Semester Project - Home Credit Default Risk

Author: Utsav Patel

Abstract
A bank provides loan only if the credit history of a customer is good. Cases when the credit history is not available, the bank has to rely on a hypothesis in order to provide loan to such customers. Our model can predict if a customer will default or not. It uses the historical loan application data to train. The challenge here is that there is more information on the non defaulters than on the defaulters in this dataset. We deal with several models and techniques and compare their performance and finally come up with the best model.

Contents

Introduction

Our project aims to use historical loan application data to predict whether or not an underserved applicant (a person with insufficient or no credit history) will be able to repay a loan. With an efficient model such as this, the banks and financial institutions can target just the potential customers. This will not only allow the banks to avoid spending resources unnecessarily but also provide a positive and safe borrowing [5] experience for the customer. The objective is especially eye catching considering the increase in the financial institutions over the years. We implement many supervised learning techniques such as Logistic Regression, Random Forest, K Nearest Neighbors, Decision Tree, Light GBM and XGBoost and compare each one of the results based on evaluation metrics We then choose the most efficient technique.

1. Models and Methodology

We use only the application train data since including other tables features lead to a reduction in their performance. LightGBM on the other hand uses the other tables and provides better accuracy as compared to XGBoost model.

We also mentioned SVM in proposal but are instead implementing Decision Tree since Decision Tree perform better on unbalanced data.

Below are the models we have implemented:
Logistic Regression
Random Forest
Decision Tree
K Nearest Neighbors
XGBoost

1.1 Exploratory Data Analysis:

The training data has 307511 observations (each one a separate loan) and 122 features including the TARGET (the label we want to predict).

1) **Imbalance**
For EDA first we examine the distribution of the Target Column : 0 -282686 and 1 - 24825,where 0 indicates the loan was repaid on time and 1 indicates the [4] client had payment difficulties. Imbalance-There are far more loans that were repaid on time than loans that were not repaid. i.e. there are more samples from class 0 than class1.
Below are considerations to tackle this imbalance issue:

- We analyse (later) various imbalance techniques with respect to each model and see which technique works for which model.
- We use only the f1_score and recall in order to determine and compare the different models and find the best working model. This is because if we classify a defaulter as non defaulter (i.e.1 as 0), it is much worse

than classifying a non defaulter as defaulter(i.e.0 as 1). i.e. we are more interested in having more true positives but at the same time have one of the better f1_scores.

Handle the imbalance issue: When the samples between the two classes are not balanced then the model is more liable to learn about the majority class. This results in poorer classification of the minority class since there are not sufficient data available from the minority class. This behavior becomes more intense if the ratio of the majority to minority is very high – which is our case. There might be some models like decision tree that are exceptions to this scenario.

We have applied four approaches to handle this issue:

- Oversampling:
  In this method the data from the minority class is replicated so that a balance can be established between the two classes.
- Undersampling:
  In this method the data from the majority class is removed in order to balance the data.
- Synthetic minority oversampling technique:
  The over sampling of the minority dataset is done synthetically using an algorithm. Data from the minority class is not replicated. This prevents over fitting which is prevalent in oversampling.
- Improve the cost function:
  There are several approaches-[1] one which we are using is class weights. The class weights are added to the minority class so that the cost function accounts more for the error in minority class.
  To determine the best f1_score at each noise level we do a grid search and get the best model to test against the test data in step2 explained in approaches section.

Noise levels for each imbalance technique:

- Sampling strategy- It is the ratio of the number of samples in minority class over the number of samples in the majority class. This parameter is used for OverSampling, UnderSampling and SMOTE.
- Class weights- It specifies the weights to be given to the error from each class in the cost function. We set more weights on the minority class in order to make the model more sensitive to its errors on the minority class. This noise level is applicable for the cost function based approach.

2) **Anomalies:**$_4$ Using the analysis from the above unique values we treat the anomalies:

- DAYS_EMPLOYED- The maximum value - 365243 is about 1000 years. So we replaced all anomalous values with nan.
- CODE_GENDER- Replace the value XNA with nan since M and F are the only possible values
- DAYS_LAST_PHONE_CHANGE-Replace 0 with nan since 0 is not a possible value for this column

3) **Label Encoding:**

Next we encode the Categorical Variables:

- Label Encoding for any categorical variables with only 2 categories
- One-Hot Encoding for any categorical variables with more than 2 categories. In total - 3 columns were label encoded.

4) **Missing Values:**

Next we look at the number and percentage of missing values in each column There are 67 columns that have missing values. We used 3 different techniques for handling missing values. ([4] Note the code for finding the missing values are referenced but the strategies and how we treat them is our approach)
Below are details on the strategies we tried to handle the missing values.

- -999 strategy:
  we replace every missing value with -999.
- mean mode approach:
  For every column with missing values (say eg. col):
  Step1) Create a new column that has value 1 if col has missing value and has 0 otherwise.
  Step2) Replace every missing value in col with mean if it is a numerical column and every missing value in categorical column with mode.

  This method works for many reasons, firstly the new column added gives new information on missing values which it can use to give better accuracy.
- Strategic imputer:
  Below are the steps:
  We identify the below datasets
  DatasetA-There are 76 continous non categorical data
  DatasetB-There are 46 categorical and continuous data.

  We treat both datasets differently:
  DatasetA:
  - It has min value 0 and max value 1 – The missing

values are replaced with the mode of the column DatasetB:
- If it is an object perform one hot encoding
- If it is float then do mode of that column

- Failed Attempts:
  1) Aggregation
  We aggregate the numeric columns based on the group by columns using the method mean or max.

  Group by columns - numeric columns- method
  1)'CODE_GENDER', 'NAME_EDUCATION_TYPE' - AMT_ANNUITY - max
  'CODE_GENDER', 'ORGANIZATION_TYPE' -AMT_INCOME_TOTAL,DAYS_REGISTRATION -mean
  2)'CODE_GENDER', 'REG_CITY_NOT_WORK_CITY'- 'CNT_CHILDREN' - mean
  3)'NAME_EDUCATION_TYPE', 'OCCUPATION_TYPE' - AMT_CREDIT, AMT_REQ_CREDIT_BUREAU_YEAR, APARTMENTS_AVG, BASEMENTAREA_AVG, NONLIVINGAREA_AVG, OWN_CAR_AGE, YEARS_BUILD_AVG -mean
  4) 'NAME_EDUCATION_TYPE', 'OCCUPATION_TYPE', 'REG_CITY_NOT_WORK_CITY'-ELEVATORS_AVG –mean

  The resulting aggregated column has more correlation with the target column and works for some models like Random Forest but might increase correlation with the group by columns.

  2) Normalization- We tried normalizing the data in application train but it reduced the performance hence was not used.

1.2 APPROACHES:

Experiment to find the best missing value strategy:

- Step1) Treat the anomalies.
- Step2) Perform the label encoding and one hot encoding on the categorical columns
- Step3) Find the columns containing missing values- (this includes the newly one hot encoded columns)
- Step4) Apply the -999 missing value strategy
- Step5) Split the data into train and test set
- Step6) Perform tree based feature selection on the train data. Find the feature importances. Based on these feature importances we subset the train data and the test data.
- Step7) Perform a grid search for each model and find the best possible accuracy and the best hyper parameters.
- Step8) Perform steps 1 to 6 for other missing value strategies
- Step9) We use the hyper parameters found in Step7. If the performance is poor than -999 strategy , only then try to find the best hyperparameters for that missing value strategy.
- Step10) Based on the results obtained we decide which method best works for which model.

Experiment on the techniques to handle imbalance:

- Step1) Perform the EDA of the dataset- anomalies and label encoding. Use the best performing missing value strategy for each model.
- Step2) split the dataset into train and test
- Step3) Only for the train dataset, do step4 onwards.
- Step4) For each model, iterate over different levels of noise and repeat step 5 to 6 for each iteration
- Step5) Perform feature engineering on the resampled train set.
- Step6) Grid Search at each sampling strategy in order to fetch the best possible accuracy (f1_score) at each noise level. To determine the best f1_score for each noise we use the best grid model against the test data in step2.
- Step7) Finally find the sweet spot - the value of sampling strategy where the model gives the highest f1 score.
- Step8) Perform steps 2 to 7 for each imbalance technique and observe the best solutions for each model.

1.3 Approach to combine all 7 tables:

Combining all tables was really a great issue for us. This was because it was having too much of Missing values. The Missing value % for a column was as high as 70% in many cases. Below is the image for the same.

```
# Missing values statistics
missing_values = missing_values_table(app_train)
missing_values
```

```
Your selected dataframe has 122 columns.
There are 67 columns that have missing values.
```

	Missing Values	% of Total Values
COMMONAREA_MEDI	214865	69.9
COMMONAREA_AVG	214865	69.9
COMMONAREA_MODE	214865	69.9
NONLIVINGAPARTMENTS_MEDI	213514	69.4
NONLIVINGAPARTMENTS_MODE	213514	69.4
NONLIVINGAPARTMENTS_AVG	213514	69.4
FONDKAPREMONT_MODE	210295	68.4
LIVINGAPARTMENTS_MODE	210199	68.4
LIVINGAPARTMENTS_MEDI	210199	68.4
LIVINGAPARTMENTS_AVG	210199	68.4
FLOORSMIN_MODE	208642	67.8
FLOORSMIN_MEDI	208642	67.8
FLOORSMIN_AVG	208642	67.8
YEARS_BUILD_MODE	204488	66.5
YEARS_BUILD_MEDI	204488	66.5
YEARS_BUILD_AVG	204488	66.5
OWN_CAR_AGE	202929	66.0

This was true for all 7 tables. Moreover, if we combine those tables into one, all of them would be having much higher missing values.

• Adding useful features: For almost all tables we entered our own columns by performing some logical operations on two columns that were already present. As a result at the end We had many self created columns in top 50 features from total of 798 features that we got after combining all tables.

• Using Mean for person's entries: Suppose a normal distribution. If we are having 1 Million entries of ages. Now suppose, we have to estimate another 1 Million data of ages that are missing. Any approach you got in mind? Best approach would be to assume that all have ages of around current mean which would be best estimate for missing values. I used this approach in indirectly filling NA's. I did grouping by SK_ID_BUREAU and then found mean and entered as a column in separate dataframe that is used later. Thus if there are 50 entries of particular SK_ID_BUREAU, corresponding to those all, we will mean all values of other columns and make it as a entry.

	MONTHS_BALANCE	SK_ID_BUREAU	STATUS_0	STATUS_1	STATUS_2	STATUS_3	STATUS_4	STATUS_5	STATUS_C	STATUS_X
	mean	mean	mean	mean	mean	mean	mean	mean	mean	mean
SK_ID_BUREAU										
5001709	-48.0	5001709	0.000000	0.0	0.0	0.0	0.0	0.0	0.886598	0.113402
5001710	-41.0	5001710	0.060241	0.0	0.0	0.0	0.0	0.0	0.578313	0.361446
5001711	-1.5	5001711	0.750000	0.0	0.0	0.0	0.0	0.0	0.000000	0.250000
5001712	-9.0	5001712	0.526316	0.0	0.0	0.0	0.0	0.0	0.473684	0.000000
5001713	-10.5	5001713	0.000000	0.0	0.0	0.0	0.0	0.0	0.000000	1.000000

• Min mean max var: Moreover for some tables , we found the mean, minimum, maximum and variance of columns of ID's by grouping them by SK_ID_CURR (for other tables this ID might differ).

```
bureau_agg.head()
```

	DAYS_CREDIT				DAYS_CREDIT_ENDDATE			DAYS_CREDIT_UPDATE	CREDIT_DAY_OVERDUE		...	CREDIT_TYPE_r
	min	max	mean	var	min	max	mean	mean	max	mean	...	mean
SK_ID_CURR												
100001	-1572	-49	-735.000000	240043.666667	-1329.0	1778.0	82.428571	-93.142857	0	0.0	...	
100002	-1437	-103	-874.000000	186150.000000	-1072.0	780.0	-349.000000	-499.875000	0	0.0	...	
100003	-2586	-606	-1400.750000	827783.583333	-2434.0	1216.0	-544.500000	-816.000000	0	0.0	...	
100004	-1326	-408	-867.000000	421362.000000	-595.0	-382.0	-488.500000	-532.000000	0	0.0	...	
100005	-373	-62	-190.666667	26340.333333	-128.0	1324.0	439.333333	-54.333333	0	0.0	...	

5 rows × 58 columns

• Replacing anomalies: For some tables, we had to replace anomalies like age being 365243 and such entries with nan. We replaced it with nan because XGBoost and LightGBM handles missing values efficiently by guessing value that reduces log loss error rather than us imputing them with values that might increase log loss error.

```
prev = pd.read_csv('previous_application.csv')
prev, cat_cols = one_hot_encoder(prev, nan_as_category= True)
# Days 365243 values -> nan
prev['DAYS_FIRST_DRAWING'].replace(365243, np.nan, inplace= True)
prev['DAYS_FIRST_DUE'].replace(365243, np.nan, inplace= True)
prev['DAYS_LAST_DUE_1ST_VERSION'].replace(365243, np.nan, inplace= True)
prev['DAYS_LAST_DUE'].replace(365243, np.nan, inplace= True)
prev['DAYS_TERMINATION'].replace(365243, np.nan, inplace= True)
```

• Using Credit Active and Credit Inactive column: For bureau table we used credit active and inactive credit as a useful information and then we approached it in a manner explained in Mean, min, max and var (only for numerical columns).

STEP 1:

```
active = bureau[bureau['CREDIT_ACTIVE_Active'] == 1]
```

```
active.head()
```

	SK_ID_CURR	DAYS_CREDIT	CREDIT_DAY_OVERDUE	DAYS_CREDIT_ENDDATE	DAYS_ENDDATE_FACT	AMT_CREDIT_MAX_OVERDUE	CNT_CREDIT_PR
1	215354	-208	0	1075.0	NaN	NaN	
2	215354	-203	0	528.0	NaN	NaN	
3	215354	-203	0	NaN	NaN	NaN	
4	215354	-629	0	1197.0	NaN	77674.5	
5	215354	-273	0	27460.0	NaN	0.0	

5 rows × 50 columns

```
active_agg = active.groupby('SK_ID_CURR').agg(num_aggregations)
```

STEP 2:

```
active_agg = active.groupby('SK_ID_CURR').agg(num_aggregations)
active_agg.head()
```

	DAYS_CREDIT				DAYS_CREDIT_ENDDATE		
	min	max	mean	var	min	max	mean
SK_ID_CURR							
100001	-559	-49	-309.333333	65110.333333	411.0	1778.0	1030.333333
100002	-1042	-103	-572.500000	440860.500000	780.0	780.0	780.000000
100003	-606	-606	-606.000000	NaN	1216.0	1216.0	1216.000000
100005	-137	-62	-99.500000	2812.500000	122.0	1324.0	723.000000
100008	-78	-78	-78.000000	NaN	471.0	471.0	471.000000

5 rows × 23 columns

We did same for inactive credit entries also.

• We approached in above mentioned 5 techniques for all the tables and at the end we had our dataframe exported to csv file.

```
df=app_train
df.shape
```

```
(307507, 260)
```

```
df = df.join(bureau_agg, how='left', on='SK_ID_CURR')
del bureau_agg
```

```
df.shape
```

```
(307507, 364)
```

```
df = df.join(prev_agg, how='left', on='SK_ID_CURR')
del prev_agg
df.shape
```

```
(307507, 613)
```

```
df = df.join(pos_agg, how='left', on='SK_ID_CURR')
del pos_agg
df.shape
```

```
(307507, 631)
```

```
df = df.join(ins_agg, how='left', on='SK_ID_CURR')
del ins_agg
df.shape
```

```
(307507, 657)
```

```
df = df.join(cc_agg, how='left', on='SK_ID_CURR')
del cc_agg
```

```
df.shape
```

```
(307507, 798)
```

1.4 MODEL IMPLEMENTATION:

Below are the models we have implemented:
Logistic Regression
Random Forest
Decision Tree
K Nearest Neighbors
XGBoost

Performance issues: There are two key factors that influence the performance of the models:

- The data is extremely unbalanced. Class 0 data is more than 10 times Class 1 in the training set:
- The strategy applied to handle the missing values (0: 226132, 1: 19876)

2. EXPERIMENTS AND RESULTS:

2.1 Experiment to find the best missing value strategy:

1) Replace missing with -999

K-Nearest Neighbor 1 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	1.000000	1.000000	1.00000	1.000000
Test	0.104965	0.108507	0.106706	0.51377	0.853812

K-Nearest Neighbor 3 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.678000	0.190833	0.298000	0.933968	0.927295
Test	0.126277	0.034957	0.054755	0.528867	0.902883

Random Forest model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.988000	0.817368	0.895000	0.994287	0.984448
Test	0.213806	0.045060	0.074433	0.594956	0.909826

Decision Tree model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.999000	0.995069	0.997000	0.999995	0.999537
Test	0.129405	0.146898	0.137598	0.530103	0.851828

Logistic Regression model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.146000	0.486265	0.225000	0.670771	0.729525
Test	0.149256	0.496868	0.229556	0.675434	0.731623

2) Replace missing values with -999 after aggregation technique

K-Nearest Neighbor 1 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	1.000000	1.000000	1.000000	1.000000
Test	0.093682	0.096181	0.094915	0.507377	0.852397

K-Nearest Neighbor 3 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.679000	0.191789	0.299000	0.934430	0.927372
Test	0.101116	0.029299	0.045433	0.519543	0.900932

Random Forest model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.988000	0.814047	0.893000	0.993971	0.984187
Test	0.232195	0.048091	0.079679	0.599819	0.910606

Decision Tree model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	0.991095	0.99500	0.999988	0.999264
Test	0.131888	0.147505	0.13926	0.531510	0.853275

Logistic Regression model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.145000	0.470165	0.222000	0.664585	0.733342
Test	0.148612	0.493433	0.228427	0.669476	0.731769

3) Mean Mode Imputation:

K-Nearest Neighbor 1 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	1.000000	1.000000	1.00000	1.000000
Test	0.105845	0.108305	0.107061	0.51412	0.854625

K-Nearest Neighbor 3 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.678000	0.193349	0.301000	0.934129	0.927409
Test	0.121681	0.033340	0.052339	0.527903	0.902850

Random Forest model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.986000	0.817418	0.89400	0.994146	0.984314
Test	0.241996	0.051930	0.08551	0.600245	0.910622

Decision Tree model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.999000	0.975750	0.987000	0.999824	0.997927
Test	0.142857	0.159022	0.150507	0.535195	0.855552

Logistic Regression model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.182000	0.592473	0.278000	0.747326	0.751813
Test	0.181003	0.595676	0.277642	0.748306	0.750581

4) Mean Mode imputation after aggregation technique

K-Nearest Neighbor 1 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	1.000000	1.000000	1.000000	1.000000
Test	0.093357	0.094565	0.093957	0.507099	0.853243

K-Nearest Neighbor 3 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.680000	0.193097	0.301000	0.93468	0.927470
Test	0.107649	0.030713	0.047791	0.51888	0.901517

Random Forest model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.989000	0.813896	0.89300	0.993799	0.984220
Test	0.219203	0.048899	0.07996	0.601270	0.909452

Decision Tree model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.999000	0.976253	0.988000	0.999879	0.998004
Test	0.135996	0.149323	0.142348	0.533789	0.855210

Logistic Regression model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.179000	0.583769	0.27400	0.743543	0.749740
Test	0.178243	0.586987	0.27345	0.744460	0.749004

5) Strategic Imputer:

Improved number of features-> 90

K-Nearest Neighbor 1 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	1.000000	1.000000	1.000000	1.00000
Test	0.107422	0.111134	0.109246	0.515163	0.85417

K-Nearest Neighbor 3 model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.685000	0.193399	0.302000	0.934264	0.927641
Test	0.129507	0.035563	0.055802	0.527927	0.903159

Random Forest model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.98800	0.812337	0.892000	0.993884	0.984029
Test	0.23653	0.052334	0.085705	0.595919	0.910151

Decision Tree model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	1.000000	0.984152	0.992000	0.999915	0.998687
Test	0.138509	0.155385	0.146462	0.535385	0.854267

Logistic Regression model:

	PRECISION	RECALL	F1_SCORE	ROC_AUC_SCORE	ACCURACY
Train	0.18100	0.581958	0.276000	0.742232	0.752809
Test	0.17962	0.582340	0.274555	0.743401	0.752370

OBSERVATIONS:

1) The K Nearest model gives neighbor 1 as the best model, hence we have taken the second best as well i.e. 3 neighbors. 1 neighbor is also a valid hyperparameter , it works well especially in binary class problems.

2) –The KNN model performs best with the -999 imputer, Strategic imputer (for neighbor3 it does not) and meanmode imputer while performing poorly on the strategies with aggregation.
It is possible that the aggregation lead to increase in correlation between the categorical groupby features and agg columns-which is not preferable-although the new agg columns have increased correlation with the output Y-which is preferable.

3) The Random Forest performs best with meanmode and strategic imputer. Note here the -999 with agg performs better than without agg. Our guess is that the correlation issue explained in point 2 did not affect RF on account of the bootstrapping involved in its process.

4) The Decision tree performs best with mean mode imputer.

5) Logistic Regression –The -999 approach works the best closely followed by the mean mode technique.

6) Overall the Mean mode technique of imputation works best for all models. Hence we choose this method as missing value strategy for all models.

2.2 Experiment on the techniques to handle imbalance:

Below are the results and observations for each model:

DECISION TREE:

1) Oversampling:

F1_score at different levels of noise (noise-f1_score) :
(0.15, 0.145716),(0.3,0.147995),(0.45, 0.142224),
(0.75, 0.144683), (0.9, 0.135496)

Best Sampling strategy– 0.3

Y train after resampling Counter(0: 226132, 1: 67839)
Improved number of features– 90

Best parameter on grid– ('random_state': 42, 'max_features': 'auto', 'max_depth': 50, 'criterion': 'gini')

DECISION TREE model:
Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	226130	2
L = 1	3	67836

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	52080	4474
L = 1	4196	753

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.999	0.999	0.999	1.00
Test	0.144	0.152	0.147	0.536

2) SMOTE:

F1_score at different levels of noise (noise- f1_score) :
(0.15, 0.073491,(0.45, 0.083753),(0.75, 0.083657), (0.9, 0.091882)
Sampling strategy– 0.9

Y train after resampling Counter(0: 226132, 1: 203518)
Improved number of features- 111
Best parameter on grid– ('random_state': 42, 'max_features': 'auto', 'max_depth': 20, 'criterion': 'gini')

DECISION TREE model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	225182	950
L = 1	14191	189327

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	5254463	2091
L = 1	4610	339

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.995	0.932	0.961	0.990
Test	0.139	0.068	0.091	0.589

3) Under Sampling:
F1_score at different levels of noise (noise-f1_score) :
(0.15, 0.163427),(0.6, 0.194770),(0.75, 0.193454), (0.9, 0.185412)

Sampling strategy– 0.6
Y train after resampling Counter(0: 33126, 1: 19876)
Improved number of features- 94
Best parameter on grid– 'random_state': 42, 'max_features': 'auto', 'max_depth': 20, 'criterion': 'gini'

DECISION TREE model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	30749	2377
L = 1	3446	16430

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	40102	16452
L = 1	2640	2309

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.873	0.826	0.849	0.963
Test	0.123	0.466	0.194	0.583

4)COST FUNCTION BASED APPROACH:

Class weights– 0: 0.1, 1: 0.9 Improved number of features- 95
Best parameter on grid– 'random_state': 42, 'max_features': 'auto', 'max_depth': 20, 'criterion': 'gini' DECISION TREE model: Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	187596	38536
L = 1	1955	17921

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	44608	11946
L = 1	2921	2028

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.317	0.901	0.469	0.941
Test	0.145	0.409	0.214	0.598

Observations:

- Oversampling-we observed that introduction of more noise i.e. replicating the minority class only leads to overfitting and the decision tree performs poorly. This is possibly the reason why the best result is found at such low sampling strategy (0.3).
- Interestingly the best f1 score obtained at 0.3 in oversampling is even worse than the BASE model. This is because the model trains on same samples of the minority class multiple times due to replication and eventually trains specific to the samples themselves and not on the class features.
- One of the better results are with undersampling and worst with SMOTE. Similar to RF it is unable to learn and benefit from the new synthetic samples. Although it benefits from the undersampling strategy where the balance is set at 0.6 and a recal of 0.46 and f1_score 0.19 is achieved. The results at undersampling and oversampling are 10 times better than that at Smote.
- It works the best with the cost function based approach with f1_score more than 0.2 and recall more than 0.4.

Random Forest:

1)Oversampling:

F1_score at different levels of noise (noise-f1_score) :
(0.15- 0.039771), (0.45, 0.063275), (0.75, 0.067977), (0.9, 0.069861)
Best Sampling strategy– 0.9

Y train after resampling Counter(0: 226132, 1: 203518)
Improved number of features- 85
Best parameter on grid– 'n_estimators': 17, 'max_features': 'auto', 'max_depth': 90, 'bootstrap': True
Random Forest model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	226131	1
L = 1	0	203518

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	56226	328
L = 1	4758	191

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.999	1.00	0.999	1.00
Test	0.368	0.0385	0.069	0.682

SMOTE: F1_score at different levels of noise (noise-f1_score) :
(0.15- 0.201405), (0.45, 0.021696), (0.75, 0.017765), (0.9, 0.015032)

Y train before resampling Counter(0: 226132, 1: 19876)
Sampling strategy– 0.15

Y train after resampling Counter(0: 226132, 1: 33919)
Improved number of features- 105
Best parameter on grid– 'n_estimators': 13, 'max_features': 'auto', 'max_depth': 90, 'bootstrap': True

Random Forest model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	226129	3
L = 1	1513	32406

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	56395	159
L = 1	4859	90

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.999	0.955	0.977	0.999
Test	0.361	0.018	0.034	0.651

UNDERSAMPLING:

F1_score at different levels of noise (noise-f1_score) : (0.15:- 0.034629), (0.6, 0.255757), (0.75, 0.246021), (0.9, 0.239339)

Sampling strategy– 0.6
Y train after resampling Counter(0: 33126, 1: 19876)
Improved number of features- 96
Best parameter on grid– 'n_estimators': 17, 'max_features': 'auto', 'max_depth': 90, 'bootstrap': True

Random Forest model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	33099	27
L = 1	148	19728

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	46656	9898
L = 1	2772	2177

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.998	0.992	0.995	0.999
Test	0.180	0.439	0.255	0.700

Sampling strategy– 0.9
Y train after resampling Counter(0: 22084, 1: 19876)
Improved number of features- 95
Best parameter on grid– 'n_estimators': 17, 'max_features': 'auto', 'max_depth': 90, 'bootstrap': True

Random Forest model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	22050	34
L = 1	68	1980

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	39331	17223
L = 1	1935	3014

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.998	0.996	0.997	0.999
Test	0.148	0.609	0.239	0.705

COST FUNCTION BASED APPROACH:

Train Confusion Matrix:
Class weights– 0: 0.6, 1: 0.4
Improved number of features- 95
Best parameter on grid– 'n_estimators': 5, 'max_features': 'auto', 'max_depth': 90, 'bootstrap': True

Random Forest model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	225899	233
L = 1	3616	16260

Train Confusion Matrix:
Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
L = 0	55684	870
L = 1	4652	297

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.958	0.818	0.894	0.993
Test	0.254	0.060	0.097	0.607

Observation:

- Oversampling:
  The performance of Random Forest increases with more samples from the minority class. It has much less tendency to overfit when compared to the decision tree. This is due to the bootstrapping which is done as part of random forest that involves random resampling of data with replacement.
- We also observe that the number of trees (estimators) are very low. This is possibly due to the data being imbalanced it tends to converge to a decision tree. i.e. there are not enough samples from the minority class in the trees inside the random forest.

- The highest performance with oversampling is less than the f1_score without oversampling.

- Smote:
  The performance with Smote is the lowest, the random forest is unable to learn the new samples added since they are different from the actual dataset.

- Undersampling:
  RF performs the best with undersampling with f1_score reaching peak at 0.26 at 0.6 sampling strategy and recall of 0.6 at 0.9 sampling strategy. This shows how sensitive RF is with imbalanced data. With balanced data its f1_score is 10 times that before resampling

- Cost function based approach is not that helpful with RF.

Logistic Regression:

1)OVER SAMPLING:
F1_score at different levels of noise (noise-f1_score):
(0.15, 0.101225),(0.45, 0.291804),(0.6, 0.293157), (0.75, 0.283418),(0.9, 0.266181)
Recall score at different levels of noise (noise-recall):
(0.15, 0.057587),(0.45, 0.358254),(0.6, 0.478683), (0.75,0.577288),(0.9,0.642756)

Sampling strategy– 0.9
Y train after resampling Counter(0: 226132, 1: 203518)
Improved number of features- 86
Best parameter on grid– 'penalty': 'l1', 'C': 0.5

Logistic Regression model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	163385	62747
L = 1	72562	130956

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	40783	15771
L = 1	1768	3181

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.676	0.643	0.659	0.748
Test	0.167	0.642	0.266	0.748

Sampling strategy– 0.6
Y train after resampling Counter(0: 226132, 1: 135679)
Improved number of features- 87
Best parameter on grid– 'penalty': 'l1', 'C': 0.2

Logistic Regression model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	190761	35371
L = 1	71021	64658

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	47710	8844
L = 1	2580	2369

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.646	0.476	0.548	0.748
Test	0.211	0.478	0.293	0.748

2) SMOTE:

F1_score at different levels of noise (noise-f1_score):
(0.15, 0.115375),(0.45, 0.295989),(0.6, 0.293079), (0.75, 0.283418),(0.9, 0.279562)

Sampling strategy– 0.45
Y train after resampling Counter(0: 226132, 1: 101759)
Improved number of features- 112
Best parameter on grid– 'penalty': 'l1', 'C': 0.8

Logistic Regression model:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	201569	24563
L = 1	64445	37314

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
L = 0	50415	6139
L = 1	3023	1926

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.603	0.366	0.456	0.764
Test	0.238	0.398	0.295	0.747

UNDERSAMPLING:

F1_score at different levels of noise (noise- f1_score) : (0.15- 0.039771), (0.45, 0.063275), (0.75, 0.067977), (0.9, 0.069861)

Sampling strategy– 0.6
Y train after resampling Counter(0: 33126, 1: 19876)
Improved number of features- 96
Best parameter on grid– ’penalty’: ’l1’, ’C’: 0.8

Logistic Regression model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
$L = 0$	27915	5211
$L = 1$	10488	9388

Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
$L = 0$	47812	8742
$L = 1$	2595	2354

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.643	0.472	0.544	0.746
Test	0.212	0.475	0.293	0.748

COST FUNCTION BASED APPROACH:

Class weights– 0: 0.1, 1: 0.9
Improved number of features- 96
Best parameter on grid– ’penalty’: ’l1’, ’C’: 0.8

Logistic Regression model:

Train Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
$L = 0$	173117	53015
$L = 1$	8101	11775

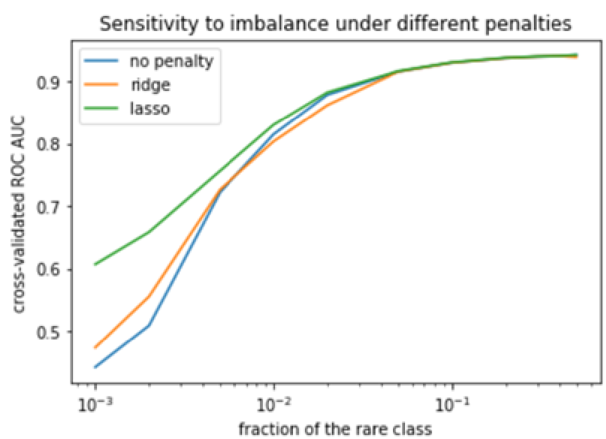
Test Confusion Matrix:

	$\hat{L} = 0$	$\hat{L} = 1$
$L = 0$	4321	13335
$L = 1$	2002	2947

	PRECISION	RECALL	F1_SCORE	AUC
Train	0.181	0.592	0.278	0.747
Test	0.180	0.595	0.277	0.748

Observation:

- All techniques of imbalance improve the performance logistic regression with almost the same intensity. The best is with SMOTE with recall score shooting as high as 0.6 at sampling strategy 0.9 and highest f1_score of approx 0.3 at sampling strategy 0.6. The statistical methods like Logistic regression tend to sharply underestimate the probability of rare events [2].
- The L1 regularization form works best with imbalanced data fig 1 [3]



-The cost function based approach is by far the best for logistic with optimal results for both recall and f1_score.

K Nearest Neighbors:

1) OVERSAMPLING: F1_score at different levels of noise (noise-f1_score):
(0.15, 0.107061),(0.45, 0.107061),(0.6, 0.107061),(0.75, 0.107061),(0.9, 0.107061)

Y train before resampling Counter(0: 226132, 1: 19876)
Sampling strategy– For all sampling strategies:
Y train after resampling Counter(0: 226132, 1: 33919)
Improved number of features- 95
Best parameter on grid– ’n_neighbors’: 1, ’algorithm’: ’auto’

K NEAREST Neighbor: model:

Train Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
$L=0$	226132	0
$L=1$	0	33919

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
$L=0$	52026	4528
$L=1$	4413	536

	PRECISION	RECALL	F1_SCORE	AUC
Train	1.00	1.00	1.00	1.00
Test	0.105	0.108	0.107	0.514

2) SMOTE:

F1_score at different levels of noise (noise-f1_score):
(0.15, 0.113533),(0.45, 0.127832),(0.6, 0.129434),
(0.75,0.134434),(0.9, 0.134869)

Sampling strategy– 0.9
Y train after resampling Counter(0: 226132, 1: 203518)
Improved number of features- 115
Best parameter on grid– 'n_neighbors': 1, 'algorithm': 'auto'

K NEAREST Neighbor: model:

Train CoMatrix:

	$\hat{L}=0$	$\hat{L}=1$
$L=0$	226132	0
$L=1$	0	203518

Test Confusion Matrix:

	$\hat{L}=0$	$\hat{L}=1$
$L=0$	46927	9627
$L=1$	3895	1054

	PRECISION	RECALL	F1_SCORE	AUC
Train	1.00	1.00	1.00	1.00
Test	0.098	0.212	0.134	0.521

3) Undersampling:

F1_score at different levels of noise (noise-f1_score):
(0.15, 0. 0.125849),(0.45, 0.149095),(0.9, 0.155352)

3

```
K NEAREST Neighbor: model:

Sampling strategy--> 0.9
Y train after resampling Counter({0: 22084, 1: 19876})

Best parameter on grid--> {'n_neighbors': 5, 'algorithm': 'auto'}
K NEAREST Neighbor: model:
Train Confusion Matrix:
[[16185  5899]
 [ 6357 13519]]
Test Confusion Matrix:
[[31818 24736]
 [ 2449  2500]]
        PRECISION    RECALL  F1_SCORE  ROC_AUC_SCORE  ACCURACY
Train     0.69621  0.680167  0.688095       0.773326  0.707912
Test      0.09179  0.505153  0.155352       0.545348  0.557989
```

Observation:

- Interestingly in the oversampling technique the KNN does not show any change in performance whatsoever. The accuracies at different noise levels are exactly the same. This is because replicating same dataset does not give any new additional information especially since the grid gives the 1 nearest neighbors as the winner. The class to which the new data belong- there is already one instance of that present before sampling so oversampling has absolutely no effect.

- The SMOTE technique also uses KNN to synthetically add new data. Hence, since it is not a replication of the existing data, the new data gives useful information to the model. We can observe (from the f1_scores above) that with more synthetic data the KNN model is able to predict more. Hence, best sampling strategy is high at 0.9

- The undersampling technique works best for KNN at sampling strategy 0.9 with the best f1_score at 0.155352. More importantly since the model can learn the minority class as much as the majority class at sampling strategy 0.9, the recall is very high i.e. the true positives are highest at 0.5. We observe an increase in the performance as the balance is increased in the dataset.

There are no class weights based approach for KNN.

XGBoost:

Why XGBoost?

XGBoost is an advanced implementation of gradient boosted decision trees designed for speed and performance. Some advantages of using XGBoost are:

- Regularization : Standard GBM has no regularization like XGBoost, thus it helps reduce overfitting.
- Parallel Processing : faster as compared to GBM (sequential process)
- High Flexibility : User can define Custom Optimization Objective and Evaluation criteria
- Handling Missing Values : Imputes missing values on side that has reduced loss
- Tree Pruning : GBM stops splitting when it encounters negative loss in the split. Thus it is more of a greedy algorithm. XGBoost on the other hand make aplits upto the max_depth specified by the user and then start pruning tree backwards and remove splits beyond which we do not have positive gain.
  Example: Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. GBM would stop as it encounters -2. But XGBoost will go deeper and it will see a combined effect of +8 of the split and keep both.
- Built In Cross Validation: XGBoost allows user to run a cross-validation at each iteration of the boosting process.

Now as our data is having many missing values and due to computation power of XGBoost, we used XGBoost.

What we tried for XGBoost?

1. XGBoost model on application_train.csv :
We did some manual feature engineering for this csv file and then we applied our model on it. After that parameter tuning was done and at the end we came up with below metric results:

```
Train AUC 0.7756
Test AUC 0.7588
```

```
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
print("Train Accuracy %.4f" % accuracy_score(y_train,y_train_pred))
print("Test Accuracy %.4f" % accuracy_score(y_test,y_test_pred))
```

```
Train Accuracy 0.8903
Test Accuracy 0.8863
```

```
# for i in range(100):
#     print(y_test_pred[i]," ",y_test_predicted[i])
```

```
from sklearn.metrics import confusion_matrix

confmat=confusion_matrix(y_test,y_test_pred)
print(confmat)
# plt.imshow(confmat, cmap='binary')
```

```
[[53121  3433]
 [ 3560  1389]]
```

2. XGBoost model on application_train.csv (Feature Engineering in previous step plus 15 most missing columns dropped):

After the 1st method, I thought to drop top 15 missing columns. Moreover, we tried to drop column by column and the efficiency of model increased up till dropping 15 columns and then it decreased. Thus, we decided not to drop more than 15 columns. After this we did parameter tuning to come up with a model that can give best efficiency for this data.

```
Train AUC 0.7904
Test AUC 0.7607
```

```
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
print("Train Accuracy %.4f" % accuracy_score(
print("Test Accuracy %.4f" % accuracy_score(y
```

```
Train Accuracy 0.8903
Test Accuracy 0.8839
```

```
from sklearn.metrics import confusion_matrix

confmat=confusion_matrix(y_test,y_test_pred)
print(confmat)
# plt.imshow(confmat, cmap='binary')
```

```
[[52896  3658]
 [ 3483  1466]]
```

We can note that compared to previous one, AUC of Train and test both increased. Moreover, having a closer look at the True Negatives, we can see that they increased from 1389 to 1466 just by dropping 15 unwanted columns.

3. XGBoost on application_train_imputed.csv:

Application_train_imputed.csv was imputed logically by us in best manner that we can come up with. It contained NO NULL values in it. It was imputed by Imputer with mode and mean as imputing metric. We applied XGBoost on that and did best parameter tuning that we could do. Below are the results:

```
y_pred=clf1.predict(X_test)
newdf=pd.DataFrame(data=y_pred)
newdf.iloc[:,0].value_counts()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\pre
ty array is ambiguous. Returning False, but in future
ray is not empty.
  if diff:
```

```
.1]: 0.0    41118
     1.0    20385
     Name: 0, dtype: int64
```

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

```
             precision    recall  f1-score   support

        0.0       0.96      0.70      0.81     56538
        1.0       0.17      0.69      0.27      4965

avg / total       0.90      0.70      0.77     61503
```

```
confusion_matrix(y_test, y_pred)
```

```
.3]: array([[39556, 16982],
            [ 1562,  3403]], dtype=int64)
```

4. XGBoost on application_train_imputed.csv + SMOTE: Taking the same data ie. Application_train_imputed.csv we applied oversampling with the help of SMOTE. SMOTE is a technique used to oversample the data label that is very few. We tried to oversample the data label that is much less ie. People who will default. The results were not as expected as this. It did not improve with a much extent. The number of defaulter detection increased but not with a great extent which we were expecting. Moreover, here what we should be concern about is innocent people being declared as defaulters. Here we can see that SMOTE declares 19000 people as defaulters which are actually not. Below is the evaluation results that we got after doing parameter tuning.

```
0.0    38343
1.0    23160
Name: 0, dtype: int64
```

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

```
             precision    recall  f1-score   support

        0.0       0.96      0.65      0.78     56538
        1.0       0.15      0.72      0.25      4965

avg / total       0.90      0.66      0.74     61503
```

```
confusion_matrix(y_test, y_pred)
```

```
array([[36928, 19610],
       [ 1415,  3550]], dtype=int64)
```

5. XGBOOST on all 7 tables combined (FINALHOMECREDIT.csv):
The data file that we prepared by combining all 7 tables was used here. It was having shape of (307507,798). This data file was providing test AUC of 0.7817 and it detected almost 1600 defaulters which was almost highest as compared to all above. We should be glad about AUC curve as the highest ranking solution of Kaggle has AUC of 0.8057

```
Train AUC 0.8229
Test AUC 0.7818
```

```
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
print("Train Accuracy %.4f" % accuracy_score(y_
print("Test Accuracy %.4f" % accuracy_score(y_
```

```
C:\ProgramData\Anaconda3\lib\site-packages\skle
ty array is ambiguous. Returning False, but in
ray is not empty.
  if diff:
```

```
Train Accuracy 0.8980
Test Accuracy 0.8917
```

```
C:\ProgramData\Anaconda3\lib\site-packages\skle
ty array is ambiguous. Returning False, but in
ray is not empty.
  if diff:
```

```
from sklearn.metrics import confusion_matrix

confmat=confusion_matrix(y_test,y_test_pred)
print(confmat)
# plt.imshow(confmat, cmap='binary')
```

```
[[53235  3246]
 [ 3417  1604]]
```

What Worked?
Using FINALHOMECREDIT.csv table worked for us as it was made up with all 7 tables. Moreover, it gave best AUC curve till now of all models that we applied within XGBoost and except XGBoost also.

What didn't work?
SMOTE was highly expected to work while we are having such a huge amount of imbalanced data(92% - 8%). Smote not only didn't worked, it misclassified 19000 people who are not defaulters as defaulters which should never happen as the Loan company will have a great loss loosing such customers. Compared to SMOTE results of detecting almost 71% of people who will default, the results of XGBoost in 5th model (on FINALHOMECREDIT.csv) is a lot better which is just detecting 30% of people who are defaulters because SMOTE misclassifies 34% of nondefaulters (19000 people), while our 5th model misclassifies only 5%(3,000) of people who are

nondefaulters.

What could be done in future?
I believe due to large amount of data, we could not do parameter tuning to the level that we do normally for small datasets. Easily the models took 1 hour sometimes to run and it is not feasible on local machine to keep it on parameter tuning phase as it would take much much longer time. Thus, if we would be having a very good GPU access, we would have achieved still better AUC metric.
Apart from this, we can try Neural Networks and CatBoost, which I had a very great affinity towards. We can also do Ensembling of XGBoost, LightGbm and Neural Networks. We would definitely increase efficiencies if we would have done so.

LightGbm Classifier:

Why did we try LightGbm?

Light GBM is a gradient boosting framework that uses tree based learning algorithm. Light GBM grows tree vertically while other algorithm grows trees horizontally. It means that Light GBM grows tree leaf-wise while other algorithm grows level-wise. It will choose the leaf with max delta loss to grow. When growing the same leaf, Leaf-wise algorithm can reduce more loss than a level-wise algorithm.
We preferred LightGBm to run because
• We have large data and it works very well on large size of data and its processing speed is also high. Moreover it also takes lower memory to run.
• We need fast computation here and LightGbm supports GPU Learning.

What did we try for LightGbm?

We tried following things:

1. LightGbm on application_train_imputed.csv file:
Application_train_imputed.csv was imputed logically by us in best manner that we can come up with. It contained NO NULL values in it. It was imputed by Imputer with mode and mean as imputing metric.

We applied LightGbm on it and the results were not satisfactory. Though it misclassified only small amount of people who were nondefaulters which is good but on the contrary it also did not detect many defaulters . Parameter tuning was done after that but all models were having almost same classification report as mentioned below. There was no change for almost all models that I found by parameter tuning.

```
newdf=pd.DataFrame(data=y_pred)
newdf.iloc[:,0].value_counts()
```

```
0.0    61012
1.0      491
Name: 0, dtype: int64
```

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

```
             precision    recall  f1-score   support

        0.0       0.92      1.00      0.96     56538
        1.0       0.46      0.05      0.08      4965

avg / total       0.89      0.92      0.89     61503
```

```
confusion_matrix(y_test, y_pred)
```

```
array([[56273,   265],
       [ 4739,   226]], dtype=int64)
```

2. LightGbm on application_train_imputed.csv file + SMOTE:
Here we have used same file as above but in addition we have also used SMOTE technique to increase the sample. It did not turn out very well. In fact it decreased the models performance.

```
newdf=pd.DataFrame(data=y_pred)
newdf.iloc[:,0].value_counts()
```

```
0.0    61174
1.0      329
Name: 0, dtype: int64
```

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

```
             precision    recall  f1-score   support

        0.0       0.92      1.00      0.96     56538
        1.0       0.52      0.03      0.06      4965

avg / total       0.89      0.92      0.89     61503
```

```
confusion_matrix(y_test, y_pred)
```

```
array([[56379,   159],
       [ 4795,   170]], dtype=int64)
```

3. LightGbm on all 7 table combined file (FINALHOMECREDIT.csv):

The data file that we prepared by combining all 7 tables was used here. It was having shape of (307507,798). So far, for LightGbm, this was the best model that worked. A small amount of parameter tuning was done for this one.

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

```
             precision    recall  f1-score   support

          0       0.92      0.99      0.96     56537
          1       0.52      0.07      0.13      4965

avg / total       0.89      0.92      0.89     61502
```

```
confusion_matrix(y_test, y_pred)
```

```
array([[56206,   331],
       [ 4611,   354]], dtype=int64)
```

4. LightGbm on all 7 tables combined file + parameter tuning: Here parameter tuning was done for a great amount of time. It took almost 12 hours on local PC to run the KFoldCV. Below were the parameters that we used for Classifier and we did K Fold CV for a shape of almost 300000. Apart from these the estimators are nearly 10000 which lead to much extensive time. In this model we achieved AUC metric of 0.7861 which was a lot better than other models that we performed.

```
clf = LGBMClassifier(
        nthread=4,
        n_estimators=10000,
        learning_rate=0.02,
        num_leaves=34,
        colsample_bytree=0.9497036,
        subsample=0.8715623,
        max_depth=8,
        reg_alpha=0.041545473,
        reg_lambda=0.0735294,
        min_split_gain=0.0222415,
        min_child_weight=39.3259775,
        silent=-1,
        verbose=-1, )
clf.fit(train_x, train_y, eval_set=[(train_x, train_y), (valid_x, valid_y)],
        eval_metric= 'auc', verbose= 200, early_stopping_rounds= 200)
```

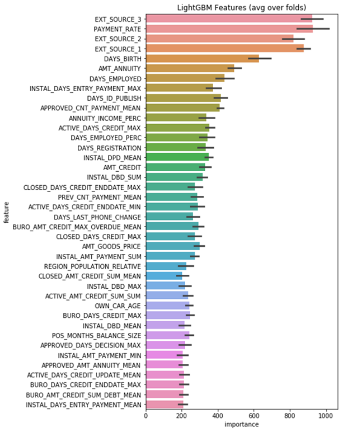
```
    fold_importance_df["fold"] = n_fold + 1
    feature_importance_df = pd.concat([feature_importance_df, fold_importance_df], axis=0)
    print('Fold %2d AUC : %.6f' % (n_fold + 1, roc_auc_score(valid_y, oof_preds[valid_idx])))
```

```
[400]   training's binary_logloss: 0.223304     training's auc: 0.825267        valid_1's binary_logloss: 0.243232      va
lid_1's auc: 0.777924
[600]   training's binary_logloss: 0.216474     training's auc: 0.841749        valid_1's binary_logloss: 0.241588      va
lid_1's auc: 0.782438
[800]   training's binary_logloss: 0.210906     training's auc: 0.855162        valid_1's binary_logloss: 0.240953      va
lid_1's auc: 0.784171
[1000]  training's binary_logloss: 0.205622     training's auc: 0.867474        valid_1's binary_logloss: 0.240699      va
lid_1's auc: 0.78471
[1200]  training's binary_logloss: 0.201178     training's auc: 0.877326        valid_1's binary_logloss: 0.240542      va
lid_1's auc: 0.785325
[1400]  training's binary_logloss: 0.196791     training's auc: 0.886674        valid_1's binary_logloss: 0.240328      va
lid_1's auc: 0.785846
[1600]  training's binary_logloss: 0.192669     training's auc: 0.895254        valid_1's binary_logloss: 0.240305      va
lid_1's auc: 0.785932
Early stopping, best iteration is:
[1528]  training's binary_logloss: 0.194195     training's auc: 0.892123        valid_1's binary_logloss: 0.240254      va
lid_1's auc: 0.786127
best iteration of clf is :  1528
Fold 10 AUC : 0.786127
```

```
print('Full AUC score %.6f' % roc_auc_score(y_train_later, oof_preds))
```

```
Full AUC score 0.788996
```

Below is the features that were found most important and we got this by doing feature_importances over K Folds.



What Worked?

A key thing to notice about LightGbm here was though the models were not detecting defaulters, it also was not misclassifying the nondefaulters. The AUC metric for 4th model was the best one and it worked well. We received a metric of 0.7886 while Kaggle top scorer got 0.80.

What did not work?

The results that we achieve with SMOTE are really bad in both XGboost and LightGbm. Here we can easily conclude that Smote surely doesn't works well for both the models used.

Detecting the number of more defaulters did not worked well , that could have been improved by much more parameter tuning then that we did. But, unfortunately our PC were not having that capability to run for such a longer time.

What could be done in future?

A lot more things could be done in future. One such thing is using Ensemble Modeling of multiple LightGbms. Our only benefit of LightGbm here was it was not misclassifying nondefaulters, which was really a great thing as company would not be willing to loose 100 nondefaulters in order to catch just one defaulter. Thus, what we can do is, we can ensemble many more weak learners that we are already having right now and ensemble them up into one model. Apart from this, another thing that could be done is try more parameter tuning with the model building phase.

3. CONCLUSION:

SUMMARY and CONCLUSION:
-The mean mode technique works best to impute the missing values
- We use the recall, F1_score and ROC_AUC score to compare models
-Oversampling stands mediocre with RF and Decision trees and performs lowest for KNN.
-Smote deteriorates or has no effect on the performance of Decision trees and Random forest.
-Undersampling boosts the performance the most for all the four models when compared to oversampling and smote.
-Cost function based approach works well for all models except for Random Forest.
- Amongst the four models (RF, DT, Logistic, KNN) Logistic performs the best with cost function based resampling at 1:9 class weights for majority:minority class. Highest f1_score–.28, recall=0.6 and ROC_AUC_Score=0.75

4. REFERENCES:

1. https://datascience.stackexchange.com/questions/9440/how-to-customise-cost-function-in-scikit-learns-model
2. https://gking.harvard.edu/files/abs/0s-abs.shtml
3. https://datascience.stackexchange.com/questions/3699/what-cost-function-and-penalty-are-suitable-for-imbalanced-datasets/5031
4. https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction
5. https://www.kaggle.com/c/home-credit-default-risk

other sources used:
https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc

https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

https://towardsdatascience.com/fine-tuning-xgboost-in-python-like-a-boss-b4543ed8b1e

https://towardsdatascience.com/machine-learning-kaggle-competition-part-two-improving-e5b4d61ab4b8

https://nycdatascience.com/blog/student-works/kaggle-predict-consumer-credit-default/
https://www.kaggle.com/aantonova/797-lgbm-and-bayesian-optimization
https://www.linkedin.com/pulse/winning-9th-place-kaggles-biggest-competition-yet-home-levinson/

https://medium.com/@mateini_12893/doing-xgboost-hyper-parameter-tuning-the-smart-way-part-1-of-2-f6d255a45dde

https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction

https://www.kaggle.com/ogrellier/lighgbm-with-selected-features/
https://www.kaggle.com/jsaguiar/lightgbm-with-simple-features
https://www.kaggle.com/ogrellier/good-fun-with-ligthgbm

Acknowledgments

We would like to express our special thanks and gratitude to our professor Gregory Rawlins for his support and guidance.

Also a special thanks to our Associate instructors - Zeeshan Ali Sayyed,Christopher Torres-Lugo ,Manoj Joshi