## Table of Contents

## 1. Problem Statement: Improving Credit Card Approval Accuracy

If we google credit card applications, we will come across many posts such as “How to Apply for a Credit Card So You’ll Get Approved.” Financial institutes such as commercial banks processed many applications for credit cards and loans. Not all applications get approved. How banks decide which applications are qualified affects not only the applicants but also the banks themselves.

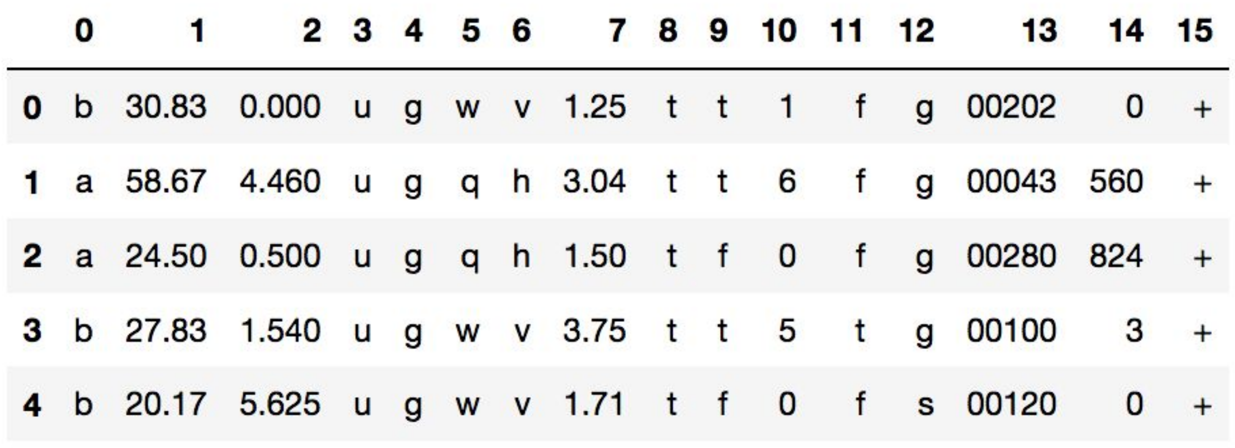
It takes about 15 seconds these days to run a credit card applications. We wonder it works. Every bank run this process automatically with the help of machine learning. Improving the accuracy of credit card approval can help the banks decrease the debt default rate. This is a major problem for banks. However, an overfitting model can decrease the number of approved application and lead to a reduce in the banks’ revenue.

In this project, we will try different models to predict credit card approvals including the newest method LGB that conquers the latest data science since.

We'll use the Credit Card Approval Dataset from the UCI Machine Learning Repository to predict credit card approval.

## 2. Exploratory Analysis

Once we read in the file, we have a table of features as follow:



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | b | 30.83 | 0.000 | u | g | w | v | 1.25 | t | t | 1 | f | g | 00202 | 0 | + |
| **1** | a | 58.67 | 4.460 | u | g | q | h | 3.04 | t | t | 6 | f | g | 00043 | 560 | + |
| **2** | a | 24.50 | 0.500 | u | g | q | h | 1.50 | t | f | 0 | f | g | 00280 | 824 | + |
| **3** | b | 27.83 | 1.540 | u | g | w | v | 3.75 | t | t | 5 | t | g | 00100 | 3 | + |
| **4** | b | 20.17 | 5.625 | u | g | w | v | 1.71 | t | f | 0 | f | s | 00120 | 0 | + |

The data frame is quite ambiguous with features being labeled by numbers, and values are alphabets and figures only. We will need to decode the most important features of a credit card application.

The features of this dataset have been anonymized to protect the privacy, but after some research, t we can guess a pretty good overview of the probable features. The probable features in a typical credit card application are Gender, Age, Debt, Married, BankCustomer, EducationLevel, Ethnicity, YearsEmployed, PriorDefault, Employed, CreditScore, DriversLicense, Citizen, ZipCode, Income and finally the ApprovalStatus.

Features 1 and 13 appear to be numerical, but once we you .info() to see which are object or numerical, we soon realize 1 and 13 are in fact objects. One can guess column 12 as zip codes since it has 5 digits, similarly, column 1 as age with a range less than 99.

## 3. Data wrangling: handling missing values

Missing data will affect our models. Data wrangling will help with modelling.

- Our dataset contains both numeric and non-numeric data. Features 2, 7, 10 and 14 contain numeric values and all the other features contain non-numeric or object values.
- The dataset also contains values from several ranges. Some features have a value range of 0 - 28, some have a range of 2 - 67, and some have a range of 1017 - 100000. Apart from these, we can get useful statistical information (like mean, max, and min) about the features that have numerical values with .describe()
- Finally, the dataset has missing values labeled with '?', which can be seen in the last cell's output.

We replace all the question marks with NaNs. Handling missing values is important because while some models can be ran with missing values, ignoring them can decrease the accuracy of a model. Plus, there are many models which cannot handle missing values implicitly such as LDA.

We are going to impute the missing values with a strategy called mean imputation using .fillna() and .mean() for numeric columns.

For non-numeric columns 0, 1, 3, 4, 5, 6 and 13, the mean imputation strategy would not work here. We are going to impute these missing values with the most frequent values as present in the respective columns using .fillna() and .value_counts()

## 4. Preprocessing data

We have handled missing data and got a clean dataset.

We need some essential data preprocessing before we build our machine learning model. Our preprocessing steps include:

1. Convert the non-numeric data into numeric.
2. Split the data into train and test sets.
3. Scale the feature values to a uniform range.

First, we will be label encoding all the non-numeric values into numeric ones for faster computation and to fit that with machine learning models (like XGBoost) that require the data to be in a strictly numeric format. We do this using LabelEncoder()

Splitting data into two sets: training and testing to ensure that testing data won't be altered in the fitting process. We import train_test_split from model selection

The ability to drive and credit approval aren't related. We drop DriverLicense as feature selection. Another column that we drop is ZipCode. As a rule against red-lining in real estate, zip code is not a determining factor in credit approval.

Once the data is split into two separate sets - train and test sets respectively, we will proceed to the final preprocessing step of scaling before fitting.

For example, we will scale CreditScore with 1 as the highest and 0 as the lowest to determine one's creditworthiness based on their credit history. The higher this number, the more financially trustworthy a person is considered to be. We're rescaling all the values to the range of 0-1.

## 5. Logistic regression model

Credit card application approval prediction is a classification task. The result is either approved or not approved. In our dataset obtained from UCI, there are more denied instances (55.5%) than approved (44.5%). This is our benchmark. Our model needs to be accurately predict the approval rate within this ratio.

In this notebook, we will conduct 4 different models: logistic regression as the classical classification learning machine method, LGB as the newest and most advanced one, decision tree and also gradient boosting methods.

## 6. Making predictions and evaluating performance

We will evaluate our models using classification accuracy score. We will look into the confusion matrix of logistic regression and LGB because the true negative or the rate of denied application got predicted as denied helps us evaluating performance of our models. Because if our model is not performing well, denied applications might get approved and resulting in defaults or collections.

Our logistic regression model was pretty accurate with the accuracy score of 0.8377192982456141It was able to yield an accuracy score of almost 84%, with true negative as 0.93 which shows the model performs pretty well.

We continue to tune our model using a grid search of the model parameters, in this case, two: tol and max_iter

## 7. Finding the best performing model

For logistic regression model, we use GridSearch to tune the model for a better result. We supply X (scaled version) and y to fit the model, then perform a cross-validation of five folds.

We also carry out 3 more models: LGB, Gradient Boosting and Decision Tree.

We find that Decision Tree has the best accuracy score of 0.868421052631579, higher that Logistic Regression with Grid Search, and LGB.