Report.md

Machine Learning Engineer Nanodegree

Capstone Project

Camila Barbosa January 27, 2020

I. Definition

Project Overview

Presently, fraud traffic is one of the most pressing matters coupled with the exponentially growing mobile phone prevalence. Data from analytic firm Data Visor shows that fraudulent apps led to at least 15% of the total amount of apps downloaded and installed on smartphones in 2017, resulting in losses of hundreds of million dollars for developers and mobile advertisers.

In this project, I trained and tested a binary classifier capable of predicting whether a given click on an advertisement by a mobile user would result in the advertised app being downloaded or not. The model used the gradient boosting algorithm, and was trained on the data provided by the big data firm TalkingData for their Kaggle competition.

Problem Statement

Talking Data is one of the most prominent big data platform, covering more than 70% of mobile devices' activities across China. They handle billions of ads clicks on mobile per day that are potentially fraudulent. Their current strategy against mass-install factories and fake click producers is to build a portfolio for each IP address and device based on the behavior of the click, flagging those who produce lots of clicks but never end up installing apps.

In their Kaggle competition, Talking Data provided a dataset detailing the clicks registered by their system; the goal is to build a binary classification model that predicts whether an user will download an app after clicking a mobile app or not. This model would help the firm increase their solution's accuracy in identifying fraudsters.

To start understanding this problem, I tried, at first to understand the training data, checking for characteristics such as target concept balance and distribution of features; also tried feature engineering with some aspects that I thought to be important. Then, I implemented a classification model based on the gradient boosting method (discussed in **Analysis > Algorithms and Techniques** below) that is fitted with the training data with all attributes to see how it performs to set a baseline expectation for my final model. At this point, having seen the technical difficulties with the engineered feature and training process, I made certain changes to improve the process, decided to reduce the amount of data used, and performed parameter tuning for the classification model (Methodology > Refinement). At last, I tested the final model design using its performance on the given testing data.

Metrics

Following Kaggle's competition metrics, a model is graded based on the area-under-the-ROC-curve score between the predicted class probability and the observed target, measured on the test data. Since the test data is not labelled, grading is done by uploading the file containing the download probability of each click in the test data to Kaggle.

> ROC (Receiver Operating characteristic) curve is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

- True Positive rate
- False Positive rate

> AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

source: Google

Using the ROC-AUC metric, we can see the closeness between the model's prediction on the testing set and that of the solution, showing how accurate the model is in terms of classification probability. Furthermore, ROC-AUC can tell how good the classifier is at separating the positive and negative. While , ROC-AUC is a quick and intuitive way of assessing the model's performance compared to the solution.

II. Analysis

Data Exploration

The dataset provided by Talking Data on Kaggle competition homepage includes approximately 200 million registered clicks over 4 days, split into training and testing sets. The training set contains more than 180 million rows of data, each has the timestamp of the click, number-encoded IP addresses, device numbered label code, device's operating system code, app code, channel code, whether the click resulted in a download or not, and time of download if applicable. The testing set contains about 18 million clicks with each click associated with an ID and other information excluding the download or not label and download time.

Files:

- train.csv (7537.65 MB): the training data
- test.csv (863.27 MB): the test data
- train_sample.csv (4.08 MB): a number of randomly-selected rows from train.csv

Line counts:

- train.csv: 184,903,890
- test.csv: 18,790,469
- train_sample.csv: 100,000

Labels:

- On both sets:
  - `ip` (integer): IP address from which the click was registered, encoded for privacy
  - `app` (integer): the app whose advertisement was clicked on, encoded for privacy
  - `device` (integer): label of the model of the device on which the click was made
  - `os` (integer): label of device's operating system
  - `channel` (integer): the channel on which the app advertisement was put and clicked
  - `click_time` (yyyy-mm-dd hh-mm-ss): the time when the click was made
- Training set:
  - `attributed_time` (yyyy-mm-dd hh-mm-ss): time of download if a download was made, None otherwise
  - `is_attributed` : whether this click resulted in a download or not; 1 - yes, 0 - no
- Testing set:
  - `click_id` : used for identifying the click for grading purpose

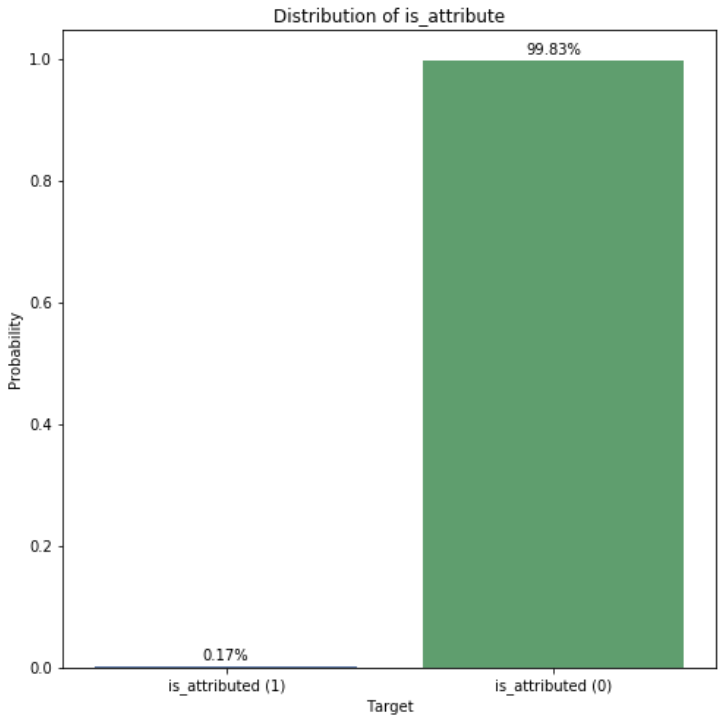
Observations:

- The data is heavily imbalanced with only less than 0.2% positive (clicks that resulted in downloads)!
- The data was collected over the span of 4 days; the training set contains data from November 6, 2017 2pm and the testing set from November 10, 2017 4am to 3pm.
- As data covered a relatively short amount of times (4 days), it is not possible to derive and use certain time-sensitive features from the data such as what day of the week the click was registered, or whether there is a public event happening, which may influence the target concept.
- In each row, only the IP address, device label, and OS are provided. Clearly this is not enough to pinpoint specific user associated with the click, as an IP address may represent many users such as IP of a school, a cafe, or any public place; device code and OS also do not specify individual user as there are many prevalent phone models and OSes such as iPhone on iOS 11.
- The data was thoroughly processed beforehand. There was no missing data in all columns of both the training and testing sets. On both sets, the IP address and app identity were encoded for privacy purpose, and all other features have been labelled in integers to facilitate the training process. This is reasonable as such information can be sensitive, but it limits the amount of feature engineering that can be done. On the training set, `attributed_time` is always present when `is_attributed==1` and is None otherwise.

- While `attributed_time` is only available in the training dataset, it can still serve some interesting analysis, for example predicting the `attributed_time` for the testing dataset.

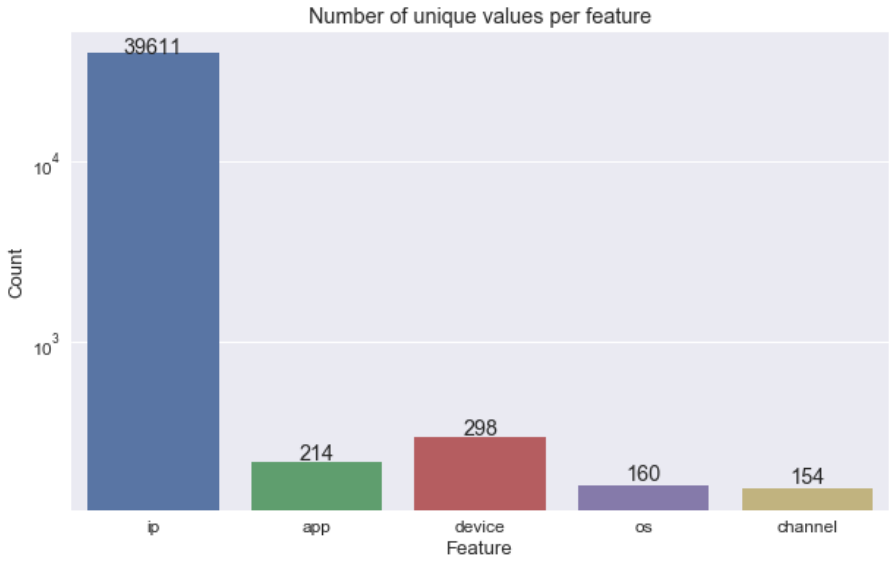
Exploratory Visualization

The plot below shows how skewed the target concept is in the first 10 million rows of the training set:



Clearly, this needs to be taken into consideration later when the classification model is chosen.

For each given feature, how many unique values does it have?



While it makes sense for the number of unique devices and oses to be so low as there are only so many phone models and OSes, and same goes for app and channel, it is quite interesting to see only about 40,000 IP addresses are responsible for 10,000,000 ad clicks in less than a day! That means on average an IP address has about 250 ad clicks in a few hours! Clearly some IP have abnormally too many clicks, possibly click factories fraudsters.

Algorithms and Techniques

Extreme Gradient Boosting (XGBoost) is an implementation of the gradient boosting machines that is highly flexible and versatile while being scalable and fast. XGBoost works with most regression, classification, and ranking problems as well as other objective functions; the framework also gained its popularity in recent years because of its compatibility with most platforms and distributed solutions like Amazon AWS, Apache Hadoop, Spark among others.

In short, XGBoost is a variation of boosting - an ensemble method algorithm that tries to fit the data by using a number of "weak" models, typically decision trees. The idea is that a "weak" classifier which only performs slightly better than random guessing can be improved ("boosted") into a "stronger" one that is arbitrarily more accurate (source: Y. Freund, R. E. Schapire)(source: R. E. Schapire). Building on the weak learners sequentially, at every round each learner aims to reduce the bias of the whole ensemble of learners, thus the weaker learners eventually combined into a powerful model. This idea gave birth to various boosting algorithms such as AdaBoost, Gradient Tree Boosting, etc.

XGBoost is an example of gradient boosting model, which is built in stages just like any other boosting method. In gradient boosting, weak learners are generalized by optimizing an arbitrary loss function using its gradient.

> XGBoost, as a variation of boosting, features a novel tree learning algorithm for handling sparse data; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning.

source: T. Chen, C. Guestrin

There is a number of advantages in using XGBoost over other classification methods:

- **Work with large data:** XGBoost packs many advantageous features to facilitate working with data of enormous size that typically can't fit into the system's memory such as distributed or cloud computing. It is also implemented with automatic handling of missing data (sparse) and allows continuation of training, or batch training which was a tremendous help for me in this project.
- **Built-in regularization:** XGBoost supports several options when it comes to controlling regularization and keeping the model from overfitting, including gamma (minimum loss reduction to split a node further), L1 and L2 regularizations, maximum tree depth, minimum sum of weights of all observations required in a child, etc.
- **Optimization for both speed and performance:** XGBoost provides options to reduce computation time while keeping model accuracy using parallelization with multi-core CPU, cache optimization, and GPU mode that makes use of the graphics unit for tree training.

Benchmark

A benchmark for this problem is a classifying model that randomly guess whether or not a registered click on an advertisement would result in a download of the advertised app. The guessed probability is either 0 or 1. On the testing dataset, the model scored 0.3893 of the ROC-AUC metrics measuring the closeness of the resulted probability graph versus that of the solution.

III. Methodology

Data Preprocessing

As previously mentioned, the given dataset had already been well-prepared and processed, all categorical features had been labelled and set to numbers, sensitive data encoded, and potentially missing data had been either filled or discarded.

Data conversion

Before being feed to the model, the time attributes need to be converted to a format that XGBoost can understand. In its raw form, the attribute `click_time` was given in the format of `yyyy-mm-dd hh-mm-ss`; I have chosen to extract this into 3 new features: `click_day`, `click_hour`, `click_minute`, corresponding to the day, hour, and minute the click was registered, and in integer format. For `click_time`, the year and month information was disregarded simply because the data was recorded in the same month, so these values should be the same for all data points. On the other hand, I did not extract the second out of `click_time` because this value by itself does not how much meaning; in my opinion, humans are precise creatures but not down to the seconds in most cases. Moreover, I later made use of the click second in an engineered feature.

Feature engineering

How many clicks does an user make to a certain app?

Between a normal mobile user who clicks on an advertisement out of personal interest and a fraudster whose aim is to make as many empty clicks as possible, the fraudster would make many more clicks on the same advertisement than the normal user. This is because in a typical case once the user have visited the ad, he/she would be able to make the decision immediately or after just a few more clicks unless he/she is very indecisive! Therefore, a fraudster would make hundreds or even thousands of clicks on the same ad compared to a handful by an user, and this could be a clear distinction for the tree classifiers to pick up.

In order to create this new feature, `avg_app_click_by_ip`, I grouped the data based on the app, and divided the number of clicks by the number of unique IP addresses. Why did I only consider IP addresses and not device, OS, and channel? This is because a common "click farm" where fraudsters mass produce clicks can be set up with various devices and operating system, as can be seen here. In recent years, mobile frauds seem to have evolved to be more sophisticated in order to avoid the countermeasures put up by authorities. They equip themselves with varying phones and tablets models in order to mislead advertisers into thinking that several users are interacting with their advertisements. However, it is not easy for fraudsters to hide or relay their IP addresses because in countries such as China where mobile frauds are prevalent, it is not easy to access services such as VPN or relay servers.

Is a given click a one-off event or the user actually is on a click-spree?

As the goal is to tell clicks by genuine users from those by fraudsters who typically make a lot of clicks but never actually download apps, I think it is important to know how long does it take for the same user to make another ad click. The intuition is that a genuine user generally takes longer before making another click, since he/she would need time to decide whether to download the app or not, while a fraudster who never intends to download the app in the first place would not need to do so and hence takes less time before clicking another ad.

This new feature, `time_to_next_click`, is the amount of time measured in seconds in between clicks made by devices of the same IP address, model, OS, and from the same channel. While this can be a decisive feature differentiating a fraudster and an ordinary user, it is clear that it is not perfectly accurate due to the previously mentioned fact that the given data cannot exactly pinpoint individual users; it is not uncommon to have users with the same device and OS on the same network in school, cafe, or building, and these people may be wrongly taken as the one user.

In the end however, due to the sheer size of the data being too large, Pandas was unable to fully construct this feature on the whole training dataset of nearly 180,000,000 lines, so I decided to construct it on a portion of the data only.

Implementation

The software requirement for the implementation is as followed:

- Python >= 3.7.5
- numpy >= 1.17.4
- pandas >= 0.25.3
- scikit-learn >= 0.22
- xgboost == 0.90

I first attempted to train an XGBoost model without the engineered features, then planned to compare the performance to the same model trained with data with added features. The model's performance is evaluated by its ROC-AUC score on the testing data. The model's parameters were as followed:

- Objective function: logistic binary
- Scale positive weight: 99 (scale up the weight of the positive data points `is_attributed==0` to counter the imbalance of the data)
- Parallel jobs: 3 (to make use of CPU cores)
- Tree method: "exact" On a system with a 4-cored CPU and 16GB of RAM, the dataset itself occupied half of the system memory, therefore XGBoost encountered memory error training on the whole dataset. Moreover, constructing `time_to_next_click` feature was also impossible due to memory shortage. This data proved to be simply impossible to train as it is. At this point I decided to use on only a portion of the data; at 10 million data points, the model scored a point of 0.5445 on the test data, significantly higher than the baseline of 0.3893 by the random-guessing model, but is still a terrible score. There was much to be done to refine this result.

Refinement

Hyperparameter tuning

The refinement process started with hyperparameter tuning. Thanks to XGBoost's versatility that its classifier object `XGBClassifier` is actually compatible with scikit-learn framework, I was able to make use of scikit-learn's `GridSearchCV` object to perform tuning. The search space contains values for the following parameters:

- `max_depth`:
  - Maximum depth of a tree
  - Used to prevent overfitting, as higher depth allows model to learn more specific relations
  - Search values: [5..10]
- `learning_rate`:
  - Control how fast the model tries to converge

    - Search values: [0.05, 0.10, ..., 0.25]
- `gamma`:
    - Minimum loss reduce for each node split
    - Search values: [0.01, 0.02, ..., 0.05]
- `min_child_weight`:
    - Minimum sum of weights of all observations required in a child node
    - Used to avoid overfitting as higher values prevent model to learn relations specific to the dataset
    - Search values: [1..5]
- `max_delta_step`:
    - Control the update step of each tree's weight estimation
    - Important when classes are imbalanced
    - Search values: [10, 12, 14, ..., 20]
- `colsample_bytree`:
    - Control the amount of features to be sampled by each tree
    - Search values: [0.5, 0.6, ..., 1.0]
- `reg_lambda`:
    - L2 regularization
    - Search values: [1000, 2000, 3000]

On fitting the grid with the sample training data (100,000 rows of train.csv randomly sampled), the best estimator with optimized ROC-AUC score is:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1.0,
       colsample_bytree=0.7, gamma=0.03, learning_rate=0.15,
       max_delta_step=20, max_depth=6, min_child_weight=4, missing=None,
       n_estimators=100, n_jobs=4, nthread=None,
       objective='binary:logistic', random_state=42, reg_alpha=0,
       reg_lambda=1000, scale_pos_weight=99, seed=None, silent=True,
       subsample=1.0, tree_method='hist')
Best ROC-AUC: 0.9733
```

Incremental training

After doing some research, I found out how to train an XGBoost model incrementally. I divided the training data into non-overlapping chunks of pre-determined size; at each iteration, I loaded one chunk into the memory and train the model with that chunk. Once this is done, the memory is cleared and ready to load the next chunk of data. This process was repeated for all data chunks.

Incremental training helped avoiding the need to have the whole training data loaded on the memory, allowing the model to have more computational resources and also made the training possible since the model can avoid memory error with smaller data. However, this also means the model is continuously trained with different data each time; without the whole data being available at all time, the tree construction is less ideal since it is impossible to choose upon optimal splits, as discussed here.

IV. Results

Model Evaluation and Validation

Even though I managed to bypass the system memory limit on XGBoost by training the model incrementally, it still remains that Pandas is unable to produce the engineered feature `time_to_next_click` on the whole dataset. Therefore, I decided to again train the model on just a subset of the data. After some trials and errors, I found that 60,000,000 was the limit to which Pandas managed to generate `time_to_next_click` without memory error.

The final model design was built with the tuned hyperparameters as mentioned earlier, using GPU mode (`tree_method`: `gpu_hist`), trained on 60,000,000 data points with `avg_app_click_by_ip` and `time_to_next_click` features along with extracted time features. Score obtained on the testing set: 0.9700

To see the effect of having a smaller dataset but one extra feature, I have also trained another model with the same hyperparameters and on the whole training dataset, but without the `time_to_next_click` feature. Score obtained on the testing set: 0.9575

In conclusion, having `time_to_next_click` improved the performance of the dataset (0.0125) over having just the `avg_app_click_by_ip` feature even when trained on just about 30% of the data. It is certain that when trained with the whole dataset, this feature would increase the model's accuracy even more.

Justification

The final model design with tuned hyperparameters trained on a third of the data with extra features scored 0.9700 on the testing data, dwarfing the score of the random guessing model at 0.3893. This means that the final model far surpasses the random guessing model in terms of learning the target concept, as a perfect predictor would give a score of 1.0.

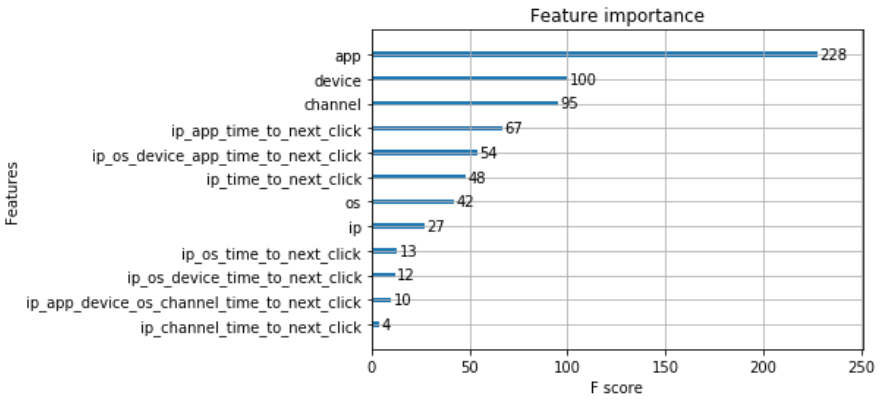
In the boundaries of the competition, I can say these results are encouraging and show that the approach taken is the right direction, although there is so much to improve upon.

V. Conclusion

Free-Form Visualization

To get a glimpse, I constructed more `time_to_next_click` features, this time using different aggregations of given features [`ip` , `os` , `device` , `channel` , `app`] to identify individual "clicking session" or "clicking spree". As previously mentioned, the intuition is that a fraudster would have very short amount of time in between advertisement clicks compared to a normal mobile user. I also paired `ip` with `app` and `channel` to construct time to next click features, because as mentioned earlier some fraudsters may make use of "click farm" with various different devices to generate fake clicks, and in those cases the IP address is the only way to identify them.

Again due to hardware limitation, I was only able to construct the new `time_to_next_click` features with 10,000,000 rows of the training data. I then fitted the data with an XGBoost classifier and used the `plot_importance` feature of XGBoost to see how much each feature has an impact on predicting the outcome of a click. The graph of feature importance is given below:



As can be seen, among the 7 newly constructed `time_to_next_click` features, groupings of [`ip` , `app`] and [`ip` , `os` , `device` , `app`] were the most impactful. This aligned with my above assertion that for some cases, [`ip` , `app`] is the best way to identify a fraudster's "clicking session"; [`ip` , `os` , `device` , `app`] on the other hand shows that a fake click can still be identified with more specific user information aside from just their IP address; perhaps these are cases where fraudsters use mostly similar devices because they were cheap?

Reflection

End-to-end problem solution

This project turned out to be one that emphasizes on handling big data and utilizing computational resources; the given data was clean and thoroughly processed, the target concept is clearly defined, the features gave lots of room for engineering and research. All that was left was working around the sheer gigantic size of the data and optimizing all computing power at hand to fit it in, so much so that the data itself became the main problem instead of the target concept throughout the project.

Therefore, my problem solution not just includes data processing and training but also hardware optimization:

- Establish basic statistics and understanding of the dataset such as imbalance, data type, etc.
  - Data cleaning was not needed as the given data was thoroughly processed by TalkingData.
  - Extracted clicking time information (day, hour, minute) into a format usable by trainer.
- Devise new features based on the given features:

    - Average click on the same advertisement by the same IP address
    - How much time does it take until the same IP address clicks again?
        - This feature proved to be impossible to produce on the whole training data due to hardware limitations
- Train and test model's performance:
    - Model was unable to fit the whole dataset, again due to hardware limitations
    - As `time_to_next_click` was also impossible to produce on the whole data, used only 10 million data points instead
    - Score: 0.5445
- Improve:
    - Fine-tune model's parameters with `GridSearchCV`
    - Implemented incremental training so that model can make use of the whole dataset
    - Setup model to train on GPU, computation time improved by 75%
- Train and test again:
    - `time_to_next_click` was still impossible to produce, so I decided to use 60 million rows of data this time; it is small enough so that Pandas can generate the feature. Score: 0.9700
    - To compare, I trained another model with the same hyperparameters, but with all the data and without the `time_to_next_click` feature. Score: 0.9575

Challenges

I encountered many difficulties at different steps of the projects.

At first, I couldn't generate one of the new features I needed for the whole dataset due to not having enough system memory; in the end I decided to settle with using just a portion of the given data instead.

Second, again due to low memory problem, the classification model I used was not able to take in the whole dataset to train. This costed me a couple of days to look into until I found the way to implement incremental training for XGBoost classifier.

Thirdly, after managing to get the training in place, I realized still it took a relatively long time for the model to train even with just a fraction of data. While this is acceptable for the training process, it made the hyperparameter tuning step impossible, as I used `Grid Search` to perform brute force method to find the optimal parameters with thousands of combinations in the search space; the long training time for each instance meant it would take days to run! Luckily, I found that XGBoost supported training using GPU which greatly decreased training time, so I decided to rebuild XGBoost with GPU mode. Unfortunately, the rebuild process turned out to be the most frustrating part of the project, as I was met with various software compatibility issues on Windows environment; Visual Studio refused to work with XGBoost build when specified with certain versions of CUDA (computing driver for GPU) included. It's only after 3 days with dozens of trials and errors with different software versions that I finally got XGBoost with GPU mode up and running; training time immediately reduced by 4 times!

In conclusion, while there were many obstacles along the way, I think it was largely due to my inexperience. In the end, I felt appreciative towards those problems that I met, they are all precious practical experiences that only come by working on a real-world project like this one. They also made me realize that I still have so much to learn.

Improvement

There is so much that can be improved upon for this project, for example the same model design training on the whole dataset with all generated features present would no doubt yield a much higher performance.

Overcome memory limits for feature engineering

I was unable to generate `time_to_next_click` feature on the whole dataset due to hardware limitations. After doing some research, I found that there were ways to workaround this such as using distributed system and paging memory.

As I understand, distributed systems such as AWS clusters help overcome the memory constraint by distributing the data into its many node, each holds a chunk of the data that can work as a whole. These systems utilizes parallelization for speed and computing power, and definitely would be a big help in problems that deal with big data like this one.

Another potential workaround that I found was using in-storage memory. Apparently, it was possible to construct certain file formats that resemble how data is stored on RAM, but instead these files can be stored on the system's hard drive which typically has more generous size than RAM. This method would the training data to be accessible by the system, without the need to have a big enough memory. This method may also obsolete the incremental step of training process, since the whole data is readily available at all time for the model to access.

More time features worth exploring

My intuition is to follow the clicking patterns of individual users, and the time between the clicks has proven to be an important feature. To exploit this further, we can also generate more time-between-click features such as how much time does is it between this click and the next 2 clicks, or how much time between this click and the previous click. The reason is that for normal users, there may be cases where the user makes more than 1 click in a short amount of time due to accident or because the user realizes something after finishes reading the advertisement, and goes back to it. These cases may be falsely identified as fake clicks, so to be able to look 2 or more clicks ahead would be helpful since it would be more rare for normal users to generate repeated clicks in any case.

More app-based features

Aside from time features, app-based and channel-based features may hold important information, since intuitively a fraudster often target a single app or advertisement to generate fake clicks, and often operates exclusively in certain app distributing channels since some channels do not have strict regulation against mobile frauds.

Similar to the time features, `app` and `channel` can be associated with individual users or "clicking sessions" by aggregating them with different groups of given identity features: [`ip` , `device` , `os`]. Upon successfully generating these new features, I would gauge their importance similar to what I did in the previous section and include the 2 most important ones for the final model training.