# PREDICTING CREDIT CARD DEFAULT

**Group F:** Nicolas Ford, To Eun Kim, Zi Lian Lim, Xiaoqi Tan, Chi Xue, Jingting Yan
Department of {Computer Science, Biomechanical Engineering}
University College London
London, WC1E 6BT

January 17, 2021

## 1 Introduction

In the past few decades, machine learning has made significant breakthroughs and contributions in the financial space, revolutionizing the way financial institutions and businesses perform their day-to-day activities. Due to increasing and varying customer expectations, and an increase in market competition of Fin-tech players, financial institutions turned to machine learning and data analysis as a key instrument to improve customer experience and relations. This paper aims to explore and experiment with a range of different machine learning techniques to identify a stronger predictive model when classifying potential defaulting accounts utilizing data sets that contribute to credit card defaults.

One of our contributions include identifying an algorithm that is more resilient to an imbalanced dataset distribution. Previously, Brown and Mues (2012) compared different techniques to classify imbalanced credit score and showed that the random forest and gradient boosting perform better in dealing with imbalanced distribution[2]. Also, Neema and Soibam (2017) analysed 7 different machine learning methods that predict credit card default and concluded that Random Forest and Artificial Neural Network are superior to other models[8]. Thus, we decided to choose three main categories of methodologies to attempt: Decision Trees (AdaBoost, XGBoost and Gradient Boosting), Random Forest and Artificial Neural Networks. After comparing their performances, we chose XGBoost as our final methodology.

## 2 Data Exploration & Transformation

### 2.1 Data Cleaning and Exploration

The goal of data exploration is to gain insight and better understanding of our dataset. This will help with data transformation which will create good-quality data to be fed into our model. Such good-quality data is a key to reaching a good performance in the field of Machine Learning.

We start by cleaning our dataset (**Notebook Section 3.2**). There were three types of undocumented values present: EDUCATION (containing 0, 5, 6), MARRIAGE (containing 0) and PAY_n (containing -2, 0). During the data cleaning process, unknown values in EDUCATION and MARRIAGE were all mapped to 'others'. Values of -2 in PAY_n were considered as 'no consumption'[10], and thus were not changed. Values of 0 in PAY_n were understood as paying 0 months late i.e paying duly, and were thus reassigned to class -1.

Next, we moved on to data exploration by searching for outliers in the numerical features (**Notebook Section 3.1.3**). First, negative values were found in the bill statement features (BILL_AMT1 to BILL_AMT6) where only values greater or equal to zero should be expected. Our solution for this was to keep these negative values since they can still represent uncommon payment behaviour, where someone pays more than what their bill states. Second, we checked if credit (LIMIT_BAL) have extremely large or small values, by selecting values that are either below the 5th percentile or above the 95th percentile. Then we checked if the amount of previous payment (PAY_AMT) and bill statement (BILL_AMT) exceeds 2 * LIMIT_BAL. If it exceeds, then the relationship between credit and payments does not make sense; and therefore, this sample worth a closer examination. This criteria filtered out 5 records. However, after further examination, they looked quite normal: their PAY_AMT and BILL_AMT are relatively balanced in general and most of these values lie within the given credit. Therefore, we did not regard them as outliers.

For categorical features (Sex Education, Marriage and PAY_n) we checked their distribution by plotting bar charts, and histograms for numerical values (**Notebook Section 3.3.1 and 3.3.2**). Next, we checked the correlation between features using box plots and heatmaps (**Notebook Section 3.3.3**). After that, to get a general understanding of importance of features and to set a certain goal for data transformation, a feature importance test was done using the cleaned data (**Notebook Section 3.3.4**). This revealed that PAY_1 to PAY_6 are the most important features in classifying defaults.

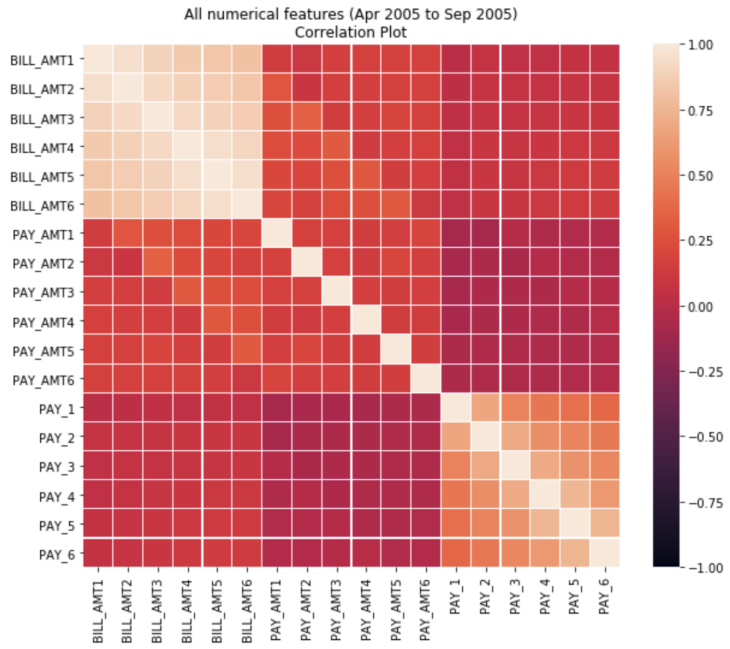



Figure 1: Correlations of features

Figure 1 is the heatmap of all numerical features, showing the correlations between them. As can be seen, PAY_n values have a higher correlation only with themselves and very low correlation with PAY_AMT. However, PAY_AMT and BILL_AMT have relatively high correlations. Considering both the heatmap and feature importance test, we came up with two ideas. Firstly, we should not transform PAY_n features because these are the most important. Secondly, new features can be made by combining two correlated features (PAY_AMT and BILL_AMT) to make semantically meaningful features (dealt with more in 2.2).

### 2.2 Data Pre-processing and making of new features

Based on the findings that we have gained in 2.1, we tranform our data as follows (**Notebook 3.4.1**):

- **Make Age bins**
  Convert Age into a categorical value using a range condition.
- **Combine Sex and Marriage**
  Encode each Cartesian combination to an integer of value 1 to 6, to reduce feature dimensionality.
- **New feature: Closeness**
  Difference between credit (LIMIT_BAL) and each bill statement
- **New feature: Difference between bill and amount paid**
  Difference between the sum of bill statements (BIL_AMT1 to BIL_AMT6) and the sum of previous payments (PAY_AMT1 to PAY_AMT6).

We applied the same cleaning and preprocessing pipeline on the test data before making predictions with it.

### 2.3 Dealing with imbalanced distribution

In the training set, there are 18,630 samples classified as 0 (no default) and 5,370 samples classified as 1 (default). The probability of default samples over all samples is around 0.22. This imbalanced dataset would cause shifting of the learning. Learning methods only need to return a learner that always predicts new samples as 1, achieving around 80 percent accuracy. Therefore, it is necessary to come up with a remedy for the imbalanced class-distribution.

Two popular techniques for balancing class-distribution are under-sampling and over-sampling. When it comes to the credit card dataset, down sampling non-default is not desirable since around 55 percent of data would need to be

dropped, which will induce severe model under-performance. Therefore, default labels are over-sampled using the Synthetic Minority Over-sampling Technique (SMOTE).

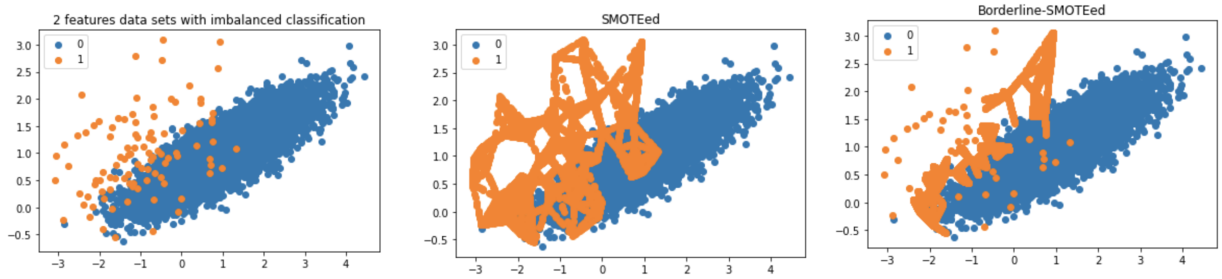
SMOTE over-samples the minority group by introducing synthetic samples along the line segments joining randomly chosen k nearest neighbours[4]. Synthetic samples can solve the problem with unbalanced classes. However, SMOTE can cause overfitting in some cases. According to Bunkhumpornpat et al. (2009), SMOTE generalised the minority classes without considering the majority classes, causing overlapping between classes since it does not consider the distribution of the majority samples around the minority samples[3]. Thus, Borderline-SMOTE was used instead to tackle this issue.

Borderline-SMOTE, created by Han et al.(2005), divides minority samples into 3 classes:

- SAFE: Half or more samples around the selected samples are in the minority class
- DANGER (near the border-line): Half or more samples around the selected samples belong to the majority class
- NOISE: All samples around the selected sample are in the majority class[6]

Samples on or near the borderline of each class are easier to be misclassified than those further away from the borderline. Borderline-SMOTE only over-samples DANGER samples, rather than all existing minority samples.

Figure 2 displays how SMOTE and Borderline-SMOTE perform in randomly generated 2-feature datasets. SMOTE over-samples all minority data (See Figure 2b) while Borderline-SMOTE over-samples only data classified as DANGER (See Figure 2c).




(a) Plot of imbalanced data set (b) After SMOTE (c) After Borderline-SMOTE

Figure 2: Comparison of performance of SMOTE and Borderline-SMOTE

For imbalanced data, using accuracy as a main metric can be deceptive in that it can go high if the learner classifies all samples to the majority class. Therefore, we need to seek other assessment methods for evaluation. ROC-Curve is one of the most popular ways to evaluate over imbalanced datasets[6]. By summarising 20 different algorithms on six criteria, Song and Peng (2019) showed that no algorithm can perform better under all criteria[9]. However, AUC, F-measure, and FN-rate were found to be the most important performance measures with the highest weight. We have adopted this fact and utilised AUC, F-measure (esp. f1-score) and FN-Rate as our standard evaluation metrics.

### 2.3.1 Data Normalisation

Data scaling is done in **Notebook Section 3.4.3**. MinMaxScaler is used as a normalisation scaler. It is first scaled with the (non-)oversampled training data. Then, the exact same scaling for training data is applied to the test data. This is done by storing the scale and offset used with the training data, and using that again on the test set.

## 3 Methodology Overview

Figure 3 describes the workflow of our methodology. We were given a dataset split into a training set and a test set in a ratio of 8:2. As specified above, we transformed, oversampled and scaled the dataset before feeding it into the models. Notable is that in the phase of oversampling, we experimented with both oversampled and non-oversampled data, thus creating 10 different paths within our workflow (ANN with and without oversampling, GBRT with and without oversampling, and so on).

After that, we trained and validated every baseline model using K-Fold cross validation. We evaluated these 10 baseline models by comparing their F1-score to choose the optimal model, which was non-oversampled XGBoost. This baseline

model then goes through various hyperparameter tuning steps to perform even better on the task with an optimal set of hyperparameters.

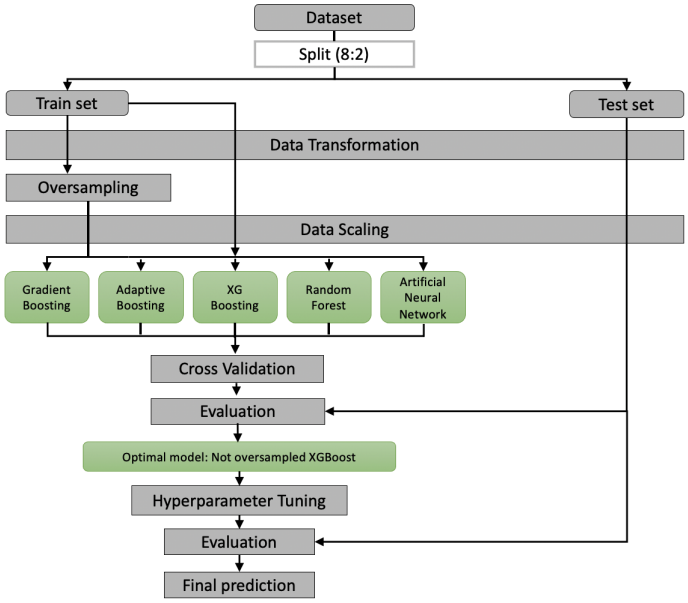



Figure 3: Overall workflow of model development for the classification

Below (Section 3.1 and 3.2) is a brief overview of each methodology we attempted.

### 3.1 Ensemble Learning

#### 3.1.1 Boosting: Adaptive Boosting Classifier (Notebook Section 4.1)

AdaBoost, or 'Adaptive Boosting', is regarded as an ensemble approach among machine learning algorithms. In each iteration, weights are reassigned to each individual model, where larger weights are assigned to the models with higher error rates [7]. Each subsequent learner is updated based on its previous respective learner by reducing the bias and variance. This speciality, as well as weight reassigning, makes AdaBoost different from other boosting approaches[7].

#### 3.1.2 Boosting: Gradient Boosting Classifier (Notebook Section 4.3)

For each given model and feature we obtain, GBRT uses an exhaustive method to get the threshold of each feature. This is done by finding the two branches that make the feature less than and more than the threshold. Two new nodes are created according to this criterion of branching and a prediction value is obtained at each node. Log loss is chosen as the loss function to find the most reliable branching method. Moreover, this approach can naturally handle different types of data with various features while being extremely robust to out-of-space outliers. This is possible due to utilising robust loss functions.

#### 3.1.3 Boosting: XGBoost (Notebook Section 4.5)

XGBoost (which stands for "Extreme Gradient Boosting"), together with the decision tree ensembles model, is used for supervised learning problems. The tree ensemble model consists of a set of classification and regression trees (CART). Unlike decision trees, each of the leaves in CART is associated with a score that takes both decision values and the model complexity into account. Also, the ensemble model sums the prediction of multiple trees together, where the trees try to complement each other.

Compared with the standard Gradient Boosting Machine, XGBoost has two advantages. Firstly, XGBoost takes regularisation into account. Secondly, it is highly flexible allowing users to define custom loss functions, from logistic regression to pairwise ranking. It also has a built-in booster parameter scale_pos_weight, which can help with imbalanced datasets.

The training set was further split into a smaller training set (60%) and a validation set (20%). Keeping the rest of the booster parameters as their default values, we trained a baseline model. During the training, the evaluation metric

('logloss') was applied on the validation set to assess performance for each round. Although we set the number of boosting rounds to a large value, we did expect the model to finish early by triggering early_stopping_rounds=10. This means that, if the performance (for validation set) did not improve for 10 rounds, the model would stop training and keep the current number of boosting rounds as the optimal.

#### 3.1.4 Bagging: Random Forest (Notebook Section 4.2)

Random Forest are made up of multiple decision trees which are composed of many classification trees. As each tree provides a classification, the whole forest will investigate all trees and finally choose the most popular classification [1]. For the formation of a single tree, a sample with capacity of N is drawn N times with a put-back, 1 sample for each, resulting in N samples. These selected N samples are used to train a decision tree at the root node. Suppose each sample has M attributes, when each node of the decision tree needs to be split, m attributes are randomly selected from these M attributes, satisfying the condition m « M. Then a strategy is implemented to select one attribute from these as the splitting attribute of the node. Each node is split by following this step in the decision tree and continues until it is no longer possible to be split. No pruning is done during the entire process of a single tree formation. After completion, a random forest will form with a large number of trees.

### 3.2 Artificial Neural Network (ANN) (Notebook Section 4.4)

An ANN is designed to simulate the way the human brain analyzes and processes information[5]. It learns data by forward and back propagation of weights, followed by weight updates. Backpropagation is a method used to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight is computed using the chain rule.

While preparing the training dataset that will be fed into the neural network, we use the same x_train, but different y_train compared to the other method. Here, we make our y labels categorical one-hot vectors, which are more suitable for an ANN input.

In our notebook **Section 4.4.2**, the ANN model structure is formed. We make 6 layers, which includes 1 input layer, 1 output layer and 4 hidden layers. For input and hidden layers, ReLu (Rectified Linear Unit) is used as the activation function for the ANN to learn the non-linearity of the dataset. Whereas in the final output layer, we have 2 neurons in charge of outputting a 2-entry vector (containing 0 and 1), followed by a softmax activation function. Softmax is employed as this is a classification task. Output values from the last layer have to be mapped to a probabilistic distribution.

To optimise the convergence of the cross-entropy loss function, the adaptive learning rates method is used with the Adam optimiser with an initial learning rate of 0.001. As the model is trained, the learning rates are varied, reducing the training time and improving the numerical optimal solution.

In addition to that, the Dropouts and Earlystopping methods are used to prevent overfitting. Dropout prevents overfitting the training data by dropping neurons with probability greater than 0. It forces the model to avoid relying too much on particular sets of features. Also, Earlystopping stops the training process as soon as the validation loss reaches a plateau or starts to increase. This can be tweaked with the 'patience' parameter.

### 3.3 10 Models Prediction Performance Comparasion

Table 1: Performance of 10 Baseline models.

| Model | Accuracy | F1-score | FP-rate | FN-rate |
|---|---|---|---|---|
| Gradient Boosting Regression Tree (GBRT) | 0.831 | 0.465 | 0.04 | 0.652 |
| AdaBoost | 0.831 | 0.448 | 0.034 | 0.675 |
| XGBoost | 0.792 | **0.525** | 0.142 | **0.455** |
| Random Forest | 0.821 | 0.37 | 0.026 | 0.75 |
| Artificial Neural Network (ANN) | 0.833 | 0.486 | 0.044 | 0.626 |
| *Oversampled* GBRT | 0.827 | 0.506 | 0.064 | 0.58 |
| *Oversampled* AdaBoost | 0.794 | 0.512 | 0.131 | 0.487 |
| *Oversampled* XGBoost | 0.798 | 0.471 | 0.102 | 0.574 |
| *Oversampled* Random Forest | 0.817 | 0.464 | 0.065 | 0.625 |
| *Oversampled* ANN | 0.803 | 0.496 | 0.105 | 0.54 |

In the scenario of credit card default detection, both false negatives and false positives should be avoided. From the perspective of credit card issuers, a false positive means losing a potential customer and a false negative means losing money. It is in their interest to minimise both of these metrics. Therefore, the F1 score is the best metric to use for model evaluation. Moreover, since our dataset is imbalanced between positive and negative samples, F1-score is more valuable than ROC AUC. F1 score keeps a balance between Precision and Recall; whereas solely looking at these two scores gives misleading results for uneven class distribution. Therefore, when the F1 score is improved, both Precision and Recall are improved, and the model's overall performance is improved. However, ROC AUC is not a good indicator for imbalanced data, because the False Positive rate does not drop drastically when the total number of real Negatives is large.

As observed in Table1, in general, with oversampling, we can see a higher F1-Score, lower False Negative rate, higher False Postive rate with similar accuracy. However, considering that the F1-Score and FN-rate are the most important metric, the XGBoost model which trained on a normal (not oversampled) dataset scored the highest F1-Score and the lowest FN-rate. Therefore, we chose the XGBoost with non-oversampled baseline model as our final methodology that will be optimised further in the following sections.

## 4 Model Training & Validation

The decision tree ensembles model learns the trees by defining an objective function and optimising it. The objective function consists of two parts: the training loss function, which measures how predictive the model is with respect to the training data, and the regularisation term, which controls the complexity of the model to avoid overfitting. Mathematically, the objective function to be optimised is given by

$$\text{obj} = \sum_{i=1}^{n} l\left(y_i, \widehat{y}_l\right) + \sum_{k=1}^{K} \Omega\left(f_k\right)$$

where K is the number of trees, f is a function in the functional space F, and F is the set of all possible CARTs.

In each training step, XGBoost uses an additive strategy to learn the tree structure. To further elaborate, since it is intractable to enumerate all possible tree structures, XGBoost optimises one level of the tree at a time. It splits a leaf into two leaves and adopts this split only if the score it gains is higher than the original leaf score. For real valued data, XGBoost would first place all the instances in sorted order. Then, a left-to-right scan is sufficient to calculate the structure score of all possible split solutions. Thus, it is efficient to search for an optimal split. We aim to obtain a model that has a good bias-variance trade-off, i.e. is both simple and predictive.

For this binary classification task of the 'Default of Credit Card Clients' dataset, our XGBoost objective function used logistic regression as our training loss function and cross-entropy loss as the evaluation metric for the validation data.

### 4.1 Setting up the XGBoost model

We first defined the tree booster parameters and the learning task parameters for constructing our model. Our objective function is Logistic Regression ('binary:logistic') and our evaluation metric for training is cross-entropy loss ('logloss'). Below is the list of booster parameters with their default values:

| | | | |
|---|---|---|---|
| 'max_depth': 6 | 'min_child_weight': 1 | 'eta': 0.3 | 'subsample': 1 |
| 'colsample_bytree': 1 | 'gamma': 0 | 'alpha': 0 | 'lambda': 1 |

'scale_pos_weight' is another XGBoost built-in parameter designed to tune the behaviour of the algorithm for imbalanced classification problem. It helps in faster convergence in case of high class imbalance. In general,'scale_pos_weight' is the ratio of the size of the negative class to the positive class. Since our training set is imbalanced, we update this parameter with the value of (no. of nondefault clients / no. of default clients) = 3.518.

Despite setting the number of boosting rounds to 1000, we expected the model to finish training early due to setting triggering early_stopping_rounds=10.

### 4.2 Hyperparmeters Tuning

Overall, we tuned five tree-specific parameters: max_depth, min_child_weight, subsample, colsample_bytree, gamma, and two regularization parameters: alpha, lambda. We also tuned the learning rate and found the optimal boosting rounds. In each tuning, we defined a set of values to be tested then performed the XGBoost built-in cross validation (5-fold) function on them. We started with setting wider ranges for the parameter to be tuned and then would perform

another iteration for smaller ranges until locating the optimal value. The '.cv' function performs cross-validation at each boosting iteration and returns the optimum number of trees required. The cross validation considered cross-entropy loss (logloss) and would return evaluation results for each testing value, which calculated 4 values: mean & std logloss for the training folds, mean & std logloss for the test fold. We decided to choose the parameter value with the minimum test-logloss-mean. We also managed to find the (at least) local minimal point by plotting the test-logloss-mean results and choosing the point at the vertex of a 'V' shape.

*(Note that due to the page limits, we converted the graphs to tables to save spaces. For a more intuitive visualisation, please refer to the graphs in the corresponding section in the notebook.)*

#### 4.2.1 Tuning 1: Max_depth and min_child_weight

'max_depth' defines the maximum depth of a tree; 'min_child_weight' defines the minimum sum of weights of all observations required in a child. They are used to control over-fitting as higher values will allow the model to learn relations which are very specific to a particular sample. Also, since these two parameters are correlated, we performed cross validation on them at the same time, i.e. conducting grid search to test the combination of their different values.

Table 2: Cross Validation Results for Different Values for **max_depth** and **child_weight**

| (**max_depth**, **child_weight**) | Log Loss Mean | Boost Rounds |
|---|---|---|
| (8, 2) | 0.518665 | 56 |
| (8, 3) | 0.521234 | 54 |
| (8, 4) | 0.520496 | 61 |
| (9, 2) | 0.516898 | 44 |
| (9, 3) | 0.515872 | 58 |
| (9, 4) | 0.517881 | 61 |
| (10, 2) | 0.516704 | 34 |
| (10, 3) | 0.517901 | 45 |
| (10, 4) | 0.520572 | 44 |

We first tested wider intervals between values for these two parameters: max_depth=[3,5,7,9], min_child_weight=[1,3,5], performing a grid search for these 12 combinations. The ideal values are 9 for max_depth and 3 for min_child_weight. We searched for optimum values one above and below the ideal values. The testing results for this step can be observed in Table 2. max_depth=9 and min_child_weight=3 remained the optimal parameters, with the lowest mean logloss and acceptable boost rounds. The graphs in **Notebook Section 5.2.1** demonstrated the above procedure more intuitively.

#### 4.2.2 Tuning 2: Subsamples and colsample_bytree

'subsample' denotes the fraction of observations to be randomly sampled for each tree; 'colsample_bytree' denotes the fraction of columns to be randomly sampled for each tree. Lower values of these two parameters can make the model more conservative, prevent overfitting and decorrelate the trees. However, too small values might lead to under-fitting. Since these two parameters are correlated, we performed cross validation on them at the same time.

Table 3: Cross Validation Results for Different Values for **subsamples** and **colsample_bytree**

| (**subsamples**, **colsample_bytree**) | Log Loss Mean | Boost Rounds |
|---|---|---|
| (0.95,0.65) | 0.514187 | 48 |
| (0.95,0.7) | 0.522073 | 47 |
| (0.95,0.75) | 0.512343 | 48 |
| (1.0,0.65) | 0.517326 | 54 |
| (1.0,0.7) | 0.516431 | 59 |
| (1.0,0.75) | 0.515612 | 42 |

We started with testing wider intervals between values for these two parameters: subsamples=[0.7,0.8,0.9,1.0], colsample_bytree=[0.7,0.8,0.9,1.0], performing a grid search for these 16 combinations. The ideal values are 1.0 for subsamples and 0.7 for colsample_bytree. We searched for optimum values 0.5 above and below the ideal values (note that 1.0 is the maximum). The testing results for this step can be observed in Table 3. We got the new optimal parameters as subsamples=0.95 and colsample_bytree=0.75, with the lowest mean logloss and an acceptable number of boost rounds. The graphs in **Notebook Section 5.2.2** demonstrated the above procedure more intuitively.

#### 4.2.3 Tuning 3: Gamma

'gamma' is a tree complexity parameter which specifies the minimum loss reduction required to make a split. Tuning it would make the model more conservative. To further explain, XGBoost prunes the tree by calculating the difference between gain and gamma. If negative, then prune.

Table 4: Cross Validation Results for Different Values for **gamma**

| **gamma** | Log Loss Mean | Boost Rounds |
|---|---|---|
| 0.0 | 0.512343 | 48.0 |
| 0.1 | 0.515057 | 59.0 |
| 0.2 | 0.516438 | 54.0 |
| 0.3 | 0.514048 | 65.0 |
| 0.4 | 0.515415 | 57.0 |
| 0.5 | 0.514597 | 45.0 |
| 0.6 | 0.515690 | 50.0 |
| 0.7 | 0.514366 | 61.0 |
| 0.8 | 0.516375 | 53.0 |
| 0.9 | 0.516102 | 47.0 |
| 1.0 | 0.515819 | 47.0 |

We first tested wider interval between values for gamma: [0,1,2 ... ,10], performing cross validation for each. The ideal value is 0. We then searched for an optimum value in a narrower interval between 0 and 1: [0,0.1,0.2, ... ,1.0]. The testing results for this step can be observed in Table 4. gamma=0 remained the optimal parameter, with the lowest mean logloss and small boost rounds. The graphs in **Notebook Section 5.2.3** demonstrate the above procedure more intuitively.

#### 4.2.4 Tuning 4: Regularisation parameters: alpha and lambda

'alpha' (analogous to Lasso regression) and 'lambda' (analogous to Ridge regression) are the regularisation terms on weights for XGBoost objective function. Both of them can be tuned to reduce overfitting. In addition, alpha can be used in case of very high dimensionality to speed up the model when implemented.

We tuned these two regularisation parameters separately. For each of them, we started testing for a broad range of values around the local minimum, and eventually converging to it. We constructed several parameter test sets until finally choosing alpha = 8.5 and lambda = 30 as the optimal values. The entire process of narrowing down the optimal values and their 'V-shape' graphical results can be found in **Notebook Section 5.2.4**.

#### 4.2.5 Tuning 5: Learning rate

'eta' (learning rate) makes the model more robust by shrinking the weights each step. However, a higher eta usually means it takes longer to train the model. The testing of different parameter values can be observed in Table 5.

Table 5: Cross Validation Results for Different Values for **eta**

| **eta** | Log Loss Mean | Boost Rounds |
|---|---|---|
| 1.0 | 0.543979 | 30.0 |
| 0.5 | 0.514181 | 70.0 |
| 0.3 | 0.503741 | 120.0 |
| 0.1 | 0.497889 | 370.0 |
| 0.05 | 0.496983 | 724.0 |
| 0.01 | 0.518110 | 999.0 (upper limit reached) |
| 0.005 | 0.532402 | 999.0 (upper limit reached) |

As can be seen from the table and also from the graphs in **Notebook Section 5.2.5**, eta=0.05 has the lowest test-logloss-mean but a high number of boost rounds (724 rounds). However, it didn't take a long time to run (63 seconds). Hence, we decided to consider 0.05 as the optimum.

# 5 Results

Below is the list of booster parameters with their optimal values after tuning:

| | | | | |
|---|---|---|---|---|
| 'max_depth': 9 | 'min_child_weight': 3 | 'eta': 0.05 | 'subsample': 0.95 | 'colsample_bytree': 0.75 |
| 'gamma': 0 | 'alpha': 8.5 | 'lambda': 30 | 'scale_pos_weight': 3.518 | |

We re-trained our model with these parameter values, observing logloss (cross-entropy loss) on the validation set. It stopped training at 544 rounds with a logloss value of 0.491. Compared with the lowest logloss value returned by our baseline model (0.505 in **Notebook Section 4.5.2**), we achieved a 2.77% reduction. Therefore, 544 is the optimal number of boost rounds for this model. We put this number into our model and removed the early_stopping_rounds parameter. Finally, we have obtained an optimally trained XGBoost model.

# 6 Final Prediction on Test Set

Before performing predictions on the given test set, we cleaned the test data using the same data transformation logic described in **Notebook Section 2**. Since our objective function is logistic regression, the predictions, ranging from 0 to 1, output the probability of being in the positive class. We classified all the predictions with probability greater than or equal to 0.5 as Default ("1"); and the rest as No Default ("0").

## 6.1 Evaluation statistics

We investigated the following five statistics for evaluating the performance for the predictions: accuracy score, f1 score, ROC AUC, False Positive, and False Negative. For a binary classification class, the accuracy score is used when the True Positives (TP) and True Negatives (TN) are more important while F1-score is used when the False Negatives (FN) and False Positives (FP) are crucial. F1 Score is a comparison indicator between Precision and Recall:

$$\mathbf{F}_1 = \frac{2}{recall + (precision)^{-1}} = \frac{\mathbf{TP}}{\mathbf{TP} + \frac{1}{2}(\mathbf{FP} + \mathbf{FN})}$$

ROC AUC, on the other hand, compares the Sensitivity (i.e. the TP rate) vs (1-Specificity) (i.e. the FP rate). The higher the ROC AUC, the greater the distinction between True Positives and True Negatives.

## 6.2 Results analysis

Overall, our model has a relatively high accuracy and ROC AUC, but a low F1 score, the number of false positive and false negative cases are large. Compared with the statistics for the baseline model in **Notebook Section 4.5.3**, our optimal model achieved a small improvement.

## 6.3 Feature Selection

One benefit of using decision tree ensemble models like XGBoost is that it automatically provides estimates of feature importance from a trained predictive model. In general, an importance score indicates how valuable each feature was in the construction of the boosted decision trees within the model. The more an attribute is used to make key decisions with decision trees, the higher its relative importance. This importance is calculated explicitly for each attribute in the dataset, allowing attributes to be ranked and compared to each other [1].

We can use the calculated feature importances to perform feature selection with **SelectFromModel**. This class can take a pre-trained model, then use a threshold to decide which features to select. In the case of XGBoost, the feature importances calculated from the training dataset are the thresholds. We wrapped the model in a SelectFromModel instance, then used this to select features on the training dataset, trained a model from the selected subset of features, and finally evaluated the model on the test set, subject to the same feature selection scheme.

For interest, we tested each subset of features by importance, starting with all features and ending with a subset with the most important features. We measured the performance of these feature subsets with three statistics: accuracy, f1, and ROC AUC. By carefully investigating the results (in **Notebook Section 7.2**), we concluded that the number of features should either be 2, 3 or 5. n = 3 or 5 is slightly better than n = 2, with higher F1 and ROC AUC but a lower accuracy. We would like to take a less complex model and accept a modest decrease in estimated accuracy from 80.8% (n = 2) down to 80.4% (n = 3) or 80.3% (n = 5). These most valuable 3 or 5 features in our optimal XGBoost model are: 'PAY_1', 'PAY_2', 'PAY_3' and/or 'PAY_5', 'PAY_4'. Therefore, we reached the conclusion that, among the 29 features, the history of past payments (PAY_n) is of the most importance.

## 7 Conclusion

In this report, we explored, processed and transformed the UCI 'Default of Credit Card Clients' dataset, including not only the disposal of undocumented labels and an imbalanced training set with borderline-SMOTE, but also drawing in additional features based on the insights obtained from Exploratory Data Analysis. We then attempted five different methodologies for this binary classification task: AdaBoost, XGBoost, Gradient Boosting, Random Forest, and Artificial Neural Networks. For each methodology, we trained a baseline model with both oversampled and non-oversampled training sets (10 models in total) and adopted the evaluation metrics: accuracy score, f1 score, false positive (FP) rate and false negative (FN) rate. After comparing the prediction performances for these 10 models, we chose non-oversampled XGBoost with the objective function set as Logistic Regression, as our final methodology because it had the highest f1 score and the lowest FN rate.

Then, we tuned the XGBoost tree-specific parameters, regularisation parameters and the learning rate via grid-search cross validation to find the optimal value for each hyperparameter. Cross-entropy loss ('logloss') was used during cross validation and each optimal value was guaranteed to be a local minimal point observed by the 'V-shape' line graph visualisation. After training and validation, we utilized five statistical criteria (accuracy, f1, ROC AUC, FP and FN) to evaluate our prediction performance, and further conducted a results analysis and feature selection. Comparing the performance of the baseline model to the performance of our final model, we achieved a small improvement: the f1 score increased by 0.007, reaching 0.532 with a moderate volume of False Negative (551) cases.

To achieve better performance, based on the outcome of this experiment, future development should aim to increase model sensitivity and recall. One possible way is to collect more client data for the 'Default' class. A more balanced dataset could significantly boost the model's performance. Another way is to develop a deeper understanding of credit card customers' behaviour and create new features that include more behavioural information rather than just their historical records of payments and bills. Additionally, innovative methods such as anomaly detection could be explored to further improve performance.

## References


[1] Leo Breiman and Adele Cutler. *Random Forests*. 2012. URL: `https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm`.

[2] Iain Brown and Christophe Mues. "An experimental comparison of classification algorithms for imbalanced credit scoring data sets". In: *Expert Systems with Applications* 39.3 (2012), pp. 3446–3453.

[3] Chumphol Bunkhumpornpat, Krung Sinapiromsaran, and Chidchanok Lursinsap. "Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem". In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2009, pp. 475–482.

[4] Nitesh V Chawla et al. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[5] Jake Frankenfield and Eric Estevez. *Artificial Neural Network (ANN)*. Aug. 2020. URL: `https://www.investopedia.com/terms/a/artificial-neural-networks-ann.asp`.

[6] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning". In: *International conference on intelligent computing*. Springer. 2005, pp. 878–887.

[7] Ashish Kumar. *AdaBoost Algorithm: Boosting Algorithm in Machine Learning*. Mar. 2020. URL: `https://www.mygreatlearning.com/blog/adaboost-algorithm/`.

[8] Shantanu Neema and Benjamin Soibam. "The comparison of machine learning methods to achieve most cost-effective prediction for credit card default". In: *Journal of Management Science and Business Intelligence* 2.2 (2017), pp. 36–41.

[9] Yongming Song and Yi Peng. "A MCDM-based evaluation approach for imbalanced classification methods in financial risk prediction". In: *IEEE Access* 7 (2019), pp. 84897–84906.

[10] I-Cheng Yeh and Che-hui Lien. "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients". In: *Expert Systems with Applications* 36.2 (2009), pp. 2473–2480.