Supervised Machine Learning for Click Fraud Detection

MLND Capstone Project
Shan Dou | Jul 21, 2018

I. Definition

1.1 Project Overview

Click fraud has been a "billion dollar" problem facing ***pay-per-click (PPC)*** advertisers. PPC is by far the most widely used compensation model for digital advertising (e.g., both Google AdWords and Facebook Ads are PPC platforms). As the name "pay-per-click" implies, PPC advertisers pay for every click on their ads. This compensation mechanism has clear benefits to advertisers because they don't need to pay for ads that don't generate clicks, but this same mechanism also is heavily abused by fraudsters through click fraud.

Click fraud is the ill-intentioned clicking of PPC ads by fraudsters to waste and/or to mislead advertisers' ad spending. According to a recent report by CNBC[1], click fraud cost advertisers $12.5 billion in 2016 and wasted nearly 20% of total ad spending. But the monetary loss is not limited to advertisers. Click fraud also hurts revenue streams for ad platforms because it degrades the overall appeal of digital advertising. As an example, the consumer giant Procter&Gamble slashed its digital ad spending by more than $200 million in 2017[2]. For the $200 billion market of digital advertising[3], the stakes for preventing click fraud are high, and this is where data mining and machine learning could come to help.

1.2 Problem Statement

The goal of the project is to build a click-fraud detector that serves ***ads platforms for mobile apps***. Quantitatively defining fraudulent clicks is a challenge in its own right. Existing studies have shown that fraud labels based on IP and device blacklists often are problematic as they are biased by the procedures used to generate those lists in the first place (e.g., Oentaryo et al., 2014[4]). As a work-around, we employ the following simplication: **Clicks followed by app downloads are legitimate, whereas clicks that don't lead to downloads are fraudulent.** With this simplification in place, we can now frame the problem as a ***supervised learning*** problem, and more specifically, we are to construct a ***binary classifier*** for predicting whether or not clicks are followed by app downloads.

Data for this project were click-traffic records provided by TalkingData, which is China's largest independent big data service platform. The raw data are accessible through a Kaggle machine learning competition titled "TalkingData AdTracking Fraud Detection Challenge"[5]. The full dataset consists of 200 million click records and takes 7GB of memory. Such a big data size is not ideal for exploratory data analysis or for evaluating performance of machine learning models. In this capstone project, we focus solely on the effectiveness of data processing and machine learning pipeline, and to keep the operations lightweight, only **0.1%** of the click records are randomly sampled and used throughout this report (downsampling was implemented in `preprocessing.csv_randomized`).

1.3 Metrics

We use the ROC AUC score as the concise and quantitative metric for evaluating the performance of binary classifliers. A classifer no better than random guesses yields a score of ~0.5, and a perfect classifier has a score of 1.

The ROC AUC score meausres the **a**rea **u**nder of the reciever operating characteristics (ROC) **c**urve. The ROC curve is a plot of true positive rate (recall) versus false positive rate (1 - specificity), where the true positive rate ($TPR = \frac{TP}{TP+FN}$) measures the fraction of the positive instances that are correctly detected by the classfier, and the false positive rate measures the fraction of the negative instances that are incorrectly classified as positive ($FPR = \frac{FP}{FP+TN}$). In **Figure 1** below, the dashed line is the ROC curve of random guess, and the ROC curve of a good classifier should stay as far away from the dashed line as possible. In ideal cases, it should almost be touching the upper left corner of the graph.

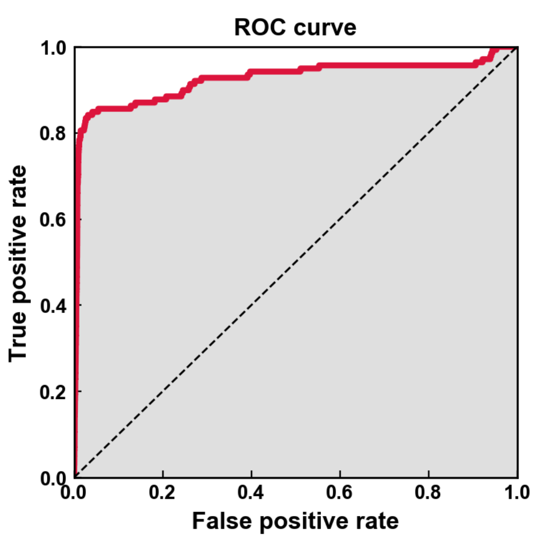

Figure 1: Example of receiver operating characteristic (ROC) curve. Shaded area denotes the area under the ROC curve (AUC).

Why use ROC AUC score as the metric? The ROC curve is a useful graphical tool that captures the tradeoff between recall and specificity: That is, the more `1`s we can capture, the more likely we also are to have missclassified `0`s as `1`s. An ideal classifier should help us capture as many `1`s as possible while misclassifying the least amount of `0`s as `1`s[6]. From this perspective, when an ROC curve is almost touching the upper left corner of the graph, this very corner represents the desired situation of haivng high true positive rate and low false positive rate. The AUC score is built on top of this concept. It is the area under the ROC curve that quantitatively describes the curve's shape. The closer AUC is to 1, the closer the ROC curve is to the upper left corner of the graph, and relatedly, the more effective the classifier is.

In addition to AUC, other ancillary metrics are used to assist the performance evaluations:

- Visualizations of ROC curves: Plots as shown in **Figure 1** are used to evaluate model performances. The closer a ROC curve can approach the upper left corner of the graph, the better performed the corresponding model is.

- Visualizations of precision-recall curves: An example of precision-recall curve is shown in **Figure 2**, which is precision values plotted against the corresponding recall values for each of the probability threshold use to separate postive and negative classes.

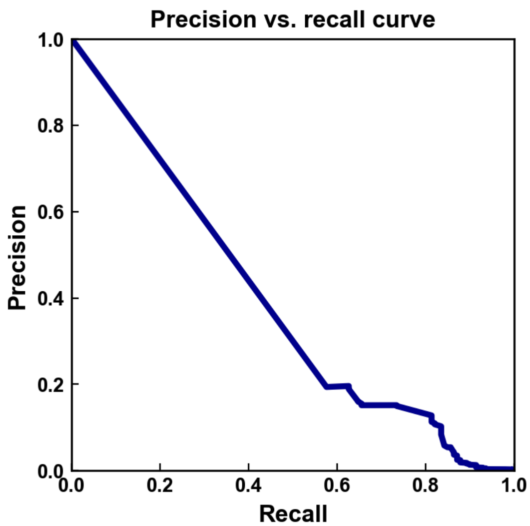

Figure 2: Example of precision-recall curve.

- Precision measures among all the positives identified by the classifier, how many of them are try positives:

$$Precision = \frac{TP}{TP + FP}$$

- Recall measures among all the positives we have in the data, how many of them are correctly identified by the classifier:

$$Recall = \frac{TP}{TP + FN}$$

Precisions and recalls must be examined simultaneously, as tradeoffs always exist between the two. An ideal classifier should have both high precisions and high recalls, which corresponds to a precision-recall curve that approaches the upper right corner of the graph. Having only high precision could result in a classifier that frequently misidentifies positive cases as being negative (i.e., high false negative), despite being able to avoid labeling negative cases as positive. Similarly, having only high recall could result in a classifier that is unable to correctly identify most of the negative cases. Even though ROC curve and precision-recall curve appear to be similar to each other, the latter has a better chance of revealing high false positives and is particularly helpful for cases where false positives should be minimized[7].

In the case of predicting if app download will occur following a click, false positives are not very critical but are worth monitoring. Therefore, we also include precision-recall curves to assist our performance evaluations.

- Visualizations of confusion matrixes: Confusion matrix shows the number of times class 1 are classifed as class 2. **Figure 3** is an example of the confusion matrix obtained from the light gradient boosting algorithm.

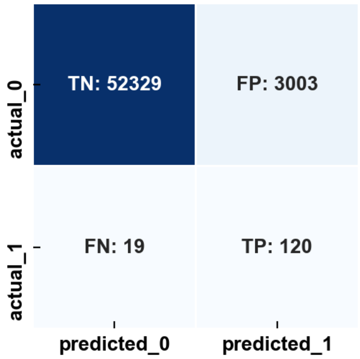

Figure 3: Example of confusion matrix (obtained from light gradient boosting algorithm tested in this project).

We can see that the diagonal elements of confusion matrix correspond to correct classifications of both positive and negative cases. The off-diagonal elements show the amount of false positives and false negatives, respectively. Confusion matrix is less concise than the other performance metrics described above, but it provides a straightforward way of visualizing model performances in the presence of class imbalances.

II. Analysis

2.1 Data Exploration

(1) Subsample 0.1% of raw data with `gshuf`

As mentioned earlier, the raw dataset consists of close to 200 million click records and thus is too big for exploratory data analysis and machine learning experimentations. A random subsample of 184,903 records (0.1% of the full data size) was generated with Linux shell command `gshuf` and used as the training and testing data throughout this report.

(2) Raw features and target values for classification

Seven (out of eight) of the orignal fields that are shared by both raw training and testing data are preserved for furthur processing and analysis. Among these seven fields, the field `is_attributed` denotes whethor or not an app download has occurred and it is the target array of the supervised learning. The rest of the six fields--`ip`, `app`, `device`, `os`, `channel`, and `click_time`--are the base ingredients for feature engineering.

(3) Categorical features are predominant and highly cardinal

The six raw features are all categorical. **Table 1** shows the amount and proportions of the unique values for each of these categorical features. We can see that they are so highly cardinal (the amount of unique values is at least 140) that commonly used one-hot-encoding treatment is inapplicable. More advanced feature engineering techniques are necessary to transform these raw features into more usable forms.

Table 1: High cardinality of raw features

	ip	app	device	os	channel	click_time
n_unique	46005.000000	186.000000	148.000000	143.000000	164.000000	125421.0000
n_unique (%)	24.880613	0.100593	0.080042	0.077338	0.088695	67.8307

(4) Severe class imbalance

Given that only **~0.24%** of the target values are positive, steps such as ***class-weights balancing*** and ***oversampling minority*** are necessary when working with such severely imbalanced classes.

2.2 Algorithms and Techniques

Because the dimension of the feature space is small (even after the feature engineering steps described in later sections), deep learning algorithms are not considered and we focus primarily on **ensemble algrithms** that use decision tree classifiers as their base learners.

What is ensemble learning?
Ensemble learning is a technique for aggregating predictions obtained from a group of base predictors. And these base predictors could be trained *in parallel* with each predictor working on a different random subset of the training data, or they could be trained *sequentially* with each added learner working on improving the predictions of its predecessor.

Bagging, pasting, boosting, and stacking
For the parallel ensemble, when the aggregated predictions are computed with a simple average across the predictions of all individual learners, we call it a **bagging** (when training data sampling is performed with replacement) or **pasting** (when training data sampling is performed without replacement) ensemble. Random forest is an example of bagging ensemble. It makes predictions by averaging the predictions of a group of decision trees that each works on a different random subset of the traing data (sampled with replacement).

Boosting ensemble refers to the technique of training a group of predictors sequentially, with each of the added predictor working on improving the predictions of its predecessor. Adaptive boosting and gradient boosting are the two major types of boosting. For adaptive boosting, the predictions of each predecessor are used to assign weights to the training data for its successor, and the prediction aggregation is achieved by taking a weighted average over the predictions of all predictors (The better performed predictor earns a higher weight in the weighted average). For gradient boosting, instead of using predecessors to tweak the weights of the training data and the predictors, each predictor is added to the sequence to fit the residual error of its predecessor.

Stacking refers to the approach used for aggregating the predictions. Instead of taking a simple average like in bagging and adaptive boosting, stacking ensemble trains a meta learner (sometimes called a *blender*) to perform this aggregation. In this way, stacking reduces the subjectivity involved in the aggregation step and allows great flexibility in the structure of the ensemble. For example, one can use a multilayer stacking in which the first few meta-learner layers could be several different blenders and the last meta-learner layer could be a single blender that performs the final aggregation.

Ensemble algorithms used in this project

Table 2: List of ensemble algorithms used in this project

Algorithm	Ensemble type	Implementation
Random forest	Bagging	`sklearn.ensemble.RandomFroestClassifier`
Stacking	Stacking	`mlens.ensemble.SuperLearner`
Extreme gradient boosting	Gradient boosting	`xgboost.XGBClassifier`
Light gradient boosting	Gradient boosting	`lightgbm.LGBMClassifier`

2.3 Benchmark

We use logistic regression as the benchmark baseline in this project. Similar as linear regression, logistic regression first computes a weighted linear sum of the input features and then feeds it to a sigmoid function (σ) to obtain a probability value $\hat{p}$ that varied between 0 and 1[8]:

$$\hat{p} = \sigma(\theta^T \cdot \mathbf{x})$$
$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

As a linear classifier, logistic regression is easy to implement and performs very well on classes that are linearly separable. Although logistic regression is unlikely to perform well when class boundaries are nonlinear, it is a good baseline for evaluating other more complicated models.

III. Methodology

3.1 Data Preprocessing

Because the raw data consist entirely of categorical features and are highly cardinal, feature engineering is necessary and crucial. In this project, a mapping between categorical labels and numerical values is obtained from training data first, and then this same mapping should be applied to both training and testing data. We use the following transforms to transform the raw data into a usable form for machine learning:

1. **Impute rare categorical labels**[9]: Rare labels are only present among a small percentage of the observations, and often times they are only present in either training or testing data but not both. If left untreated, rare labels could cause noticeable overfitting. In this project, all categorical labels that are only present among fewer than 0.05% of the training data are replaced with a uniform categorical label (we use the large number 1×10^{10} as the replacement) and then the imputed data are fed into subsequent feature engineering steps.

2. **Replace categorical labels with counts**[10]: After rare label imputation, features named as `count_*` are generated by replacing raw categorical labels with their respective counts in the training data.

3. **Replace categorical labels with the target mean**[11]: Another set of features named as `risk_*` are generated by replacing raw categorical labels with their corresponding target mean (also known as the "risk factor"). This is an example of *targeted-guided feature encoding* that creates a monotonic relationship between the categorical labels and the taget values.

After preprocessing, we have **12 features** to be used for machine learning:

- Counts: `count_ip`, `count_app`, `count_device`, `count_os`, `count_channel`, `count_click_hour`
- Risk factors: `risk_ip`, `risk_app`, `risk_device`, `risk_os`, `risk_channel`, `risk_click_hour`

Data inspection after feature engineering

Figure 4 shows correlation matrix of post-engineering features. If we use an absolute value of `0.7` as a threshold for defining "noticeable linear correlations", the following feature pairs are correlated:

- `risk_device` and `risk_app`: $r_{Pearson} = 0.7$
- `risk_channel` and `risk_app`: $r_{Pearson} = 0.7$
- `risk_os` and `risk_device`: $r_{Pearson} = 0.8$

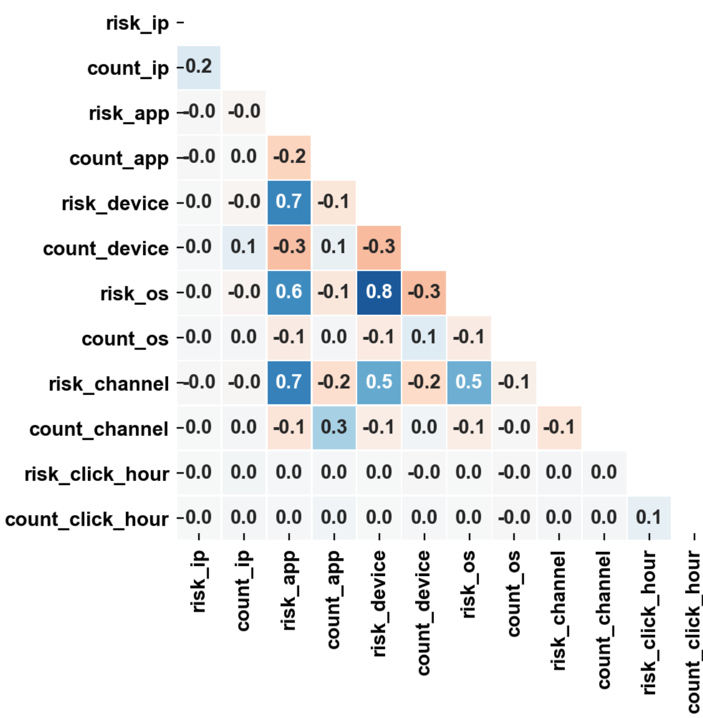

Figure 4: Correlation cofficients of the 12 engineered features

The presence of these correlations makes intuitive sense, as operating systems and devices are likely to be correlated, and the types of apps are expected to show correlations with devices and channels of the app platforms. This also indicates reduced information content in the risk-factor features. The count features, on the other hand, don't appear to show strong correlations with any of the other features. The lack of correlations is less intuitive, which could imply weaker signal strengths in these count features.

3.2 Implementation

Three Python modules are set up for preprocessing (`preprocessing.py`), modeling (`modeling.py`), and miscellaneous tasks such as visualization and generating result summary tables (`utils.py`).

1. The `preprocessing` module consists of seven functions:

- `csv_randomized_downsamp`: Performs randomized subsampling on raw training data. Although this operation could be carried out with `pandas`, it is done by calling shell command `gshuf` for its faster speed.

- `csv_list_fields`: Helper function for listing all fields in the raw csv files. It generates a list of strings that are later used to exclude data fields that only appear in traning data but not in testing data.

- `mapper_label2count`: Generates mapping dictonary that maps raw categorical labels into corresponding counts. The mapping is always generated with training data only and then applied to both training and testing data. The output dictionary is then used in `pd.DataFrame.map()` to convert raw categorical features into numbers.

- `mapper_label2riskfactor`: Similar as `mapper_label2count`, but performs mapping between raw categorical labels and the corresponding target means (also known as rick factor).

- `df_rarelabel_imputer`: Performs rare label imputation. It uses an unusual but uniform value (1×10^{10}) to replace feature values that only appear in less than 0.05% of the observations. This operation, which is always performed before the label-to-number encoding, allows all the rare labels to be grouped into a single category and then being encoded either with counts or with risk factors. Similar as the encoding operation, the imputer mapping is always generated with training data and then applied to both traing and testing data--despite that the same label could be rare in training data but not rare in testing data.

- `df_label2num_endoding`: A one-step wrapper that performs the label-to-number encoding and drops the original categorical features from the data frame. It calls both `mapper_label2count` and `mapper_label2riskfactor` and is used after rare label imputation.

- `df_to_Xy`: Extract features matrix `X` and target array `y` from training and testing data frames.

2. The `modeling` module consists of one decorator, two functions, and one class:

- `timer`: A decorator that displays training time of the machine learning models.
- `pipeline`: A wrapper function based on `imblearn.pipeline.make_pipeline()`. This is an end-to-end pipeline that performs feature scaling (`sklearn.preprocessing.StandardScaler()`), randomized oversampling of the minority class (`imblearn.over_sampling.RandomOverSampler()`), and machine learning (`estimator` that with either user specified hyperparameters or hyperparameters returned by gridsearch or randomized search).
- `gridsearch`: A wrapper function for hyperparameter tuning. It takes in `pipeline` object and decide whether to perform grid search (when hyperparameter grid is small) or randomized search (when hyperparameter grid is large).
- `Classifier`: A class that is the work horse of the machine learning workflow. It consists of `assess`, `fit`, `predict`, and `predict_proba` methods. The `assess` method is used for model evaluation and selection. It performs a 2×5 nested cross validation, with an outer loop of 2-fold cross validation for hyperparameter selection and inner loop of 5-fold cross validation for evaluating model performances. The `fit` method trains models with hyperparameters that are either specified by users or are determined via grid or randomized hyperparameter search. The `predict` method uses the trained models to make predictions

and outputs class labels of 0s and 1s, and the `predict_proba` method does similar operations as the `predict` method but outputs probability values instead.

[Note] The `assess` method exists and functions, but it is NOT used to produce results for this report because it is too time consuming for the datasize at hand.

3.3 Refinement

Overfitting occurs when a machine learning models performs well on training data but does not generalize well on testing data, and it is a sign that the model is so overly complex that it becomes heavily "distracted" by noise in the training data. Overfitting is a common problem facing machine learning and must be mitigated.

Regularization is the mathematical operation used to constrain model complexity, which in turn reduces overfitting. The use and choice of regularization is crucial for this project. Here we use both logistic regression and random forest as examples to illustrate its effects. In particular, we compare the AUC scores obtained from the training and the testing data. For models that are overfitting, we expect to see a good training score but a noticeably worse testing score.

(1) Effects of regularization on logistic regression

For logistic regression, we examine the effects of L_2 regularization (also known as "Tikhonov regularization") that adds a term $\frac{1}{2C}||\theta^2||$ to the cost function, where C is the inverse of regularization strength (smaller C correponds to a stronger regularization), and θ denotes the weights of the model. With this term added to the cost function, the model is forced to not only fit the training data, but also keep the weights of the model as small as possible. From **Figure 5** we can see that the best C value is around 1×10^{-6}.

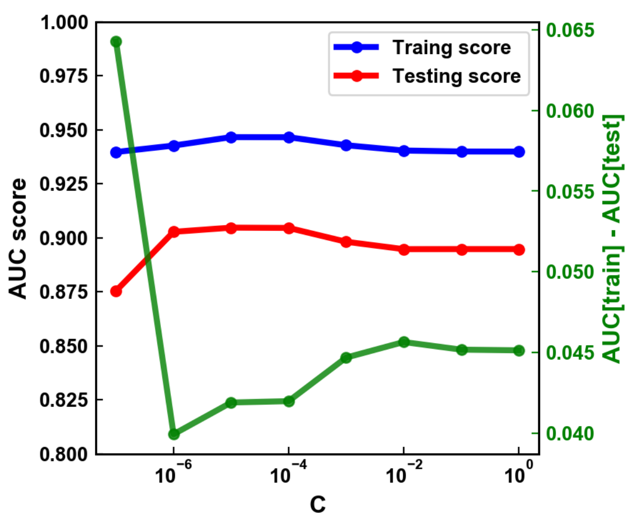

Figure 5: Effects of L_2 normalization on performance of logistic regression.

(2) Effects of regularization on random forest

For random forest, `max_depth` is the most important regularization parameter that contrains the depth of the decision trees. A few other hyperparameters can also for regularization. Examples include `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `max_leaf_node` (maximum number of leaf nodes), and `max_features` (maximum number of features that are evaluated for a split at each node). Increasing `min_*` or decreasing `max_*` will increase the amount of regularization applied to the model.

Here we only examine the effects of `max_depth` given its critical role. The number of decision trees used to form the forest is fixed at `50` to keep training time short. From **Figure 6** we can see that the `max_depth` of 1 yields the best performance. Because only one tree split is allowed, such decision trees are also called "decision stumps"[12].

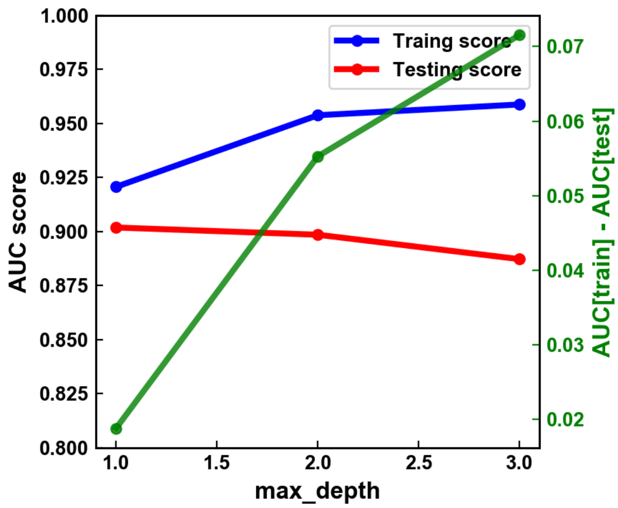

Figure 6: Effects of tree depth regularization on performance of random forest.

Tree learners are used in all the ensemble models tested for this project, and similar as for random forest, `max_depth = 1` provides the best performance and thus is used in all tree learners.

IV. Results

4.1 Model Evaluation and Validation

We use two training datasets to evaluate the performance and robustness of the models. The first dataset is 0.1% subsample of the raw data, and the second dataset is 0.2% subsample of the raw data. **Table 3** and **Table 4** list all the scores from the two sets of training. Although ideally a learning curve should be obtained to examine how model performance varies as a function of training data size, but the large size of the raw data makes it too time consuming for a more thorough robustness test. For this project, we only validate the robustness of the model by comparing model performances of the above-mentioned two data sizes.

From **Table 3** and **Table 4** we can see that the performance of all models deteriorate once the data size doubles. This indicates the likely presence of overfitting and that a more complicated model structure could be required to capture the signal content of the data.

Because the testing scores of light gradient boosting model are the highest for both datasets, and its run time is only slightly longer that logistic regression, we choose it as the preferred model for this project.

Table 3: Model comparison with subsampling proportion of 0.1%

Model	Training time (s)	Training score	Testing score
Logistic regression	1.9	0.943	0.903

Random forest	9.8	0.921	0.902
Stack ensemble	3.3	0.902	0.904
Extreme gradient boosting	12.8	0.969	0.905
Light gradient boosting	4.6	0.969	0.905

Table 4: Model comparison with subsampling proportion of 0.2%

Model	Training time (s)	Training score	Testing score
Logistic regression	4.5	0.945	0.878
Random forest	22.6	0.928	0.886
Stack ensemble	6.8	0.916	0.887
Extreme gradient boosting	27.9	0.964	0.890
Light gradient boosting	8.5	0.964	0.890

4.2 Justification

To double check the performance of the selected model--light gradient boosting, we visualize all of its performance metrics against other models that we have tested so far. These metrics are ROC curves, precision-recall curves, and confusion matrixes.

(1) ROC curves

Figure 7 shows the ROC curves of all models obtained with the two data sizes. Regardless of data sizes and models, the ROC curves all show desirable shapes of approaching the upper left corner. Noticeable differences are present between the two data sizes. With the smaller data size, the shapes of the ROC curves are close to each other; with the bigger data size, however, the better performance of the light gradient boosting model becomes clearer.

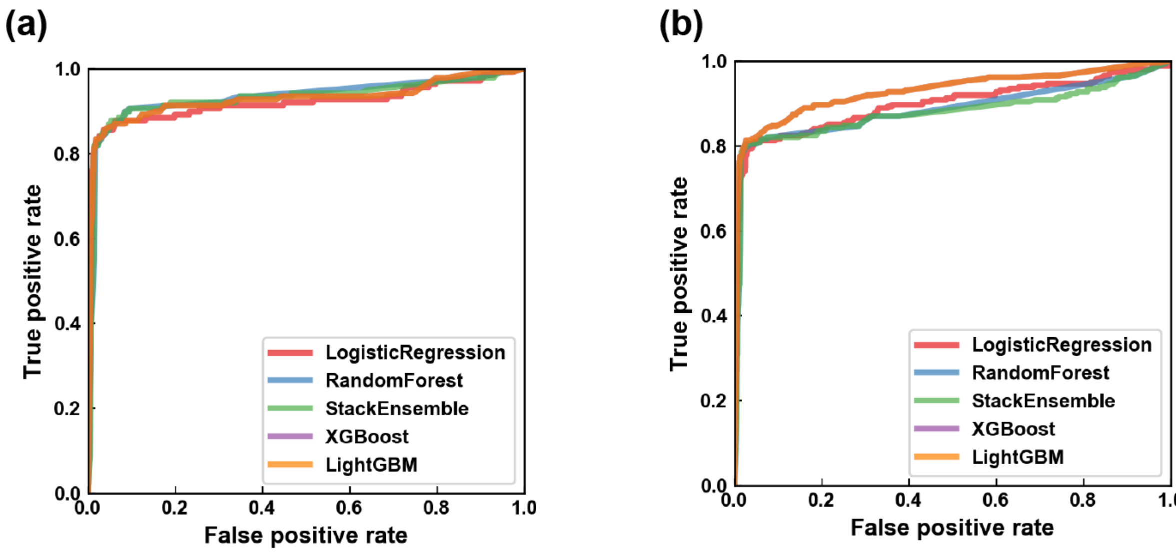

Figure 7: Comparisons of ROC (receiver operating characteristics) curves. (a) 0.1% subsample and (b) 0.2% subsample.

(2) Precision-recall curves

Different from ROC curves, the shapes of the precision-recall curves are not ideal (**Figure 8**). Instead of "bulging" toward the upper right corner of the graph, most of the curves bend at the lower left corner and then lean close to the bottom. Such shapes indicate that models only have high precisions when recalls are very low, and high recalls are mostly associted with low precisions. Although high recalls are desirable for this fraud detection problem, it comes with the costs of high false positives that is likely to be the consequence of the severe class imbalance. Once again, the perferred model, light gradient boosing, has the best precison-recall shape, which confirms the validity of our choice.

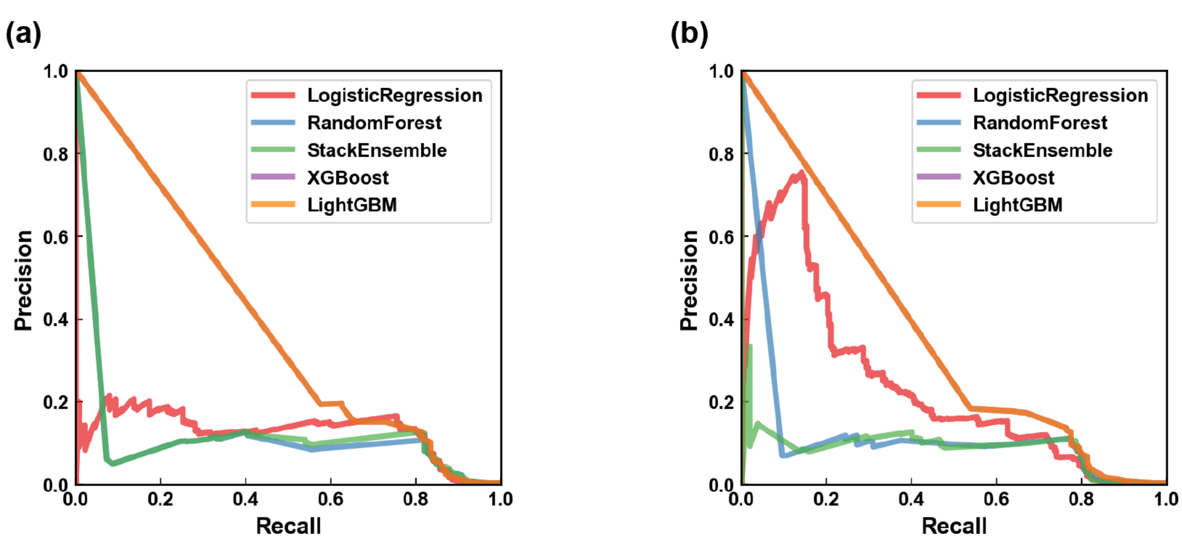

Figure 8: Comparisons of precision-recall curves. (a) 0.1% subsample and (b) 0.2% subsample.

(3) Confusion matrixes

The above-mentioned high false positive rate also stands out in the confusion matrixes (**Figure 9**). Although the amount of false negatives is small, the true positives are so scarce that the high amount of false positives is not surprising.

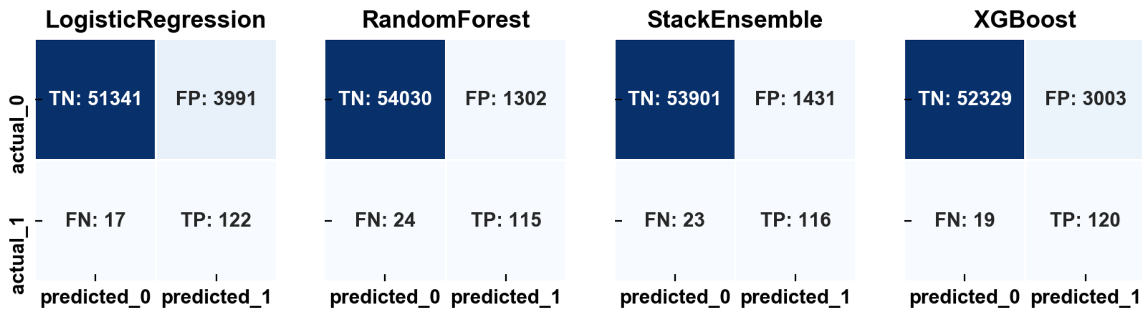

Figure 9: Comparisons of confusion matrixes (results from the 0.1% subsample).

V. Conclusion

5.1 Free-form visualization

Finally, we examine feature importance values returned by all the models tested in this project (for logistic regression, the absolute values of the model coefficients are used as feature importances). **Figure 10** shows that features `risk_channel`, `risk_app`, and `risk_os` are deemed important by all models, and none of the `count_*` features are given high importance by any of the boosting ensembles (i.e., `XGBoost` and `LightGBM`). Such ranking highlights the better effectiveness of the target-guided label encoding than using counts.

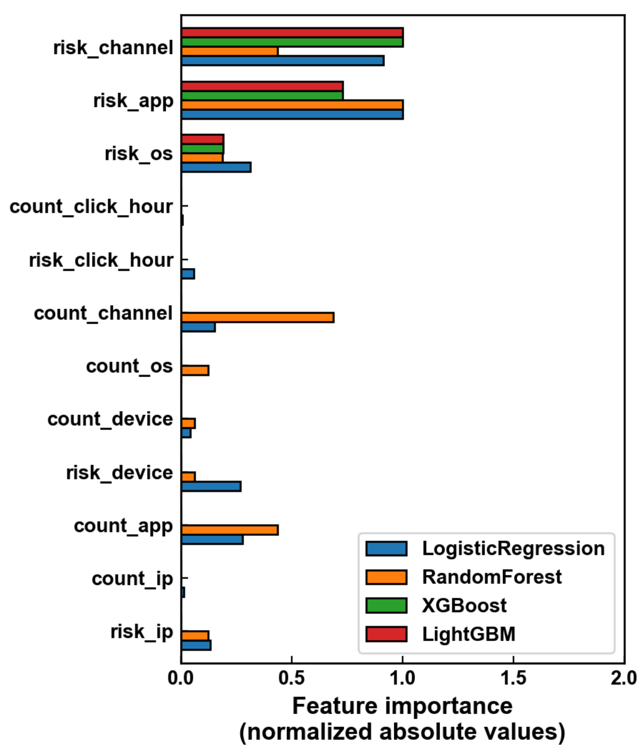

Figure 10: Feature importance appraisals.

Contrasting againt the low importances of `ip`, `click_hour`, and `device` features, the high importances of `rick_channel` and `risk_app` imply that the content and genres of the apps largely determine whether app downloads are likely to occur. Because operating system of the app is also correlated with the app contents (e.g., some apps may only be available for either iOS or Android, and/or only supports certain versions of the OS), the high importance of `risk_os` is to be expected as well.

5.2 Reflection

In this project, we focus mostly on finding effective processing and machine learning pipelines. The scalability of the chosen methods to the entire dataset is beyond the scope of this experiment.

(1) Summary of the end-to-end solution

To keep the data exploration, processing, and model evaluation steps lightweight, we only work with 0.1% of the raw records. With this small subset, we carry out feature engineering to impute categorical labels that are extremently rare (appear in fewer than 0.05% of the training data) and to convert categorical labels into either counts or target means. Note that all the feature transform rules are first obtained from training data and then the same rules are applied to testing data. The transformed features are then fed into machine learning models, and between feature scaling and model training, a random oversampling step is used to combat the severe class imbalance. For machine learning, we focus mainly on Logistic regression and tree-based ensemble models for their fast speeds and interpretability (via feature importance). After examining the effects of hyperparameters, and the robustness of the models against larger data sizes (double the data size), we select light gradient boosting as the most suitable algorithm for this fraud detection problem. **Figure 11** summaries the end-to-end solution we have presented in this report.

Figure 11: Summary of the end-to-end solution pipeline. Most challenging steps are highlighted with color.

(2) Critical steps and main challenges

The most critical as well as challenging part of the project is feature engineering. All of the raw features are categorical labels with extremely high cardinality, which requires all usable features to be generated through engineering. In addition, the presence of rare labels, if left untreated, will result in severe over fitting that is not resolvable through oversampling alone. The large datasize, even after downsizing, still is too large for exhaustive model tuning. In this project, only a very small hyperparameter space is explored through sensitivity tests to strive for a good enough instead of near-perfect solution.

(3) Have we solved the problem?

The solution we have presented is good enough as an alpha prototype. Even though the entire pipeline is devised based on results obtained from downsized training data, the selected light gradient boosting model is expected to perform well on larger-size data. The high false positive rate, however, leaves room for improvements. It mainly results from the extremely rare occurrence of the positive cases, and to some degree, could also have been intensified by the presence of rare labels that are not fully treated (note that the threshold 0.05% used for rare label imputation is quite arbitrary). For the next iteration of the pipeline desigh, as well as the final model deployment, the false positive rates should be further reduced.

5.3 Improvement

As mentioned earlier, hyperparameters used in the model are not thoroughly tuned. If we want to further improve the results, more comprehensive tuning of the hyperparameters is the first thing we should try.

More experimentations of class imbalance treatments should also be carried out. In this project, we only used random oversampling of the minority class for its simplicity. Other techniques such as SMOTE oversamplin (Synthetic Minority Over-sampling TEchnique), ADASYN oversampling (ADAptive SYNthetic sampling), under sampling of the majority class, and the combination of over- and under-sampling methods are available in `imblearn`[13] for further improvements.

The most ambiguous step in the current pipeline is rare label imputation. The threshold used for defining rare label is somewhat arbitrary--with the only constraint of being smaller than the proporition of the positive class in the data. Without rare-labe imputation, the pipeline suffers from severe overfitting. Nevertheless, the approach we have used is a good one but not guaranteed to be the best one.

Another caveat is that despite having applied rare label imputation, we still have categorical labels that are present in the training data but not in testing data. This has led to the presence of missing values in the testing data after label encoding. In this project, we simply filled these missing values with 0, but whether this operation is the most suitable is yet to be investigated further.

References

[1]
[2]
[3]