Santander Customer Transaction Prediction

MSBA 6420, Spring 2021

Jane Jian, Xiaoshuo Liu, Xiaowen Tang, Yuanzhe Wang, Weizhong Yao

Project Definition & Introduction **2**

Solution Overview **2**

Technical Specification **2**

Methodology **3**

Feature Engineering **4**

Model Building **4**

Baseline Model 4

LightGBM 5

XGBoost 6

Neural Network 6

Business Insights **8**

Future Steps **8**

Challenges Faced **9**

Reference **9**

Appendix **9**

Project Definition & Introduction

Santander Bank is a wholly-owned subsidiary of the Spanish Santander Group. As a retail bank, Santander Bank is continually working with the global data science community to explore new ways to solve one of the most challenging tasks: binary classification problems such as if a customer is satisfied, if a customer can pay the loan, or if a customer will make a transaction.

The goal of this project is to discover the best approach to predict which customers will make a transaction in the future. This goal brings up two key challenges: first, transforming the raw data into features that better represent the underlying problem to the predictive models; second, designing a predictive model that has the best performance.

The data provided by Santander for this project are de-identifying real data in CSV format, which have two hundred predictor variables and one target variable. The predictor variables are the historical customer data that are de-identified by Santander and the target variable is whether the customer made a transaction or not. One represents yes and zero represents no.

Solution Overview

Our final solution contains two major strategies, feature engineering and predictive model building.

Our feature engineering process focuses on three major steps, detecting fake (pseudo) testing data, adding "magic features", and data augmenting. The detection step enabled us to remove noise records that may disturb our model training and testing process which can lead to poor model performance. The magic-feature step helped us to improve our predictive model by providing additional features that relate to the target outcome. The data augmentation step allowed us to remedy the unbalanced data in order to improve predictive accuracy.

Our predictive-model-building process focuses on exploring and creating models that can achieve the best performance based on the ROC-AUC metric. We used the Naive Bayes model as a baseline and tried XGBoost, LightGB, neural network, and stacking models. We found that LightGB had the best performance. Knowing that the revenue or cost for each category of predicted outcome may be different, we also conducted a profitability analysis and plotted margin curves to show the impact of profitability regarding prediction thresholds.

Technical Specification

As all the predictive modeling techniques in the course are taught with Python, the team solely used Python for this project. The list of Python packages is below:

Python Packages:
• pandas • numpy • scikit-learn • xgboost

- lightgbm
- keras
- keras-tuner
- tensorflow
- bayesian-optimization
- scikit-optimize

We heavily utilized Google Collaboratory for this project to take advantage of Google's GPU processing power. The platform also enables us to edit, share, and view each team member's scripts in a centralized Google Drive disk. At the time of writing, the Python version of Colab is 3.7.10.

Methodology

In this customer transaction prediction project, the ultimate goal is to predict whether a customer will make a transaction or not with Santander's service, which is a binary classification problem.

A quick glance over the data showed us that these variables are not in their actual names. They are simply labeled as "Var_" with the column number. Because of this finding, we quickly determined that domain knowledge would have very limited help in this project. With the suggestions from the professor, we agreed to focus on feature engineering techniques to improve the prediction accuracy. Knowing that some Kaggle contests would add fake data in the sets to disturb predictive model training, we first explored the possibility of that issue. We then utilized the concept of magic features. These features are only added when a feature has a unique value. These unique features provide additional information to the model training algorithm and allow the model to gain more insights regarding the feature significance. Last, we used data augmentation, which is a technique that increases the amount of data by adding slightly modified copies of the existing data or newly created synthetic data from the existing data. It worked as a regularizer and helped us reduce possible overfitting issues during the training process.

As for model building, we mainly focused on the advanced models that are known to do well in binary classification problems. These models use the concepts of boosting, stacking, and neural networks. To provide a base-line benchmark, we first ran a Naive Bayes Classification model on the processed data. Then, we tested LightGB, XGB, neural networks, and stacking of LGB and XGB by using the evaluation metric of ROC-AUC specified by the competition rules. Because this project relates to potential business revenue and cost by Banco Santander, we also conducted cost analysis using our models by analyzing the relationship between prediction threshold and total cost. Based on a survey conducted by Parvathy P (2017), it costs a bank around 80 USD to acquire a credit card customer who returns about 120 USD in profit. Therefore, we used 80 as false positive cost (cost of mis-classifying a customer that will not make a transaction as a potential customer) and 120 as false negative cost (cost of mis-classifying a potential customer).

Feature Engineering

Before we started doing feature engineering, two important things that we learned from Kaggle discussion are first there are fake testing data and second some features are more informative than the others and we called those more informative ones - “magic features”.

First, people have found that the testing data consists of real testing and fake testing, which are synthetic testing data generated from the real testing data. Based on that, we assume that if at least one feature of a sample has unique value among all the data, then this sample is a real one. Then we found there are 100,000 real testing data and 100,000 fake testing data.

Second, similar logic is applied for finding magic features, if a feature has a unique value, which means this value only appears once among this feature of all the data points, then this feature is a magic feature. Another thing hidden behind the magic features is in order to find the true magic features, we have to combine the training data with real testing data. Then we group by each feature and compute the variance of each group, containing the same value. Therefore, if a group contains only one value, then the variance is NaN, which means the new features added are NaN.

One limitation of this feature engineering method is that only XGBoost and LightGBM treat NaN as a unique value, but other models don’t. Therefore, based on the essential characteristic of magic features, we simply counted the number of values for each feature and added the counts as new features.

Another thing that we noticed is that the data is unbalanced, data points labeled 1 are 10.049% of the whole dataset. One way to solve that is to increase or decrease the number of samples by duplicating the smaller class or removing data points from the majority class to make the data more balanced. So we adopted Imblearn to achieve that but that eventually caused overfitting. We think the overfitting issue is caused by the fact that this method actually changed the original distribution of data to a balanced distribution (50% to 50%). Then, we implemented another method to resampling the data but without changing the original distribution a lot. Since simply duplicating data points doesn’t add any new information, we went down another way – First, we filter the data by target class and create a new sample of each target class. Then, the code goes through every feature column and randomly assigns it a value from the same column of a different sample that matches the target class. This adds some new information (new data points) to the dataset, since the feature column values of the created sample are a new combination (of known data).

Model Building

Baseline Model

Before we dive deeper into more complex morelds, we first tried Gaussian Naive Bayes as a benchmark.

Hyper-parameter Tuning

We used default settings of hyper-parameters: prior probability calculation is adjusted according to the data, portion of largest variance of all features is 1e-9.

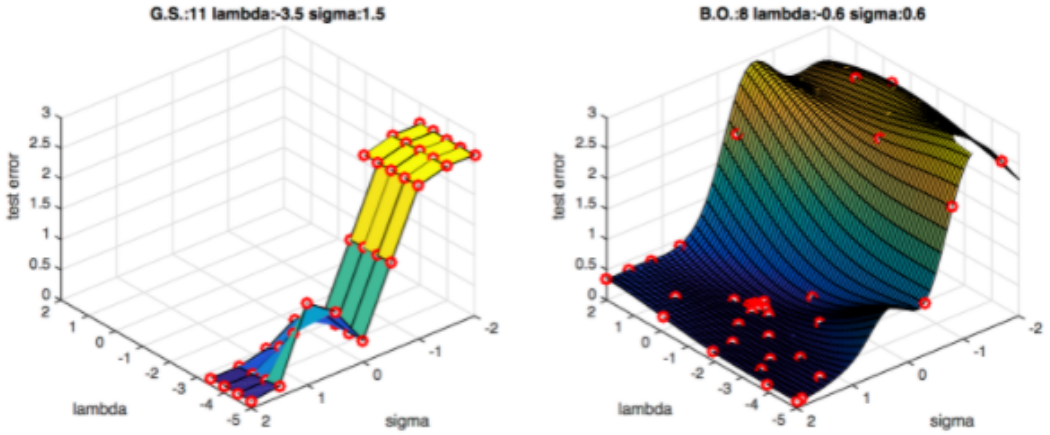
Model Performance

The best performance based on AUC was 0.888.

LightGBM

Hyper-parameter Tuning

For the hyper-parameter tuning process, we decided to use Bayesian global optimization with gaussian processes. This optimization attempts to find the maximum value of an evaluation function (AUC) in as few iterations as possible so that parameters will maximize AUC in both training and validation. Another reason that we chose to use it is that Bayesian Optimization can speed up the tuning process and has its own advantage. Because unlike grid search which try out all the possible combinations, Bayesian Optimization firstly will try out some random values at the beginning then focus on values that could be more promising (have lower validation error) and the advantage is that with the prediction on each point, it also gives how uncertain it is, which tells if it's worth to explore or not. Therefore, it can find the best hyper-parameters much faster and much better than grid search.

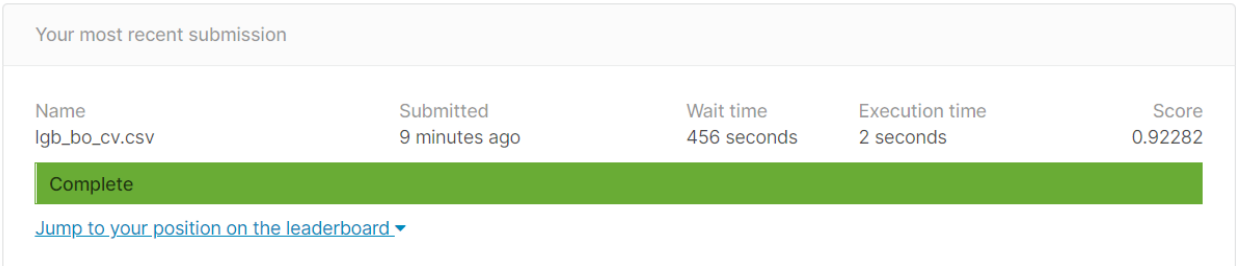


First, construct a function with training and validation data defined inside of it. Then this function takes values: num_leaves, min_data_in_leaf, learning_rate, min_sum_hessian_in_leaf, feature_fraction, lambda_l1, lambda_l2, min_gain_to_split, max_depth. Next, we gave bounds for these parameters and Bayesian optimization will only search inside the bounds. One thing we want to mention is that num_leaves, min_data_in_leaf, and max_depth should be integers so we forced them to be integers inside the function.

5-folder LightGBM Model

We decided to use 5-folder cross-validation to evaluate the performance of the LightGBM model. But a single run of the k-fold cross-validation procedure may result in a noisy estimate of model performance. Therefore, we repeated 5-fold cross-validation to improve the estimated performance of the model. This involves simply repeating the cross-validation procedure 5 times and reporting the mean result across all folds from all runs. This mean result is expected to be a more accurate estimate of the true unknown underlying mean performance of the model on the dataset, as calculated using the standard error.

With this model, the AUC is 0.92282.



XGBoost

Hyper-parameter Tuning

We used grid search for tuning the algorithm's hyperparameters. The parameters we focused on were max_depth, n_estimators, and learning_rate. Other hyperparameters and wider ranges can be possibly included in the search, but the hardware limitation only allowed us to search within a narrower scope. A reasonable range was then assigned to the three hyperparameters and 160 different models were tested. The parameter ranges are below:

```
[ ]  parameters = {
         'max_depth': range (2, 10, 1),
         'n_estimators': range(60, 220, 40),
         'learning_rate': [x*0.01 for x in range(5, 30, 5)]
     }
```

With a 10-fold cross validation, a total of 1,600 train-validation processes were performed. The configuration of the best estimator is below:

```
param = {}
param['booster'] = 'gbtree'
param['max_depth'] = 5
param['n_estimators'] = 180
param['learning_rate'] = 0.25
param['objective'] = 'binary:logitraw'
param['eval_metric'] = 'auc'
param['silent'] = 1
param['tree_method'] = 'gpu_hist'
gpu_res = {}
```

Model Performance

With the hyperparameters, we trained a XGBoost model with 10,000 boost rounds and an early-stop setting of 3000 rounds. The best performance based on AUC was 0.912138. Based on the FP and FN costs we mentioned, the lowest cost was 8.038 with a threshold of 0.0674.

Neural Network

Hyper-parameter Tuning

We used Keras Tuner to tune the Neural Network's hyperparameters. The parameters that we aimed to tune were the number of hidden layers, the number of neurons, and the learning rate for Adam optimizer. A reasonable range was assigned to the three hyperparameters, and RandomSearch was used to test 500 different models. The result shows that the best performing model is a 4-hidden layer Neural Network model with the number of neurons equal to 416, 256,128, and 96 respectively, also with a learning rate of 0.01.

4-hidden layer Neural Network Model

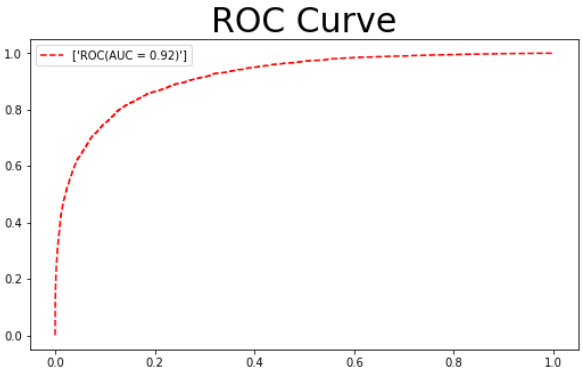
With the hyperparameters, we trained a 4-hidden layer Neural Network model with the batch size of 128 and epochs of 1000. The best performance based on AUC was 0.88149.

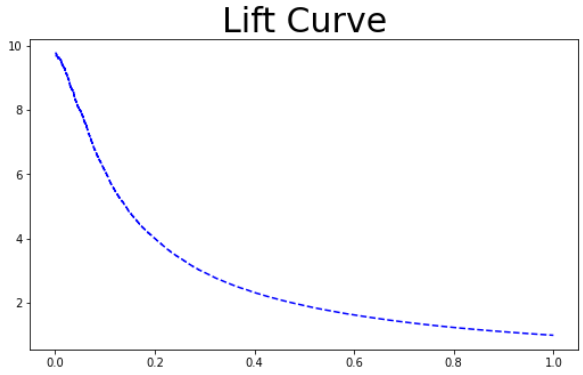
Model Evaluation and Cost Analysis

We used AUC(Area Under the Curve) to select the best performed model.

Model	**AUC**
LightGBM	0.92282
XGBoost	0.91745
Neural Network	0.88149
Naive Bayes	0.888

The optimal model is Light GBM which achieves an AUC of 0.92282.





Ultimate goal of every predictive model is to generate business value. Misclassifying a potential customer or non-potential customer will both incur cost. After choosing our optimal model, we did the Cost-Benefit Analysis to choose the best threshold that can minimize the misclassification cost.

The analysis was based following assumptions:

1. Cost of mis-classifying a customer that will never make a transaction as a potential customer is 80
2. Cost of mis-classifying a potential customer is 120

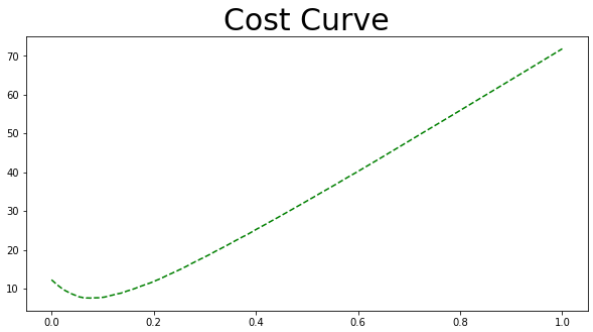
*According to a survey conducted by Parvathy P (2017) [1], it costs a bank around $80 to acquire a credit card customer who returns about $120 in profit.

We calculate misclassification cost for every threshold and choose the one that has the lowest value.

Misclassification cost was calculated as :

$$\frac{False\ Positive\ \times False\ \ Positive\ Cost + False\ Negative \times False\ Negative\ Cost}{Total\ Number}$$

Based on our analysis, 7.6% is the optimal threshold. Since false negative cost is higher than false positive cost, it is reasonable to apply a more conservative threshold. The corresponding average misclassification cost is 7.56, compared with 32 using the naive threshold of 50%. The corresponding precision is 71.10%, recall is 52.72% and accuracy is 92.96%.



Confusion Matrix

		Actual Value	
		Positive	Negative
Predict Value	Positive	TP: 2,160	FP: 878
	Negative	FN: 1,937	TN:35,025

Since the proportion of potential customers is small, randomly selecting target customers will incur high cost. The average profit per customer applying our model to acquire customers is 62.19 compared with -59.51 if randomly choose target customers.

$$Revenue\ = \ Potential\ Customer\ Number\ \times\ Revenue\ per\ person$$

$$Cost\ = Total\ Targeted\ Customer\ Number\ \times\ Cost\ per\ person$$

$$Average\ Profit\ Per\ Customer\ = \frac{Revenue - Cost}{Total\ Target\ Customer}$$

Business Insights

Financial institutions want to predict whether their customers would make a specific transaction in the future, and we are helping them to identify those customers with a large amount of data representing their behaviours. The tradeoff is between the cost of promoting our product to customers and the profit of

customers making transactions. We conducted cost analysis using our best performing model LightGBM and we found that we could achieve a 62.19 average profit compared with -59.51 if we randomly choose target customers.
The threshold of minimizing the cost is 7.6%, meaning that we are targeting the customers whose probabilities of making transactions are more than 7.6% based on our predicted results. The corresponding average misclassification cost is 7.56.

Future Steps

Further improvement of the proposed model including model fine-tuning, feature selection and the association with economic factors.

1. Implementing stacked generalization of models is able to reduce both bias and variance and further improve model performance.
2. Better acknowledgement of the features of customer behaviours may help conduct empirical feature selection and yield more informative results.
3. Different macroeconomic environments could influence the customer behaviours as well as the prediction results.

Challenges Faced

The two biggest challenges were feature engineering, and model selection.

Feature engineering including finding the fake data in testing dataset and indicating magic features which means feature has a unique value only appears once among this feature of all the data points. We also apply data augmentation to solve the imbalanced data issue.

Another challenge was model training and selection. We covered some advanced models and applied grid search to get the best parameters of these models. Then we trained the models with thousands of iterations to get the best performance of the models. However, we have a large dataset and the computations required massive computing power so we spent a large amount of time training and tuning models. We use Colab to do all the coding and modeling but it crashes randomly in the middle of computing.

Reference

[1] What is the average customer acquisition cost that a bank pays to acquire a credit card customer?, Parvathy P. and 2 others

https://askwonder.com/research/average-customer-acquisition-cost-bank-pays-acquire-credit-card-customer-i-m-fy512dtei

Appendix

File System

1.File Location

Every file, including the dataset and the code/models that are referenced in the analysis are located in the shared Google Drive:

https://drive.google.com/drive/folders/16q9m0O8TorU0OjonLyQBtBJXOqKLUMi4?usp=sharing

The codes are in the Scripts folder and the datasets are in the Data folder.

2.File Descriptions

File	Description
train.csv	Raw training dataset provided by Santander
test.csv	Raw testing dataset provided by Santander
train_simulated.csv	The data splitted from the training data as validation training data
test_simulated.csv	The data splitted from the training data as validation testing data
Naive Bayes.ipynb	Code for Naive Bayes model, which is our baseline model
PDA-Neural Network.ipynb	Code for the Neural Network model
XGB.ipynb	Code for XGBoost model
XGB_Misclassification.ipynb	Code for misclassification cost analysis of XGBoost model
SimulatedTrainTest.ipynb	Code for split the training data into validation training and testing data
LGB_BO_CV.ipynb	Code for the LightGBM model with future engineering
StackingClassifier.ipynb	Code for stacking model with LightGBM model and XGBoost model
FE Baseline with LGB.ipynb	Code for the LightGBM model without future engineering
Cost_Analysis_LGB.ipynb	Code for misclassification cost analysis of LightGBM model