How to Achieve a Silver Medal within the Kaggle Competition as a Beginner

***Shinan Xu**, Nanjing University of Science and Technology, China, shinan.xu97@gmail.com*
***Zhitong Ye**, Beijing Normal University-HongKong Baptist University United International College (UIC), kelly_0928@qq.com*
***Zisheng Liang**, Northeastern University, China, i.liangzisheng@gmail.com*
***Yufeng Ma**, The Chinese University of Hong Kong, Shenzhen, 115010201@link.cuhk.edu.cn*

Kaggle is a data science and machine learning competition platform, owned by Google, upon which companies, including Facebook, Google, Quora, and others, publish their data and problems. Many data scientists, data-mining experts, and machine-learning engineers from all over the world, who call themselves "Kagglers" attempt to produce the most accurate predictions by analyzing the data and building predictive models.

The contest that we took part in was *the TalkingData AdTracking Fraud Detection Challenge* held by Kaggle. TalkingData is China's largest independent big data service platform. Online advertisement companies, such as TalkingData, encounter click fraud often, resulting in inaccurate click data, and thus wasting both time and capital when it occurs in large volumes. TalkingData handles over 3 billion clicks per day, thus building an IP and device blacklist record of users who do not install apps after clicking on them by employing machine learning methods.

The data presented by TalkingData contains information relating to users who click on advertisements, such as IP address', apps, devices like iPhone, Xiaomi, and Huawei, operating systems like IOS or Android, channels, the time of the clicking, and so on. Our job was to employ the training data relating to circa 200 million clicks over a four day period, including the features mentioned above, in order to build a reasonable model with which to predict whether the click is fraudulent or not. The score will be determined by the precision of our prediction based on the testing data.

1. Introduction

The training dataset provided by Talkingdata was around 8G in size, covering approximately 200 million clicks occurring between Nov, 06, 2017 and Nov, 09 2017. We were tasked with using the dataset to predict malicious IP's capable of producing high volumes of clicks but which never install said apps.
The test dataset was over 700M, covering approximately 19million clicks taking place on 10/10/2017, from 4:00 to 15:00.

IP: IP address;
App: app id provided by advertisers;
Device: user's mobile device id, such as iPhone 6, iPhone 7;
OS: the operating system version id of the user mobile device;
Channel: advertising channel id;
Click_time: click time (UTC time), in the format yyyy-mm-dd hh:mm:ss;
Attributed_time: if the user downloads the app, this is the download time;
Is_attributed: whether the app is downloaded, this is the target variable;

Here, IP, app, device, OS, and channel are all classified variables and are processed by coding.

2. Work Flow

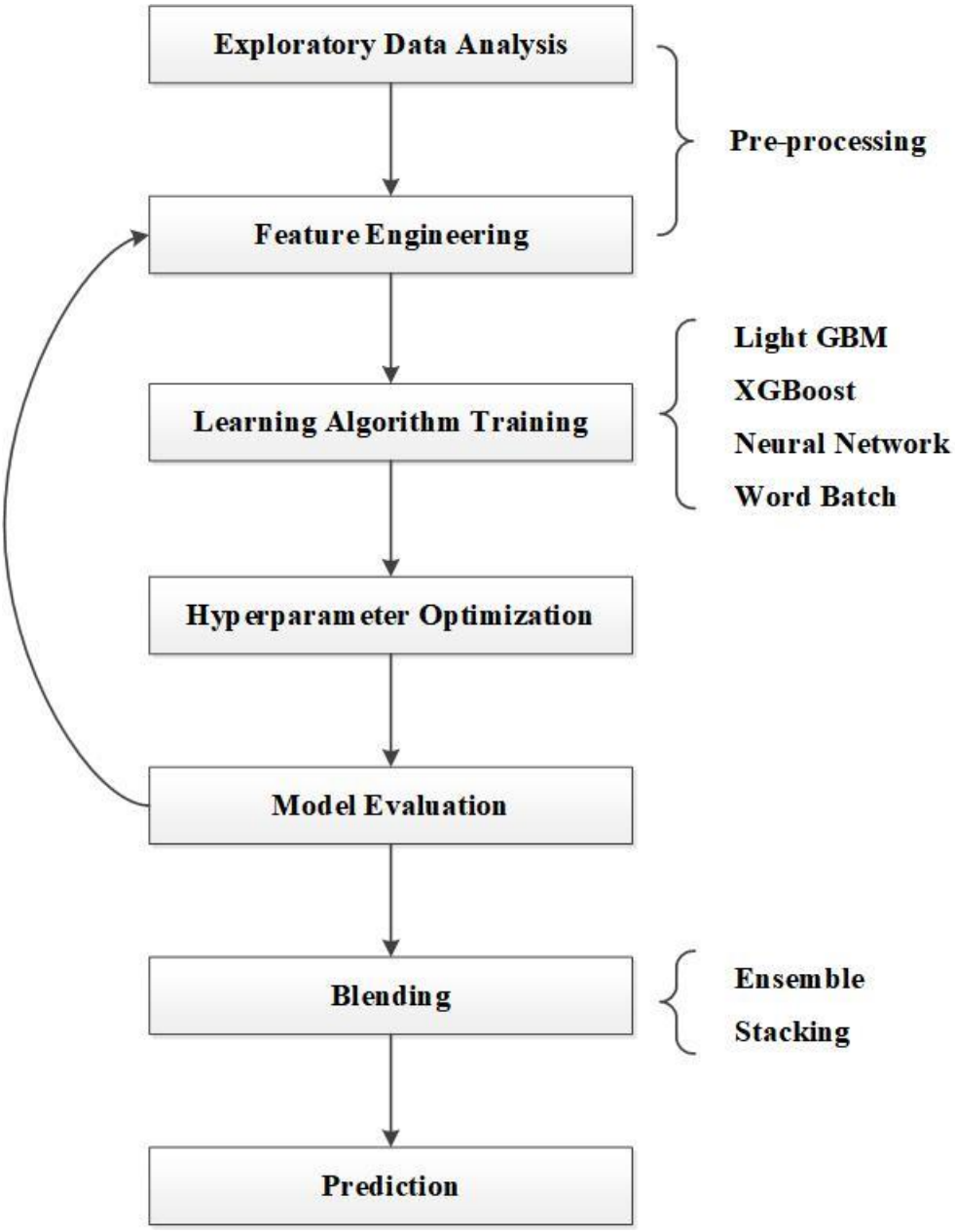

Figure 2-1 Work Flow

3. Pre-processing

3.1 Exploratory Data Analysis

3.1.1 Data Distribution Pattern

By handling the whole dataset, we get the number of unique values of each feature (shown in Figure 3-1). We can then find that the uniqueness count of the IP is much more than that of the application, operating system, and channel.

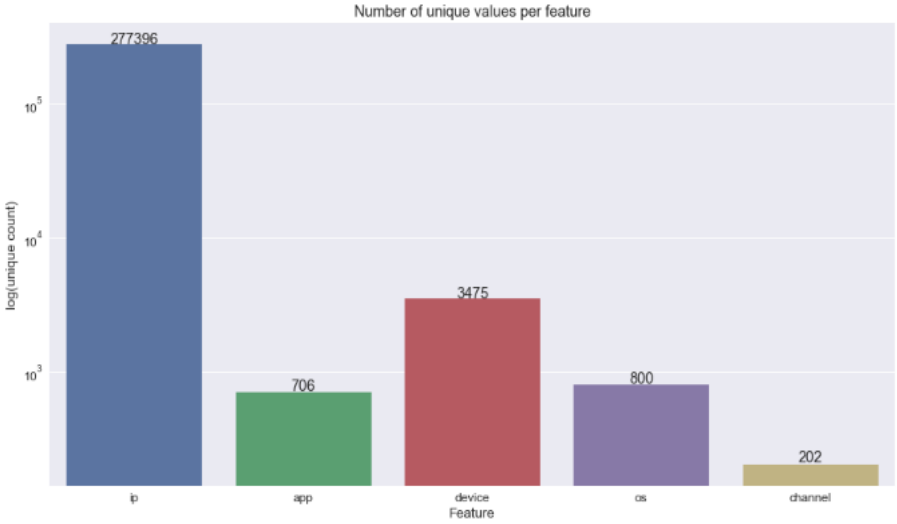

Figure 3-1 The number of unique values of each feature

The occurrences plots IP, application, device, OS, and channel values (Figure 3-2), and thus we see that some operations are possibly "fraud" operations, as their occurrence time is greater than expected.

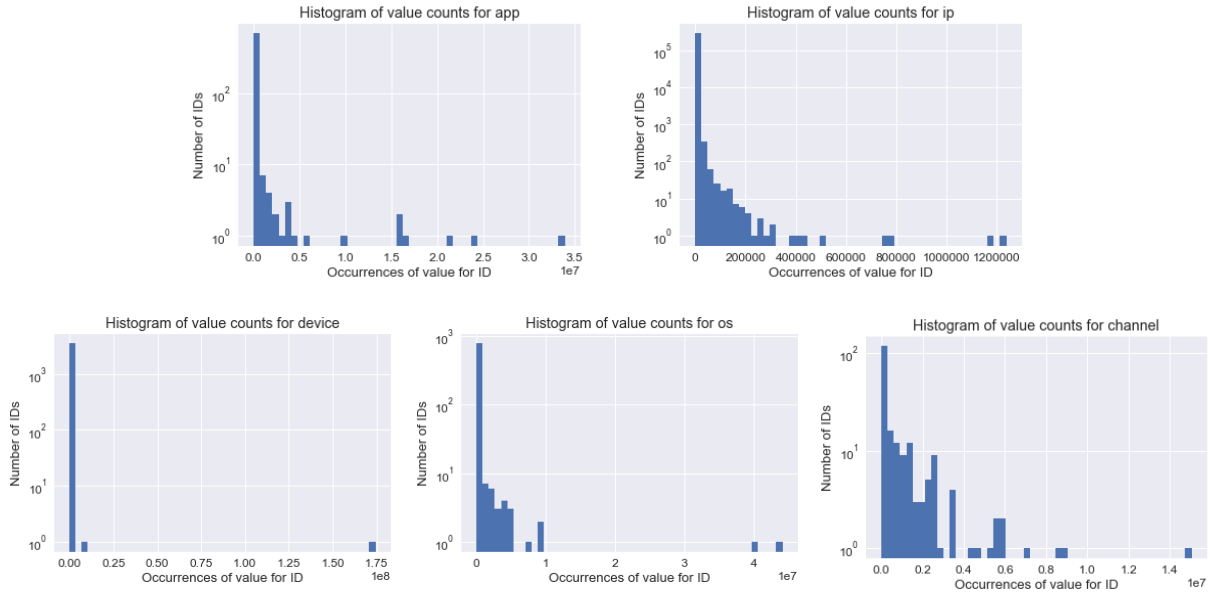

Figure 3-2 Value counts for app, IP, device, OS, and channel

In addition, Figure 3-3 shows us that the proportion of users who download the app versus those who do not is quite unbalanced.

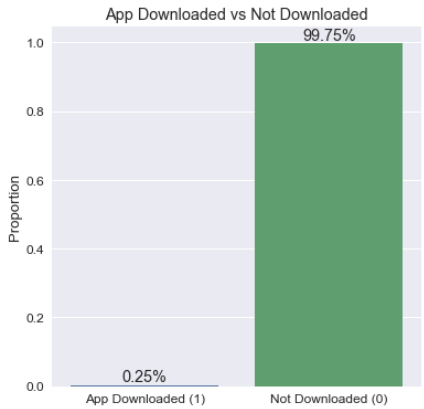

Figure 3-3 App Downloaded Compared with Not Downloaded

3.1.2 Data Time Series Pattern

We employed click time features to build hourly patterns. Figure 3-4 displays the time patterns of clicks by the hour over the course of these days. The time pattern of hourly click frequency is definite, however the hourly time pattern in ratios is not as obvious as said click frequency.

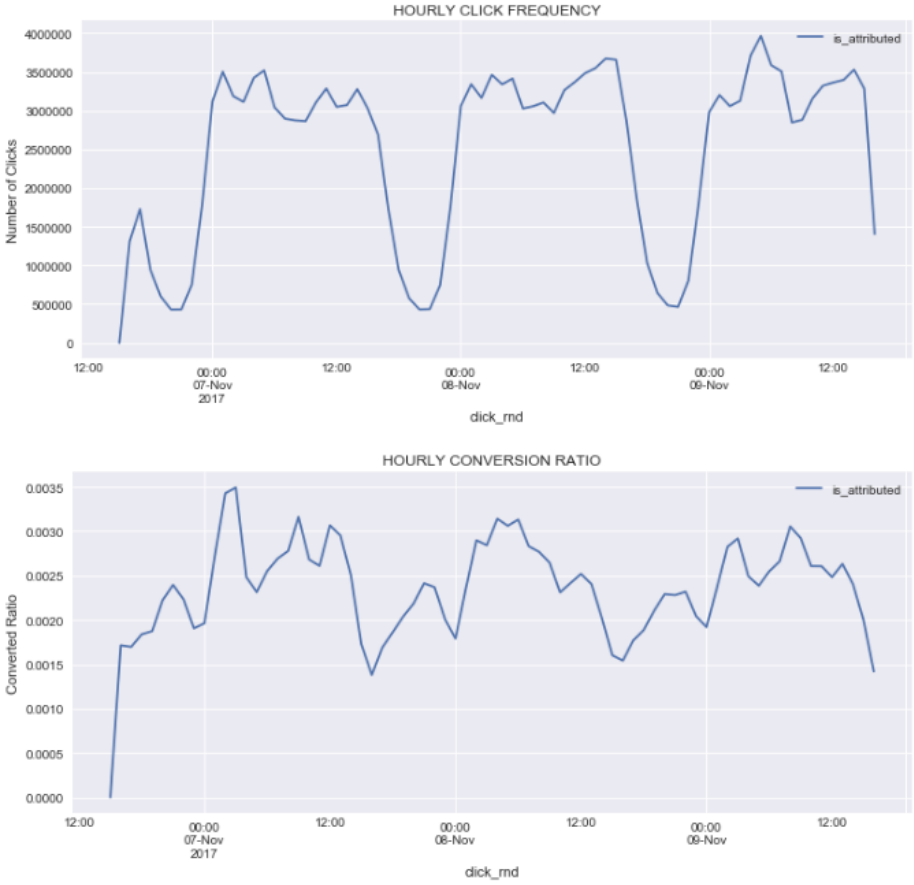

Figure 3-4 Hourly Click Frequency and Conversion Ratio

Then we try to merge these four days together and extract the hour of day from each day as a separate feature (Figure 3-5). By checking the click count compared with the proportion converted each hour, we find that between 12:00 and 13:00 the number of clicks and the proportion converted are both high, and after that these values both drop. However, from 15:00 the proportion converted suddenly increases, but that click count continues going down, which is a strange phenomenon that may need further analysis.

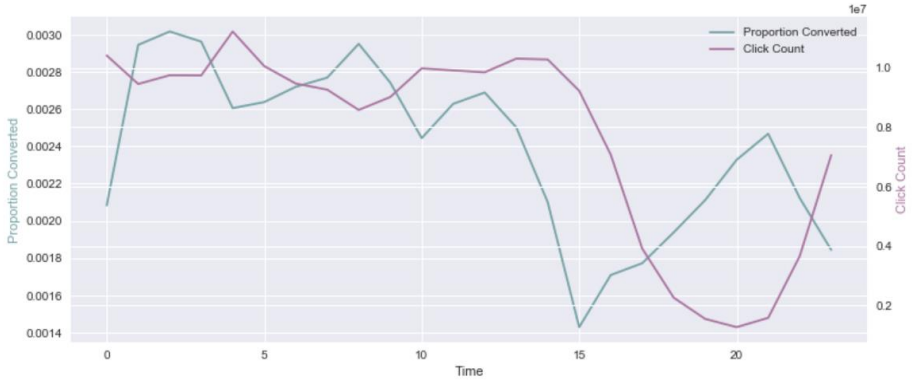

Figure 3-5 Click Count comparted with Proportion Converted

3.2 Feature Engineering

3.2.1 Click Time

We first transpose 'Click_time' into day and hour categorical features. We do not believe that the details, click minute and second, are helpful within the classification task, and therefor don't include them.

3.2.2 Groupby Related

We groupby some of the original categorical data (E.g., group samples by the 'IP', 'app', 'channel'), and then get unique values, variances, counts, and the cumulative counts of that group.

Feature	Method
X0	Group by ['IP', 'channel'], calculate the number of unique items
X1	Group by ['IP', 'device', 'OS', 'app'], and calculate the cumulative count.
X2	Group by ['IP', 'day', 'hour'], and calculate the number of unique items
X3	Group by ['IP', 'app'], and calculate the number of unique items
X4	Group by ['IP', 'app', 'OS'], and calculate number of unique items
X5	Group by ['IP', 'device'], and calculate the number of unique items
X6	Group by ['app', 'channel'], and calculate the number of unique items
X7	Group by ['IP', 'OS'], and calculate the cumulative count
X8	Group by ['IP', 'device', 'OS', 'app'], and calculate the number of unique items
ip_tcount	Group by ['IP', 'day', 'hour'], and calculate the count
ip_app_count	Group by ['IP', 'app'], and calculate the count
ip_app_os_count	Group by ['IP', 'app', 'OS'], and calculate the count
ip_tchan_count	Group by ['IP',' day', 'channel'], and calculate the variance
ip_app_os_var	Group by ['IP', 'app', 'OS'], and calculate the variance
ip_app_channel_var_day	Group by ['IP', 'app', 'channel'], and calculate the variance
ip_app_channel_mean_hour	Group by ['IP', 'app', 'channel'], and calculate the mean

3.2.3 Next Click

Since repeated clicks from the same IP address is highly likely a fraud click, we calculate the nextClick attribute within the first group by employing some categorical features and then calculating the differences between 'clicktimes' for the next sample within the group.

3.3 Correlation

The correlation matrix plot of our added numerical data is shown below. From the figure we can deduce that many of them are highly correlated (i.e., correlation coefficients larger than 0.5). The main reasons are that those numerical features are all generated by the original 5 categorical features, so they are internally correlated. Meanwhile, the high correlation amongst features is one reason for us to apply highly non-linear models such as neural networks and CART based on the high correlation among features.

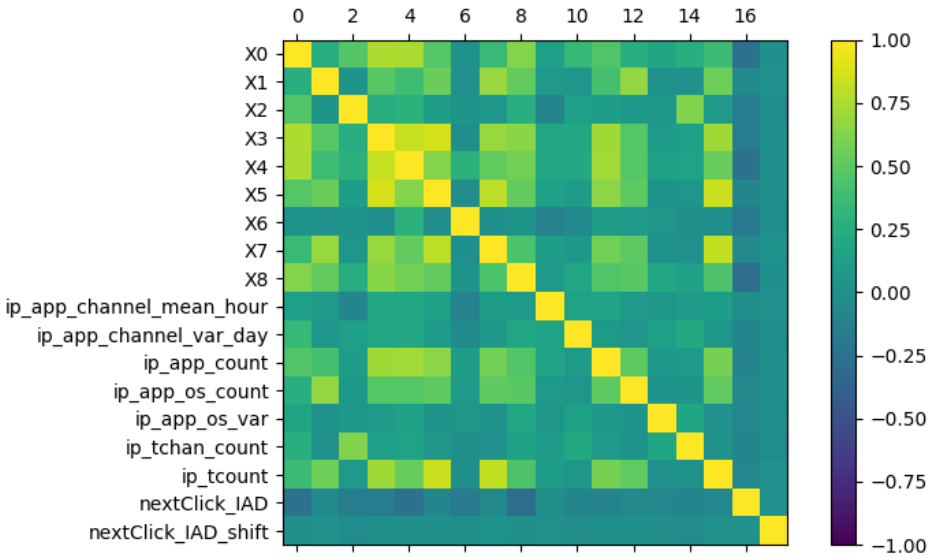

Figure 3-6 Correlation

4. Learning Algorithm Training

4.1 Overview of Machine Learning Methods

We have tried a number of different machine learning methods. Below is a quick review of the methods with a general description, their pros and cons.

Classifier	Description	Pros	Cons
Light GBM	• Uses histogram based algorithms, which separates feature (attribute) values into discrete bins • Reduces calculation cost of split gain and uses histogram subtraction for further speed-up	• Speeds up training and reduces memory usage • Supports parallel processing • Higher accuracy (especially when handling categorical features) due to Leaf-wise (Best-first) Tree Growth and optimal split for categorical features	• Histogram based algorithm which cannot find exact split point • May grow a deep decision tree, resulting in overfitting
XGBoost	• An advanced implementation which produces a prediction model in the form of an ensemble of decision trees • More regularized model to control overfitting	• Wide usage (classification, regression or ranking), can be used to extract variable importance • Supports parallel processing • Built-in cross validation • Makes splits up to the max_depth specified and then start • Uses regularization to reduce overfitting	• Cannot handle category features (only regarded as numeric features) • Harder to tune parameters than within other models • Susceptible to outliers • Lack of interpretability and higher complexity

Neural Networks	• Attempts to identify underlying relationships in a set of data by using a process that mimics the way the human brain operates • Has the ability to adapt to changing input so that the network produces the best possible result without the need to redesign the output criteria	• Could produce high nonlinearity • Less hyper parameters to tune, compared with other methods like XGboost.	• Relative slow to train • Computationally expensive • Hard to judge whether converged or not, during a limited period of time.
Wordbatch	• Wordbatch produces parallel feature extraction from raw text data for uses such as deep learning and text analytics. • Wordbatch additionally provides customizable preprocessing and feature extractors that improve predictive performance.	• Wordbatch works best with large minibatches of text data. • Wordbatch internally stores per-batch statistics of the data, and applies these for uses such as dictionary selection, spelling correction, and online IDF weighting. The larger the batches, the better choices Wordbatch can make in extracting features.	• Unable to learn the relationship between three or more features, the work of cross-feature selection is still unavoidable.

4.2 Light GBM

Conventional implementations of GBDT need to, for every feature, scan all the data instances in order to estimate the information gain of all the possible split points, which makes these implementations very time consuming when handling big data. Light GBM proposes two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) with which to tackle the challenge in the tradeoff between accuracy and efficiency by improving both of them.

With GOSS, the model can exclude a significant proportion of data instances with small gradients, and only uses the rest to estimate the information gain. Since the data instances with larger gradients play a more important role in the computation of information gain, GOSS can obtain comparatively accurate estimations of the information gain with a much smaller data size. With EFB, it bundles mutually exclusive features in order to reduce the number of features. Using a greedy algorithm can achieve superior approximation ratios when finding the optimal bundling of exclusive features.

So Light GBM is faster and more accurate than traditional tree models. It reduces the calculation cost of split gain and uses histogram subtraction in order to optimize both speed and memory usage. Additionally, Light GBM grows tree based on Leaf-wise (Best-first) Tree Growth algorithms and supports categorical features because of the optimal split for categorical Features.
Since four of five features within the data set are categorical features, Light GBM is really an excellent machine learning algorithm.

Here's several parameters with which to tune the Light GBM.

PARAMETER	NOTE	RANGE	CHOSEN VALUE
learning_rate	shrinkage rate	[0, 1]	0.05, 0.1
num_leaves*	number of leaves in one tree	[4, 31]	7, 9, 15, 31
max_depth*	limit the max depth for tree model	[3, 5]	4, 5
min_child_samples	minimum number of data need in a child	100	100
max_bin	number of bucketed bin for feature values	100	100
subsample*	subsample ratio of the training instance	[0.7, 0.9]	0.7, 0.9
subsample_freq	frequency of subsample	1	1
colsample_bytree*	subsample ratio of columns when constructing each tree	[0.7, 0.9]	0.7, 0.9
min_child_weight	minimum sum of instance weight needed in a child	0	0
scale_pos_weight*	weight of positive class in binary classification task	[99, 400]	200

* is the key turning parameters

We first tried a variety of different features and parameter combinations, applying each to our data sets of 40 million on our local computer with 16G memory. Then, we trained 100m and 185m（whole datasets）on the server with 64G memory and 128G swap space. The training results are as below:

No.	Train / Valid	Parameter	Train-auc	Valid-auc	Public LB
0.1	40 / 10m	7 / 4 / 0.7 / 0.7 / 99.7	/	0.999333	0.9690
0.2	40 / 10m	7 / 4 / 0.7 / 0.7 / 99.7	/	0.9996	0.9735
2	40 / 10m	7 / 3 / 0.7 / 0.9 / 200	0.983257	0.983732	0.9782
6	40 / 5m	7 / 4 / 0.7 /0.7 / 200	0.984383	0.987396	0.9790
19	100 / 6.25m	7 / 4 / 0.8 / 0.7 / 200	0.985363	0.988485	0.9792
22	100 / 6.23m	31 / -1 / 0.9 / 0.7 /200	0.98790	0.988559	0.9794
27	185 / 6.25m	15 / -1 / 0.9 / 0.7 /200	0.986786	0.988858	0.9797
30	185 / 5m	31 / -1 / 0.9 / 0.7 /200	0.987276	0.989441	0.9798

Parameter Column: num_leaves / max_depth / colsample_bytree / scale_pos_weight
No. column above is the version number we used internally when we submitted the lgb model. As we can see, this report only shows a few representative ones.

No.2 – No.30 in the table above employs 24 features (including 17 additional feature and next click）. However, No.0.1 only uses 11 features (5 additional), and No.0.2 uses 16 features (10 additional), neither of which include next click.
In addition to the features, we also tuned the early stopping rounds from 50 down to 30 to prevent overfitting.

Finally, we added test supplement data sets to perform feature engineering and use the new feature created to predict the test dataset. The result was significantly improved and achieved 0.981x.

The feature importance of Light GBM is shown below. It shows that “channel”, “app”, “OS” and “hour” are the three most important features, and that the feature nextClick we creates also greatly contributed to the training model.

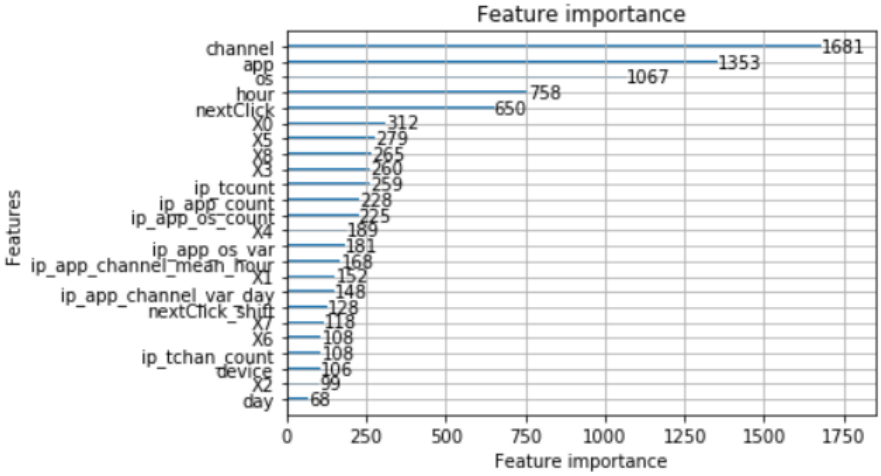

Figure 4-1 Feature importance of Light GBM model

4.3 XGBoost

We used Xgboost to train our tree model. It is a fast and efficient boosting model. There are ***the 10 parameters we use*** to tune the Xgboost.

PARAMETER	NOTE	RANGE	CHOSEN VALUE
eta	step size shrinkage used in update to prevents overfitting.	[0, 1]	0.1
gamma	minimum loss reduction required to make a further partition on a leaf node of the tree, the larger, the more conservative the algorithm will be.	[0, ∞]	5
max_depth*	maximum depth of a tree, the increase of this value will make the model more complex/likely to overfit.	[1, ∞]	[5, 10]
subsample	subsample ratio of the training instance.		0.9
colsample_bytree*	subsample ratio of columns when constructing each tree.	(0, 1]	0.7, 0.8
colsample_bylevel	the subsample ratio of columns for each split, in each level.	(0, 1]	0.7, 0.8
min_child_weight	minimum sum of weights of all observations required in a child.	(0, ∞]	3
alpha	L1 regularization term on weight.		3, 4
lambda	L2 regularization term on weights.		3, 5
scale_pos_weight*	a value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.		9, 100

The training result of the three representative models are:

No	Data size	Parameter	Train-auc	Valid-auc	Public LB
1	50m	0 / 0.8 / 0 / 100	0.993718	0.982350	0.9736961
2	75m	0 / 0.8 / 0 / 100	0.997923	0.988792	.9763157
Final	185m	5 / 0.7 / 3 / 9	0.987301	0.986006	0.9781754

Parameter column: max_depth / colsample_bytree / min_child_weight / scale_pos_weight

We attempted a number of combinations of these parameters and even tried other tuning parameters. However, the tuning parameter for the first and second model are the same, the score increased because we used a larger training dataset. We found that, at times, the larger dataset is more helpful than merely employing tuning parameters. Then we noticed that the train-auc was above 0.99, which means that it faces the possibility of producing an overfitting

problem, so we tried to fit the tuning parameter used to handle overfitting throughout the entire dataset.

We used a sparse matrix to better handle the categorical features “app”, “device”, “OS”, “channel” and “hour” and to achieve a figure of all its feature importance, but the training result is not better than that of Xgboost without the sparse matrix. Moreover, it uses a lot of memory and makes the fitting process slower. So we gave it up.

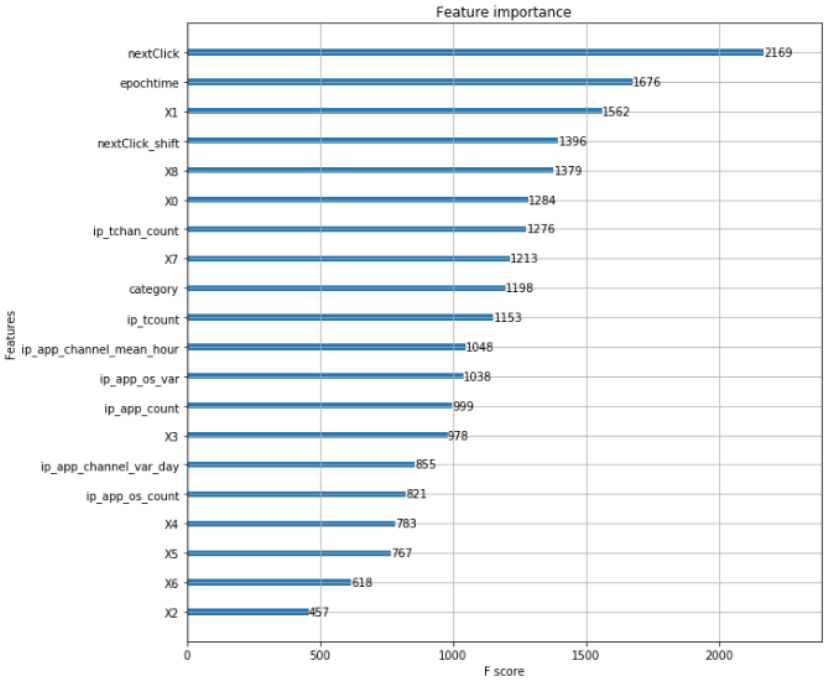

Figure 4-2 Feature importance of the Xgboost with sparse matrix

However, the feature importance of the final training of Xgboost is shown below. It shows that the “app”, “channel” and “nextClick” are the three most important features, which was not the case in the other models we used.

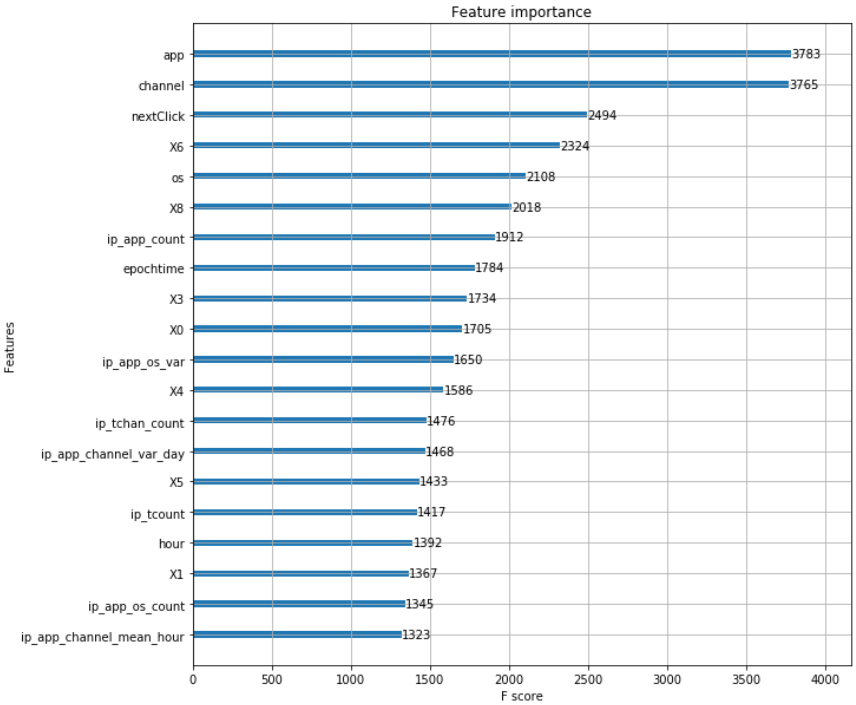

Figure 4-3 Feature importance of final model

4.4 Neural Networks

4.4.1 General Description:

We applied the 'Keras' package in Python in order to build our neural network. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to performing good research. Compared with other packages in Python for the neural network, Keras is more user friendly, putting user experience front and center. Also, it follows the best practices for reducing cognitive loads, offering consistent & simple APIs, minimizing the number of user actions required for common use cases, and providing clear and actionable feedback upon user error. Meanwhile, we ran those codes on GPU, which improved the running time quite a lot. Thus we chose Keras to support our neural network.

Initially we employed a sparse matrix to represent each categorical feature, however the total number of categories for some was too large (thousands) and the training result was not good enough. So we applied an embedding layer for every feature and then combined it with the numerical data input layer, and then applied this newly fully connected layer.

4.4.2 Detail Structure

We applied an embedding layer for every feature through the embedding input channel, where the embedding weights are trainable parameters. For numerical data we first arranged them into categorical data.
We also input other numerical data, which was created within the feature engineering part through the Numerical Input Channel, where normalizations are performed by min-max normalization to keep every feature within the range of (-0.05, 0.05).
Then 2 channels are concatenated and thus a longer vector forms. Then we apply several fully connected layers, where dropout and 'Relu' activation functions are applied. Finally, the output layer is equipped with the 'Sigmoid' activation function so as to produce probabilities.

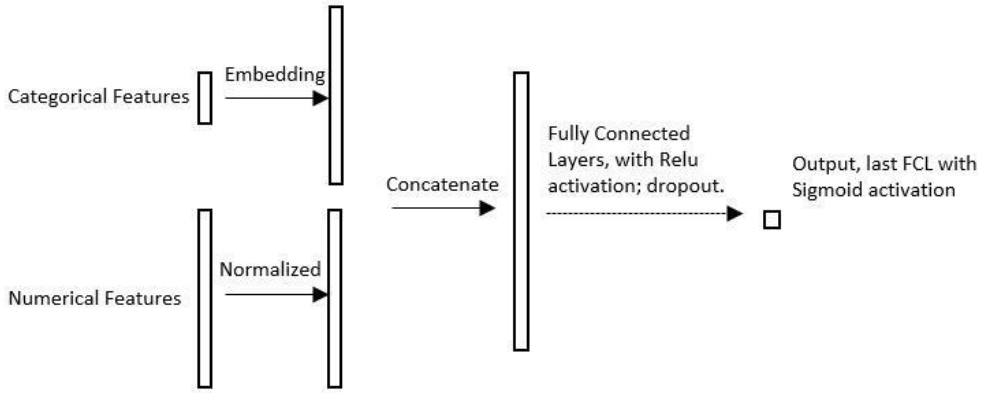

Figure 4-4 Neural Network structure

4.4.3 Hyper parameters

Several hyper parameters in this nn structure are: dropout rate, positive sample weight, epoch, batch size, and embedding size.

PARAMETER	NOTE	RANGE	CHOSEN VALUE
Dropout Rate	The proportion of neurons at each layer to ignore at each training step	[0, 1]	0.2
Positive Sample Weight	Weight on samples with positive label	[1, 400]	200
Epoch	Total number of rounds for a full round of training	>=1	7
Batch Size	Number of samples used for training every time	>=1	50000
Embedding size	Embedding size for every feature in the embedding layer	>=1	50
Optimizer	Optimization method	NA	Adam
Learning rate	Weight updating weight at each training round	NA	0.001

We have set dropout rate of 0.2, positive sample weight of 200, epoch 7, batch size 5000 and embedding size 50.

The training result for three representative models are:

No	Data size	Parameter	Train-auc	Valid-auc	Public LB
1	100m	0.2 / 99 / 5	0.987923	0.984792	0.9787691
2	185m	0.2 / 99 / 5	0.989343	0.985330	0.9790559
3	185m	0.2 / 200 / 5	0.991352	0.986245	0.9794871
Final	185m	0.2 / 200 / 7	0.992623	0.987006	0.9795161

4.5 Wordbatch - Factorization Machine

Inspired by the kernel provided by anttip, we tried the fm_ftrl method using the wordbatch module. wordbatch is a python module written by anttop.

fm_ftrl is The Factorization Machine (FM), and is a machine learning algorithm based on matrix decomposition proposed by Steffen Rendle. FM algorithms can deal with three kinds of problems: 1. Regression problems; 2. Binary Classification, which is the problem we were dealing with; and 3. Sorting (Ranking) FTRL (Follow- regulation-leader) algorithms, proposed and implemented by Google, and which performs well on issues such as logistic regression with non-smooth regularization items. the fm_ftrl method, provided in the wordbatch module, has Linear effects estimated with FTRL and factors which effects estimations within the adaptive SGD. Prediction and estimation multithreaded across factors, and without optimization, and using 40m of data, the result is 0.9752695 on a private dataset and 0.9712230 on a public dataset. It's a pity that we are not familiar with this algorithm, and thus were not able to tone it further.

5. Blending

5.1 Stacking

Stacking is the method in which the base models are combined using another machine learning algorithm. The basic idea is to train machine learning algorithms with the training dataset and to then generate a new dataset with these models. Then this new dataset is used as input for the combiner machine learning algorithm.

Base model	Stacking model	Public LB
0.9782_lgb + 0.9787_nn + 0.9736_xgb	Logistic Regression	0.9786
0.9794_nn + 0.9790_lgb	Logistic Regression	0.9801
0.9794_nn + 0.9795_lgb + 0.9798_lgb	Neural networks	0.9803

We use two models to perform the stacking, one is logistic regression, the other is neural networks.

5.2 Ensemble

Ensemble methods are techniques that create multiple models and then combine them to produce improved results. Ensemble methods usually produce more accurate solutions than a single model would.
After some trials, as stacking models is time-consuming and the result is close to that of the ensemble model, we used the ensemble model in the later stages of the competition.

Base model	Ensemble method	Public LB
0.9782_lgb + 0.9787_nn + 0.9736_xgb	average	0.9792
0.9797_lgb + 0.9795_nn	average	0.9804
0.9798_lgb + 0.9795_lgb + 0.9795_nn + 0.9794_nn	weighted: [0.4, 0.2, 0.2, 0.2]	0.9806
0.9811_lgb + 0.9798_lgb + 0.9795_lgb + 0.9799_nn + 0.9795_nn + 0.9794_nn	weighted	0.9813

We also used two methods to create the ensemble, one was simple averaging, the other was weighted averaging.

6. Result

Our public leaderboard journey is shown below (x-axis: date, y-axis: ranking). The highest score being 0.9813, which was submitted four hours before the competition finished.

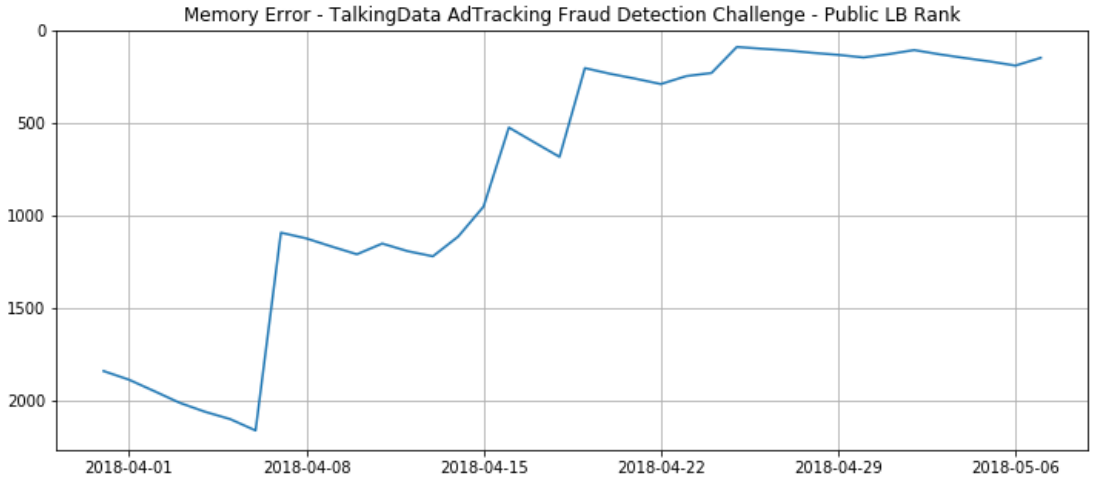

Figure 6-1 public leaderboard journey

After two months fighting, our final score on the private leaderboard is 0.9813159, ranked 133 over 3,967 Teams and achieving a silver medal.

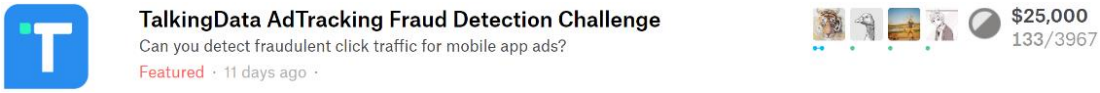

Figure 6-2 rank and medal