CREDIT RISK MODELLING ASSIGNMENT

FM9528A Banking Analytics

Word Count:2152

By:251253766

Objective

With this coursework, our objective is to create a fully compliant PD model to determine good and bad applicants.

Approach

My approach to solving this problem can be broadly outlined into the following steps:

1. Data Cleaning and Exploratory Data Analysis. Creation of 3 new variables that capture the pattern in the data better
2. Calculating Weight of Evidence to perform variable selection
3. Creating our base model of predicting good and bad applicants using Logistic Regression model and creating the scorecard
4. Comparing base model with ensemble models like Random Forest Classifier and XGBoost.
5. Finally creating a two-cut-off point strategy for the scorecard.

All the steps will be detailed below:

1. Data Cleaning and Preprocessing

The main objective of data cleaning is to be remove incorrect entries, inconsistent information, and correct human error in data entry as much as possible. The goal is also to treat any NULL values present in the data so that our models can use the data as input. The exploratory data analysis was done on the entire dataset. We then split the data in train and test sets in a ratio of 70:30, and any imputation with sample means, medians or modes was done based on the train sample.

a. **NULL Handling:** This is how the NULLS were detected:

Let us find out how many NULLS are there in each column

```
[ ] null_columns = credit_modelling_df.columns[credit_modelling_df.isnull().any()]
    null_columns

    Index(['residence_type', 'months_in_residence', 'professional_city',
           'professional_borough', 'profession_code', 'occupation_type',
           'mate_profession_code', 'mate_education_level'],
          dtype='object')
```

```
[ ] credit_modelling_df[null_columns].isnull().sum()

    residence_type            1349
    months_in_residence       3777
    professional_city        33783
    professional_borough     33783
    profession_code           7756
    occupation_type           7313
    mate_profession_code     28884
    mate_education_level     32338
    dtype: int64
```

After detecting the columns that have NULLs in them, we examine them one by one.

Imputing values

- For **months in residence**, only about 7 % data had NULLS, and we imputed the Median value of the column into NULLs.
- For **residence type**, I have replaced the NULL values (about 3% of the data) with the Mode of this column. I have assumed that all applicants have a valid value for residence type and that none of them were homeless and therefore missing this value for valid reasons.
- For variables **profession code** and **occupation type**, there was 10% data with NULLS and they were replaced with a new category, because it would be incorrect to assume they belong to anyone of the existing categories.
- For variables like residence City, residence state, professional city, professional state

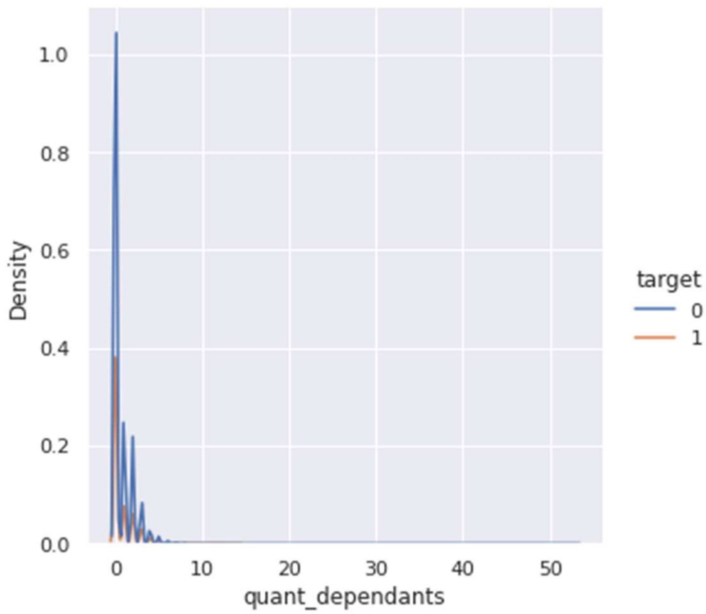
Dropping columns

- **Professional City** and **Borough** had more 60% NULLs, same with **Mate Profession** code and **Education** level, so we dropped them from our dataset. We assume that **Professional Zip Codes** will have the relevant information.
- **Residential City** and **Borough** had more 60% NULLs, same with **Mate Profession** code and **Education** level, so we dropped them from our dataset. We assume that **Residential Zip Codes** will have the relevant information.
- I dropped all columns that had a single value for all observations, for example **Clerk Type** and **Quant Additional Cards** and a few flags.
- I dropped **Sex** because identity should not contribute to credit scoring.

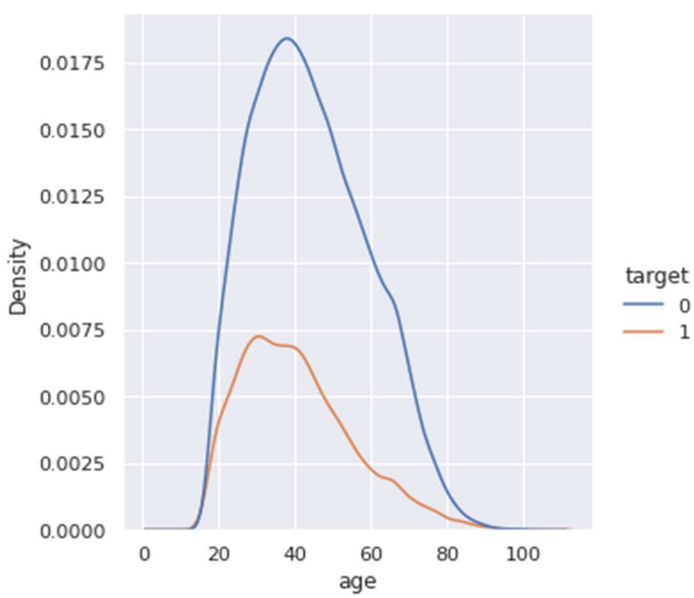
b.Outlier Handling

To detect outliers, we need to understand how our data is distributed. We run some univariate exploration of our data to see if we have outliers and whether they can be explained or need to be handled. A few example plots are shown below:

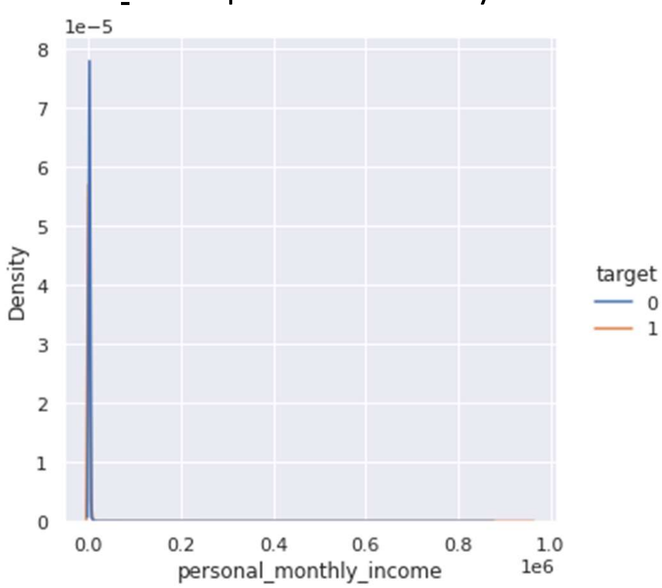
Quant_dependants: It appears that this field has an outlier at greater than 50. Most of the distribution is concentrated between 0 to 10.

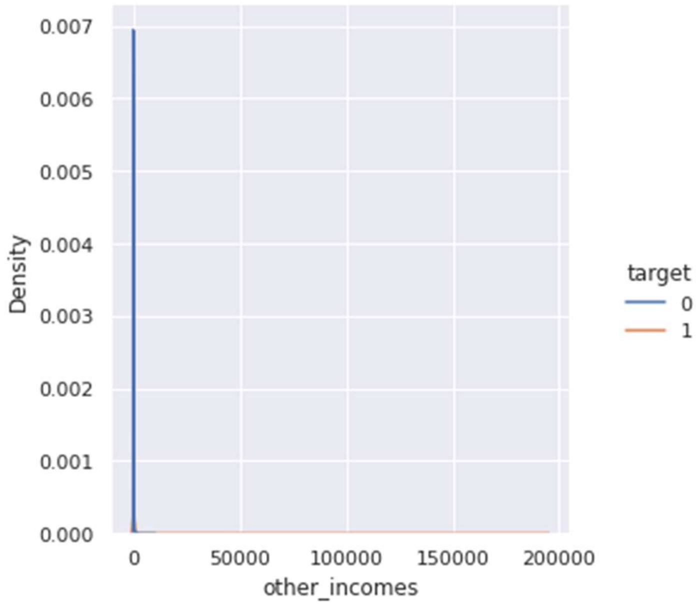


Age:



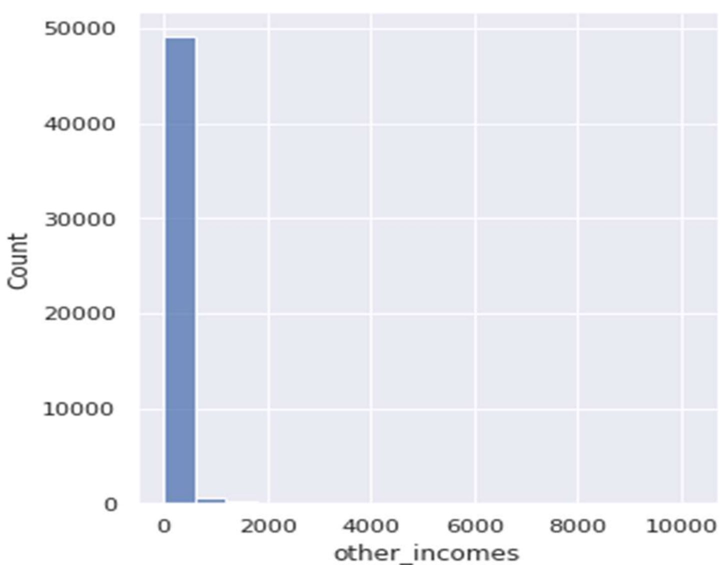
Income:_ Both personal monthly income and other income show outliers in the distribution.



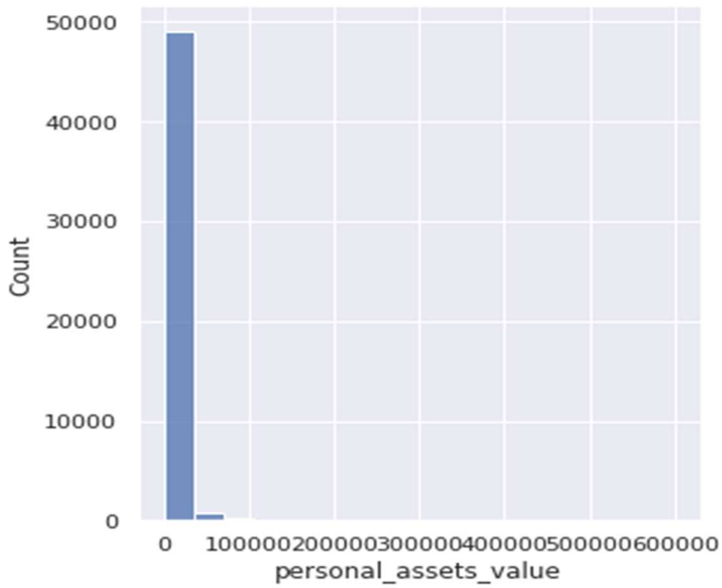


We investigated these variables further, and we decided to take the following approach:

1. We replaced the outlier value of **quant_dependants** by the median of the population
2. We did not do any changes to **personal income**, and since we had only one observation for **other incomes** that was disproportionately larger than the average distribution, we decided to filter it out. We handled the **personal assets** similarly.



3.

4. 

<u>c.Creating new variables:</u>

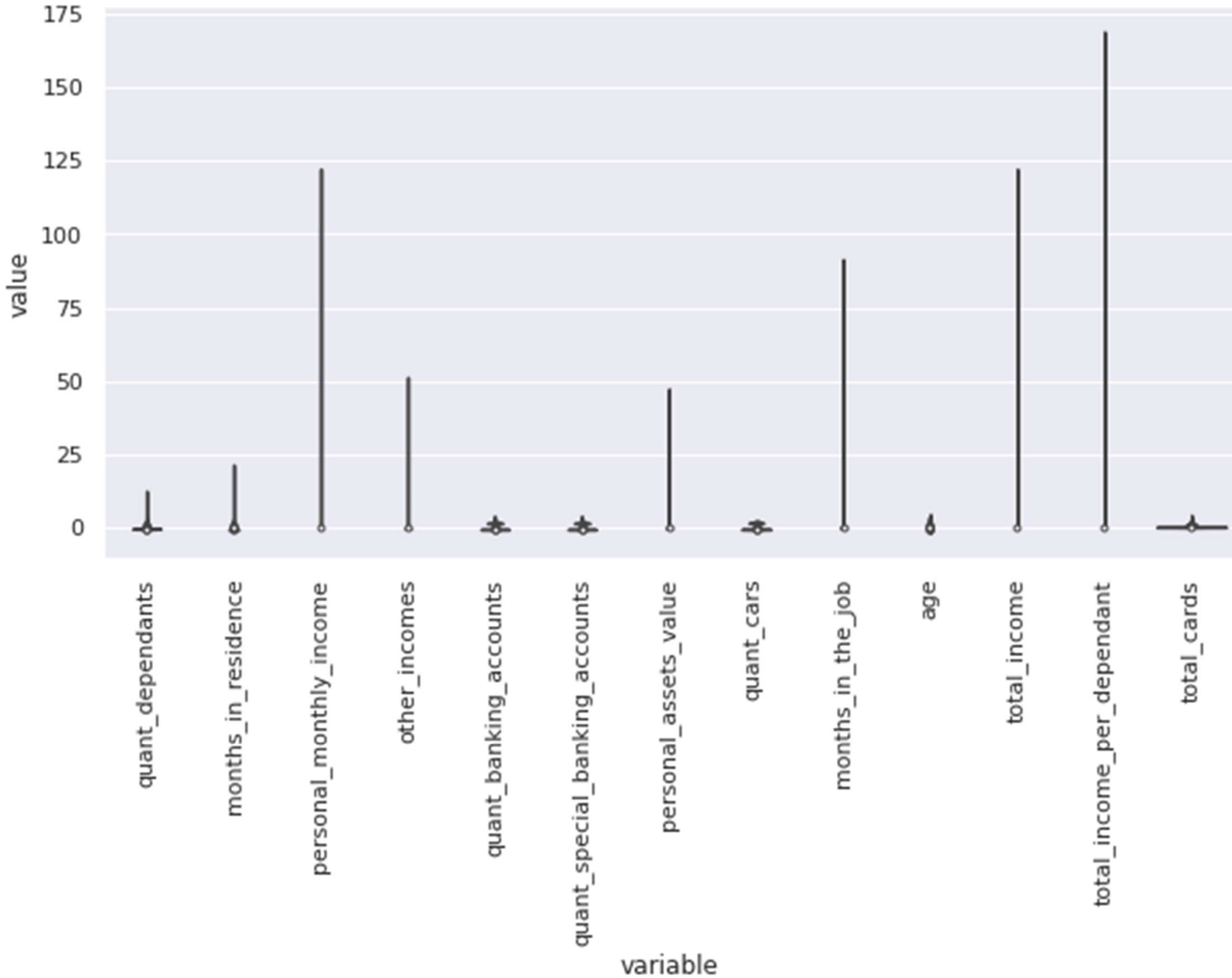
I created the following 3 new variables to understand the data behavior better:

i. Total Income: Sum of personal monthly income and other income. I wanted to consolidate the earnings in one variable to understand how the earnings impact credit card defaults

ii. Total Income per dependent: This gives an idea about how much money an individual has available on dependents and whether, having less money per dependent might lead to higher rate of default.

iii. Total Number of Cards: I wanted to understand if an individual has too many credit cards and if that impacts default rate.

After we have created our new variable and are done with cleaning the dataset, we normalize our numeric variables to bring them to the same scale to make sure everything looks ok.



2. Weight of Evidence(WoE) and Variable Selection:

We do a 2-stage variable selection here, Information Value(IV) filtering and correlation filtering.

IV Filtering

In this step we group our categorical and continuous variables into bins. The optimum number of bins is decided based on the weight of evidence provided by the grouping of each variable. If our goal is to predict a good or bad outcome (0 or 1 in our case), Weight of Evidence is given by the following equation:

$WoE_{category} = \ln(p\_good_{category} / p\_bad_{category})$,

where $p\_good_{category}$ = number of $goods_{category}$ / number of $goods_{total}$
$p\_bad_{category}$ = number of $bads_{category}$ / number of $bads_{total}$

If $p\_good_{category} > p\_bad_{category}$ then $WoE_{category} > 0$
If $p\_good_{category} < p\_bad_{category}$ then $WoE_{category} < 0$

Using the WoE value, we can calculate the IV of each category. IV is the calculated predictive power of each variable, and generally IV < 0.2 is considered to have weak predictive power.

I used the scorecard.py package in python to calculate the WoE and IV.
Given the size of our dataset, at 50,000 observations and about 53 predictors, I felt that a final of maximum 8 bins per variable, with at least 5% data in each bin was sufficient for our purpose. I think an initial 100 cuts is enough for our dataset.

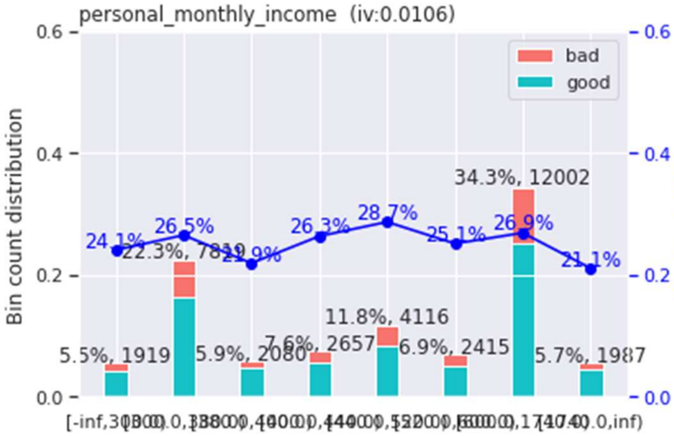
Here is our set up for WoE binning

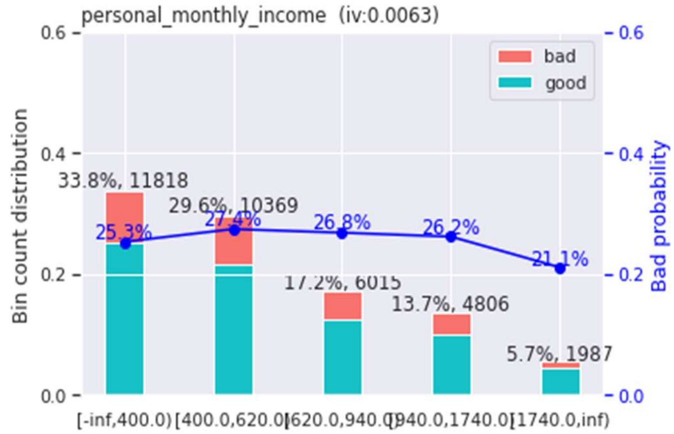
```
bins = sc.woebin(train,
                 y = 'target',
                 min_perc_fine_bin=0.01, # How many bins to cut initially into
                 min_perc_coarse_bin=0.05,  # Minimum percentage per final bin
                 stop_limit=0.02, # Minimum information value
                 max_num_bin=8, # Maximum number of bins
                 method='tree'
                 )

bins

sc.woebin_plot(bins)
```

I then manually adjusted the bins for some variables, so that trend of percentage of default in each bin is reasonable and explainable and as free of noise as possible.

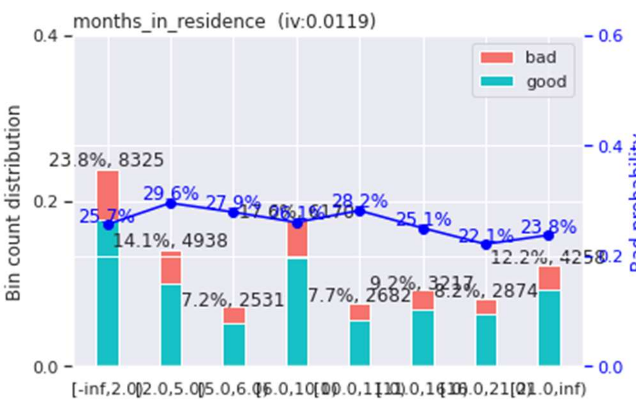
Examples of manual adjustment

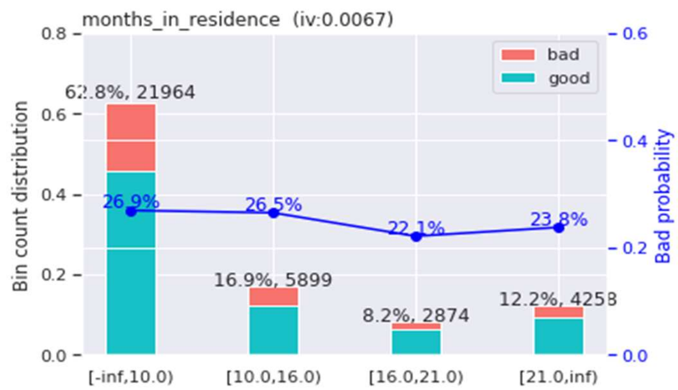
Personal Monthly Income: Left Image is before manual bin adjustment, right image is after





Months in Residence: Left Image is before manual bin adjustment, right image is after





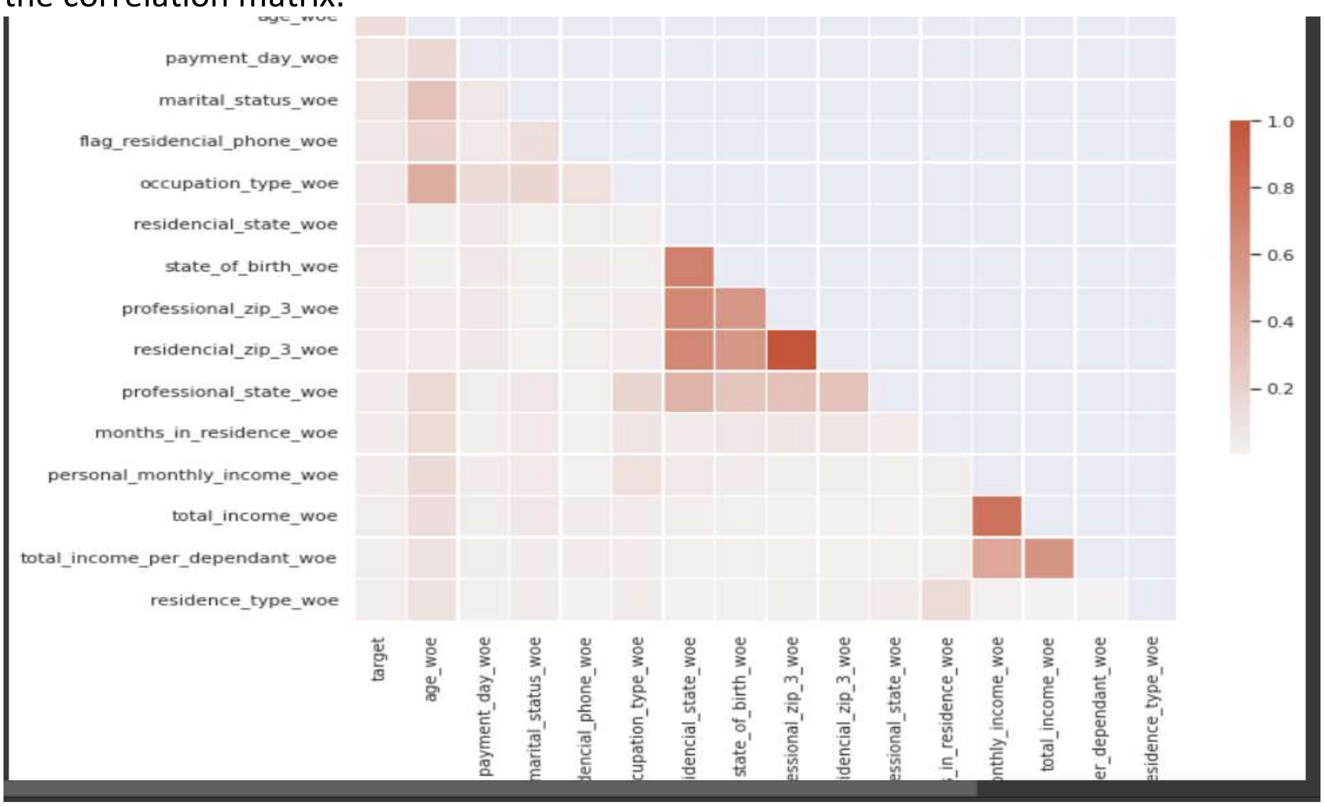
The calculated Information Value for the variables is as follows. We choose any variable with an **IV > 0.002** because our data is not very informative, and we need at least a few variables to create a viable, if weak model.

All the variables and their IV

variable	info_value
age_woe	0.078221
payment_day_woe	0.031669
marital_status_woe	0.027454
flag_residencial_phone_woe	0.019313
occupation_type_woe	0.018752
residencial_state_woe	0.017281
state_of_birth_woe	0.015231
professional_zip_3_woe	0.011580
residencial_zip_3_woe	0.011580
professional_state_woe	0.010326
months_in_residence_woe	0.006746
personal_monthly_income_woe	0.006308
total_income_woe	0.005412
total_income_per_dependant_woe	0.004517
residence_type_woe	0.004122
profession_code_woe	0.001638
flag_professional_phone_woe	0.001528
quant_dependants_woe	0.001432
application_submission_type_woe	0.000941
quant_banking_accounts_woe	0.000489
product_woe	0.000480
other_incomes_woe	0.000438
quant_cars_woe	0.000317
company_woe	0.000283
total_cards_woe	0.000121

Correlation Filtering

We drop any variable that is highly correlated with another predictor variable as displayed in the correlation matrix.



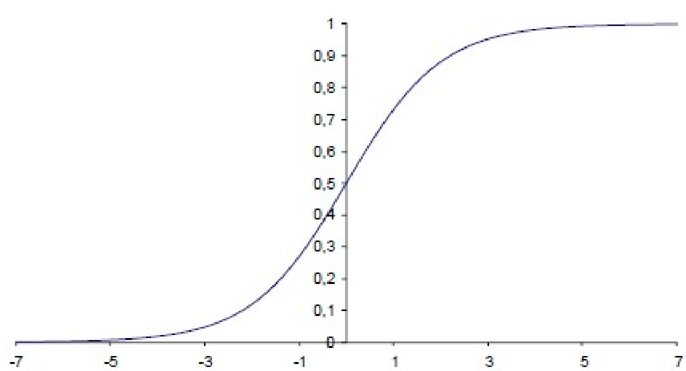
We drop total_income, residential and professional state, professional zip based on their high correlation with other predictors (0.6). The final list of columns is:

	column	0
0	age_woe	0.957131
1	payment_day_woe	0.763664
2	marital_status_woe	0.633783
3	flag_residencial_phone_woe	1.719996
4	occupation_type_woe	0.085793
5	state_of_birth_woe	0.783603
6	residencial_zip_3_woe	0.422154
7	months_in_residence_woe	0.382343
8	personal_monthly_income_woe	0.411285
9	total_income_per_dependant_woe	0.214759
10	residence_type_woe	0.542823

3.ScoreCard Creation to model the probability of default

We are dealing with a binary classification problem. The purpose of this section is to create a model that can predict whether a creditcard holder as a defaulter or a non-defaulter, based on the parameters in the dataset. We choose a LogisticRegressionCV classifier model for our purpose to perform automatic cross validation. It takes the following function form
take the following function form:

$$f(z)=\frac{1}{1+e^{-z}}$$



Steps to build the model:

a. LogisticRegression object creation: For this step, we take as input the data set from the previous step, where we have applied WoE to the predictors. We the split the dataset into **train** and **test** data in 70:30 ratio.
We train our model using the train set, and then apply our predictions on the test set. The hyperparameter setting is as follows:

```
[ ] # Logistic Regression Model and scorecrad
    creditdef_logreg = LogisticRegressionCV(penalty='l2', # Type of penalization l1 = lasso, l2 = ridge, elasticnet
                                             Cs = 3,        # How many parameters to try. Can also be a vector with parameters to try.
                                             tol=0.000001, # Tolerance for parameters
                                             cv = 3,      # How many CV folds to try. 3 or 5 should be enough.
                                             fit_intercept=True, # Use constant?
                                             class_weight='balanced', # Weights, see below
                                             random_state=251253766, # Random seed Student ID
                                             max_iter=100, # Maximum iterations
                                             verbose=1, # Show process. 1 is yes.
                                             solver = 'saga', # How to optimize.
                                             n_jobs = 2,      # Processes to use. Set to number of physical cores.
                                             refit = True,     # If to retrain with the best parameter and all data after finishing.
                                             #l1_ratios = np.arange(0.1, 1.01, 0.1), # The LASSO / Ridge ratios.
                                            )
```

Hyperparameter selection:

- I chose l2 (ridge regularization) because all my variables are very weak predictors, and I did not want to drop them.
- Class_weights has been set to 'balanced' so that the class weights are proportionately applied to our class imbalanced dataset.
- Tolerance is chosen as such because this value is widely accepted across finanicl organisations and also because our variables are weak predictors.
- Solver is to 'saga' as it is an efficient option
- We choose refit as our Dataset is sufficiently small.

The coefficients for the parameters building the model are:

	column	0
0	age_woe	0.957131
1	payment_day_woe	0.763664
2	marital_status_woe	0.633783
3	flag_residencial_phone_woe	1.719996
4	occupation_type_woe	0.085793
5	state_of_birth_woe	0.783603
6	residencial_zip_3_woe	0.422154
7	months_in_residence_woe	0.382343
8	personal_monthly_income_woe	0.411285
9	total_income_per_dependant_woe	0.214759
10	residence_type_woe	0.542823

The trained model is the applied on the test set, and we calculate the prediction accuracy. The AUC for the model is **0.602.** It can be said that the model performance is fair, if not outstanding, but this was expected because almost all variables are weak predictors based on their IV.

I recommend using the above 10 predictors for our model.

b. Scorecard building: Using our trained model and using the scorecard.py package, we calculate the credit score all the cardholders.

Creating scorecard

```
creditdef_sc = sc.scorecard(bins_adj,          # bins from the WoE
                            creditdef_logreg,  # Trained logistic regression
                            train_woe_new.columns[1:], # The column names in the trained LR
                            points0=800, # Base points
                            odds0=0.02, # Base odds bads:goods
                            pdo=50
                            ) # PDO
```

For our scorecard, we choose **our Base Points as 800**, our odds ratio as **100:1** for good to bad (amongst 100 people of the same score, 1 person will default), and a PDO(points to double odds) as 50, which means, if the odds double, say from **100:1 to 200*sqrt(2):1**, score increases by 50. Here is the summary statistic of the score card column. It ranges from 408-623.

	score
count	49993.000000
mean	520.490709
std	31.989120
min	408.000000
25%	499.000000
50%	519.000000
75%	542.000000
max	623.000000

4.Ensemble Models:

In this section, we examine two ensemble models- RandomForest Classifier and XGBoost Classifier.

a. **Random Forest** is a popular Machine Learning Algorithm which combines the output of multiple decision trees to reach a single result. It creates an uncorrelated forest of decision trees.
   We converted all our categorical variables to dummy variables using the Pandas method **pd.get_dummies().** We split the dataset into train and test, and fit our model on the train set. Here are the hyperparameter settings for our model, which we feel are optimal given the dimension of our dataset.

```
from sklearn.ensemble import RandomForestClassifier

#Define the classifier
creditcard_rf = RandomForestClassifier(n_estimators=1000, # Number of trees to train
                       criterion='entropy', # How to train the trees. Also supports gini.
                       max_depth=None, # Max depth of the trees. Not necessary to change.
                       min_samples_split=2, # Minimum samples to create a split.
                       min_samples_leaf=0.0001, # Minimum samples in a leaf. Accepts fractions for %.
                       min_weight_fraction_leaf=0.0, # Same as above, but uses the class weights.
                       max_features='auto', # Maximum number of features per split (not tree!) by default is sqrt(vars)
                       max_leaf_nodes=None, # Maximum number of nodes.
                       min_impurity_decrease=0.00001, # Minimum impurity decrease. This is 10^-4.
                       bootstrap=True, # If sample with repetition. For large samples (>100.000) set to false.
                       oob_score=True,  # If report accuracy with non-selected cases.
                       n_jobs=2, # Parallel processing. Set to the number of cores you have. Watch your RAM!!
                       random_state=251253766, # Seed
                       verbose=1, # If to give info during training. Set to 0 for silent training.
                       warm_start=False, # If train over previously trained tree.
                       class_weight='balanced' # Balance the classes.
                                   )
```

b. **XGBoost** : Here, we create smaller , less number of correlated trees, every subsequent tree learns from the error of its predecessor and aims to minimize the error at each stage. Here is our XGBoost object.

```python
from xgboost import XGBClassifier
#Define the classifier.
XGB_Creditcard = XGBClassifier(max_depth=2,                  # Depth of each tree
                            learning_rate=0.1,            # How much to shrink error in each subsequent training. Trade-off with no. estimators.
                            n_estimators=50,             # How many trees to use, the more the better, but decrease learning rate if many used.
                            verbosity=1,                  # If to show more errors or not.
                            objective='binary:logistic',  # Type of target variable.
                            booster='gbtree',             # What to boost. Trees in this case.
                            n_jobs=2,                    # Parallel jobs to run. Set your processor number.
                            gamma=0.001,                  # Minimum loss reduction required to make a further partition on a leaf node of the tree. (Controls growth!)
                            subsample=0.632,              # Subsample ratio. Can set lower
                            colsample_bytree=1,           # Subsample ratio of columns when constructing each tree.
                            colsample_bylevel=1,          # Subsample ratio of columns when constructing each level. 0.33 is similar to random forest.
                            colsample_bynode=1,           # Subsample ratio of columns when constructing each split.
                            reg_alpha=1,                  # Regularizer for first fit. alpha = 1, lambda = 0 is LASSO.
                            reg_lambda=0,                 # Regularizer for first fit.
                            scale_pos_weight=1,           # Balancing of positive and negative weights. G / B
                            base_score=0.5,               # Global bias. Set to average of the target rate.
                            random_state=251253766,        # Seed
                            missing=None,                 # How are nulls encoded?
                            tree_method='hist',          # How to train the trees?
                            #gpu_id=0                      # With which GPU?
                            )
```

We use GridSearchCV(cross validation) on 50% of training data to find the best hyperparameters, esp number of trees, max depth of trees and the learning rate, in order to avoid overfitting. We then use those hyperparameters to fit our training set and test our prediction on our train set. The optimal hyperparameters are :

```python
# Show best params
print('The best AUC is %.3f' % GridXGB.best_score_)
GridXGB.best_params_
```

```
The best AUC is 0.609
{'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 150}
```

c. **Model comparison:** We compare the models based on their AUC , prediction accuracies and ROC curve.

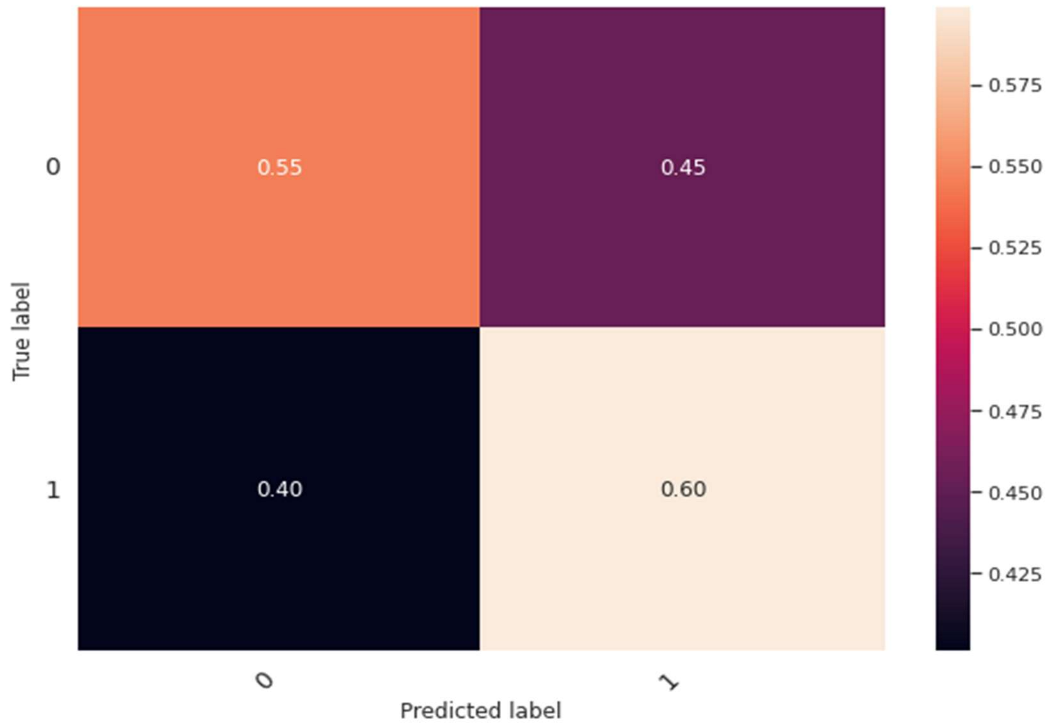
AUC Comparison:

Model	AUC
Logistic Regression Classifier	0.602
Random Forest Classifier	0.63
XGBoost Classifier	0.623

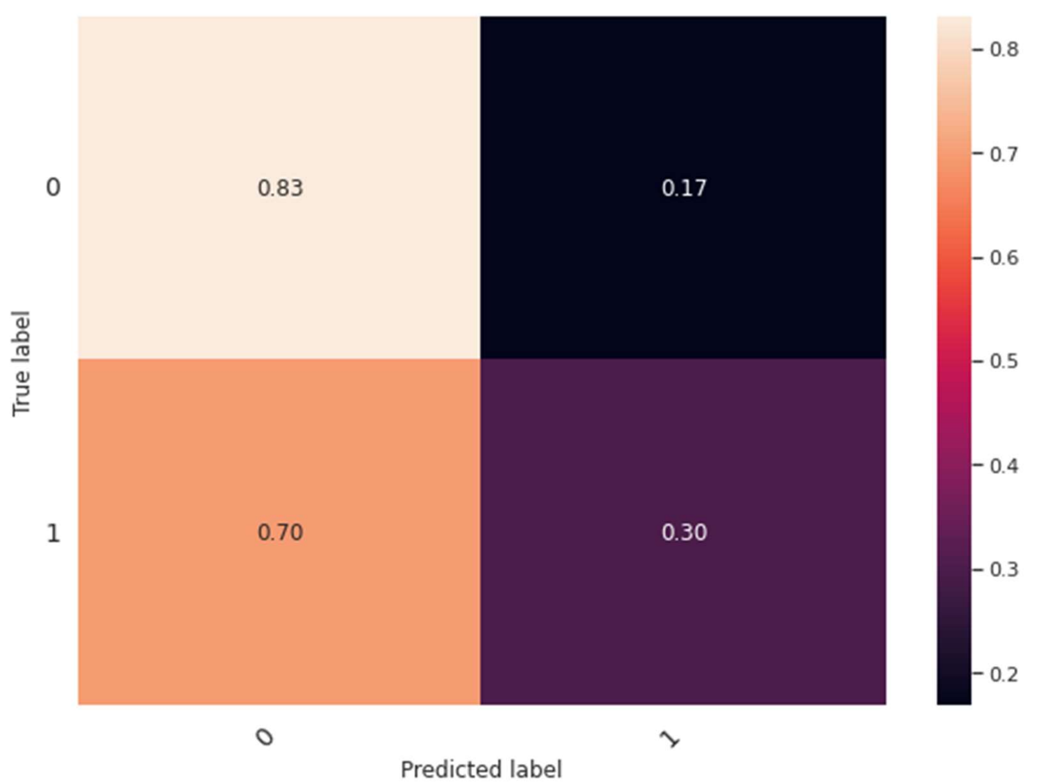
Prediction Accuracies:

Confusion Matrices for the models tell us the sensitivity and specificity of the model.

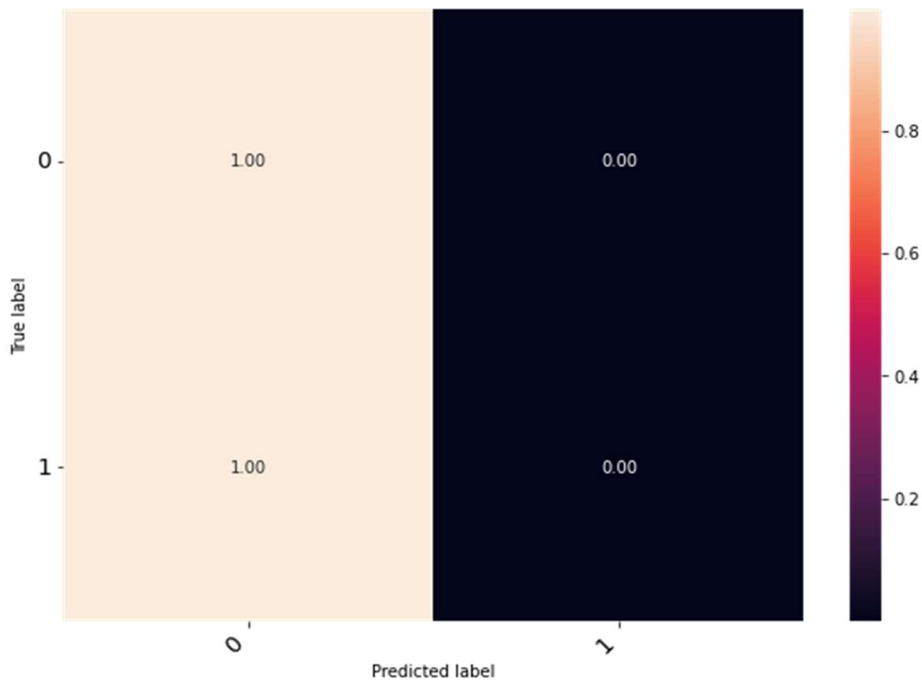
Logistic Regression



Random Forest



XGBoost

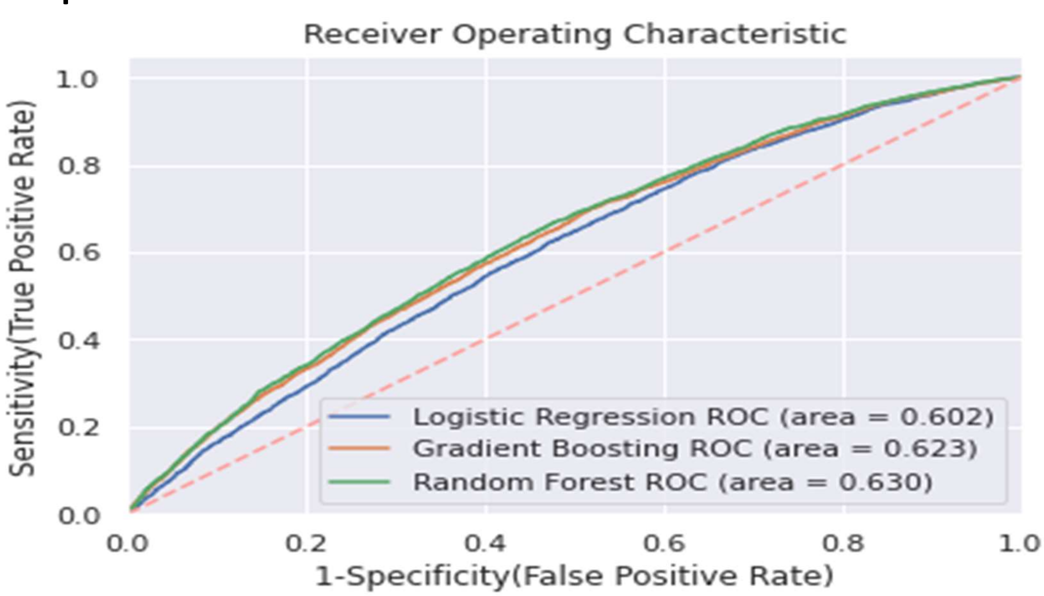


:

From the above, we conclude that LR does a fair job of predicting the good(target label 0), but RF does a better job, however, RF performs worse than LR in detecting bad(target label 0) applicants.
XGB does great job of predicting the good, however, it completely fails to detect the bad. I tweaked the decision boundary from 0.5 to 0.8, but there was no remarkable change. I assume that because of the poor quality and very small size of the class unbalanced dataset, the model doesn't have enough **class '1'** observation to train from.

ROC curve: Different models perform differently at different cut off points, therefore it is difficult to decide one single model based on ROC.

Comparison between of ROC curves:



Conclusion: Based on the different metrics like AUC score, ROC curve and confusion matrix, we can say that Random Forest should be our model of choice.

5.Feature Importance and Selection:

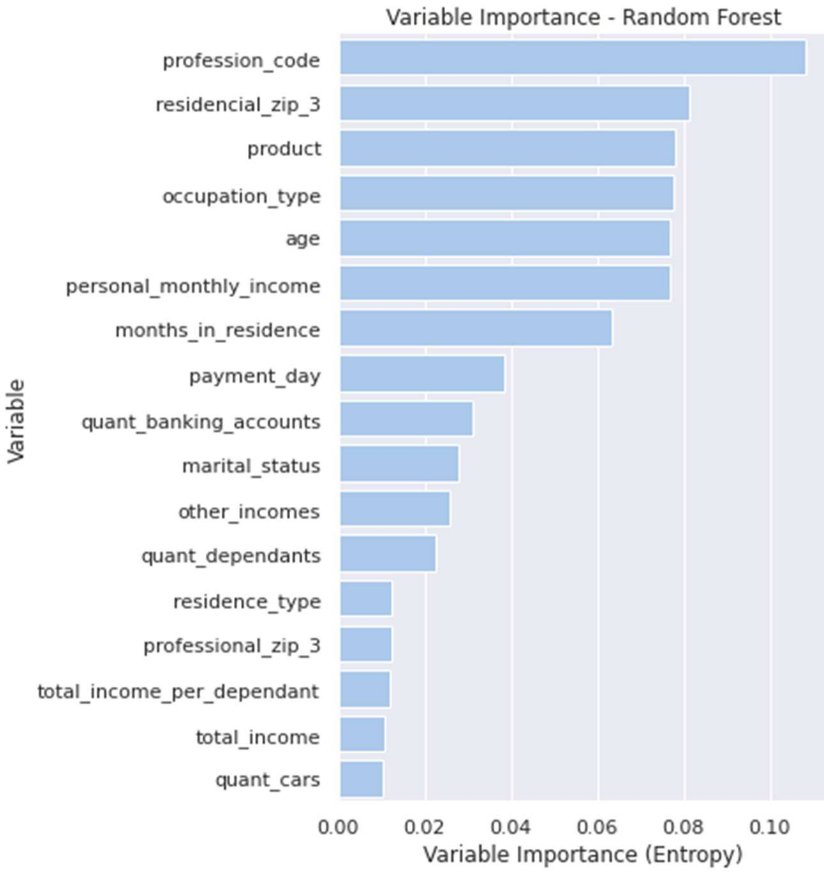
The variable selection for LR, RF and XGB algorithms is as follows:

a. **Logistic Regression (features with IV >0.002)**

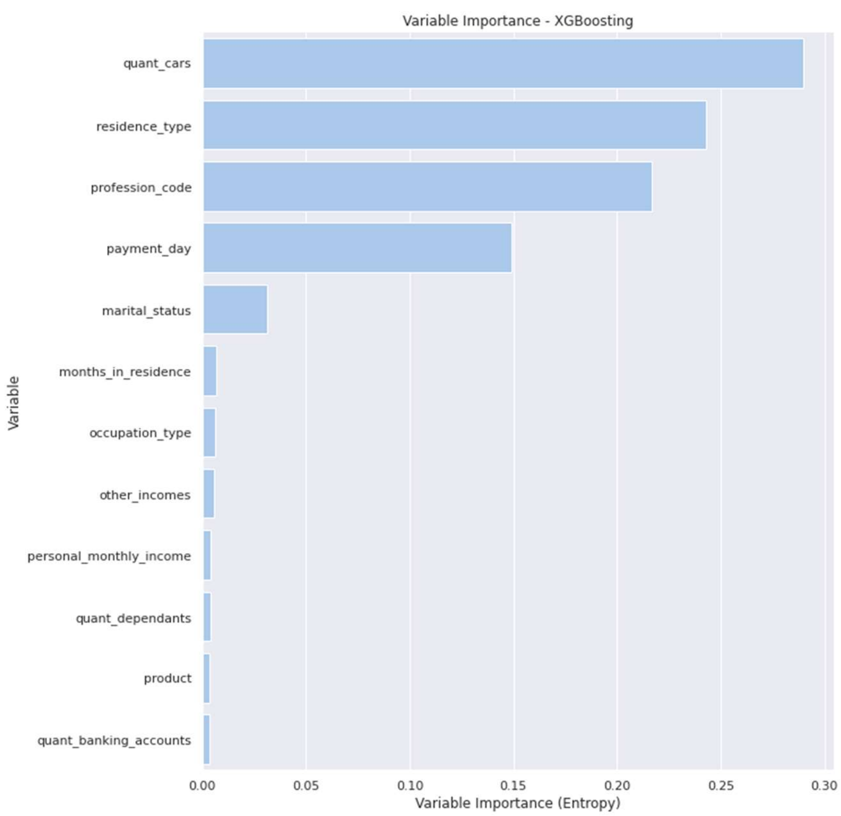
:

	column	0
0	age_woe	0.957131
1	payment_day_woe	0.763664
2	marital_status_woe	0.633783
3	flag_residencial_phone_woe	1.719996
4	occupation_type_woe	0.085793
5	state_of_birth_woe	0.783603
6	residencial_zip_3_woe	0.422154
7	months_in_residence_woe	0.382343
8	personal_monthly_income_woe	0.411285
9	total_income_per_dependant_woe	0.214759
10	residence_type_woe	0.542823

b. **<u>Random Forest：(Features with Entropy >0.01)</u>**

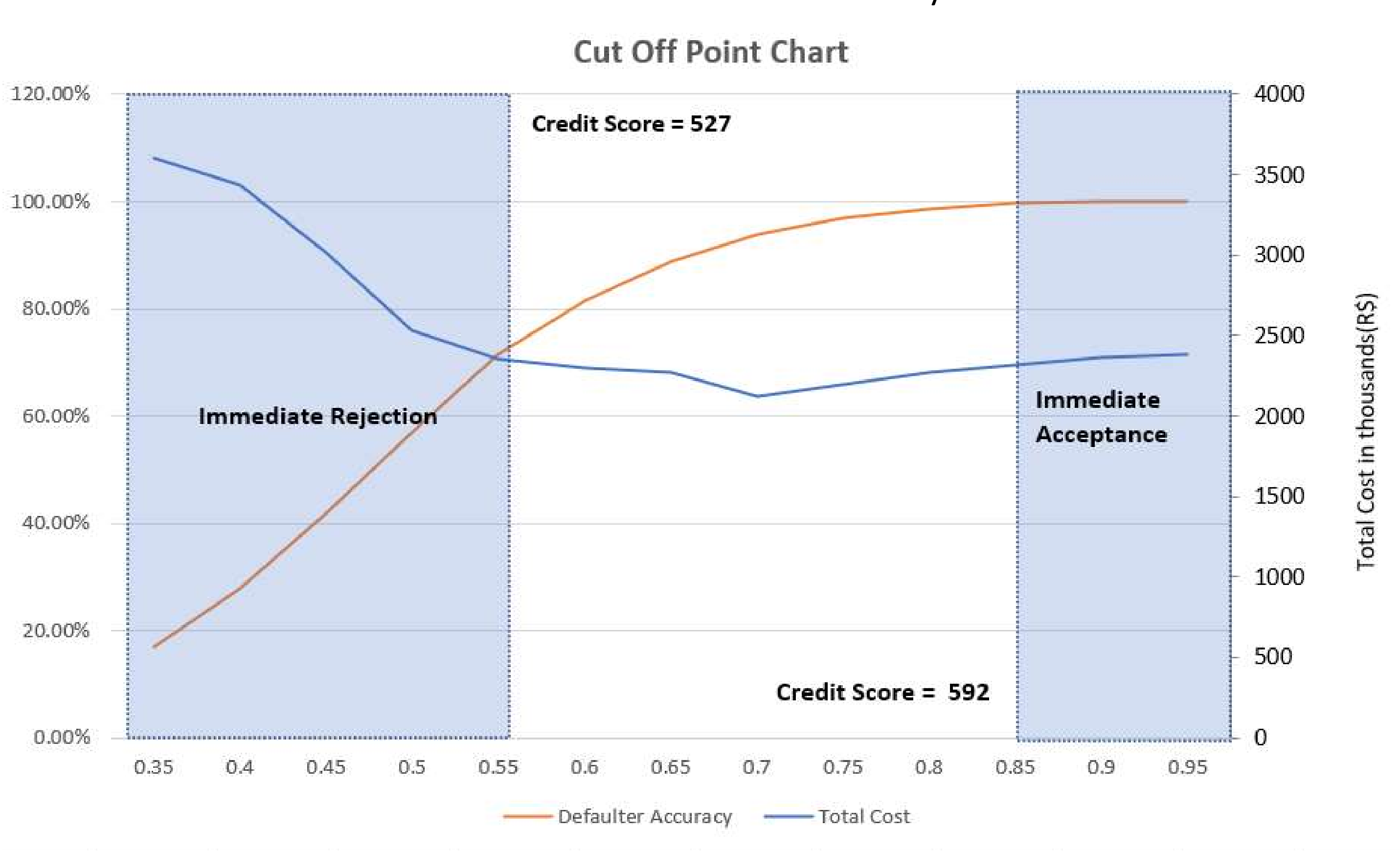


c. **<u>XGBoost: (Features with Entropy >0.005)</u>**



The variable choices are not completely similar between models, however there are some variables like **personal income, months in residence**, **age** that are common to each. The reason for this is that all models choose their optimal cuts for the predictor variables differently using different sampling methods and learning algorithms, and therefore arrive at different Entropy or Information Values.

6.<u>Two Cut Off Point Strategy:</u> Using the scorecard we calculated in step 3, we design a two-point cut off strategy. Any borrower having a credit score lower than the lower cut off point is immediately rejected, and any borrower having a credit score higher than the upper cut off point is immediately accepted. Those who fall in between the range (grey zone) usually are referred to the committee that examine the cases manually.



Here is the cutoff point table for our model:

Credit Score	Cut off	Accepted %	Accuracy			Cost(in R$)		
			Goods	Bads	Total	Good(goods misclassified as bad)	Defaulter(misclassified as good)	Total
409		0.00%	100.00%	1.54%	73.92%	0.00	3821172.41	3821172.41
452	0.2	73.27%	99.12%	1.54%	73.67%	12044.33	3855145.20	3867189.53
463	0.25	72.13%	97.58%	4.10%	73.20%	33569.05	3762590.27	3796159.32
474	0.3	69.95%	94.64%	9.01%	72.30%	77494.46	3636650.15	3714144.61
484	0.35	66.16%	89.50%	17.05%	70.60%	159453.67	3446899.22	3606352.89
495	0.4	60.90%	82.39%	27.96%	68.19%	272773.25	3164872.22	3437645.47
506	0.45	53.04%	71.76%	41.82%	63.95%	459234.04	2557892.45	3017126.49
517	0.5	43.35%	58.65%	56.82%	58.17%	700482.50	1834235.50	2534718.01
527	0.55	33.53%	45.37%	71.52%	52.19%	1014899.37	1342941.70	2357841.07
538	0.6	24.76%	33.50%	81.60%	46.05%	1315241.36	987171.14	2302412.50
549	0.65	17.49%	23.66%	88.80%	40.65%	1560191.98	710796.95	2270988.93
560	0.7	11.39%	15.41%	93.99%	35.91%	1789121.20	332936.95	2122058.15
570	0.75	6.59%	8.92%	96.96%	31.88%	1979402.69	212829.05	2192231.74
581	0.8	3.27%	4.43%	98.71%	29.02%	2221807.17	52134.06	2273941.24
592	0.85	1.25%	1.69%	99.65%	27.24%	2304177.09	13039.72	2317216.81
603	0.9	0.50%	0.67%	99.92%	26.56%	2364860.53	3095.72	**2367956.25**
613	0.95	0.05%	0.06%	99.99%	26.13%	2381517.22	576.00	2382093.22
624	1	0.00%	0.01%	100.00%	26.09%	2385738.40	0.00	2385738.40

Using our scorecard, we calculate the percentage of acceptance, accuracy of defaulters and non- defaulters and the Total Cost using the following formulae. We have completed all the calculations in excel.

% Of Acceptance = Count of 0's above cut-off point/Total Count of borrowers

Accuracy:

Accuracy % of Goods = Count of 0's above cut off point/ (Count of 0's above cut off point + Count of 0's below cut-off point)

Accuracy % of Bads = Count of 1's below cut off point/ (Count of 1's above cut off point + Count of 1's below cut-off point)

Total Accuracy % = (Count of 0's above cut off point + Count of 1's below cut off point)/ Total Number of Observation

Cost:

Opportunity Cost (Goods Classified As Bad) = Sum of Income (Credit Limit) of Rejected Goods*Average Utilization (32%)*Interest Rate(20%) for each cut off

Revenue Loss (Bads Classified As Good) = Sum of Income (Credit Limit) of Accepted Bads*Average Utilization (32%)

Based on the chart above, we decide to set the Lower and Higher Cut Offs as **527** and **592** respectively. We set the lower cutoff at the point where the decline of the Total Cost starts to flatten, any lower and we might risk accepting too many potential defaulters. We set the higher cutoff at around 0.85 and make a tradeoff between having very high accuracy of defaulters (minimizing potential revenue loss) and accepting higher number of borrowers and growth of customer base.

Appendix::

1. Link to Collab Notebook:
https://colab.research.google.com/drive/1ugjeY6Xyk9DVeIg75BqKGHq_TbavsAl0#scrollTo=sDrjUAgjN2sW

2. Code for the coursework: