UNIVERSITI TUNKU ABDUL RAHMAN

LEE KONG CHIAN FACULTY OF ENGINEERING AND SCIENCE

UECS3483 Data Mining

January 2024 Trimester

NO.	STUDENT NAME	STUDENT ID
1.	**Thanirmalai (Group Leader)**	**2207087**
2.	**Teoh Wei Hong**	**2203247**
3.	**Sim Jingjia**	**2207626**

	SimJingjia			
CLO2	EDA & Data Understanding: **(15 marks)**	12.5		
CLO2	EDA and Data Visualization **(15 marks)**	12.5		
CLO2	Data Preprocessing **(20 marks)**	18	**43**	
CLO3	Data Mining Algorithms (25 marks)	24		
CLO3	Evaluation: Determining the Best Model to Use (25 marks)	23	**47**	
CL04	Conclusion (25 marks)	23		Final Mark=
CL04	Presentation (25 marks)	23	**46**	**90.67**

((Report does not show the final decisions on the 20 applications to be determined.

((Good, relevant presentation, good analysis (especially on algorithms not covered in this subject))

Table Of Content

1. Summary Of Finding

	Model	Accuracy	Recall	Precision	AUC	F1_Score	time
0	Decision Tree	0.805585	0.275421	0.225344	0.575396	0.247879	5.149729
1	Random Forest	0.703345	0.625253	0.223237	0.669438	0.329007	140.807648
2	Linear SVC	0.565621	0.792088	0.183416	0.663949	0.297860	0.126904
3	k-NN	0.637861	0.647980	0.190065	0.642254	0.293918	14.665785
4	Naive Bayes	0.658364	0.694444	0.208797	0.674029	0.321062	0.047808
5	XGBoost	0.840794	0.316498	0.315966	0.613153	0.316232	1.886666
6	Gradient Boosting	0.763933	0.522896	0.251967	0.659278	0.340067	153.505294
7	Logistic Regression	0.565641	0.792593	0.183498	0.664180	0.298003	5.832941
8	Neural Networks	0.696510	0.659091	0.225155	0.680263	0.335648	21.549155

These are the results of the models we have trained. The model with the best accuracy is XGBoost. However , since the dataset is imabalanced, accuracy doesn't mean much here. The model with the best recall is Support Vector Machine.The model with best AUC is Neural Networks with 0.68. The model with the highest F1_Score is Gradient Boosting with 0.34. The model with the shortest time of execution is naïve bayes as expected since it uses simple probability to make prediction.

As the executive summary to this entire research, the model that performed the best and should be used in production is the Neural Network model since it has the highest AUC.

The main focus of this research is to predict potential loan defaulters and provide strong evidence of their highly possible default. With evidence from the machine learning model, their loan applications would not be permitted as the institution will suffer from great losses if they default on their loans. Hence, the model with the best ROC AUC score will be chosen to be used.

Aside from ROC AUC score, in the scenario of this research, a false negative prediction is more costly than a false positive prediction, as the latter might only lead to a lost opportunity of profit, but the former will most likely lead to great amount of financial losses. Recall becomes more important than the precision in cases like this to avoid a Type II error, where the model fails to identify a positive (default) case when it should have. Therefore, the recall score will be the second most important metric to be taken in account when choosing the best model.

After comparing the performances of each model mainly in the terms of ROC AUC score and recall score, the Neural Network model was chosen for its best overall performances. It has achieved a ROC AUC score of 0.680263, which is the highest among all models. Secondly, its recall score of 0.659091 is also considered intermediate among all models. Furthermore, it takes only 21.55 seconds to generate its output, which is an acceptable time length for a better performance. Its well-rounded performance allows it to stand out among all the models in the current scenario.

After applying the Neural Network model to the 20 new loaners that were previously unknown, the model predicted that 13 of them will not default, whereas the other 7 has a high possibility of defaulting. The detailed outcome is as follows:

Results of the new applicants can be seen here

	LoanID	default
0	A01	0
1	A02	0
2	A03	1
3	A04	0
4	A05	0
5	A06	1
6	A07	1
7	A08	0
8	A09	1
9	A10	0
10	B01	0
11	B02	0
12	B03	0
13	B04	1
14	B05	0
15	B06	1
16	B07	1
17	B08	0
18	B09	0
19	B10	0

2. Variable Summarization

In the 18 columns of the dataset, 16 of them are the actual features that affect whether an individual loaner will default on their loans or not. The other two columns are the LoanID being the unique identifier for each loan but not contributing to occurrence of default The columns, their data types and their descriptions are shown in the following table:

No	Column name	Data type	Description
1.	LoanID	String	Unique identifier for each loan.
2.	Age	Integer	The age of the loaner.
3.	Income	Integer	The annual income of the loaner.
4.	LoanAmount	Integer	The amount of money being loaned by the loaner.
5.	CreditScore	Integer	The credit score (creditworthiness) of the loaner.
6.	MonthsEmployed	Integer	The number of months the loaner has been employed.
7.	NumCreditLines	Integer	The number of credit lines the loaner has opened.
8.	InterestRate	float	The interest rate for the loan the loaner has applied.
9.	LoanTerm	integer	The term length of the loan in months.
10.	DTIRatio	float	The Debt-to-Income ratio, indicating the debt of the loaner compared to their income.

11.	Education	String	The highest level of education attended by the loaner. Consists of 4 different values: High School, PhD, Master's, and Bachelor's.
12.	EmploymentType	String	The type of employment status of the loaner. Consists of 4 different values: Unemployed, Full-time, Part-time, and Self-employed.
13.	MaritalStatus	String	The marital status of the loaner. Consists of 3 different values: Married, Divorced, Single.
14.	HasMortgage	String	Boolean variable on whether the loaner has mortage (Yes) or no (No).
15.	HasDependents	String	Boolean variable on whether the loaner has dependents (Yes) or no (No).
16.	HasCoSigner	String	Boolean variable on whether the loaner has co-signer (Yes) or no (No).
17.	LoanPurpose	String	The purpose of the loan. Consists of 5 different values: Education, Auto, Home, Business, Other.
18.	Default	Integer	Binary target variable indicating whether the loaner defaulted on their loan (1) or not (0).

3. Data preprocessing

Transforming qualitative values to quantitative values

Label encoding

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

The qualitative variables in the Education, EmploymentType, MaritalStatus, HasMortage, HasDependents and HasCosigner are transformed into quantitative variables so that they can be used in modeling. Simple label encoding was used as a hierarchy relationship exists between the different values in the said features. Take Education for example, the level of highest education is hierarchical with one level greater than another, causing the difference between PhD and High School to be greater than between Bachelor's and High School. This gives a more precise distance between different values compared to one-hot encoding.

One-hot encoding

```
one_hot_encoded = pd.get_dummies(df["LoanPurpose"], prefix='LoanPurpose')

df = df.drop(["LoanPurpose"], axis=1)
df = pd.concat([df, one_hot_encoded], axis=1)
```

The qualitative variables in the LoanPurpose feature are transformed into quantitative variable using one-hot encoding method as there is no clear hierarchy relationship between the loan purposes. After that, the original LoanPurpose column is dropped, and the one-hot-encoded columns are concatenated into the original dataset.

Data Splitting

The entire dataset is split into three different subsets, namely the train, validation and test datasets with the ratio of 6:2:2. The train dataset is used to train the model, the validation dataset is used to validate the performance of the model to ensure the model does not overfit nor underfit, whereas the test dataset serves as a final test to evaluate and compare the performance of multiple models to choose the best model. The stratify parameter ensures the split preserves the percentage of samples for each class in all 3 subsets, allowing each class to be represented in the same proportions in the subsets as they are in the original dataset.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state = 42, stratify = Y)

X_train, X_validate, Y_train, Y_validate = train_test_split(X_train, Y_train, test_size = 0.25, random_state = 42, stratify = Y_train )
```

```
print(f"Train sample size: {X_train.shape}\nValidation sample size: {X_validate.shape}\nTest sample size: {X_test.shape}")
```

```
Train sample size: (153195, 20)
Validation sample size: (51066, 20)
Test sample size: (51066, 20)
```

The training dataset has around 153k rows of data which is 60% of the original 250k rows of data, while the validation and test dataset has 51k each which is 20% of 250k.

Data cleaning

The data cleaning step consists of three tasks: removal of null values, removal of duplicate values and treatment of outliers values.

Removal of null values

After applying the isnull() function on the dataset, it can be observed that there are no null values present in the dataset.

```
print(f"Null values: \n\n{df.isnull().sum()}")
```

Output:

```
Null values:

LoanID             0
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
Default            0
dtype: int64
(255327, 18)
```

Hence, none of the values were removed for being a null value.

Removal of duplicate values

It can be observed that there are no duplicate values present in the dataset.

```
duplicate_rows = df[df.duplicated()]
print(f"Number of duplicate rows: {duplicate_rows.shape}")
```

Output:

```
Number of duplicate rows: (0, 18)
```

Hence, none of the values were removed for being duplicated.

Removal of outlier values

The detection of outliers is carried out in two methods, namely the Interquartile Range (IQR) method and the Z-score method.

1) IQR

Using this method, the upper limit and lower limit of each feature is determined through Q3 + 1.5 * IQR and Q1 - 1.5 * IQR respectively. All data points higher than the upper limit and lower than the lower limit are considered as outliers and removed.

```
Q1 = X_train.quantile(0.25)
Q3 = X_train.quantile(0.75)

IQR = Q3 - Q1

X_train2 = X_train[~((X_train < (Q1 - 1.5 * IQR)) | (X_train > (Q3 + 1.5 * IQR))).any(axis=1)]
```

2) Z-score

In this method, the Z-score of each individual record is determined and if it is greater than 3, it is considered as an outlier and removed.

```
z = np.abs(stats.zscore(X_train))
X_train3  = X_train[(z > 3).all(axis = 1)]
X_train3.shape
```

Output:

```
X_train2: (0, 20)
X_train3: (0, 20)
```

After using both methods to detect the outliers of each feature, it can be seen that none of the records were considered as an outlier. Hence, none of them are removed from the dataset.

4. Exploratory Data Analysis on Training Dataset

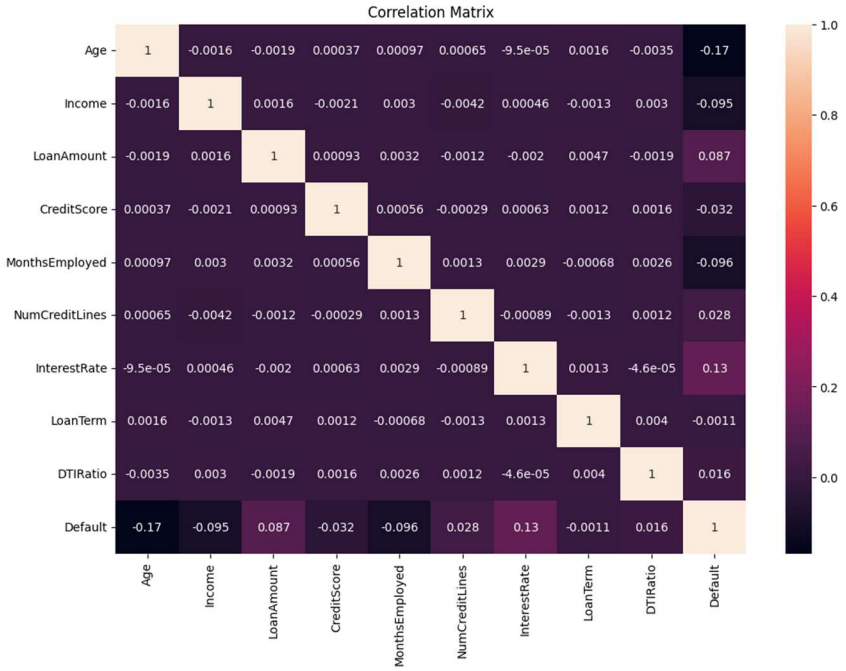
Questions to be Addressed

- What is the distribution of the numeric features
- What are the relationship between the input features and the output feature

Distribution of data

The dimension of the data is 153196 rows and 18 columns.
Here are the first 5 rows of dataset

	LoanID	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner	Default
158474	CWJK9JQCJI	68	80816	152988	780	78	4	4.18	12	0.38	PhD	Full-time	Single	Yes	No	Education	Yes	0
142166	5889B6EF76	48	50777	153455	320	73	3	6.88	60	0.40	Bachelor's	Self-employed	Married	Yes	Yes	Home	No	0
91504	NMVDWJX8BX	36	41022	224211	420	2	2	6.45	60	0.65	Master's	Full-time	Divorced	No	Yes	Auto	No	0
109239	MUJF5DPW8Y	54	73446	85362	790	17	4	24.16	24	0.15	High School	Self-employed	Married	No	No	Home	No	0
48269	2TYGLODH4C	69	41347	156778	682	79	1	8.40	24	0.44	Master's	Self-employed	Divorced	Yes	No	Auto	Yes	0

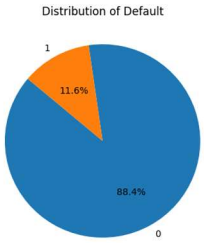


Correlation Matrix

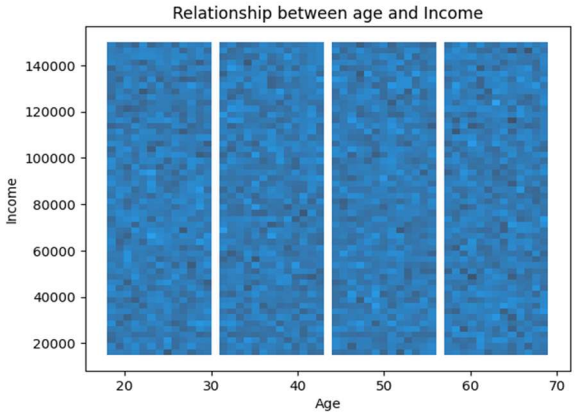
The correlation matrix indicates that there are no strong correlations (values close to 1 or -1) between any pair of features in the dataset. This suggests that the features are largely independent of each other and do not exhibit strong linear relationships. Therefore, there is no need to remove any features due to high correlation.

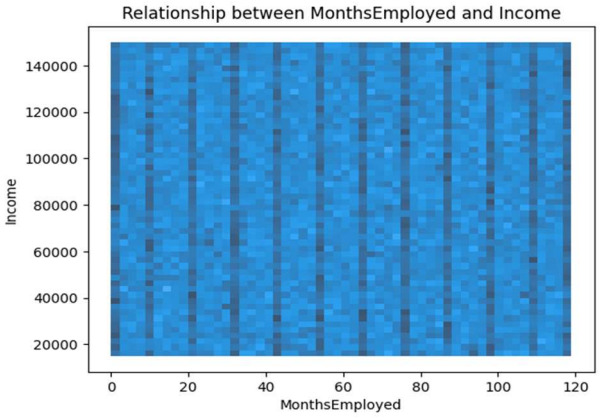
Distribution of default feature (output label)

The pie chart shows that most of the data points fall into the category (1), indicating a significant imbalance. This means there are far more instances of defaults compared to non-defaults which can affect accuracy of analysis and modeling and require special attention to ensure reliable results.



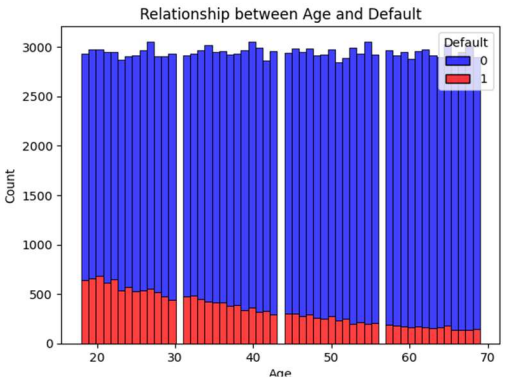
Relationship between input features

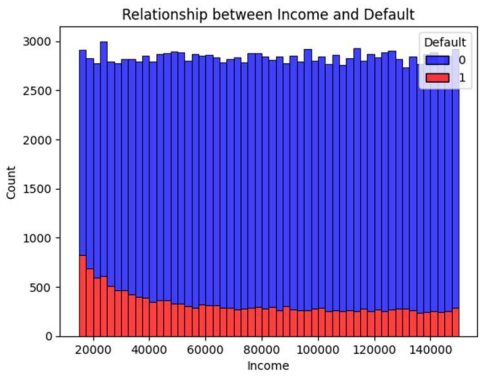


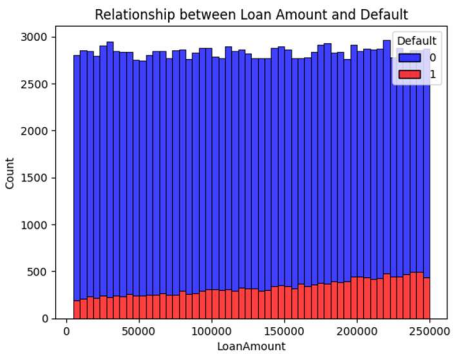


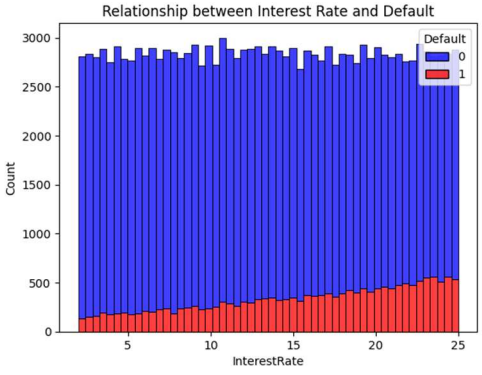
The scattered distribution observed between **Age** and **Income** as well as **Months Employed** and **income** is representative of the overall pattern seen across all input variables in the dataset. This lack of clear structure or trend extends to the relationships between all pairs of input variables, indicating a general absence of strong correlations or linear dependencies. The scattered nature of the points across various feature combinations underscores the dataset's complexity and the diverse range of interactions among its variables, highlighting the need for comprehensive analysis techniques to uncover underlying patterns or relationships. For a detailed visual analysis of all relationships between input features , please refer to the Colab Notebooks provided.

Relationship between distributed input features and output features (default)

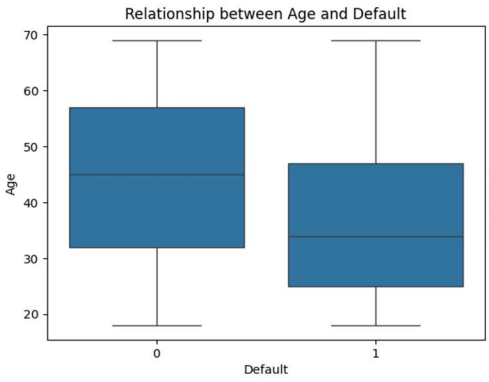


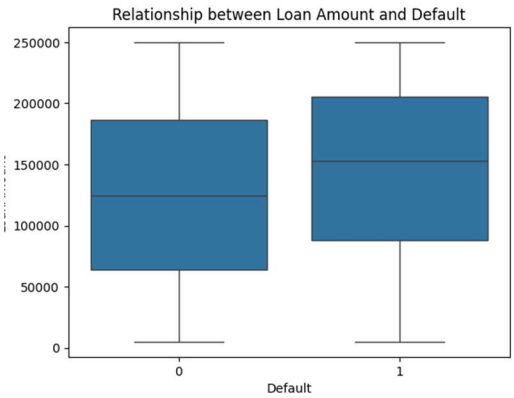


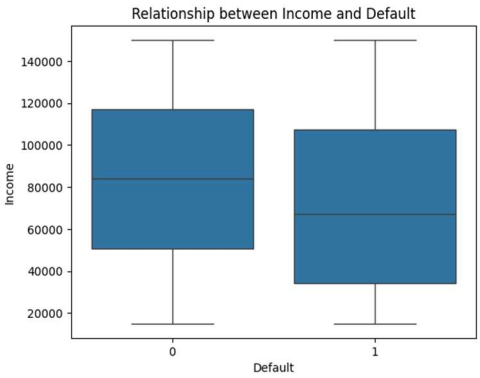




These are the relationship between input and output features. We have noted that **Age** , **Income** , **Loan amount** and **Interest Rate** shows a significant relationship to **Default**. Firstly, when considering age, our analysis revealed a notable trend: as individuals' age increases, there appears to be a corresponding decrease in the likelihood of defaulting on a loan. This trend suggests that older individuals, on average, are less prone to defaulting compared to their younger counterparts. Similarly, our examination of income levels uncovered a compelling relationship with default rates. As income increases, we observe a tendency for the probability of default to decrease. This pattern implies that individuals with higher incomes are generally more financially stable and less likely to default on their loan obligations. Furthermore, we found that as the loan amount increases, there tends to be a decrease in the likelihood of default. This trend suggests that applicants seeking larger loan amounts are often subjected to loan default. Besides, we found that as the interest rate increases, there tends to be an increase in the likelihood of default. This trend suggests that applicants seeking high interest loans are often highly likely to loan default.

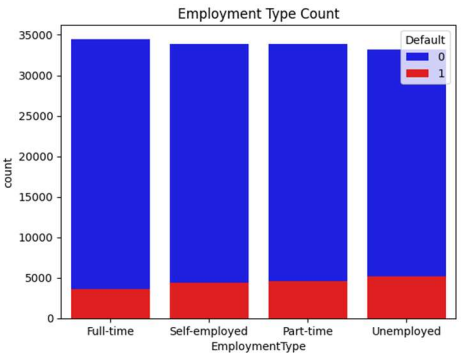


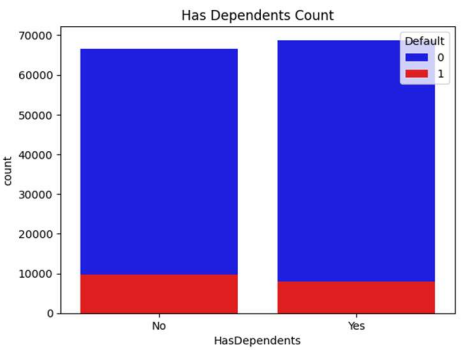


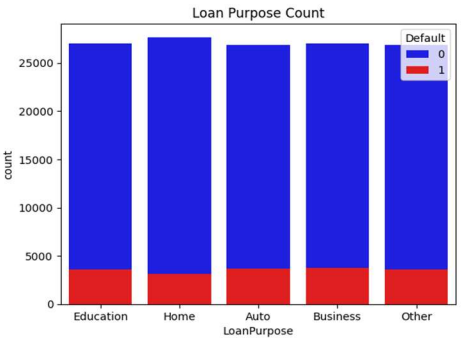


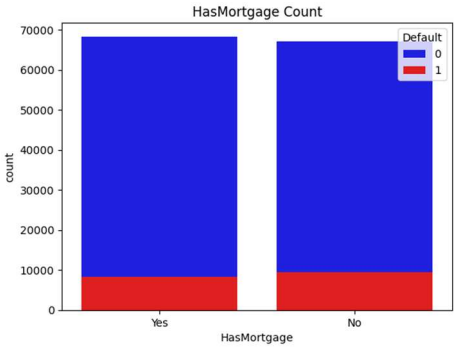
The boxplot provides a visual representation of important statistical metrics, including median values, quartiles, and the absence of outliers. Notably, there are no outlier data points present for any of the features analyzed. Examining the plot, we observe distinct trends among individuals who defaulted compared to those who did not. Firstly, defaulters tend to be younger, as evidenced by a lower median **age**. Secondly, defaulters typically borrow higher median loan amounts. Thirdly, defaulters generally have lower median **incomes**. Lastly, defaulters face higher median **interest rates** compared to non-defaulters. These findings collectively suggest clear differences between the two groups across multiple financial attributes.

Relationship between categorical input features and output features (default)









These histograms show the distribution of categorical features and relationship between the categorical features and default. Firstly, the histogram displays the frequency distribution of default rates across four **employment** categories: Full Time, Self Employed, Part Time, and Unemployed. Notably, the range frequency for all employment types is consistent, peaking between 32000 and 34000. Among these categories, the highest default rate is observed among the Unemployed group at 29.1%, followed by Part Time individuals at 25.6%. Self Employed individuals exhibit a default rate of 24.8%, while Full Time workers have the lowest default rate at 20.5%. Secondly, the analysis of the **dependent** variable reveals that individuals with a dependent exhibit a default rate of 45.2%, while those without dependent have a higher default rate at 54.8% . Furthermore, The analysis of default rates across different **loan purposes** reveals varying percentages: "Other" loans have a default rate of 20.3%, "Home" loans have a default rate of 17.7%, "Education" loans have a default rate of 20.4%, "Business" loans have a default rate of 21.4%, and "Auto" loans have a default rate of 20.6%. Lastly, The analysis of the **mortgage** variable indicates that individuals who answered "Yes" to having a mortgage exhibit a default rate of 46.9%, while those who responded "No" have a higher default rate at 53.1%. This finding suggests a somewhat counterintuitive trend where individuals without a mortgage default more frequently than those with a mortgage.

Data Standardization and Normalization

```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
X_validate = scaler.fit_transform(X_validate)
```

All 3 datasets are standardized to ensure the values of each feature are normalized so that their weights are the same in the calculation of the models. This can prevent certain variables from having more weight than the other variables and dominate the model, which will lead to inaccurate modeling. The datasets are standardized individually to prevent data leak, which allows the model to gain extra information about the test and validation dataset.

5. Feature Engineering

Feature Engineering that worked out well.

Feature Engineering is an essential in shaping the performance of machine learning models; by performance I meant both in the metrics of the machine learning model in question but also in the speed of its execution. A

robust machine learning model on paper might not be very applicable in real world environments if the speed of its execution is horrendous; KNN and SVM being the two very well-known culprit in those regards; henceforth, the job of a data scientist is to improve the speed of the model to an acceptable level while sacrificing little of its performance with regards to metrics.

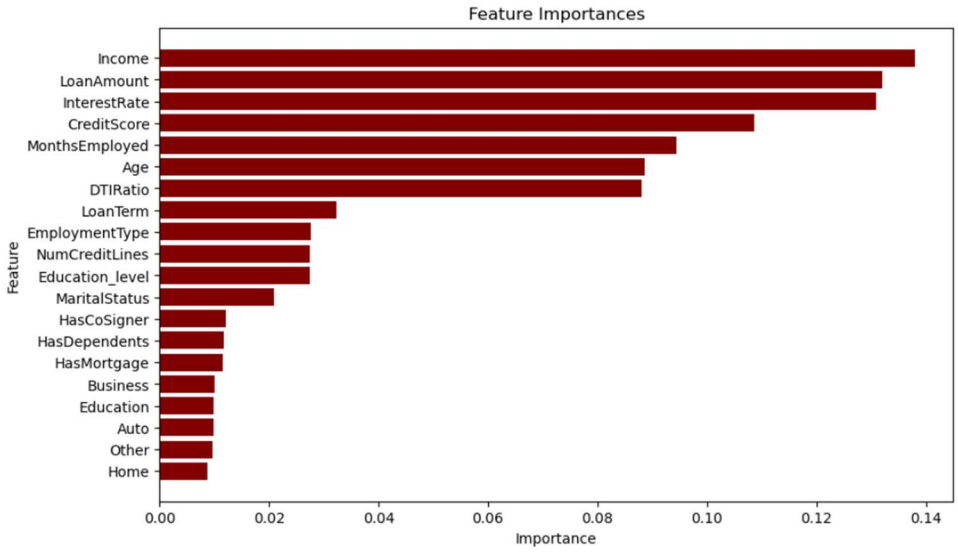
```
X_train = train_df.drop(['Default'], axis=1)
y_train = train_df['Default']
```

```
rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
feature_importances = rf.feature_importances_

importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='maroon')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importances')
plt.gca().invert_yaxis()
plt.show()
```

For this assignment, random-forest regressor is chosen to select relevant features from the 14 variables (before one-hot encoding). The way it works is by measuring how much each feature contributes to decreasing the impurity (such as Gini impurity or entropy) in the decision trees. Features that result in greater impurity reduction across all trees are considered more important. While linear regressor and other types of regressors does fine when it comes to the topic of feature selections, there are a few things that random forest regressors have an edge over the other models. The main part is random forest's robustness to noisy and irrelevant data, since it is an ensembled learner, it averages predictions over multiple trees, it minimizes the effect of outliers and irrelevant features, unlike its simple linear regression counterpart.

```
X_train = train_df.drop(['Default'], axis=1)
y_train = train_df['Default']
```

```
rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
feature_importances = rf.feature_importances_

importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='maroon')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importances')
plt.gca().invert_yaxis()
plt.show()
```



For this assignment, the 7 most significant variables are chosen; that being (Income, LoanAmount, InterestRate, CreditScore, MonthsEmployed, Age, DTIRatio). Why 7 you may ask? It is because of the discrepancy of the importance between variable 7 and variable 8. While we found out that feature reduction have little impact on the metrics of the machine learning models, the speed of all models improved significantly; KNN being the main beneficiary of feature reductions.

Feature Engineering that did not worked.

Creating new features.

Now onto the less fun part of the assignment, concepts that did not make the final cut. Besides feature reduction, our team also tried creating new variables out of existing variables. The main findings of our assignments is that the troubles that arise from the dataset not only comes from the imbalance of classes, but also from the poor quality of data, that is that the data itself does a very poor job of predicting defaults. Our team came up with several variables that might potentially help the machine learning models to predict the outcome and I shall explain the variables in detail.

```
train_df.drop(columns=['Lifespan_after_loan'], inplace=True, errors='ignore')
train_df['Lifespan_after_loan'] = 72 - train_df['Age'] - train_df['LoanTerm']
train_df.head()
```

This variable attempt to measure how close the borrower is to the average lifespan of a human since death is one of the reasons of default.

```
train_df['InterestCoverage'] = train_df['Income']-((train_df['InterestRate']/100) * train_df['LoanAmount'])
train_df.head()
```

This variable measures the interest coverage of the borrower by measuring how much income the borrower has left after paying off the interest on their debt.

```
train_df['CreditScore_log'] = np.log(train_df['CreditScore'])
train_df.head()
```

This variable applies log transformation to credit score to see if the model performs better with a different transformation.

```
train_df['CreditScore_Log_by_InterestRate'] = np.log(train_df['CreditScore']) / train_df['InterestRate']
train_df.head()
```

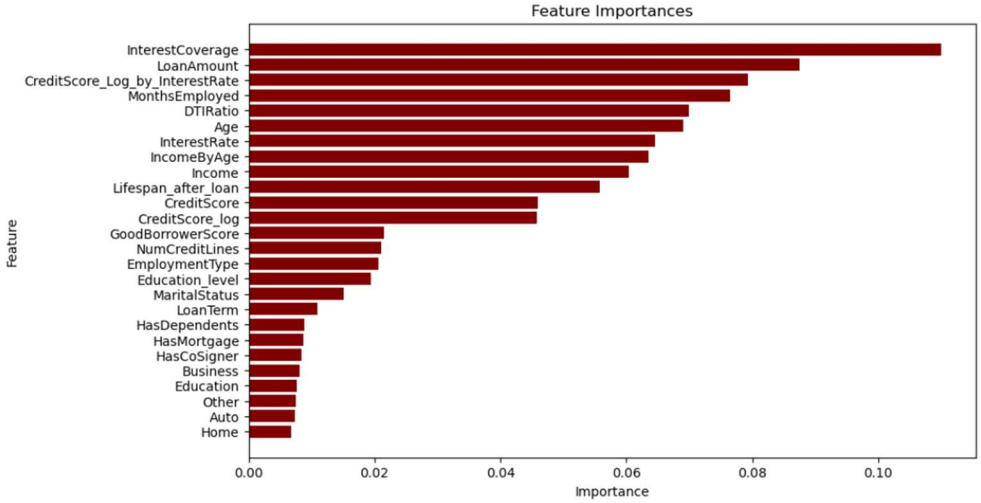
This variable is an extension of the last variable. Since credit score is inversely proportional to the probability of default and interest rate is directly proportional, we apply division to both.

```
train_df['IncomeByAge'] = train_df['Income'] / train_df['Age']
train_df.head()
```

This rationale behind this variable is that a person generally becomes richer as they reached middle age and then income drops as they retire. By applying this division, we attempt to penalized borrowers that are both poor and old.

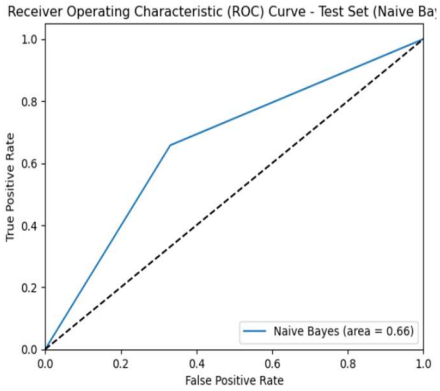
```
# Define conditions for good borrower score
good_borrower_conditions = (
    (train_df['Education_level'] > 0).astype(int) +
    (train_df['EmploymentType'] == 2).astype(int) +
    (train_df['MaritalStatus'] == 1).astype(int) +
    (train_df['HasCoSigner'] == 1).astype(int)
)

# Create new column 'GoodBorrowerScore' based on conditions
train_df['GoodBorrowerScore'] = good_borrower_conditions
```

This variable is an attempt to make the 4 irrelevant variables relevant by combining all of them into a metrics called good borrower score. Generally, people with higher education, have a full-time job, have a cosigner and is married are less likely to default.

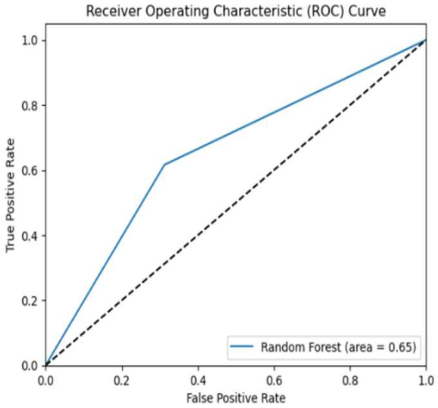


To test the efficacy of this feature, 7 features are selected (InterestCoverage, LoanAmount, CreditScore_Log_by_InterestRate, CreditScore, MonthsEmployed, Age, DTIRatio). Income and InterestRate are removed due to high correlation to InterestCoverage and CreditScore_Log_by_InterestRate

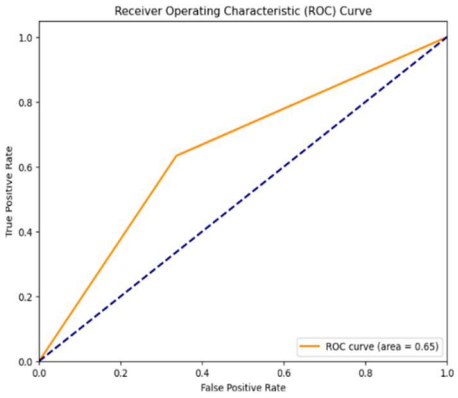
Modified Variables



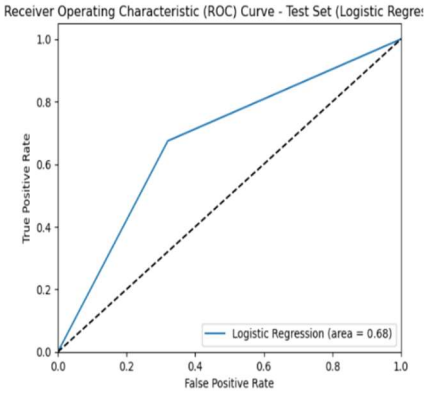
Naïve Bayes



Random Forest

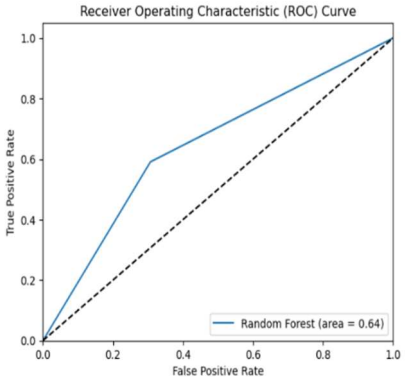


KNN

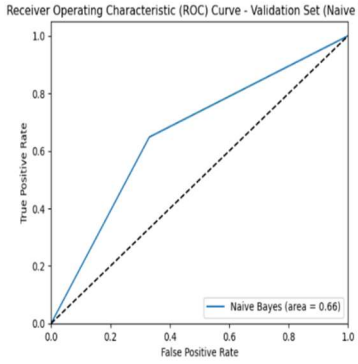


Logistic Regression

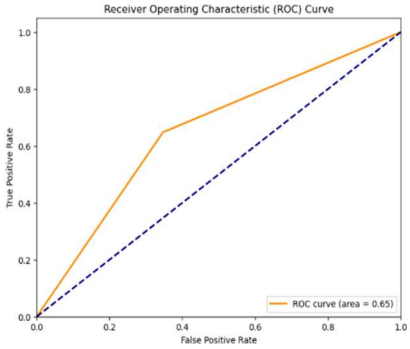
Original



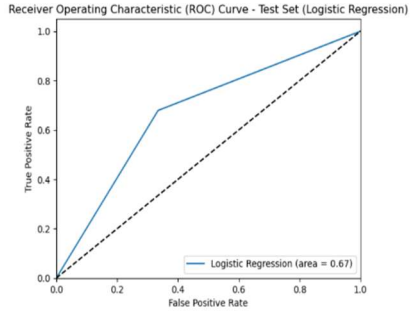
Random Forest



Naïve Bayes



KNN



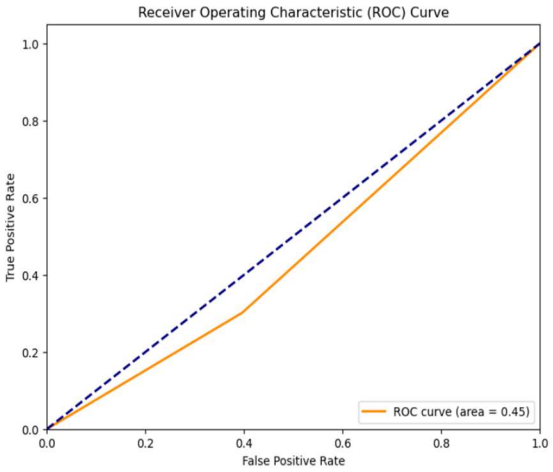
Logistic regression

The main finding is that the modifications of the variables does not bore much improvement in performance. AUC score of logistic regression, random forest and KNN improved by 0.01 compared to their original counterpart while Naïve Bayes bears no difference. Due to the miniscule impact and time constraint, we did not include this feature engineering in our final notebook.

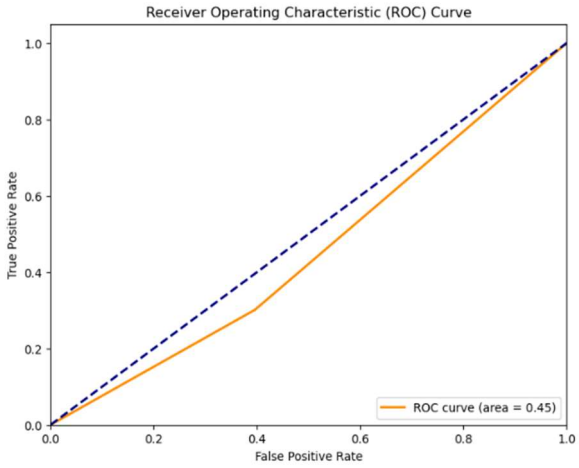
9.2.2 Dimensionality reduction

We also tried dimensionality reduction algorithms like PCA and t-SNE. Needless to say, all of the model's performance deteriorated upon applying this algorithm. To avoid complications, we shall only show the performance of KNN to prove our point. For PCA, we select 7 features, compress them into 5 variables using PCA and apply SMOTETOMEK. For t-SNE, the process is the same except we reduce the 7 variables into 3 dimensions.

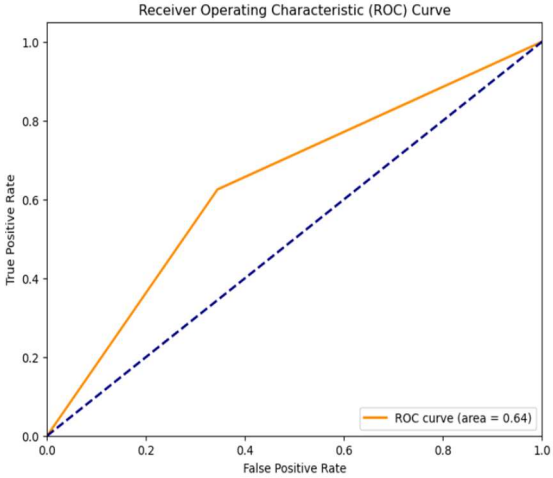
PCA



t-SNE



Without dimensionality reduction



In conclusion, the speed did in fact increased a lot but at the cost of deterioration in metrics upon applying dimensionality reduction algorithms.

6. Sampling techniques

SMOTE-Tomek

For this dataset, sampling is needed for the performance of the model to be acceptable (especially recall). Our findings shows that all models prior to oversampling display terrible performance across all metrics (F1-score, recall, AUC). For our models, we decided that oversampling is a superior choice over under sampling due to the amount of data loss using the latter method. Our weapon of choice for oversampling is SMOTE-Tomek. SmoteTomek has some advantages over smote oversampling, one being that it uses undersampling to remove noisy synthetic data generated by the oversampling, which is helpful for models like KNN, SVM and Naïve Bayes which are especially sensitive to noisy data. SMOTE-Tomek also focuses on generating synthetic samples in regions of the feature space where the decision boundary between classes is ambiguous or poorly defined. By oversampling these regions, SMOTE-Tomek can help the classifier better capture the underlying distribution of the minority class, leading to improved classification performance, particularly for complex datasets with overlapping classes.

```
from imblearn.pipeline import make_pipeline
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import TomekLinks  # Import TomekLinks
#Create Smote pipeline
# Create SMOTETomek pipeline
SMOTETomek_pipeline = make_pipeline(SMOTETomek(tomek=TomekLinks(sampling_strategy='majority')))

# Fit the pipeline to your training data
SMOTETomek_pipeline.fit(X_train, y_train)

# Transform your original training data to obtain the balanced dataset
X_train_resampled, y_train_resampled = SMOTETomek_pipeline.fit_resample(X_train_scaled, y_train)
```

```
y_train_resampled.value_counts(normalize=True)
```

```
Default
1    0.501008
0    0.498992
Name: proportion, dtype: float64
```

7. Modelling

Logistic Regression

Logistic regression is a statistical method that is used for classification. In the model, the relationship between the features and the probability of the outcome is modeled using the logistic function (or the sigmoid function), that transforms any real-valued input into a numerical value that ranges between 0 and 1. This numerical value is interpreted as the probability for the positive class.

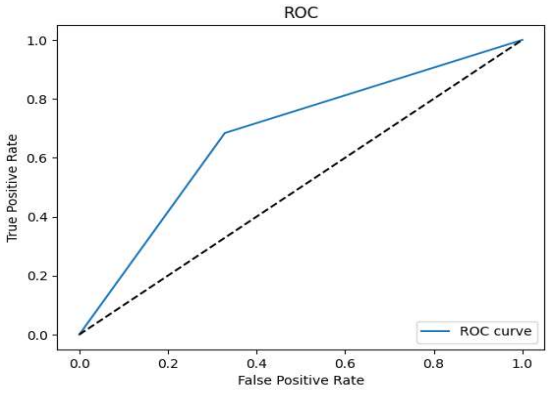
The LogisticRegression() function from sklearn.linear_model was used to develop the LogisticRegression model. In the first iteration, the base version of Logistic Regression without any hyperparameters was ran.
The outcome produced from the base Logistic Regression is shown below:

```
model = LogisticRegression()
output_model = model.fit(X_resampled, Y_resampled)
Y_pred = output_model.predict(X_test)
```

	precision	recall	f1-score	support
0	0.94	0.67	0.78	45048
1	0.22	0.68	0.33	6018
accuracy			0.67	51066
macro avg	0.58	0.68	0.56	51066
weighted avg	0.86	0.67	0.73	51066

ROC AUC Score: 0.6778101364526559

In the classification report above, it can be seen that the overall metrics are slightly unsatisfying. The metrics for the value of 1s were worse than the values of 0s, producing an outcome of only 0.22 for precision, which indicates that the model can only correctly predict 22% of the default class, and 0.33 for f1-score. From the confusion matrix, it can be seen that the ability of the model to predict both positive and negative results are similar, despite being given an imbalance dataset. This is most likely owing to the sampling done by SmoteTomek which allows the model to be less biased. However, the accuracy of only 0.67 and ROC AUC score of only 0.6714 is quite unsatisfying.



Hence, the second iteration was carried out with hyperparameter tuning using RandomSearchCV. GridSearchCV was not used due to requiring long processing time. The hyperparameters for LogisticRegression() that were tested are as follows:

```
hyperparams = {"penalty": ["l2", None],
               "solver" : ["lbfgs", "newton-cg", "newton-cholevsky", "sag", "saga"],
               "multi_class" : ["auto", "ovr", "multinomial"],
               "C": [i / 10 for i in range(1, 101)],
                "class_weight": ["balanced"],
                "intercept_scaling": [i / 10 for i in range(1, 51)],
                "fit_intercept": [True, False],
                "warm_start": [True, False],
                }

LR_model = RandomizedSearchCV(estimator=LogisticRegression(max_iter = 10000), scoring = "balanced_accuracy", param_distributions=hyperparams, cv=StratifiedKFold(), refit = True, verbose = 3,n_jobs=-1, random_state = 1)
LR_model.fit(X_resampled, Y_resampled)
print(f'Best score: {LR_model.best_score_} with param: {LR_model.best_params_}')

model = LogisticRegression(**LR_model.best_params_)
output_model = model.fit(X_resampled, Y_resampled)
Y_pred = output_model.predict(X_test)
```

Hyperparameters	Values
Penalty	"l1", None
solver	"lbfgs", "liblinear", "newton-cg", "newton-cholesky", "sag", "saga"
multi_class	"auto", "ovr"
C	0.1 ~ 10.0 with interval 0.1 (0.1, 0.2, 0.3 … 10.0)

class_weight	“balanced”
fit_intercept	True, False
intercept_scaling	0.1 ~ 5.0 with interval 0.1 (0.1, 0.2, 0.3 … 5.0)
warm_start	True, False

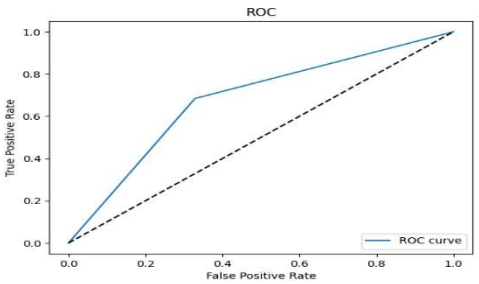
After applying hyperparameter tuning on the LogisticRegression() model, the RandomSearchCV returned the following best parameters:

```
Best score: 0.6770141482141172 with param: {'warm_start': True, 'solver': 'sag', 'penalty': None, 'multi_class': 'ovr', 'intercept_scaling': 1.1, 'fit_intercept': True, 'class_weight': 'balanced', 'C': 9.9}
```

With the best AUC Score of 0.6770141482141172. After the tuning process, the Logistic Regression model produced the following output:

	precision	recall	f1-score	support
0	0.94	0.67	0.78	45048
1	0.22	0.68	0.33	6018
accuracy			0.67	51066
macro avg	0.58	0.68	0.56	51066
weighted avg	0.86	0.67	0.73	51066

```
AUC Score: 0.6778212357245436
```



An overview on the comparison between the metrics produced before and after hyperparameter tuning:

Metrics	Before Tuning	After Tuning
Accuracy	0.67	0.67
Recall	0.68	0.68
F1-Score	0.33	0.33
Precision	0.22	0.22
ROC AUC	0.6778101	0.6778212

LogisticRegression() model that achieve ROC AUC of 0.6778. As the output is similar, it is quite unsatisfying as well in general terms.

It can be seen that with the hyperparameter tuning, the model improved slightly in the ROC AUC score. However, it is worth noting that when the hyperparameter penalty is set to None, which is the best parameter returned by RandomSearchCV, the other hyperparameters of C and l1_ratio are ignored.

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:1181: UserWarning: Setting penalty=None will ignore the C and l1_ratio parameters
  warnings.warn(
```

In conclusion, for the LogisticRegression() model, the model performed slightly better after hyperparameter tuning and the tuned model should be used in prediction of the unknown dataset.

Support Vector Machine

Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane, a line dividing a plane into two, where data points laying on one side are considered to be in the same class. In the training phase, the SVM searches for the hyperplane with the largest margin, as its prediction will be more accurate with future unknown data than the hyperplane with the smaller margin, which is more likely to misclassify data points close to the decision boundary.

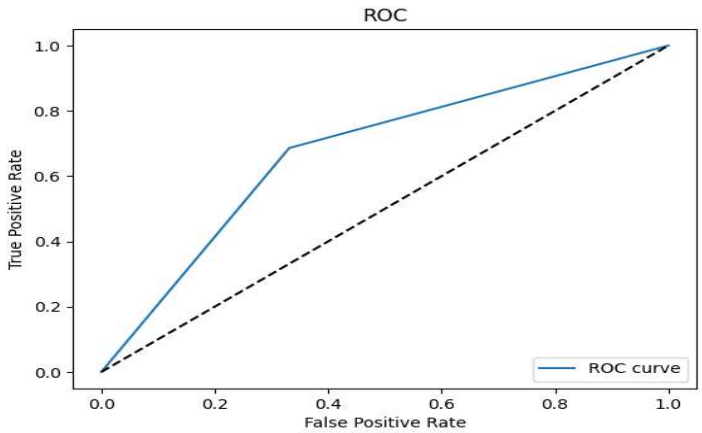
Although SVM has many advantages such as having a number of kernels to choose from and being fairly robust against overfitting, it does not work well when the dataset is large due to the high required training time. In fact, as the dataset used for modeling is very large, the conventional SVC() model from the sklearn.svm library could not be used. The documentation on Scikit-learn noted that the fit time of SVC() scales at least quadratically with the number of samples. For a dataset beyond thousands of samples, the SVC() model is impractical.

Hence, the LinearSVC() model was used in its place. LinearSVC() is similar to SVC with parameter kernel = “linear”, but it is implemented in liblinear instead of the libsvm in SVC(), so it has more choices of penalties and loss functions. Both models mainly differ in the loss function used by default, and in the handling of intercept regularization.

In the first iteration, the base LinearSVC() model without any hyperparameters is ran. The output produced by the base LinearSVC() model is as follows:

```
model = LinearSVC()
output_model = model.fit(X_resampled, Y_resampled)
Y_pred = output_model.predict(X_test)
```

	precision	recall	f1-score	support
0	0.94	0.67	0.78	45048
1	0.22	0.69	0.33	6018
accuracy			0.67	51066
macro avg	0.58	0.68	0.56	51066
weighted avg	0.86	0.67	0.73	51066




```
ROC AUC Score: 0.6773477516305637
```

In the classification report above, it can be seen that the overall metrics are unsatisfying as well. On top of it, the overall metrics of LinearSVC() were somewhat similar to of the previous LinearRegression() model. The only difference between the two models is the recall, where the LinearSVC() model achieved 0.69 as compared to 0.68 of the LinearRegression() model, and the ROC AUC, where the LinearSVC() model performed slightly worse with score of 0.6773 compared to the

Hyperparameters	Values
penalty	"l1", "l2"
loss	“hinge”, “squared_hinge”
dual	True, False

Hence, the second iteration was carried out with using RandomSearchCV. hyperparameter tuning LogisticRegression() that The hyperparameters for were tested are as follows:

multi_class	"ovr", "crammer_singer"
C	0.1 ~ 10.0 with interval 0.1 (0.1, 0.2, 0.3 … 10.0)
fit_intercept	True, False
intercept_scaling	0.1 ~ 5.0 with interval 0.1 (0.1, 0.2, 0.3 … 5.0)
class_weight	balanced
verbose	1 ~ 50 with interval 1 (1, 2, 3… 50)

```
hyperparams = {"penalty": ["l2", "l1"],
               "loss": ["hinge", "squared_hinge"],
               "dual": [True, False],
               "multi_class" : ["ovr", "crammer_singer"],
               "C": [i / 10 for i in range(1, 101)],
               "fit_intercept": [True, False],
               "intercept_scaling": [i / 10 for i in range(1, 51)],
               "class_weight": ["balanced"],
               "verbose": [i for i in range(1, 51)]
               }

SVM_model = RandomizedSearchCV(estimator=LinearSVC(class_weight = "balanced_"), scoring = "balanced_accuracy", param_distributions=hyperparams, cv=StratifiedKFold(), refit = True, verbose = 3,n_jobs=-1, random_state = 1)
SVM_model.fit(X_resampled, Y_resampled)
print(f'Best score: {SVM_model.best_score_} with param: {SVM_model.best_params_}')

model = LinearSVC(**SVM_model.best_params_)
output_model = model.fit(X_resampled, Y_resampled)
Y_pred = output_model.predict(X_validate)
```

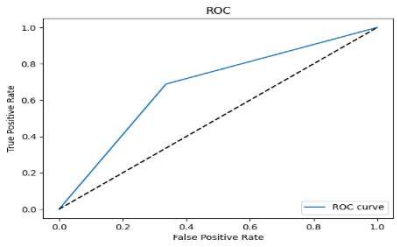
The RandomSearchCV returned the following best parameters:

```
Best score: 0.6762996339585187 with param: {'verbose': 10, 'penalty': 'l1', 'multi_class': 'crammer_singer', 'loss': 'hinge', 'intercept_scaling': 1.6, 'fit_intercept': True, 'dual': False, 'class_weight': 'balanced', 'C': 1.5}
```

With the best score of 0.6762996339585187.

The overall performance of the model after applying the best hyperparameters has slightly increased as compared to the base version of the models:

	precision	recall	f1-score	support
0	0.94	0.66	0.78	45048
1	0.21	0.69	0.33	6018
accuracy			0.67	51066
macro avg	0.58	0.68	0.55	51066
weighted avg	0.86	0.67	0.73	51066




```
AUC Score: 0.676312952015911
```

An overview on the comparison between the metrics produced before and after hyperparameter tuning:

Metrics	Before Tuning	After Tuning
Accuracy	0.67	0.67
Recall	0.69	0.69
F1-Score	0.33	0.33
Precision	0.22	0.22

ROC AUC	0.6773	0.6763

It can be observed that after hyperparameter tuning, the performance of the LinearSVC() model became worse in terms of precision and ROC AUC score as compared to before it. This may be due to several reasons, including the fact that a number of fits failed during the tuning, as certain parameters are not compatible with each other. For example, the combination of penalty = “l1” and loss = “hinge” is not supported.

```
ValueError: Unsupported set of arguments: The combination of penalty='l1' and loss='hinge' is not supported,
```

However, the best parameters returned by the RandomSearchCV included penalty = “l1”, which means that although it is not compatible with some other parameters, it is still the best parameter. In order to increase the performance of the model as much as possible, such parameters were still included for the tuning, despite producing a worse performance.

Another possible reason is due to the usage of RandomSearchCV instead of GridSearchCV for the hyperparameter tuning. Despite being able to produce a good solution in a shorter time for the sake of efficiency, RandomSearchCV suffers from the disadvantage of the lack of exhaustiveness. While GridSearchCV searches through all possible combinations of hyperparameters specified in the grid to ensure an optimal combination, RandomSearchCV randomly takes a fixed number of samples of hyperparameter combination from the specified distributions. This will cause the RandomSearchCV to potentially miss the optimal combination and eventually lead to a worse performance.

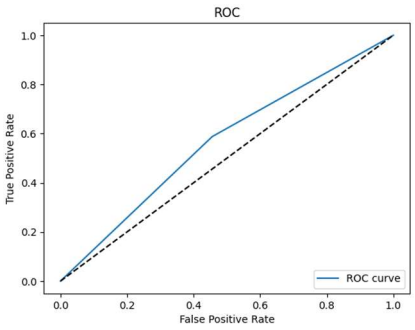
In conclusion, the LinearSVC() model before hyperparameter tuning displayed a better performance and will be used to predict the unknown data.

Decision Tree

Decision Tree is a non-parametric supervised learning method used for decision-making where each internal node represents a decision based on a specific attribute, and each leaf node represents a class label or a decision outcome. It recursively partitions the data space into smaller regions, aiming to maximize the purity of the classes within each partition. DecisionTreeClassifier from sklearn was used to develop this model. For the first iteration , we ran the base DecisionTreeClassifier without any hyperparameters.

```
dt = DecisionTreeClassifier()
dt.fit(X_train , Y_train)

find_metrics("Training", Y_train , dt.predict(X_train))
find_metrics("Validation", Y_val , dt.predict(X_val))
find_metrics("Testing", Y_test , dt.predict(X_test))
```

```
Classification Report for Decision Tree (Test):
              precision    recall  f1-score   support

           0       0.90      0.85      0.87     45126
           1       0.19      0.26      0.22      5940

    accuracy                           0.78     51066
   macro avg       0.54      0.56      0.55     51066
weighted avg       0.81      0.78      0.80     51066

ROC AUC for Decision Tree (Test): 0.56
```



This is the classification report for the decision tree on testing dataset which is unseen by the model. The model achieved 78% accuracy.The result shows that model is good at predicting non default class and bad at predicting default class as per their recall. To solve this , we have tried hyperparameter tuning on Decision Tree to prune to tree to prevent overfitting.

```
# Define the Decision Tree classifier with balanced class weights
dt = DecisionTreeClassifier(class_weight="balanced")

# Define the parameter distribution for Randomized Search
criterion = ['gini', 'entropy', 'log_loss']
splitter = ['best', 'random']
max_depth =  [i for i in range(1, 41, 5)]
min_samples_split =  [i for i in range(2, 21, 2)]
min_samples_leaf = [i for i in range(1, 21, 2)]
hyperparam = { "criterion" : criterion ,
               "max_depth" : max_depth,
               "min_samples_split" : min_samples_split,
               "splitter" : splitter ,
               "min_samples_leaf" : min_samples_leaf
                }

# Set up the random search with stratified cross-validation on the training data
random_search_dt = RandomizedSearchCV(estimator=dt, param_distributions=hyperparam, scoring='f1'
, cv=stratified_cv, n_iter=10, n_jobs=-1, random_state=42, verbose=2)

# Start timer
start_time_dt = time.time()
```

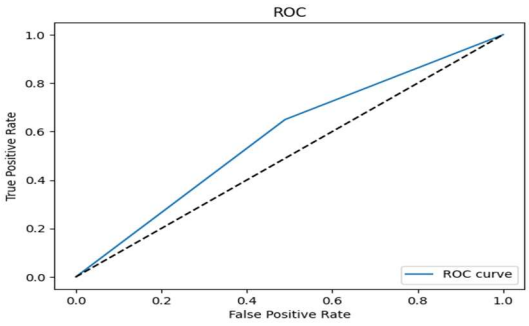
This is the code snippet which is used for tuning the parameters for the decision tree classifier. We have tried `RandomizedSearchCV` to get the best parameters by applying these methods that are optimized by cross-validated search over parameter settings. `RandomizedSearchCV` does not try out all parameter values, but rather a fixed number of parameter settings is sampled from the specified distributions . Additionally , do note that we have also implemented cost sensitive learning, an approach where the misclassification costs of different classes are explicitly considered during model training.Instead of treating all errors equally, assigning a different penalty weight for each class. This is done by using the class_weight parameter by setting it to `'balanced'`. Not only that , we have also utilized `'balanced_accuracy'` as a scoring metric to find the best hyperparameter which is a scoring metric to deal with imbalanced datasets as mentioned in sklearn.

Hyperparameters	Values
criterion (The function to measure the quality of a split)	`'gini'` , `'entropy'` , `'log_loss'`
splitter (The strategy used to choose the split at each node)	`'best'` , `'random'`
max_depth (maximum depth of the tree)	`range(1,41,5)`
min_samples_split (minimum number of samples required to split an internal node)	`range(2,21,2)`
min_samples_leaf (minimum number of samples required to be at a leaf node)	`range(1,21,2)`

```
Best parameters found:  {'splitter': 'best', 'min_samples_split': 14, 'min_samples_leaf': 9, 'max_depth': 36, 'criterion': 'entropy'}
```

These are the best hyperparameters that were found during the hyperparameter tuning process.

	precision	recall	f1-score	support
0	0.90	0.87	0.89	45126
1	0.22	0.29	0.25	5940
accuracy			0.80	51066
macro avg	0.56	0.58	0.57	51066
weighted avg	0.82	0.80	0.81	51066



After undergoing the hyperparameter tuning process , the decision tree model displayed a better performance level , improving recall , precision , F1-scores.This model is better at predicting default class than the base decision tree model. However, this performance is not satisfying enough since it is still not good in predicting default class.

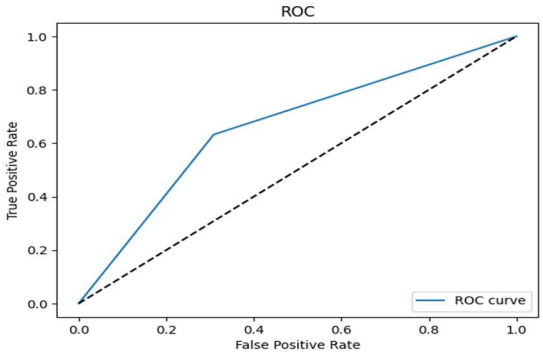
Metrics	Before Tuning	After Tuning
Accuracy	0.78	**0.80**
Recall	0.26	**0.29**
F1-Score	0.22	**0.25**
AUC score	0.56	**0.58**
Precision	0.19	**0.22**

Naive Bayes

Naive Bayes is a probabilistic classifier based on Bayes' theorem, which assumes that the features are conditionally independent given the class label. It calculates the probability of a class given a set of features by multiplying the prior probability of the class with the likelihood of the features given that class and normalizing by the evidence.

```
nb = GaussianNB()
nb.fit(X_train , Y_train)


find_metrics("Training", Y_train , nb.predict(X_train))
find_metrics("Validation", Y_val , nb.predict(X_val))
find_metrics("Testing", Y_test , nb.predict(X_test))
```

```
Classification Report for Naive Bayes (Test):
              precision    recall  f1-score   support

           0       0.94      0.65      0.77     45126
           1       0.21      0.69      0.32      5940

    accuracy                           0.66     51066
   macro avg       0.58      0.67      0.55     51066
weighted avg       0.86      0.66      0.72     51066

ROC AUC for Naive Bayes (Test): 0.67
```



This is the classification report on the testing dataset. The result is overall good as it can predict default and non default classes well. We analyzed the performance of the Gaussian Naive Bayes model, which assumes that the features follow a Gaussian (bell curve) distribution. It turned out that the model performed well, scoring high on metrics such as recall, f1-score, and accuracy. Given that this type of model doesn't have many knobs to adjust (hyperparameters), we didn't attempt to fine-tune it. However, when we did experiment with tuning these parameters, the improvements were minimal, and the model's performance remained largely similar to its original state.

K Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple algorithm that is commonly used for both classification and regression tasks in machine learning. In essence, KNN operates on the principle of similarity: it classifies the variables based on the nearest neighbor it is associated with; for this reason, it is called a lazy learner. One surprising benefits of KNN is how well it performed with the help of oversampling. Since it's training time is zero (it just fits all of the training data), oversampling does not reduce the speed of the algorithm unlike other algorithms. For this assignment, we only use the most basic K-nearest neighbors for simplicity purposes.

For the base model before the hyperparameter tuning (K values), we used K-value of 3

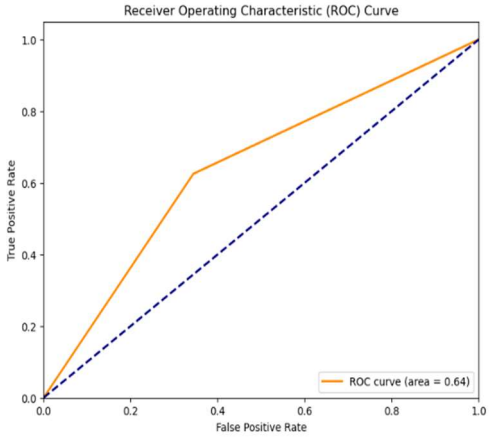
```
# Create KNN model with default k value
knn = KNeighborsClassifier(n_neighbors=3)

# Train the model on resampled training data
knn.fit(X_train_resampled, y_train_resampled)

# Validate the model on validation dataset
y_pred = knn.predict(X_validation_scaled)

# Generate classification report
print("Classification Report:")
print(classification_report(y_validation, y_pred))

# Test the model on the test dataset
y_pred_test = knn.predict(X_test_scaled)
```

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

           0       0.91      0.75      0.82     45126
           1       0.18      0.41      0.25      5940

    accuracy                           0.71     51066
   macro avg       0.54      0.58      0.54     51066
weighted avg       0.82      0.71      0.76     51066
```

fter tuning the model, we found out that the best K-values in terms of f1-score is 37.

```python
# Initialize variables to store best k value and corresponding f1-score
best_k = None
best_f1_score = 0

# Initialize lists to store f1-scores and ROC AUC scores for each k
f1_scores = []
roc_auc_scores = []

# Loop through k values from 11 to 41
for k in range(11, 42):
    # Create KNN model with current k value
    knn = KNeighborsClassifier(n_neighbors=k)

    # Train the model on resampled training data
    knn.fit(X_train_resampled, y_train_resampled)

    # Validate the model on validation dataset
    y_pred = knn.predict(X_validation_scaled)

    # Calculate f1-score
    f1 = f1_score(y_validation, y_pred)

    # Store f1-score for current k value
    f1_scores.append(f1)

    # Store ROC AUC score for current k value
    roc_auc_scores.append(roc_auc_score(y_validation, y_pred))

    # Check if current f1-score is better than the best f1-score
    if f1 > best_f1_score:
        best_f1_score = f1
        best_k = k

# Print the best k value and corresponding f1-score
print("Best k value:", best_k)
print("Best f1-score:", best_f1_score)

# Create KNN model with the best k value
best_knn = KNeighborsClassifier(n_neighbors=best_k)

# Train the final model on the resampled training data
best_knn.fit(X_train_resampled, y_train_resampled)

# Test the final model on the test dataset
y_pred_test = best_knn.predict(X_test_scaled)
```

```
Best k value: 37
Best f1-score: 0.2922504126204276
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.64      0.76     45126
           1       0.19      0.64      0.29      5940

    accuracy                           0.64     51066
   macro avg       0.56      0.64      0.53     51066
weighted avg       0.85      0.64      0.70     51066
```



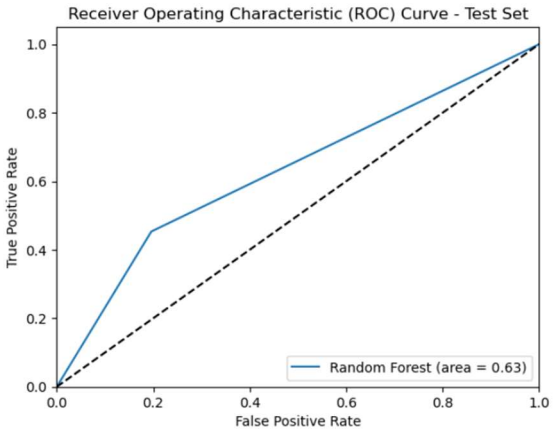
Random Forest

Random Forest is an ensemble learner primarily used for both classification and regression. It operates by constructing numerous decision trees during training, each trained on a random subset of the data and features. At each node of the tree, the algorithm selects the best split among a random subset of features. During prediction, the algorithm combines the results of all the trees by majority voting for classification tasks or averaging for regression tasks. One of our main experiences with Random Forest is how much it relied on tuning and cost-sensitive training (class weights) for it to be effective for these datasets. The upside is that compared to other algorithms, it is very fast and reliable, provided you have the correct parameters to train it. Before tuning, random forest's performance are subpar at best, better than most other untuned algorithms but it takes a long time for you to find the right parameters for this machine learning model. Here are the snippets of my code for untuned random forest

```python
# Define the classifier with default parameters
rf_classifier = RandomForestClassifier(random_state=13)

# Train the model on resampled training data
rf_classifier.fit(X_train_resampled, y_train_resampled)

# Predict on the validation data
y_pred_validation = rf_classifier.predict(X_validation_scaled)

# Generate classification report for validation data
print("Validation Performance:")
print(classification_report(y_validation, y_pred_validation))
```

Testing Performance:

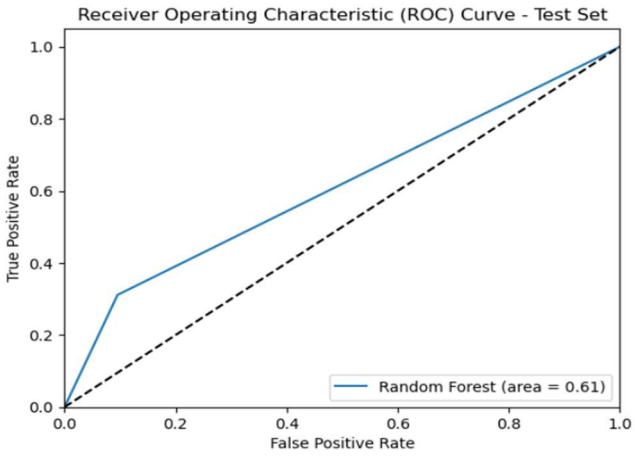
	precision	recall	f1-score	support
0	0.92	0.80	0.86	45126
1	0.23	0.45	0.31	5940
accuracy			0.76	51066
macro avg	0.58	0.63	0.58	51066
weighted avg	0.84	0.76	0.79	51066



Randomized search is not very suitable for random forest. However, for the sake of time, I reluctantly include it in the report. As you can see, the performance worsens after tuning.

```
# Define the classifier with default parameters
rf_classifier = RandomForestClassifier(random_state=13, criterion='entropy', class_weight='balanced', n_estimators=100)

# Define the parameter distribution for Randomized Search
param_dist = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [None, 10, 20],  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split a node
    'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required at each leaf node
}

# Set up Stratified K-Fold cross-validation
stratified_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Set up the random search with stratified cross-validation on the training data
random_search = RandomizedSearchCV(estimator=rf_classifier, param_distributions=param_dist, scoring='f1',
                                   cv=stratified_cv, n_iter=10, n_jobs=-1, random_state=42, verbose=2)

# Start timer
start_time_rf1 = time.time()

# Execute the random search on the resampled training data
random_search.fit(X_train_resampled, y_train_resampled)

# Output the best parameters and corresponding score
print("Best parameters found: ", random_search.best_params_)
print("Best F1 score found: ", random_search.best_score_)

# Predict on the validation data using the best model from random search
best_rf = random_search.best_estimator_
y_pred_validation = best_rf.predict(X_validation_scaled)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found:  {'n_estimators': 100, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_depth': None}
Best F1 score found:  0.9138147014294242
Validation Performance
              precision    recall  f1-score   support

           0       0.91      0.90      0.91     45145
           1       0.29      0.30      0.30      5920

    accuracy                           0.83     51065
   macro avg       0.60      0.60      0.60     51065
weighted avg       0.84      0.83      0.84     51065
```



The best hyperparameters I found for the random forest throughout my months of investigations are listed on the table below. This model yields a more respectable result.

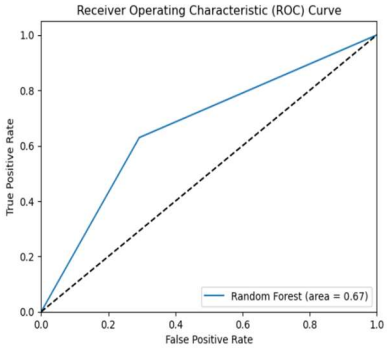
Best hyperparameters for random forest

criterion	entropy
Class weights	balanced
N Estimators	300
Max depth	10
Minimum sample splits	5
Minimum sample leaf	2

```
# Define the classifier with the specified parameters
rf_classifier_2 = RandomForestClassifier(random_state=13, criterion='entropy', class_weight='balanced',
                                         n_estimators=300, max_depth=10, min_samples_split=5, min_samples_leaf=2)

# Start timer
start_time_rf2 = time.time()

# Define the StratifiedKFold
stratified_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Perform stratified cross-validation
y_train_pred_cv = cross_val_predict(rf_classifier_2, X_train_resampled, y_train_resampled, cv=stratified_kfold)

# Calculate and print evaluation metrics for the training set
print("Training Performance (Cross-Validation)")
print(classification_report(y_train_resampled, y_train_pred_cv))

# Validate on the validation dataset
rf_classifier_2.fit(X_train_resampled, y_train_resampled)  # Fit the model on the entire training set
y_val_pred = rf_classifier_2.predict(X_validation_scaled)
```

Testing Performance

	precision	recall	f1-score	support
0	0.94	0.71	0.81	45126
1	0.22	0.63	0.33	5940
accuracy			0.70	51066
macro avg	0.58	0.67	0.57	51066
weighted avg	0.85	0.70	0.75	51066

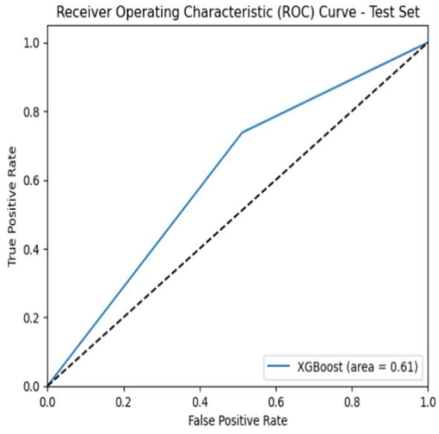


XGBoost

XGBoost, short for Extreme Gradient Boosting belongs to the family of gradient boosting algorithms, which sequentially trains many weak learners (typically decision trees) and ensembles them to minimize a predefined loss function. In that sense, it is very similar to Random Forest. XGBoost optimizes performance by employing tree pruning and column subsampling to prevent overfitting. Like its cousin, Random Forest, it requires exhaustive tuning to find the right parameters.

These are the snippets of code before tuning:

```
# Define the classifier with default parameters
xgb_classifier = XGBClassifier(random_state=13)

# Train the model on resampled training data
xgb_classifier.fit(X_train_resampled, y_train_resampled)

# Predict on the validation data
y_pred_validation = xgb_classifier.predict(X_validation_scaled)

# Generate classification report for validation data
print("Validation Performance:")
print(classification_report(y_validation, y_pred_validation))
```



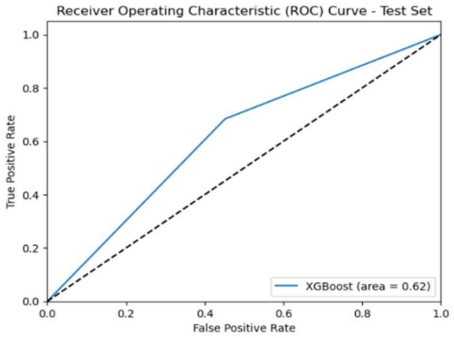

```
Testing Performance:
              precision    recall  f1-score   support

           0       0.93      0.49      0.64     45126
           1       0.16      0.74      0.26      5940

    accuracy                           0.52     51066
   macro avg       0.55      0.61      0.45     51066
weighted avg       0.84      0.52      0.60     51066
```

These are the snippets of the code after tuning.

```
# Define the classifier with default parameters
xgb_classifier = xgb.XGBClassifier(random_state=42)
# Define the parameter distribution for Randomized Search
param_dist_xgb = {
    'n_estimators': [100, 200, 300],  # Number of boosting rounds
    'learning_rate': [0.01, 0.05, 0.1],  # Boosting learning rate
    'max_depth': [3, 4, 5],  # Maximum depth of a tree
    'min_child_weight': [1, 3, 5],  # Minimum sum of instance weight (hessian) needed in a child
    'subsample': [0.8, 0.9, 1.0],  # Subsample ratio of the training instances
    'colsample_bytree': [0.8, 0.9, 1.0],  # Subsample ratio of columns when constructing each tree
    'gamma': [0, 0.1, 0.2],  # Minimum loss reduction required to make a further partition on a leaf node of the tree
}

# Set up the random search with stratified cross-validation on the training data
random_search_xgb = RandomizedSearchCV(estimator=xgb_classifier, param_distributions=param_dist_xgb,
                                       scoring='f1', cv=stratified_cv, n_iter=10, n_jobs=-1, random_state=42, verbose=2)

# Start timer
start_time_xgb = time.time()

# Execute the random search on the resampled training data
random_search_xgb.fit(X_train_resampled, y_train_resampled)

# Output the best parameters and corresponding score
print("Best parameters found: ", random_search_xgb.best_params_)
print("Best F1 score found: ", random_search_xgb.best_score_)

# Predict on the validation data using the best model from random search
best_xgb = random_search_xgb.best_estimator_
y_pred_validation_xgb = best_xgb.predict(X_validation_scaled)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found:  {'subsample': 0.9, 'n_estimators': 300, 'min_child_weight': 1, 'max_depth': 5, 'learning_rate': 0.1, '
amma': 0.1, 'colsample_bytree': 0.8}
Best F1 score found:  0.8432849377928953
```

```
Testing Performance
              precision    recall  f1-score   support

           0       0.93      0.55      0.69     45126
           1       0.17      0.68      0.27      5940

    accuracy                           0.56     51066
   macro avg       0.55      0.62      0.48     51066
weighted avg       0.84      0.56      0.64     51066
```



As you can see by the results. It didn't improved by much

Best hyperparameter for XGBoost

criterion	entropy
Subsample	0.9
N Estimators	300
Min child weight	1
Max depth	5
Learning rate	0.1
Gamma	0.1
Col Sample By Tree	0.8

Gradient Boosting

Gradient Boosting is a machine learning ensemnle technique that combines the predictions of multiple weak learners, typically decision trees sequentially. It aims to improve overall predictive performance by optimizing the model's weights based on the errors of previous iterations, gradually reducing prediction errors and enhancing the model's accuracy. This is the code snippet for gradient boosting classifier from sklearn.

```
gb = GradientBoostingClassifier()
gb.fit(X_train , Y_train)

find_metrics("Training", Y_train , gb.predict(X_train))
find_metrics("Validation", Y_val , gb.predict(X_val))
find_metrics("Testing", Y_test , gb.predict(X_test))
```

These are the results for base gradient boosting classifier. The result is very good at predicting non default class but very bad at predicting default class. To mitigate this result , we have decided to perform hyperparameter tuning.

```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     45126
           1       0.60      0.05      0.10      5940

    accuracy                           0.89     51066
   macro avg       0.75      0.52      0.52     51066
weighted avg       0.86      0.89      0.84     51066

ROC AUC for Gradient Boosting (Test): 0.52
```

```python
# Define the classifier with default parameters and balanced class weights
gb_classifier = GradientBoostingClassifier(random_state=13)

# Define the parameter distribution for Randomized Search
param_dist_gb = {
    'n_estimators': [100, 200, 300],  # Number of boosting stages
    'learning_rate': [0.01, 0.05, 0.1],  # Learning rate shrinks the contribution of each tree
    'max_depth': [3, 4, 5],  # Maximum depth of the individual regression estimators
    'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required to be at a leaf node
    'subsample': [0.8, 0.9, 1.0]  # Fraction of samples used for fitting the individual base learners
}
```

These are the hyperparameters that is used to tune gradient boosting classifier. As you can see the hyperparameters are similar to decision tree and random forest. We have utilized RandomizedSearchCV to find the best parameters as it is much more time efficient than GridSearchCV.

These are the best hyperparameters for gradient boosting classifier.

```
Best parameters found:  {'subsample': 0.9, 'n_estimators': 200, 'min_samples_split': 2, 'min_samples_leaf': 2, 'max_depth': 4, 'learning_rate': 0.05}
```

These are the results of Gradient Boosting after hyperparameter tuning.

Testing Performance

	precision	recall	f1-score	support
0	0.93	0.80	0.86	45126
1	0.25	0.52	0.34	5940
accuracy			0.77	51066
macro avg	0.59	0.66	0.60	51066
weighted avg	0.85	0.77	0.80	51066

The result overall improved as the recall for default class increased but performance. However , further improvement can be done. Thus, we have opted to combine multiple base models to create a generalized model. This technique is called ensembling learning where you can see in the next section.

Stacking Ensemble

Stacking is an ensemble technique combines multiple base models to create a generalised model. In stacking, the predictions of diverse base models are used as input features for a meta-model, which then makes the final prediction. This meta-model learns to weigh the predictions of the base models based on their performance, effectively leveraging the strengths of each base model while mitigating their individual weaknesses. But the flaw with this approach is that it requires many models to create an effective meta-model, which in this case 8 models are certainly not enough. Below is result I got from layer 0 ensemble which comprised of Random Forest, SVC, KNN, Naïve Bayes, XGBoost, Gradient Boost, Logistic regression and neural network model. Needless to say, the performance is subpar at best.

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

           0       0.91      0.86      0.88     45126
           1       0.24      0.34      0.28      5940

    accuracy                           0.80     51066
   macro avg       0.58      0.60      0.58     51066
weighted avg       0.83      0.80      0.81     51066

Test ROC AUC: 0.6012016857848529
```

3-layer stacking ensemble

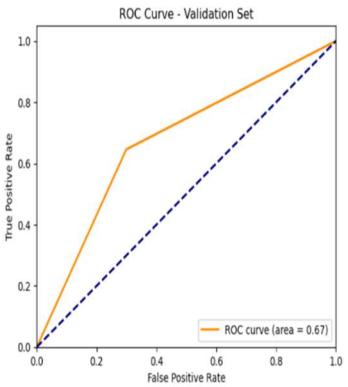
For this model we added two additional layer, layer 1 being Neural Network, Gradient Boosting and Random Forest and layer 2 being Neural Network, stacked on top of the base layer that we previously mentioned above. The result did improved but we determined that its metrics is still not good enough.

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

           0       0.92      0.76      0.83     45126
           1       0.22      0.51      0.30      5940

    accuracy                           0.73     51066
   macro avg       0.57      0.63      0.57     51066
weighted avg       0.84      0.73      0.77     51066

Test ROC AUC: 0.6318826142021197
```

Voting classifier

Voting classifier is basically taking all of the output of the input models, combines and make a final decision. Since we do not have enough models to make the stack ensemble sufficiently robust, voting classifier shows itself as a solid alternative. Here, we take the 6 best models and ensemble them into one voting classifiers, that being Random Forest, Logistic regression, Naïve bayes, Gradient Boost, Neural Network and SVM. The result is satisfactory, though not the best result we have gotten so far, it is reasonably robust and it has decent metrics and speed.

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

           0       0.94      0.70      0.80     45126
           1       0.22      0.65      0.33      5940

    accuracy                           0.70     51066
   macro avg       0.58      0.68      0.57     51066
weighted avg       0.86      0.70      0.75     51066
```

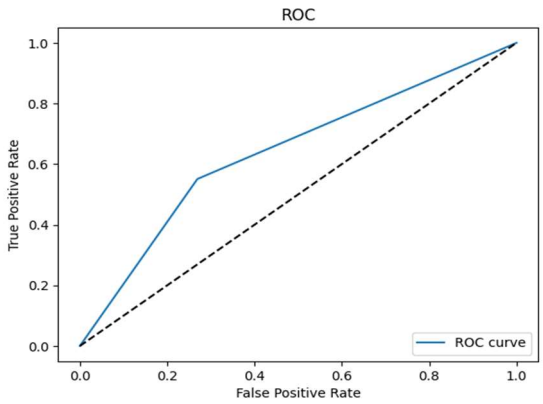


Neural Networks

Neural networks are a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. The model we first developed was the MLPClassifier from sklearn.MLPClassifier implements a multi layer perceptron algorithm that trains using backpropagation.

```
mlp = MLPClassifier()
mlp.fit(X_train , Y_train)
find_metrics("Training", Y_train , mlp.predict(X_train))
find_metrics("Testing", Y_test , mlp.predict(X_test))
```

```
Classification Report for Neural Network (Test):
              precision    recall  f1-score   support

           0       0.94      0.66      0.78     45126
           1       0.21      0.67      0.31      5940

    accuracy                           0.66     51066
   macro avg       0.57      0.66      0.55     51066
weighted avg       0.85      0.66      0.72     51066

ROC AUC for Neural Network (Test): 0.66
```



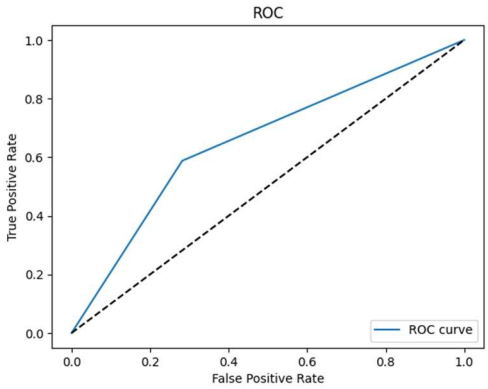
The accuracy is high compared to the rest of the model.However, we also wanted to implement cost sensitive learning , an approach where the misclassification costs of different classes are explicitly considered during model training.Instead of treating all errors equally, assigning a different penalty weight for each class. However, sklearn did not provide the class_weight attribute to perform cost sensitive learning. To mitigate this , we implemented neural networks using Tensorflow Keras.

```
class_weights = compute_class_weight(class_weight="balanced",classes=np.unique(Y_train), y=Y_train )
class_weights = dict(enumerate(class_weights))
model = Sequential()
model.add(Dense(7, activation='relu', input_dim=7, name="Inputlayer", kernel_initializer='he_uniform'))
model.add(Dense(64, activation="relu"))
model.add(Dense(32, activation="relu"))
model.add(Dense(2, activation="softmax", name="OutputLayer"))
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

model.compile(loss='binary_crossentropy', optimizer="adam",  metrics=['accuracy', AUC(),Recall() ])
model.fit(X_train, Y_train_one_hot,  epochs=30,  batch_size=100 , class_weight=class_weights ,
          validation_data=(X_val , to_categorical(Y_val)) , callbacks=[early_stopping] )
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
Inputlayer (Dense)           (None, 7)                 56

dense (Dense)                (None, 64)                512

dense_1 (Dense)              (None, 32)                2080

OutputLayer (Dense)          (None, 2)                 66

=================================================================
Total params: 2714 (10.60 KB)
Trainable params: 2714 (10.60 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This is the code snippet for the Tensorflow Neural Networks. There are 1 Input layer , 1 Output layer with SOFTMAX activation function for binary classification and 2 hidden layers with RELU activation function. There are a total of 2714 trainable parameters. We have also utilized adam algorithm as the optimization algorithm to minimize the loss function. We have used 30 epochs (feedforward iteration) to train the neural network where in each iteration the validation dataset is used to measure the loss function.

```
Classification Report

              precision    recall  f1-score   support

           0       0.93      0.72      0.81     45050
           1       0.22      0.59      0.32      6016

    accuracy                           0.70     51066
   macro avg       0.57      0.65      0.56     51066
weighted avg       0.85      0.70      0.75     51066

AUC SCORE : 0.652956101524311
```



As you can see, the accuracy, AUC score , recall , F1-score in neural networks using Tensorflow Keras is slightly better than MLPClassifier.These are the results using MLPClassifier and Neural Networks using Tensorflow Keras.

Metrics	MLPClassifier	Neural Networks using Tensorflow Keras
Accuracy	0.66	**0.68**
Recall	0.67	**0.68**
F1-Score	0.31	**0.33**
AUC score	0.66	**0.6798**
Precision	0.21	**0.22**

Conclusion

The most significant improvement to our models came from two main strategies: oversampling and feature reduction. These steps were crucial because they helped address some major issues and made our models work much better.Firstly, oversampling helped fix a problem called class imbalance, where one type of data was much rarer than the others. By creating more examples of the rare data, our models could learn better from all types of examples.Secondly, feature reduction simplified our data by picking out the most important parts and getting rid of the rest. This made our models more focused and less likely to get confused by irrelevant information.Overall, using oversampling and feature reduction together made a big difference in how well our models performed. They helped us tackle tough problems and make better predictions.Overall , we have used Area under the curve metrics to determine the best model. The best model we encountered was Neural Networks.