# Fraud Detection From Customer Transactions

Diwen Lu (dl3209), Yuhan Liu (yl7576), Wenxin Zhang (wz2164), Bella Lyu (hl4229), Zhifan Nan (zn2041)
DS-GA 3001 Advanced Python, New York University

**ARTICLE HISTORY**
Compiled May 12, 2021

**ABSTRACT**

In this project, we focused on the fraud detection of customer transactions and applied optimization tools to improve the efficiency based on Chris Deotte's work, which won the 1st prize of the Kaggle competition with his XGBoost implementation [1]. We optimized the procedures of imputing missing values with itertools, selecting informative features with multiprocessing, doing feature encoding with RAPIDS cuDF, and training the data with various models and techniques. In the following sections of this paper, we will discuss each procedure in detail and elaborate on the powerful tools that we have focused on as well as the speed improvement that we have ever achieved.

## 1. Introduction

Frauds have always existed throughout human history, but in this age of digital technology, the strategy, extend and magnitude of financial frauds is becoming more wide-ranging than ever. In the Kaggle competition held by IEEE-CIS and Vesta Corporation [2], more than six thousands of teams from the worldwide have aimed at employing advanced machine learning techniques to efficiently detect frauds based on the data provided by Vesta. However, the challenges can be various. We not only have to address the large dataset (nearly 1.84G) that comprises hundreds of variables, but also deal with encrypted features for the concern of the customers' privacy, which indeed highly increases the difficulty of constructing exploratory data analysis and feature engineering.

The goal of our project is to predict the probability that an online transaction is fraudulent, as denoted by the binary target isFraud, and more importantly, optimize each procedure with various techniques to make it more efficient. The given data is broken into two files named after 'identity' and 'transaction', which consist customers' identities and transaction information respectively and joined by 'TransactionID'. In the dataset, most of the encrypted features in 'transaction' data are clustered and arranged in sequence in this setting. For example, C1-C14 features combine to count how many addresses are found to be associated with the payment card; D1-D15 are timedelta features; and the Vxxx ones are those informative features that are engineered by Vesta, including ranking, counting, and other entity relations.

[1] https://www.kaggle.com/cdeotte/xgb-fraud-with-magic-0-9600
[2] Vesta Corporation. IEEE-CIS Fraud Detection. Retrieved April, 2021 from https://www.kaggle.com/c/ieee-fraud-detection

## 2. Data Exploration & Feature Engineering

### *2.1. V selection*

One important feature selection step is to select useful V columns from the massive amount of V columns we have. There are 340 columns of V in total, and their true meaning are encrypted by the data publisher. However, it turned out that these V columns are highly correlated and a large portion of them are redundant. We can get rid of some using the following strategy:

(1) First we group these columns into different buckets by looking at the number and the structure of their missing values. If they have the same number of null values, then they are in the same group. For simplicity, we will call them **NaN groups**, Here is one group of Vs that share the same number of NaNs ( of NaNs = 76073):

- 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'V29', 'V30', 'V31', 'V32', 'V33', 'V34'

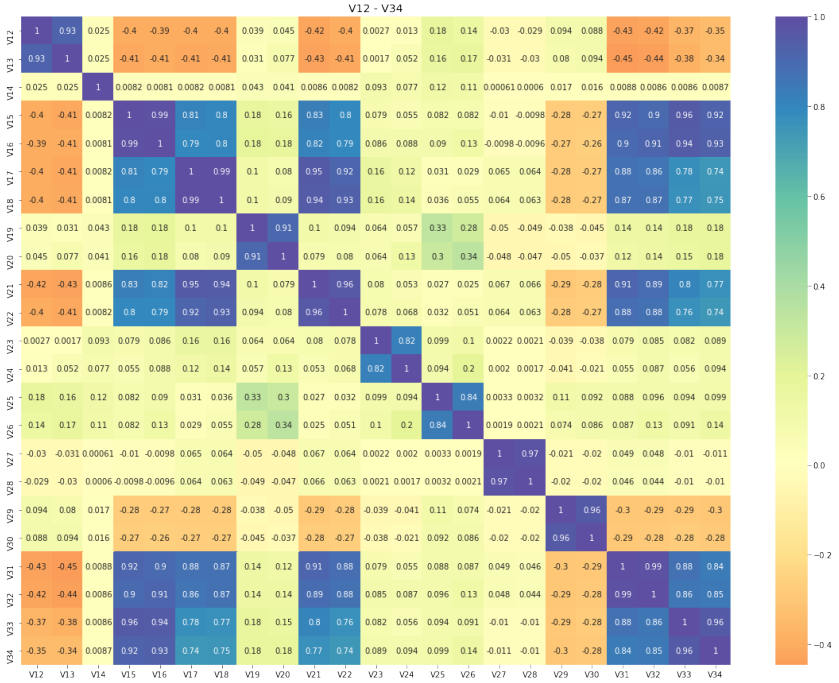


Figure 1.: correlation heatmap of the NaN group

(2) Within each NaN group, if two or more columns have correlation over 0.7 with each other, then they will be assigned to the same subgroup. Figure 1 shows the correlation matrix of the group of V columns we mentioned above. These purple blocks shows the subgroups that have high correlation ($> 0.7$) between them. In this case, we have 8 of them, and we list them as follows:

- 'V12', 'V13',
- 'V14',
- 'V15', 'V16', 'V17', 'V18', 'V21', 'V22', 'V31', 'V32', 'V33', 'V34',
- 'V19', 'V20',
- 'V23', 'V24',
- 'V25', 'V26',
- 'V27', 'V28',
- 'V29', 'V30'

(3) The V columns within each subgroup share correlation over 0.7 with each other. To get rid of the redundant columns, in each subgroup, we only keep the one that has the most number of unique values, which we assumed to be the most informative among the group, and discard others. In our example, the final selection will be:

- 'V13', 'V14', 'V17', 'V20', 'V23', 'V26', 'V27', 'V30'

**Optimization Techniques**:

We did this whole selection process using the multiprocessing module because of the independence nature between NaN groups. By distributing these NaN groups to different processes, they can run in parallel to accomplish step 2 and step 3 at the same time, and then we collect the results from all processes. It takes only 6.23s to finish the whole task, and we ended up reducing the 340 Vs to only 122 Vs.

### *2.2. Impute Missing Values*

As we've already seen in the V selection section, there are significant amount of missing values in our data, so imputing these null values effectively is an important step in our preprocessing steps. According to the Kaggle report written by Gunes Evitan [3], there are high dependencies between columns. For example, card columns represent payment card information, such as card type, card category, issue bank and country. Therefore, one-to-one correspondences may exist between two columns, which enables us to impute values.

To check the dependency between columns, Evitan built a helper function: CheckDependency with two inputs: independent variable and dependent variable. The function checks the value counts of two given variables and outputs the percentage of one-on-one correspondence between two variables, so that we can understand causality between two vectors which can't be seen by Pearson correlation. For example, in Figure2 we can see that more than 95% of distinct card1 value have only one unique card2 value. Therefore, it is possible for us to impute missing values in card2 by looking at the card1 column and finding its corresponding value of card2.

| indep variable | dep variable | only 1 uniq% | >1 unique% | only missing | null_perc |
|---|---|---|---|---|---|
| card1 | card2 | 0.952197 | 0.017319 | 0.030484 | 0.016029 |
| card1 | card3 | 0.996314 | 0.001697 | 0.001989 | 0.004162 |
| card1 | card4 | 0.000000 | 1.000000 | 0.000000 | 0.004250 |
| card1 | card5 | 0.966649 | 0.017963 | 0.015388 | 0.008026 |
| card1 | card6 | 0.988708 | 0.009245 | 0.002048 | 0.004172 |

Figure 2.: Check dependency for card columns

**Optimization Techniques**

Applying this function to the whole dataset is time consuming, because there are thousands of pairs of variables by doing permutations among all columns. To speed up the process, I used itertool.permutations and starmap, to iterate CheckDependency function for pairs. The itertool module achieves memory efficiency without providing intermediate results and optimizes the runtime by avoiding nested loops. As the result, the speed is boosted by 6.5%.

### *2.3. Feature Encoding*

Feature encoding is one important feature engineering work to get more fruitful and informative features. Here we list some of the encoding techniques we have done:

- **Label Encoding**: Convert categorical features to integers.
- **Frequency Encoding**: Help to identify whether a certain column value are rare or common, for example, it can find out which credit cards are frequently used and which are not. Let's say if one card has only been used once but it has a huge transaction amount, then our model will probably mark it as suspicious.

[3] https://www.kaggle.com/gunesevitan/ieee-cis-fraud-detection-dependency-check

- **Aggregate Encoding**: Compute simple statistics, such as the average or the standard deviation of the transaction amount for a card.
- **Feature Combining**: Feature combining allows us to identify a specific user more accurately. Because all these features have been processed and encrypted, for example, card1 only tells the first few digits of the card number, by combining it with the address information and the card starting date, we can identify a user most of time. We used this combined feature together with the frequency encoding and aggregate encoding to get more relevant and useful features in the user level.

**Optimization Techniques**:

| Operation | pandas dataframe | RAPIDS cuDF |
|---|---|---|
| I/O speed | 29.6s | 1.98s |
| Frequency Encoding | 1.27s | 0.25s |
| Label Encoding | 1.82s | 0.14s |
| Aggregate Encoding | 8.04s | 2.09s |
| Feature Combining | 1.23s | 0.21s |

Table 1.: pandas vs RAPIDS

We did all these feature encoding process based on the RAPIDS cuDF API[4] instead of the pandas library. cuDF is a GPU accelerated dataframe library, and it optimizes operations like data loading, filtering, joining and other kinds of manipulations on dataframes. Table 1 compares the performance between pandas and cuDF, and based on our experiment, the speedup is obvious and significant on almost every facet, especially the I/O speed, we can see that it's almost 15 times faster.

One thing I want to mention is that because of the local memory limitation of GPU and the large size of our dataset, we did some memory reduction techniques to make it work smoothly on GPU, for example, we convert the data type of each numeric column to the most appropriate ones based on the range of values in that column:

- -128 to 127 $\rightarrow$ int8
- -32768 to 32768 $\rightarrow$ int16
- float64 $\rightarrow$ float32

With this memory reduction technique, we reduced the memory usage from 1755.3 Mb to 1032.3 Mb (44 percent reduction).

## 3. Modeling

### *3.1. XGBoost*

Extreme Gradient Boosting (XGBoost) is a kind of gradient boosting decision trees. For hyper-parameter tuning, we applied Bayesian Optimization to find out the best paramter. The AUC achieves around 0.937. In order to speed up the run time, we tried both CPU and GPU(Mitchell and Frank, 2017).

- **CPU**: run time: 188s
- **GPU**: run time: 79s

It is obvious that GPU decreased the training time dramatically for our XGBoost model. It is straightforward to enable GPU XGboost model just by changing the hyperparameter "tree_method" from "hist" (CPU) to

[4]https://docs.rapids.ai/api/cudf/stable/

"gpu_hist"(GPU). GPU performs much better for large datasets like the one we used. However, what needs to be mentioned is that, CPU indeed has a better performance for very small datasets. It takes longer to call GPU API for computations than CPU. Although computations on GPU takes less time, considering the time of overhead of requests, CPU is more suitable for very small datasets.

### *3.2. LightGBM*

XGBoost has dominated competitions for structured or tabular data like Kaggle since 2015. But later on, many variants algorithms have been developed. One of them is LightGBM, initially released by Microsoft in 2016. Besides it leaf-wise node splitting strategy that targets to enhance model accuracy in comparison to level-wise node splitting in XGBoost, it mainly has three advantages over XGboost from the perspective of computational time (Machado et al., 2019).

LGBM adopted histogram based splitting. Since in XGBoost, each splitting requires finding the optimal point, which can be costly when there are a lot of continuous features, as it needs to test every possible split and calculate the node impurity. In comparison, LGBM buckets the continuous features into discrete bins, so at each iteration, number of calculation is reduced significantly and it also helps to save memory usage.

LGBM also uses Gradient-based One-Side sampling, a technique aiming to reduce computational time of optimal split finding, which is proportional to number of records and number of features. The core idea is to subsample the instances such that instances with little impact on the training are discarded and those with high impact are kept, so that performance can be maintained while greatly reducing training time.

The last reason is that LGBM adopts Exclusive Feature Bundling. It bundles sparse features together so that less effort is needed on a big dataset. Even though finding optimal bundling of sparse features is NP-hard, but a good approximation suffices in general.

In our case, LGBM reached similar AUC at 0.927 as XGBoost after 400 epochs in just 22s, 9 times faster than XGB-CPU and almost 3 times faster than XGB-GPU version.

## 4. Conclusion

We adopted multiprocessing module and itertools methods learned from this course to fasten our EDA and feature engineering. And for modeling there are available third party packages that already optimize towards the tabular data large in size. We also tried XGBoost and LightGBM under PySpark API, but our cleaned data has more than 200 columns and turning the PySpark dataframe to PySpark RDD form, which is required by PySpark API, takes a very long time as RDD stores data column-wise. We haven't had enough time to come up with a solution for that, which can be the future work. Another future work could be performing more model hyper-parameter tuning to reach the same AUC as is reported on Kaggle leaderboard, but since our main focus is to speed up EDA and feature engineering and reach similar results in less time, we believe our current results already manifest very well.

## References

Machado, M. R., Karray, S., & de Sousa, I. T. (2019). Lightgbm: An effective decision tree gradient boosting method to predict customer loyalty in the finance industry. *2019 14th International Conference on Computer Science  Education (ICCSE)*, 1111–1116.

Mitchell, R., & Frank, E. (2017). Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science 3*, 127.