82	0.79	0.87	0.83
	Testing	0.82	0.79	0.86	0.83
Support Vector Machine	Validation	0.78	0.78	0.80	0.79
	Testing	0.78	0.78	0.80	0.79

The table above shows the result of accuracy, precision, recall, and f1-score for each of the model either before tuned and after tuned. We can see that the accuracy, precision, recall and f1-score for Random Forest Classifier before tuning and after tuning the hyperparameters is higher than other different models. So, we decided to choose the Random Forest Classifier as our model for making prediction on different data. Besides that, we choose to use the Random Forest Classifier before tuning as our model before the time to train the Random Forest Classifier model after tuning is very long and the result for these two models is almost the same. Thus, we choose the Random Forest Classifier model as our model to making prediction for the later part.

4.0 Interaction

After identifying the best model to use in our dataset, now we will interact with this model which is the Random Forest Classifier without tuning the hyperparameters to predict the outcome of loan application in another csv file.

4.1 Storing the Model

```
import joblib
```

```
joblib.dump(rf, 'loan_application_predict')
```

Now, we need to store the model that has the best performance result. Firstly, we import the joblib. Joblib is a Python library primarily used for caching and parallelizing computations, particularly in machine learning and data analysis workflows. It is designed to efficiently save Python objects to disk and reload them, which can be particularly useful for tasks like model persistence or saving intermediate results. Some of the key features and functionalities of Joblib include caching, parallelization, serialization, and integration with NumPy arrays: Then, we use the joblib.dump() function to serialize the machine learning model of ('rf') into a file with the name of 'loan_application_predict'. After running this code, a file with the name of 'loan_application_predict.pkl' appears in our working directory, which contains the Random Forest Classifier that we have trained.

4.2 Loading the Model

```
model = joblib.load('loan_application_predict')
```

After that, we use the joblib.load() function to load a serialized object from a file that we have saved just now into a variable 'model'.

4.3 Interacting with the Model

```
new_applicant = pd.read_csv('((GAssign) NewApplicants.csv')
```

Since the loan application that we need to predict is in another file, thus we need to read the csv file that contains the loan application. We use the read_csv() method in the pandas library to read the file name '((GAssign) NewApplicants.csv' and store it in the new_applicant variable.

```
new_applicant.head()
```

Then, we print the first five rows of the data to view the data in the csv file.

	LoanID	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner	Default
0	A01	56	85994	50587	520	80	4	15.23	36	0.44	Bachelor's	Full-time	Divorced	Yes	Yes	Other	Yes	NaN
1	A02	69	50432	124440	458	15	1	4.81	60	0.68	Master's	Full-time	Married	No	No	Other	Yes	NaN
2	A03	46	84208	129188	451	26	3	21.17	24	0.31	Master's	Unemployed	Divorced	Yes	Yes	Auto	No	NaN
3	A04	32	31713	44799	743	0	3	7.07	24	0.23	High School	Full-time	Married	No	No	Business	No	NaN
4	A05	60	20437	9139	633	8	4	6.51	48	0.73	Bachelor's	Unemployed	Divorced	No	Yes	Auto	No	NaN

The figure above shows the result from the 'new_applicant.head()' function. We can see that the data that we need to predict contains the same number of columns with our training dataset.

```
new_applicant.info()
```

We also print the information like data types of each column, the number of non-null values and memory usage.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 18 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   LoanID          20 non-null     object
 1   Age             20 non-null     int64
 2   Income          20 non-null     int64
 3   LoanAmount      20 non-null     int64
 4   CreditScore     20 non-null     int64
 5   MonthsEmployed  20 non-null     int64
 6   NumCreditLines  20 non-null     int64
 7   InterestRate    20 non-null     float64
 8   LoanTerm        20 non-null     int64
 9   DTIRatio        20 non-null     float64
 10  Education       20 non-null     object
 11  EmploymentType  20 non-null     object
 12  MaritalStatus   20 non-null     object
 13  HasMortgage     20 non-null     object
 14  HasDependents   20 non-null     object
 15  LoanPurpose     20 non-null     object
 16  HasCoSigner     20 non-null     object
 17  Default         0 non-null      float64
dtypes: float64(3), int64(7), object(8)
memory usage: 2.9+ KB
```

The figure above shows the result from the 'new_applicant.info()' function. We can see that the data we are using did not contain any missing values.

```
application = new_applicant.drop(['Default'], axis=1)
application_new = application.drop(['LoanID'], axis=1)
```

This code is to create a new dataset called 'application' and remove the 'default' column and 'LoadID' column. Removing these two columns which are not needed is to reduce data size and improve the efficiency of the model.

```
application_new['Education'] = application_new['Education'].map({"Bachelor's": 0, "High School": 1, "Master's": 2, "PhD": 3} ).astype(int)
application_new['EmploymentType'] = application_new['EmploymentType'].map( {'Full-time': 0, 'Part-time': 1,'Unemployed':2, 'Self-employed': 3} ).asty
application_new['MaritalStatus'] = application_new['MaritalStatus'].map( {'Divorced': 0, 'Married': 1, 'Single': 2}).astype(int)
application_new['HasMortgage'] = application_new['HasMortgage'].map( {'No': 0, 'Yes': 1}).astype(int)
application_new['HasDependents'] = application_new['HasDependents'].map( {'No': 0, 'Yes': 1}).astype(int)
application_new['LoanPurpose'] = application_new['LoanPurpose'].map( {'Auto': 0, 'Business': 1, 'Education': 2, 'Home': 3, 'Other': 4}).astype(int)
application_new['HasCoSigner'] = application_new['HasCoSigner'].map( {'No': 0, 'Yes': 1}).astype(int)
application_new.head()
```

Then, we also need to transform the value in the categorical variable into the numerical variable. The categorical columns that we need to transform include ‘Education’, ‘EmploymentType’, ‘MaritalStatus’, ‘HasMortgage’, ‘HasDependents’, ‘LoanPurpose’, and ‘HasCoSigner’ column. After we have transformed the above categorical columns, we need to show the first 5 rows of the dataset to check whether these categorical variables values have been transformed to numeric values.

	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner
0	56	85994	50587	520	80	4	15.23	36	0.44	0	0	0	1	1	4	1
1	69	50432	124440	458	15	1	4.81	60	0.68	2	0	1	0	0	4	1
2	46	84208	129188	451	26	3	21.17	24	0.31	2	2	0	1	1	0	0
3	32	31713	44799	743	0	3	7.07	24	0.23	1	0	1	0	0	1	0
4	60	20437	9139	633	8	4	6.51	48	0.73	0	2	0	0	1	0	0

The figure above shows the result after transforming the categorical variables into numeric values.

```
application_scale = scaler.fit_transform(application_new)
```

This code is to scale the data using scaling parameters such as mean and standard deviation from the dataset. Then apply the scaling to transform the data.

```
result = model.predict(application_scale)
```

Then, we use the previously trained machine learning model to make predictions on the scaled dataset.

```
result
```

After that, we print the model’s predictions.

```
array([0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0])
```

The figure above shows the model's predictions about which applicant to approve the loan. There are thirteen '0' and seven '1' on this result, which shows that there are 13 loan applications will be accepted, and 7 loan applications will be rejected.

5.0 Conclusion

This assignment illustrates the steps to develop a comprehensive machine learning project. The result from our assignment shows that the Random Forest Classifier was the best classification method for our dataset, because it has the highest accuracy, precision, recall, f1-score, and auc score among other classification methods. In short, Random Forest Classifier is the best model to make prediction on loan application dataset.