# Elo Merchant Category Recommendation

**Fang Feng, Qirui He and Xichen Tan**

*Duke University*

**Elo[1],has built partnerships with merchants in order to offer promotions or discounts to cardholders. To identify and serve the most relevant opportunities to individuals, customer loyalty need to firstly uncover. Firstly, out liners and missing data have been prepossessed in three different ways to ensure the reliability and reduce bias of later results. After that, several features are created based on the given features in provided data sets and feature aggregation is settled for feature selection. LightGBM is adopted as regression model predicting the loyalty of our customers. RMSE is the performance evaluation criterion.**

## 1 Problem description

Our project comes from a featured prediction competition on Kaggle [2]. Imagining that when someone is travelling or on business in an unfamiliar city, feeling hungry, he (she) opens an APP (application) and want to get some restaurant recommendations that exactly base on his (her) personal taste. The good news is that Elo has built partnerships with merchants in order to offer promotions or discounts to cardholders. However, the recommendation system of Elo is not specifically tailored for an individual or profile, and Elo wants to know whether their promotions work for consumers. To identify and serve the most relevant opportunities to individuals, we need to firstly uncover signal in customer loyalty. Basically, the predicted loyalty is the evaluation criteria of this competition and it can reflect how satisfied consumers are with the recommendation system. So, our project is to use the data sets provided by Elo and apply the features in them to predict customers' loyalty. Basically, we assume that all data can reflect a real-life scene. Since features like purchase data, purchase amount are all relevant to customers' daily transaction and it could be used as reflections of customers' loyalty. We can evaluate our model performance by comparing our predicted loyalty with real loyalty provided in the datasets, setting some special criteria to measure the degree of closeness between model's outcomes and true labels. After building and refining our model, we can try to apply it in the real industry and to make a contribution to the restaurant recommendation systems.

[1]One of the largest payment brands in Brazil.

[2]An online community of data scientists and machine learners, allowing users to explore and build model to solve data science challenges.

## 2 Data description

Basically, we have 6 datasets provided by Kaggle.

- *historical_transactions.csv*: contains up to 3 months' worth of historical transactions for each card_id. And historical transactions are made before recommendations.
- *new_merchant_transactions*: contains the transactions at new merchants (*merchant_ids* that this particular *card_id* has not yet visited) over a period of two months. These transactions are made after recommendations. (its features are the same as *historical_transactions.csv* dataset).
- *merchants.csv*: contains aggregate information for each merchant_id represented in the data set.
- *train.csv* and *test.csv*: contain *card_id* and information about the card itself. *train.csv* has 201917 entries and 6 columns (5 features + 1 target). *test.csv* has 123623 entries and 5 columns (except target column, which is the loyalty we need to predict). Also, since we don't really join this competition, there is no correct prediction data for us to test, so we can only judge the performance of our model on training dataset.
- *sample_submission.csv*: a sample submission file in the correct format, so it doesn't matter here. We just ignore it.

## 3 Data pre-processing

### 3.1 Outliers removal

Figure 1 shows us many interesting statistical information within the data. First of all, the *target* feature in train data, which on the other hand is the loyalty score) contains data are far away from mainly distributed area (below -30). We believe those rows should be eliminated from the data. Secondly, the *purchase_amount* over historical transactions figure shows that there are abnormal transactions which are over 6000000 per transaction, and should be removed from our data. Third, the *Number_of_cards* over *first_active_month* figure shows that more cards are active in the market as the time closer to now, and the target (loyalty score) also increases at the same time. We collected a lot of information while we are visualizing the data in the summary statistics, which are really helpful for us to do feature engineering.

### 3.2 Handling missing data

The *historical_transactions* data and *new_transactions* data contains a large amount of missing data (or **NaN** data) in **category_2**, **category_3** and **merchant_id** these three features as shown in **Table 1**.

The way we were doing in the progress report is dropping the missing data. This time we optimized the model by fitting the missing data with (1) random selected non-missing value, (2) fixed high frequency value in the same feature (column), which eliminated the bias. The comparative performance are shown in **Table 3**.

### 3.3 Feature extraction

Basically, our main idea is to make as many features as possible and then fit them all into our training model. After doing this, we can perform feature reduction using feature importance from the model or other techniques such as PCA.
We first convert *purchase_date* in *historical_transactions.csv* and *new_merchant_transactions.csv* from string to numerical datetime. And then extracting day, month, year and other time-related features out of it. Also, we did some simple normalization, like mapping value 'Y' to 1, and 'N' to 0.
Secondly, we encode *purchase_date* as **categorical** data which gives the model more flexibility. In order to extract information from date feature and give decent result, we group date values together into some number of sets, and use the set as a categorical attribute. For example, we group day_of_week data into weekday, weekend; *elapsed_time* is the difference between today and *first_active_month*.

### 3.4 Feature aggregation

We have done a lot success on feature aggregations. To begin with, for the progress report, we have discussed that the *card_id* for each cardholder is unique and we can use it to index users' profile. So we adopted major feature aggregations on *historical_transactions* and *new_transactions* datasets. After that, we keep delving into the feature aggregations on merchants dataset, by performing operations on unique merchant_id. So the final procedures are blow:
1. Perform feature aggregations on merchant datasets: group by *merchant_id* and perform **min**, **max**, **sum**, **mean**, or **std** on features.
2. merge aggreagated merchant data into *historical_transactions* and *new_transactions* data.
3. Perform aggregation on *historical_transactions* and *new_transactions* data: group by *card_id* and perform **min**, **max**, **sum**, **mean**, or **std** on features.
4. merge *historical_transactions* and *new_transactions* data into *train* and *test* datasets.

Some of typical features after aggregation are showing below:
a. hist_month_diff_mean. The **mean** of current date minus *purchase_date* group by *card_id* in *hist_transaction* dataset.
b. *merch_avg_purchases_lag3_mean*. The **mean** of monthly average of transactions in last 3 months group by *merchant_id*, then group by *card_id* in historical_transactions and new_transactions.
c. *new_purchase_amount_sum*. The **sum** of *purchase_amount* group by *card_id* in *new_transactions* dataset.

## 4 Method

### 4.1 LightGBM

Predicting loyalty score is essentially a supervised regression problem. Instead of using single model, such as Lasso and Logistic Regression, a weak learner, boosting algorithm is introduced in this project for converting weak learner into strong one, so that improving performance. The mothed we use so far is **LightGBM** (Light Gradient Boosting Machine).
LightGBM is a powerful gradient boosting framework, based on the decision tree. It has faster training efficiency, higher classification accuracy, and it is easier to handle big datasets. Before introducing LightGBM, we need to introduce XGBOOST first, since LightGBM is developed from XGBOOST. XGBOOST stands for "Extreme Gradient Boosting", which is used for supervised learning problems, using the training data (with multiple features , just like the cleaned data features we collect from our transaction, merchants, and customers' datasets) to predict a target variable (like the customers loyalty). Before introduce the trees

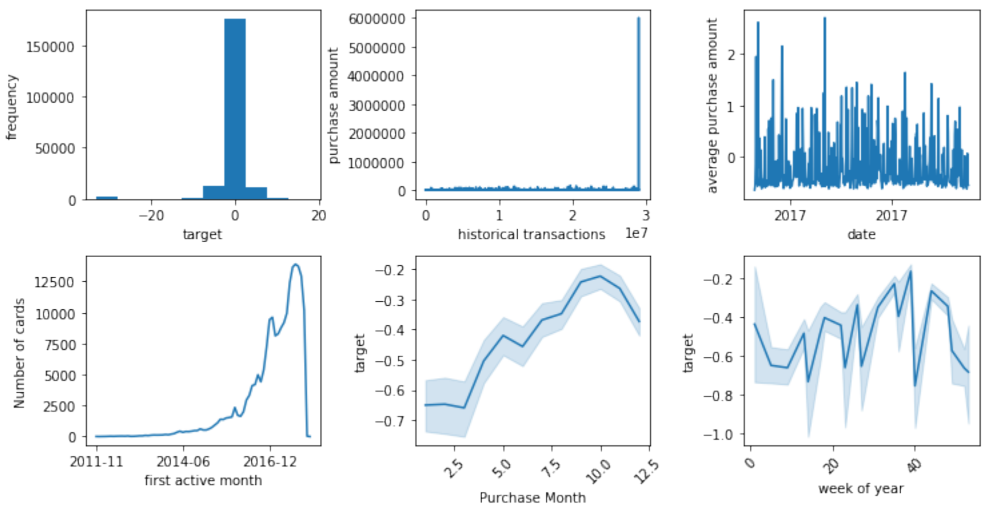


**Figure 1:** *Visualization of Dataset features*

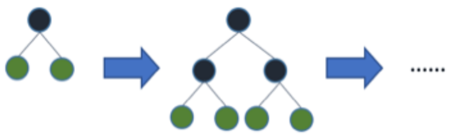
| feature | historical_transactions | new_transactions | merchants |
|---|---|---|---|
| category_2 | 2652864 | 111745 | 11887 |
| category_3 | 178159 | 55922 | 0 |
| merchant_id | 138481 | 26216 | 0 |

**Table 1:** *the number of missing data in historical and new transactions*

of XGBOOST in details, let's first talk about supervised learning.
Basically, the objective function of a supervised model can be expressed as following:

$$Obj(\theta) = \sum_{i}^{n} l(y_i, \hat{y_i}) + \sum_{k=1}^{K} \Omega(f_k) \qquad (1)$$

$\theta$ is the vector of parameters of the model which is $\theta = (\theta_1, \theta_2, ..., \theta_n)^T$. The task of the model is to find the optimal to fit the training data and the target , which is measured by the objective function. The first term of RHS (right hand side) is loss function (like mean square error, logistics error, etc.) which measures prediction accuracy. The second one of RHS is regularization term (like L1 and L2 regularization, dropout in neural network) which controls the complexity of the model, helping us to avoid overfitting. What we need to do is to balance the tradeoff between model's accuracy and model's complexity.
After knowing about supervised learning, here comes the decision and regression trees (CART). The difference between CART and the normal decision tree is that the leaves of decision tree are classification outputs, however, the leaves of CART is assigned with regression scores, which contains more information than simple classification. Just like random forest, we consider assembling many trees together to obtain better predictions.
The solution about how to assemble trees is what we mean tree boosting. We simply define every single structure of a tree as a function , thus, every time we add a new tree, we choose the tree that can minimize our objective function mentioned above. And this kind of training method is called additive training. To optimize our loss function, and to make it more general, we take the second-order Taylor expansion of the loss function, which only depends on first-order derivative and second-order derivative of the loss function. Using these derivatives, we can accurately measure how good a tree structure is. Supervised learning, CART, additive training and Taylor expansion are all main and principal ideas used by XGBOOST. Moreover, XGBOOST also takes more techniques in machine learning and systems optimization into consideration. After the brief introduction to XGBOOST, let's take a quick look at LightGBM. When searching for an optimal split, the model needs to iterate through the data many times, which is so time consuming. Thus, many bosting algorithms use pre-sorted method to train the trees structure. This is a simple solution but the drawbacks are space-consuming (store original data and sorted data), cache miss, etc. In contrast, LightGBM uses histogram-based algorithm, converting continuous features into discrete integer, to accelerate training procedure and reduce memory usage.



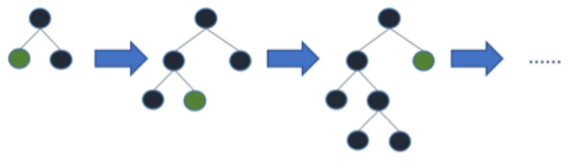


Fig. 4.1 level-wise tree growth strategy (first), Leaf-wise tree growth strategy (second)

Besides, LightGBM uses leaf-wise method instead of level-wise method to develop tress, which is more efficient. The reason is that finding a leaf with maximum split gain can save time and reduce error. There are many other advantages such as sparse feature optimization, multi-thread optimization, etc. We omit these deeper topics here since they have little to do with our model.
Back to the project, to predict the loyalty score of users, we need to fit a regression model with our datasets. From data science class, one advanced model to do this is random forest. It takes a random subset of features and a random subset of training data to build trees, however, it's easy to choose some noise data as split criteria, which cannot represent real data. And it's not as good at solving regression problems as they are at classifying, because random forest doesn't give a continuous output. Then, we consider XGBOOST model. It uses gradient Boost method to build CART trees in the negative gradient direction of loss function, adding regularization to control model complexity, and can handle missing data, by working out the missing data's split direction automatically. But finally, we take the most state-of-art model, which is LightGBM. It has advantages over XGBOOST either on training time or space usage and the main reasons have been mentioned above. In order to get the best prediction performance of our training datasets, we decide to choose LightGBM as our core method.

### 4.2 Model Selection

We split the training dataset into train and valid data and perform cross validation. And in the model, We follow the way in one of kaggle kernel [4] to tune the hyper-parameters. the writer used BayesianOptimization to get the best parameters. Below are some of the important parameters we run experiments in our model:
1. **num_leaves**. This is the main parameter to control the complexity of the tree model. Theoretically, we can set $num\_leaves = 2^{max\_depth}$ to obtain the same number of leaves as depth-wise tree. However, this simple conversion is not good in practice. The reason is that a leaf-wise tree is typically much deeper than a depth-wise tree for a fixed 2. number of leaves. Unconstrained depth can induce over-fitting. Thus, when trying to tune the *num_leaves*, we should let it be smaller than $2^{max\_depth}$. For example, when the $max\_depth = 7$ the depth-wise tree can get good accuracy, but setting num_leaves to 127 may cause over-fitting, and setting it to 70 or 80 may get better accuracy than depth-wise.
2. **min_data_in_leaf**. This is a very important parameter to prevent over-fitting in a leaf-wise tree. Its optimal value depends on the number of training samples and num_leaves. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset.
3. **max_depth**. You also can use max_depth to limit the tree depth explicitly.
Some of parameter range we used in our model:
a. max_depth: (4, 10)
b. num_leaves: (5, 130)
c. min_data_in_leaf: (10, 150)
d. feature_fraction: (0.7, 1.0)
e. bagging_fraction: (0.7, 1.0)
And the final optimal parameters we acquired are shown in **table 2**.

| | num_leaves | min_data_in_leaf | max_depth |
|---|---|---|---|
| value | 31 | 30 | -1 |

**Table 2:** *optimal hyper-paramters for LightGBM*

After parameter tuning, we can evaluate the performance of model by doing cross validation on training datasets. The performance will be determined by RMSE (Root of Mean Square Error).

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_l})^2} \qquad (2)$$

$\hat{y_i}$ is the predicted loyalty for each customer and is the actual loyalty

## 5 Results

With employment of LightGBM model, we firstly try filling missing data by permutation, using features only from historical data, and then attempt to simply drop missing data. Finally features extracted from new merchant data set is added into our model. With 5-fold Cross Validation, performance of each of the model is evaluated by RMSE (Root Mean Square Error) on test set as shown in **Fig.1**.
As we can see in the comparison table, different ways of treating missing data have non-trivial influence on prediction performance. Between the two method, filling missing data with randomization is 1.5% better than dropping the data directly, and filling with most frequent values is 0.2% better than filling with random values. What is interesting is features of new merchants does not enhance performance at all, making things worse on contrast. Dropping random data delete missing data and other useful data at the same time. This actually create bias, thus resulting in bad

performance. If taking a look at new merchant data, these data are actually features for merchants rather than customer, which means there is not necessarily relationship between customer behavior with these data.

| Drop | Random | Fixed | Add merchant features |
|---|---|---|---|
| 3.713 | 3.655 | 3.648 | 3.657 |

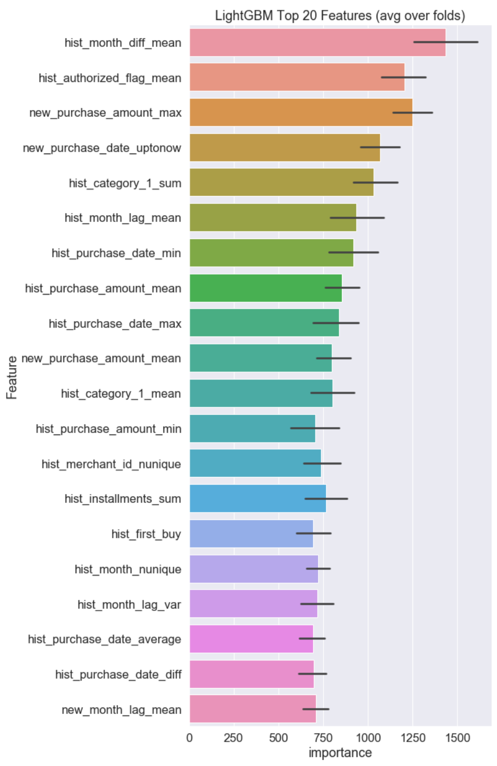
**Table 3:** *the score (RMSE) of All models*

This method is essentially a strong learner with higher prediction precision than weak one. Additionally, it supports taking category parameter directly, which save tons of work for us from transform them to one-hot encoding. For millions of pieces of data in this problem, this method enhances efficiency of training. However, we have to be very careful about protect it from overfitting. Generally speaking, this method should be helpful in reality to help the company solve the problem, for it has enough prediction precision, and model simplicity. And we have not yet figure out a more advanced approach to solve this problem.

## 6 Conclusion

At the beginning, we had a really hard time understanding the problem requirements, datasets and features, and have no idea how should we do it. But as we do more and more work like processing the data, visualizing the features, and understanding other people's helpful research, we gradually found traces to get our model working. And it is so excited to finally see our model performance on the competition's score board. We know we haven't done every procedure of an excellent data science project should have, but we have already learned many useful techniques, tricks, and knowledge during the whole process, which are definitely significantly important for our future data science work. We find that one of the important feature engineering work is extracting and aggregating features with fundamental statistical methods like **sum**, **mean**, and **max** and etc. And we also get hands on experience of using one of the well-known data science algorithm **LightGBM** and its implementation, which inspired us to delve into more data science interesting algorithms, libraries, and projects.
In the future, We are going to optimize our model by exploring more useful features, trying other algorithms like Neural Network, and using some advanced methods like ensemble and stacking. But we know that the most important procedure is still the feature engineering, in order to achieve a good score we've already collected and created hundreds of features. But as far as I am concerned, that is far from enough. Some competitors with really good score get thousands of features with different combination of datasets. On account of that we need more domain knowledge to inspire us to gain more useful features.

## 7 Appendix



## References

[1] Tianqi Chen, *Introduction to Boosted Trees* Washington University, 2014

[2] Guolin Ke, Taifeng Wang, *Microsoft Research*, Neural Information Processing Systems 2017

[3] Will Koehrsen, *Introduction to Manual Feature Engineering*, https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering

[4] FabienDaniel, Data Scientist at Continental https://www.kaggle.com/fabiendaniel/hyperparameter-tuning