Machine Learning Engineer Nanodegree

Capstone Project: TalkingData AdTracking Fraud Detection

Pooya Rezaei
May 23, 2018

I. Definition

Project Overview

Click fraud, which is clicking on advertisements with no real interests in the product or the offered services, is an important problem in online advertising that leads to wasted money for the service providers. TalkingData, China's largest independent big data service platform handles three billion clicks per day, of which about 90% are potentially fraudulent. For this reason, TalkingData has turned to Kaggle community and defined a competition to detect fraudulent clicks using a dataset of approximately 200 million clicks over four days as the training set. The goal is to predict fraudulent clicks in the testing set, which includes clicks on a fifth day. The data is available in the competition website on Kaggle. For each click in the training set, the following features are available:

- ip address of the click
- app id for marketing
- device type id that clicked (e.g., iphone 6, Huawei mate 7, etc)
- os version id of mobile phone
- channel id of mobile ad publisher
- click time
- download time (if the user downloaded the app)
- is_attributed indicating the app was downloaded, which is the target that is to be predicted

Problem Statement

The challenge is to build an algorithm that predicts clicks that lead to downloading the apps. So this is a binary classification problem that uses all available features of the clicks mentioned above. Many machine-learning algorithms solve binary classification problems including Logistic regression, SVM, deep learning, bagging and boosting methods. Among these, ensemble methods especially boosting algorithms have been very promising in recent Kaggle competitions.

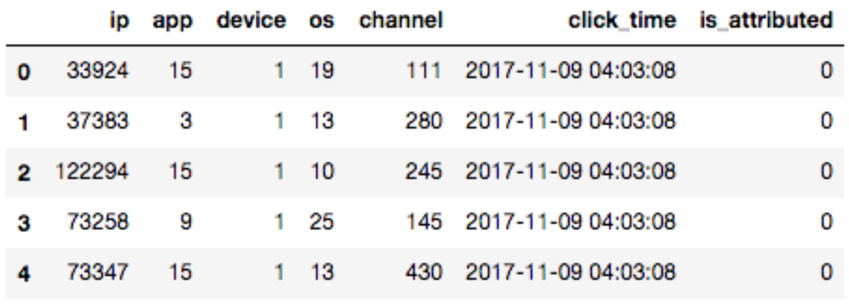
Metrics

There are many ways to evaluate the results of a binary classification problem like the one here. These include accuracy, precision, sensitivity/recall, F1 score and Area Under the ROC Curve (AUC). The first four metrics are all affected by class imbalance that exists to a high degree in this problem. However, AUC is not affected by class imbalance, and so we are going to use that as the evaluation metric. This is also the metric that the competition is going to be evaluated on.

II. Analysis

Data Exploration

Two separate datasets are provided in this competition including the training and the testing sets. The output variable (is_attributed) is only provided in the training data. The data has seven features, from which five are categorical. These include ip, app, device, os and channel. All of these features are encoded to integer values to obfuscate the actual values and preserve privacy of the users. Two other features are in date-time format: click time and download time. Because download time only exists for the users who ended up downloading the app it does not provide useful information generally and so we do not use it in our analysis. Table 1 shows the first five rows of the training data.



	ip	app	device	os	channel	click_time	is_attributed
0	33924	15	1	19	111	2017-11-09 04:03:08	0
1	37383	3	1	13	280	2017-11-09 04:03:08	0
2	122294	15	1	10	245	2017-11-09 04:03:08	0
3	73258	9	1	25	145	2017-11-09 04:03:08	0
4	73347	15	1	13	430	2017-11-09 04:03:08	0

Table 1. Five first rows of the training set

An important challenge about this problem is the size of the data. The training data is about 8 GB, and so loading it needs enough RAM and takes some time. There are a total of 184,903,891 records in the training set and 18,790,469 records in the testing set. The training data includes user click data on Monday through Thursday of a week and the testing data has the click data for Friday of the same week. Because of the size of the data we loaded the last 40,000,000 rows of the data for data exploration and training.

Another challenge in this problem is the high class imbalance in the predicted variable. The number of downloads in the training set is 100,508, which is only 0.25% of the total number of records. So we need to use special means that are appropriate for datasets with class imbalance in this problem. Table 2 shows some characteristics of the training data.

	ip	app	device	os	channel	click_time	is_attributed
count	40000000.0	40000000.0	40000000.0	40000000.0	40000000.0	40000000	40000000.0
unique	104523.0	469.0	2192.0	443.0	185.0	43013	2.0
top	5348.0	3.0	1.0	19.0	107.0	2017-11-09 12:00:11	0.0
freq	318367.0	6198926.0	37343882.0	9368370.0	2304914.0	1471	39899492.0
first	NaN	NaN	NaN	NaN	NaN	2017-11-09 04:03:08	NaN
last	NaN	NaN	NaN	NaN	NaN	2017-11-09 16:00:00	NaN

Table 2. Information about the subset of training data

Note that there are many unique values for each categorical feature. Figure 1 shows a bar plot of the number of unique values for the categorical features.

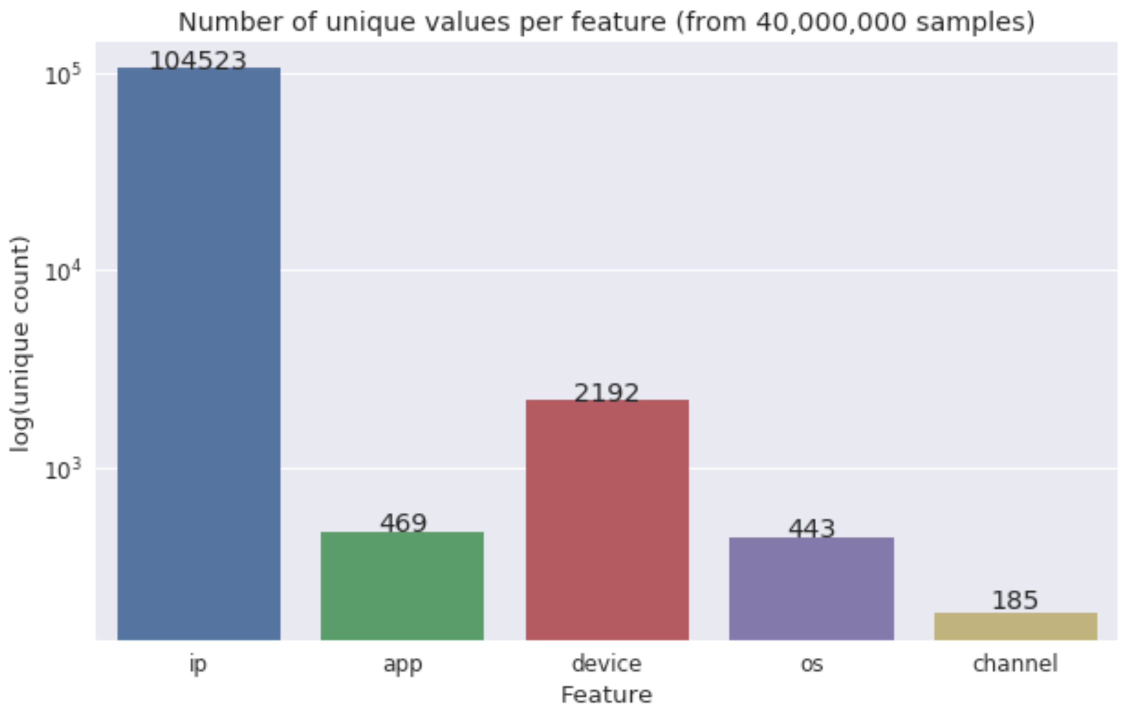


Figure 1. Number of unique values per feature (recreated from [1])

Another interesting issue is the frequency of the most repeated value for each feature from Table 2. Some features come from a limited number of possible values, e.g., app, device, os and channel can be shared among a number of users, and so it will not be surprising to have a high number for the most repeated value. However, IP should have a specific value for each user, but we see the top value for the ip has a frequency above 300,000. There are at least two theories why this might happen. One is that most of these clicks are fraudulent. Another possibility is to have many users on the same network such that they all have the same IP. Table 3 shows the 10 most repeated IP values in the training data with the number of repetitions (counts) and the number of downloads among all the repetitions (counts_attributed). This table shows that many of the most repeated IP's end up downloading the app. So many users that share the same network perhaps have these same IP's.

	ip	counts	counts_attributed
0	5348	318367	525
1	5314	284299	508
2	73516	155728	244
3	73487	154130	237
4	53454	116348	39
5	112302	114715	54
6	17149	113777	52
7	26995	97968	81
8	114276	95276	15
9	95766	90304	120

Table 3. The 10 most repeated IP values in the training data with their count (counts) and the number of downloads for that IP (counts_attributed)

Exploratory Visualization

This section explores any potential patterns in the click time for each user. Here we round the hour of the click and look at the histogram of the click hours. We cannot use the training data for this purpose as the data is sorted sequentially in time and we just loaded a sequential subset of the data. A sample of the data is provided by the competition that has 100,000 randomly distributed samples from the training data. Figure 2 shows the number of clicks at each hour of each day in the simulation.

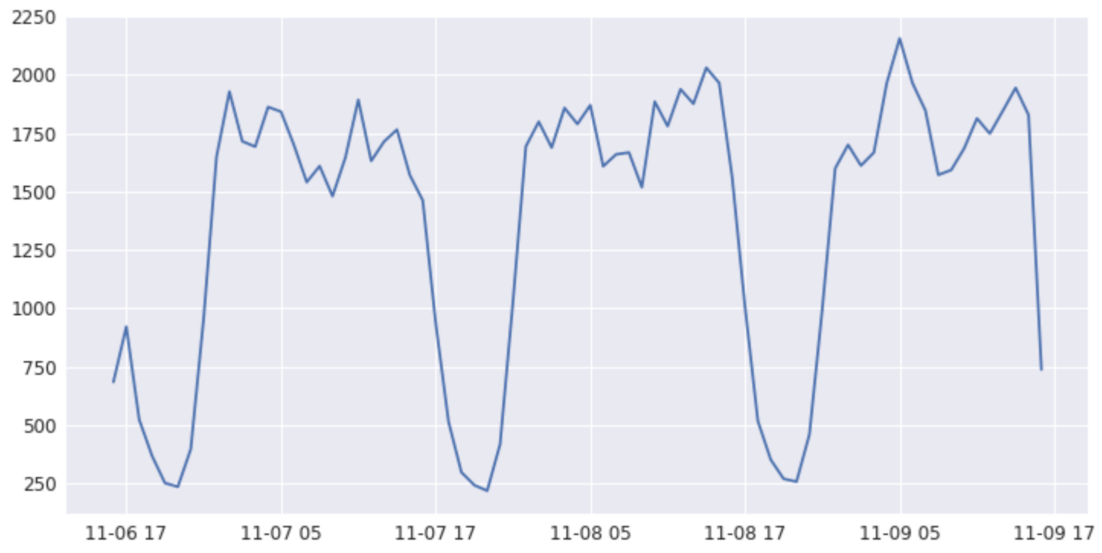


Figure 2. The number of clicks at each date and hour for the 100,000 sample data (recreated from [1])

It is clear that there is a daily pattern in the number of clicks. The clicks start to increase after midnight and are almost at the same rate until the afternoon, when they start to decrease until midnight. Figure 3 shows the number of clicks in each hour irrespective of the date for the same sample data. We see more clearly that the number of clicks starts to increase after midnight when they remain almost at the same level until they decrease at 4 PM.

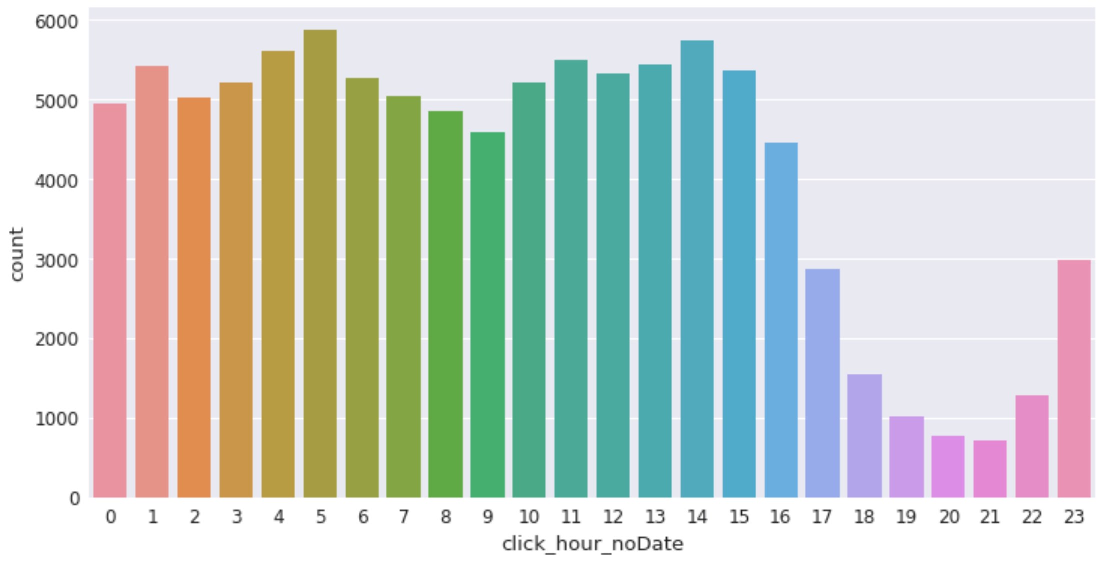


Figure 3. The number of clicks at each hour for the 100,000 sample data

Figure 4 shows the average number of clicks that led to a download in each hour. This follows more or less the same pattern as the total number of clicks. However, it is difficult to draw any conclusion on the click time pattern for downloads using a small sample data, because the variability is too high and the number of downloads are a very small portion of the total number of records.

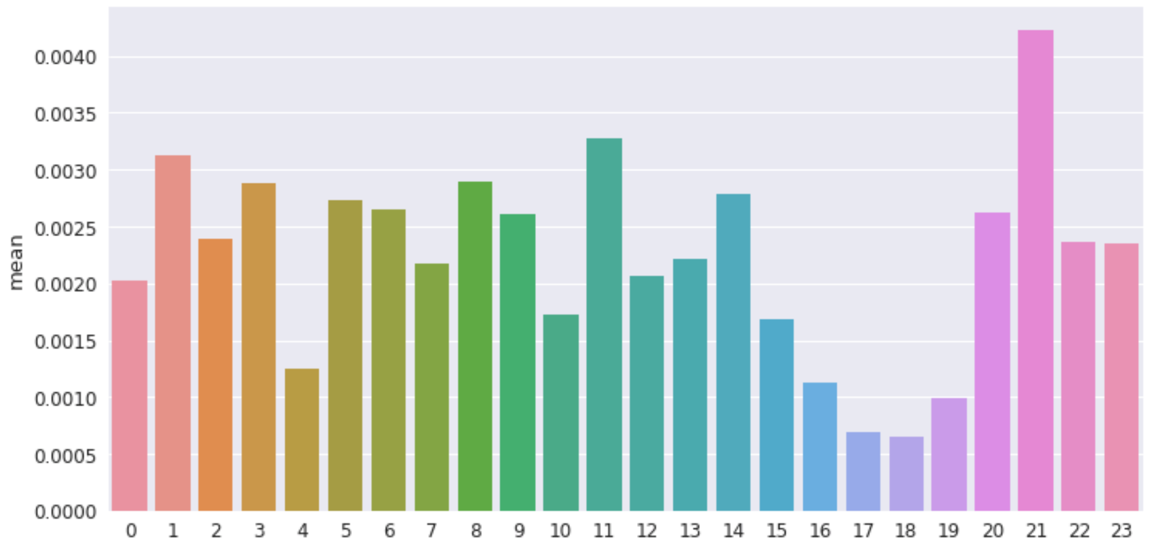


Figure 4. The number of clicks that led to a download at each hour for the 100,000 sample data

Algorithms and Techniques

Among algorithms that are suitable for binary classification problems, we have used Random Forest for benchmarking and Gradient Boosting Method as the final algorithm. As mentioned before, ensemble methods have been very promising for binary classification problems in many Kaggle competitions. Random forest is a modified bagging method that trains an ensemble of decision trees and uses averaging or voting mechanisms to predict the output variable. Similar to bagging, for each decision tree bootstrapping is used to train the tree with a limited number of records in the training set. The only difference of Random Forest with bagging is that a subset of the features is used at each split of a decision tree. Using a subset of the features and a subset of the data (bootstrapping) Random Forest reduces variance of the prediction effectively.

Boosting methods are another ensemble technique that use sequential models (instead of parallel models in bagging methods) in learning a model. Specifically, Gradient Boosting Methods train weak learners (algorithms that can do just slightly better than random guess) sequentially. The training starts with a model to predict the output. The sequential learning works such that the goal is to predict the error of the previous model at each step. The final output is then the sum of predictions from all the sequential models. In this project, we have used LightGBM, which is a fast, distributed, high performance gradient boosting framework based on decision tree algorithms.

Benchmark

One solution to the problem is to use Random Forest algorithm with the given features. We show in the next section how feature engineering is essential in this problem to come up with good models. However, for the base benchmark model we are going to use ip, app, device, os and channel as the features and train a random forest algorithm. As mentioned before, these are all encoded categorical variables. The typical approach is to use one-hot encoding of the categorical features, but because of many unique values for each feature we are not going to do that here, and will use the features as they are.

Because of the high volume of data we have trained the benchmark model with only 4 million ending rows of the training data, and used 10% of the data for validation. We have trained a random forest algorithm with this data with max_depth of 10 and other default values in the sci-kit learn package. The training AUC is 0.97565 and the evaluation AUC is 0.97562, which is very close to the training data AUC value. After we submit the test data result to the competition website we got the Leaderboard AUC score of 0.9288 for the testing set, which means that we are over-fitting to the training data. Although the model generalizes very well to the evaluation data, it does not generalize to the test data. This is de to the limited number of data used for training. In the next section, we show how using more data with many more informative features engineered from the existing features can result in a much better prediction for the test data.

III. Methodology

Data Preprocessing

The benchmark was done with the features as they were given, however, it is very typical to look at the total number of clicks with the same IP, device, etc in advertisement fraud. First, we extracted day and hour from click time, which is provided, and added them to the features. We also created new features by counting the number of clicks that are from a combination of the features. This requires using the groupby function first and then counting the number of records. We have used combining the features by grouping ip-day-hour, ip-app, ip-app-os, and ip-app-device-os. In addition to count, one can take the mean and variance of some features for the group. Experiments have shown that the following features have been helpful:

- variance of hour when grouping by ip-day-channel
- variance of hour when grouping by ip-app-os
- variance of day when grouping by ip-app-channel
- mean of hour when grouping by ip-app-channel

Implementation

We used LightGBM in python to predict whether or not the users downloaded the app. LightGBM grows trees leaf-wise while other algorithms grow level-wise. It will choose the leaf with max delta loss to grow. When the number of leaves is fixed, a leaf-wise algorithm can reduce more loss than a level-wise algorithm. However, leaf-wise growth may cause over-fitting when dataset is small. Figures 5 and 6 show level-wise and leaf-wise tree growth respectively.

LightGBM has some core and some learning control hyper-parameters that need to be tuned for training. A complete list of these parameters is available in [2].

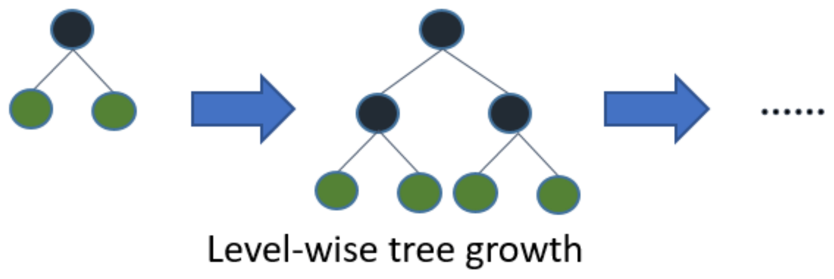


Figure 5. Level-wise tree growth in decision tree learning algorithms [2]

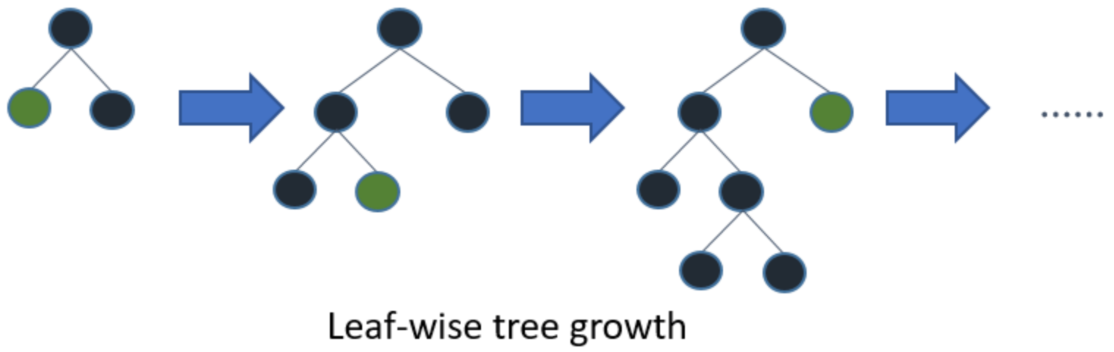


Figure 6. Leaf-wise tree growth in decision tree learning algorithms [2]

A function is responsible for the modeling. This function takes LightGBM paramters and training data and returns the trained LightGBM class object. We used 5% of the training data for validation. Below are the parameters used for LightGBM with their explanation:

- learning_rate: 0.15 (determines the impact of each tree on the final outcome)
- scale_pos_weight: 99 (to increase the weight of positive class as we have a very large class imbalance in this problem)
- num_leaves: 7 (number of leaves in each tree)
- max_depth: 3 (maximum depth of each tree)
- min_child_samples: 100 (Minimum number of data in one leaf. This parameter is used to avoid over-fitting.)
- max_bin: 100 (maximum number of bins that feature values will be bucketed in. Small number of bins may reduce training accuracy but may increase general power to deal with over-fitting)
- subsample: 0.7 (this will randomly select part of data without resampling and can be used to speed up training and deal with over-fitting)
- subsample_freq: 1 (frequency for bagging, 0 means disable bagging and k means will perform bagging at every k iteration)
- colsample_bytree: 0.9 (fraction of features to selclt on each iteration if feature_fraction smaller than 1.0. Take 90% of features before trainging each tree.
- early_stopping_rounds: 30 (will stop training if one metric of one validation data doesn’t improve in last 30 rounds)
- num_boost_round: 500 (number of boosting iterations)

IV. Results

Model Evaluation and Justification

The final model was run on Kaggle kernels for about 1.5 hours (before early stopping criterion stopped the training process). Note that because of the volume of the data we need at least 16 GB of RAM to be able to run this model (even with the partial data that is used here). The final AUC score was 0.969794 on the training set and 0.985616 on the validation set (5% of the training). The fact that validation set was getting a score close to the training set shows that this model can be trusted.

Furthermore, after submitting the solution to the Kaggle competition the model got a public score of 0.96779 and a private score of 0.9689007. Public leaderboard score is computed on a fraction of the testing set, and is available during the competition. Private leaderboard score is computed on the full testing set and is available only after the competition has ended. If a model is over-fitting to the testing set, the public score will be high but the private score will be much lower. The fact that the private score is higher than the public score means that the model is not over-fitting to the testing set at all. It should also be noted that the public and private scores are much higher than what the benchmark model gave using Random Forest with the original features. The private and public scores for LightGBM are even higher than the training and validation score in the benchmark model.

V. Conclusion

Reflection

In this project, we presented a model for predicting click fraud on mobile app advertisements. There were a few challenges involved with the modeling. First, some of the categorical features had many unique values, e.g., more than 300,000 IP's in 40,000,000 records. Not all algorithms can handle this many unique values for a feature. Secondly, the size of the data in number of records was quite large. This meant that loading the full dataset needs higher than 8 GB of RAM. Lastly and most importantly, there was a high class imbalance in this problem, such that only about 0.25% of the data were from positive class (were fraudulent). We used LightGBM, which is a high performance gradient boosting package based on decision tree algorithms. Using LightGBM we could achieve an AUC of 0.9689 on the full test dataset. This became possible not only due to the superiority of LightGBM but also because of engineering many new features based on different groupings of the original features and measuring aggregate properties (e.g., counts of records) of the groups.

Improvement

Potential improvements can be done by more rigorously searching the high-dimensional parameter space of LightGBM, instead of using some parameter examples that we used from [3] in this project. The computational resources and time for this task is quite noticeable, as each round of training the model took about 1.5 hours on Kaggle Kernel, which has 17 GB of RAM (and a maximum simulation time of 2 hours). Another potential improvement can be done by making sure that the validation data is for a different day than the training data. This may help the training process because we know that the test data is for a different day than the training data and this way, we can mimic the same conditions for the validation data.

VI. References

[1] https://www.kaggle.com/yuliagm/talkingdata-eda-plus-time-patterns

[2] LightGBM documentation https://lightgbm.readthedocs.io/en/latest/

[3] https://www.kaggle.com/pranav84/lightgbm-fixing-unbalanced-data-lb-0-9680/code