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.

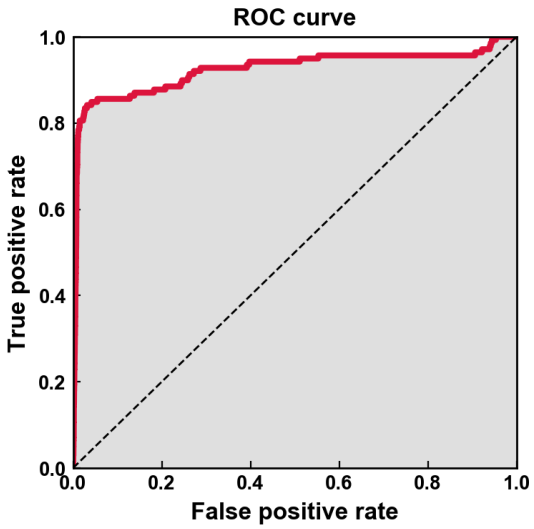

Figure 1: Example of receiver operating characteristic (ROC) curve. Shaded area denotes the area under the ROC curve (AUC).

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. 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**. We can see that it is precions plotted against recalls.

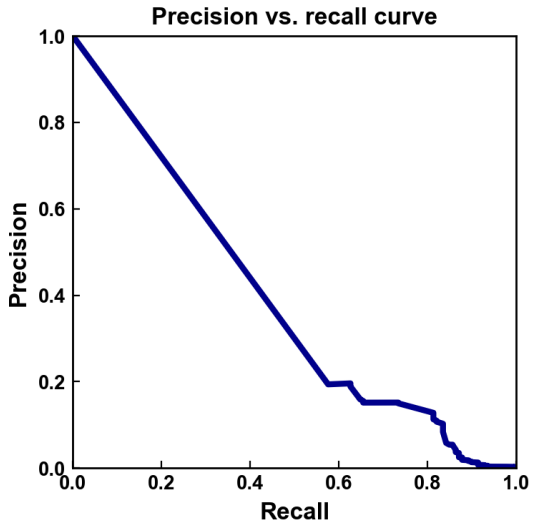

Figure 2: Example of precision-recall curve.

  - 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 corresponding 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 misidentify 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 still 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.

  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 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.

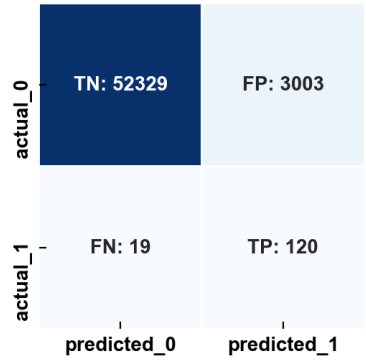

Figure 3: Example of confusion matrix (obtained from light gradient boosting algorithm tested in this project).

We can see that the diagonal elements of confusion matrix correspond to correct classification 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 being used here, but it provides a straightforward way of visualizing model performance in the presence of performance 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 experimentation. 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 ingredient 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 greater than 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. Feature engineering steps used in this project are detailed in the **Methdology** section.

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 balacing*** and ***oversampling minority*** are necessary for dealing 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 opted out 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 alternatively, they can 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 an weighted average over the predictions of all predictors; and 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 predictor, 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 bagging and adaptive boosting, stacking describes the process of using a meta learner (sometimes called a *blender*) to perform the aggregation. By training a model instead of taking averages, stacking reduces the subjectivity in the aggregation step, and allows the use of multilayer stacking in which the first few meta-learner layers could be several different blenders and last meta-learner layer could be a blender for 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 ($\hat{p}$) that varied between 0 and 1.

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

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 machine learning models.

III. Methodology

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 always 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 subsequent machine learning steps:

1. **Impute rare categorical labels**: 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, they often result in noticeable overfitting. In this project, all categorical labels that only present among fewer than 0.05% of the training data are replaced with a uniform categorical label (we use the large value of 1×10^{10} as the replacement) and then the imputed data are fed into subsequent feature engineering steps.
2. **Replace categorical labels with counts**: 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**: 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, the **12 features** used for machine learning are:

- 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`: $r_{Pearson} = 0.8$

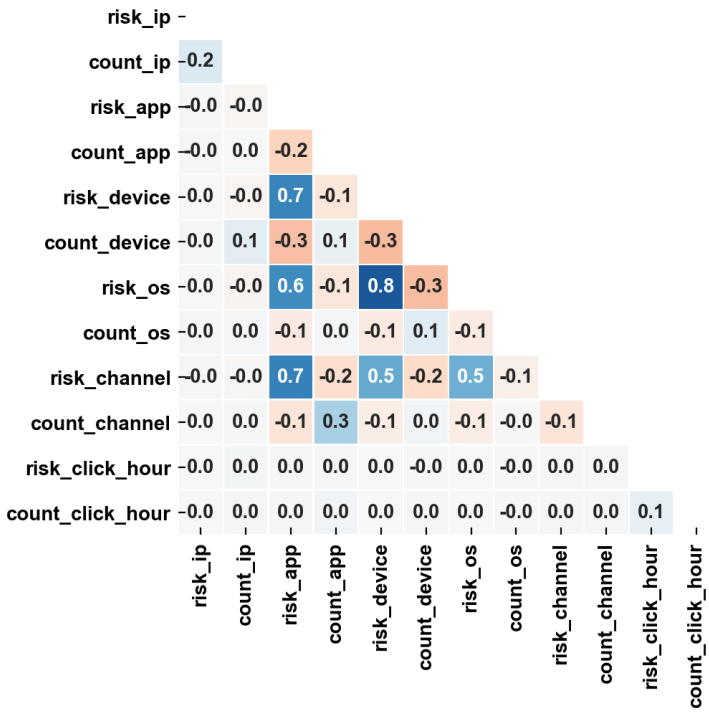

Figure 4: Correlation cofficients of the 12 engineered features

The presence of these correlations make 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 that of these risk-factor features may not contribute additional information due to such correlations. The count features, on the other hand, don't appear to show strong correlations with any of the other features.

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 is possible to be carried out with `pandas`, it is done by calling shell command `gshuf` for faster operations.
- `csv_list_fields`: Helper function for listing all fields of 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 traning 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 mean (also known as rick factor).
- `df_rarelabel_imputer`: Performs rare label imputation by using the unusual , uniform value of 1×10^{10} to replacing 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 these rare labels to be grouped into one category and then being encoded. Similar as the encoding operation, the imputer mapping is always generated based on training data and then applied to both traing and testing data--despite that the same label could be very rare in training but not as rare in testing data.
- `df_label2num_endoding`: The one-step wrapper that performs label-to-number encoding and drops the original categorical features. 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 traing and testing data frames.

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

- `timer`: A decorator that provides 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 standardization (`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 with cross validations.
- `gridsearch`: A wrapper function for hyperparameter tuning. It takes in `pipeline` object and decide to perform grid search (when hyperparameter grid is small) or randomized search (when hyperparameter grid is large).
- `Classifier`: A class that is the main 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.

[Note] The `assess` method exists and functions, but it is NOT used in producing core results for this report. The inclusion of these steps will be further discussed in later *Improvement* section.

Software requirements:

The implementation relies mostly on sklearn, mlen, and bla for oversampling minority classes to combat the case of severe class imbalance. Only tree stumps are used instead of deeper trees.

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 the noise in training data. Overfitting is a common problem facing machine learning and must be carefully 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 training and testing data. For models that are overfitting, we expect to see a noticeable differences between traing and testing scores.

(1) Effects of regularization on logistic regression

For logistic regression, we examine the effects of $l2$ regularization (also known as "Tikhonov regularization") that adds the term $\frac{1}{2C}\,||\theta^2||$ to the cost function, where C is the inverse of regularization strength (smaller C correponds to stronger regularization), and θ is 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 to keep the weights of the model as small as possible. From **Figure 5** we can see that the best regularization C value is around 1×10^{-6}.

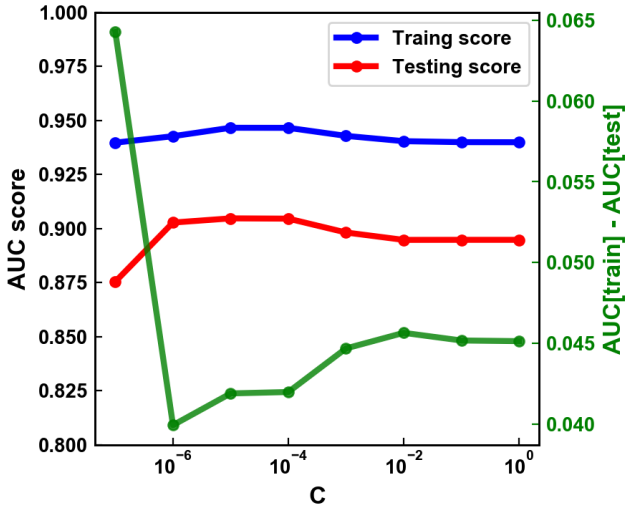

Figure 5: Effects of $l2$ normalization on performance of logistic regression.

(2) Effects of regularization on random forest

For random forest, `max_depth` is the most important regularization parameter that contraints the depth of the decision trees. A few other hyperparameters can also be used to control the complexity of individual trees in the random forest, such as `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 maximum tree depth given its critical role, and the number of decision trees used to form the forest is fixed at `50`. From **Figure 6** we can see that `max_depth` of `1` provides the best performance. In this case, only one split is use and sometimes we call the decision trees "decision stumps".

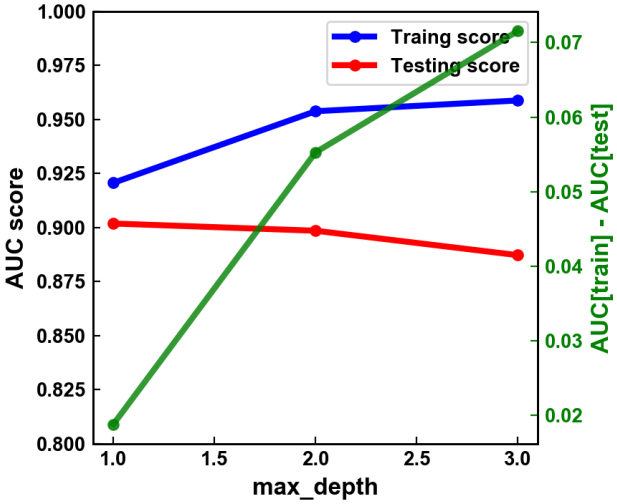

Figure 6: Effects of tree depth regularization on performance of random forest.

IV. Results

Model Evaluation and Validation

Justification

V. Conclusion

Reflection

Improvement
