Ad Tracking – Attribution Prediction

I. Definition

Project Overview

Rare Event Prediction (Kaggle competition)

Reference: https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection#description

The following two paragraphs have been taken verbatim from the link above:

Fraud risk is everywhere, but for companies that advertise online, click fraud can happen at an overwhelming volume, resulting in misleading click data and wasted money. Ad channels can drive up costs by simply clicking on the ad at a large scale. With over 1 billion smart mobile devices in active use every month, China is the largest mobile market in the world and therefore suffers from huge volumes of fraudulent traffic.

TalkingData, China's largest independent big data service platform, covers over 70% of active mobile devices nationwide. They handle 3 billion clicks per day, of which 90% are potentially fraudulent. Their current approach to prevent click fraud for app developers is to measure the journey of a user's click across their portfolio, and flag IP addresses who produce lots of clicks, but never end up installing apps. With this information, they've built an IP blacklist and device blacklist.

In order to further developing their solution, we will **build an algorithm that predicts whether a user will download an app after clicking a mobile app ad**.

Domain Background

The problem at hand could be characterized as a rare event prediction. Although the data is very clean, we only have 6 features and the positive class accounts only to about 0.2% of the entire data set. The main tasks involved as part of the solution would be to build a richer feature-set, followed by techniques and algorithms to address the highly imbalanced class problem during prediction.

Problem Statement

The objective of this project is to be able to predict whether a particular click on an app ad catered by a channel from an IP address on a mobile device will subsequently result in its download. We will go about solving this by feature creation, train data set balancing, feature selection, and finally, modeling. We will use a variety of models to test their performance prior to arriving at the final model. We will try out the traditional logistic regression, ensembles like random forest, AdaBoost, and also newer more powerful ensembles like the XGBoost and light gradient boosting algorithms. The binary classification model thus built would be evaluated based on the area under the ROC curve.

Datasets and Inputs

The dataset used in this project was provided by the host of the Kaggle competition, TalkingData, and acquired from Kaggle using their API. The main training/validation dataset has 1 million rows and 8 columns, which would be used to build the models. The final test data set that we will assess our models on has 500,000 rows. (Note: The proposal mentioned that the data set would have 100000 rows; I have since been able to take a larger sample, hence the change) Each record corresponds to a click. The columns in the data set are as follows:

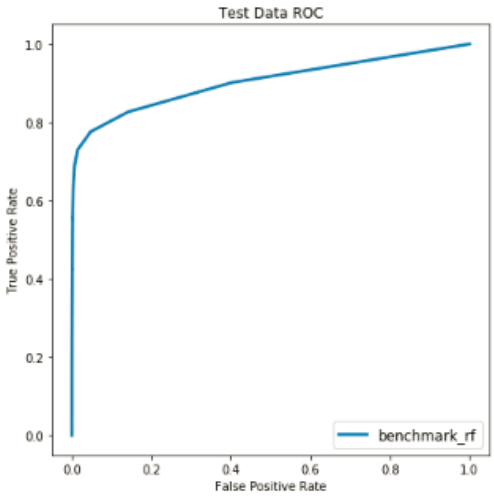
- ip: ip address of click.
- app: app id for marketing.
- device: device type id of user mobile phone (e.g., iphone 6 plus, iphone 7, huawei mate 7, etc.)
- os: os version id of user mobile phone
- channel: channel id of mobile ad publisher
- click_time: timestamp of click (UTC)
- attributed_time: if the user downloaded the app after clicking an ad, this is the time of the app download
- is_attributed: the target that is to be predicted, indicating the app was downloaded

Metrics

The target variable is binary in nature and hence the nature of the prediction problem is that of binary classification. The data set is highly imbalanced though, with the positive class accounting for only 0.3% of the data. As a result of the aforementioned points, we cannot use common metrics like accuracy as they would be misleading. Instead, we will

measure the strength of the models and the solution model using the area under the Receiver Operating Curve metric (ROC AUC score).

Accuracy only looks at the percentage of misclassified data points and also for only a single threshold value (typically 0.5) on the predicted probability. ROC on the other hand helps us visualize and understand the model performance for both the positive and negative classes individually, and also at varying threshold levels thus making the performance measure independent of the threshold selection.



The ROC is a plot of the False Positive Rate against the True Positive Rate for differing values of the cutoff/threshold. At the threshold value 0 for predicted probability, all the data points are classified as 1, as all the probabilities are greater than or equal to 0. This would correspond to TPR = 1 as well as FPR = 1; all the positive class data points are classified correctly and all the negative class data points are also classified as positive. This corresponds to the top-right corner of the plot. At threshold of 1, TPR = FPR = 0, corresponding to the bottom-left corner of the plot. TPR and FPR for all other thresholds within this range, when plotted, create the ROC curve. The closer the curve is to the top-left corner of the plot, i.e., low FPR but high TPR, the better the model performance. The ROC AUC score is nothing but the area under the ROC curve, with the best model having the value 1 and worst 0.

We will also look at how the model is performing in terms of precision (proportion of the predicted positives that are actually positive), recall (proportion of the true positives predicted as positive), their harmonic mean, the F1 score, F2 score which weighs recall higher than precision and lastly F0.5 score which weighs precision higher than recall.

II. Analysis

Data Exploration and Visualization

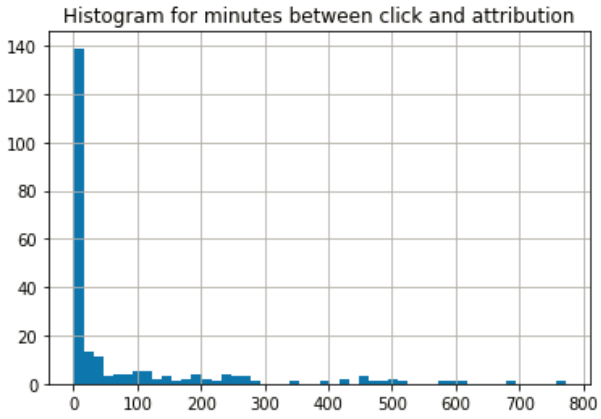
The data used in this project has been provided by TalkingData. Here is a snippet of the same:

	ip	app	device	os	channel	click_time	attributed_time	is_attributed
0	87540	12	1	13	497	2017-11-07 09:30:38	NaN	0
1	105560	25	1	17	259	2017-11-07 13:40:27	NaN	0
2	101424	12	1	19	212	2017-11-07 18:05:24	NaN	0
3	94584	13	1	13	477	2017-11-07 04:58:08	NaN	0
4	68413	12	1	1	178	2017-11-09 09:00:09	NaN	0

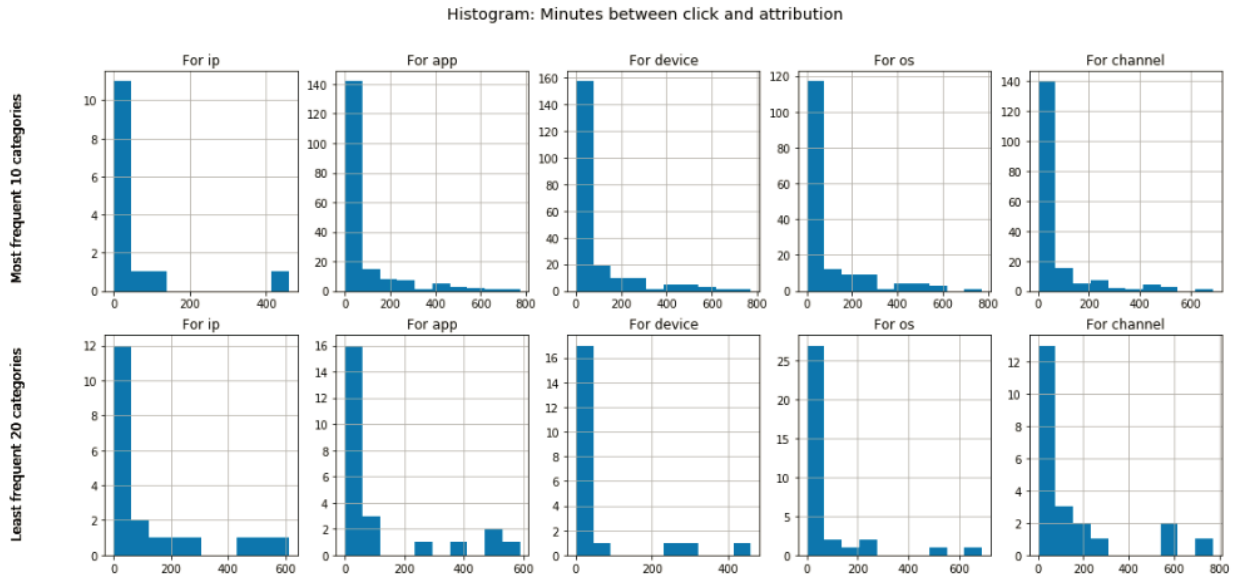
The attributes have been described in the earlier section. One point to note would be that the first five attributes have been encoded, in order not to disclose the actual values. They are categorical in nature. The last column contains the target variable. There are 227 positive label records which corresponds to 0.23% of the dataset. The column 'attributed_time is only populated when 'is_attributed' is 1. All but the 'attributed_time' column have no missing values.

When we are making predictions, we would not have the 'attributed_time' available so it cannot be made part of the feature set for modeling. However, we can explore it to see if we gain any insight into other features in the dataset for the purposes of data cleaning if at all; following that we will drop the column.

We look at the time difference between click_time and attributed_time and notice that on most instances the difference is a couple minutes, as expected.

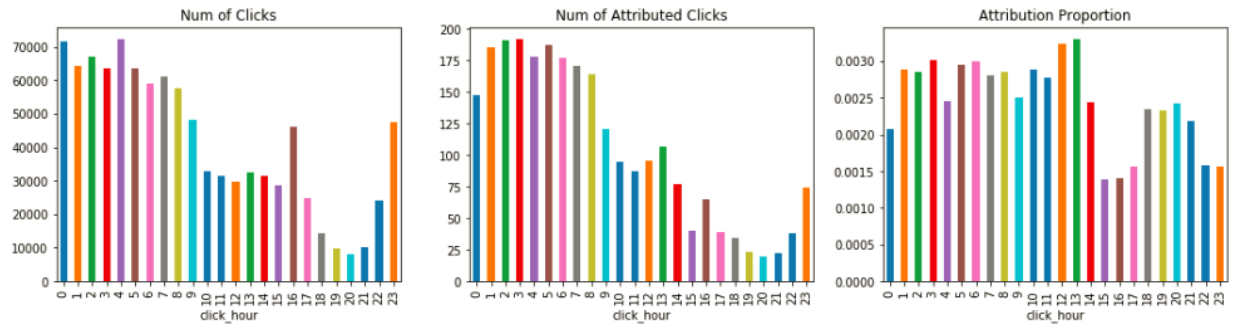


We then look at the most and least frequent categories in each of features in the data set and plot the histogram for the difference in time between click and attribution:



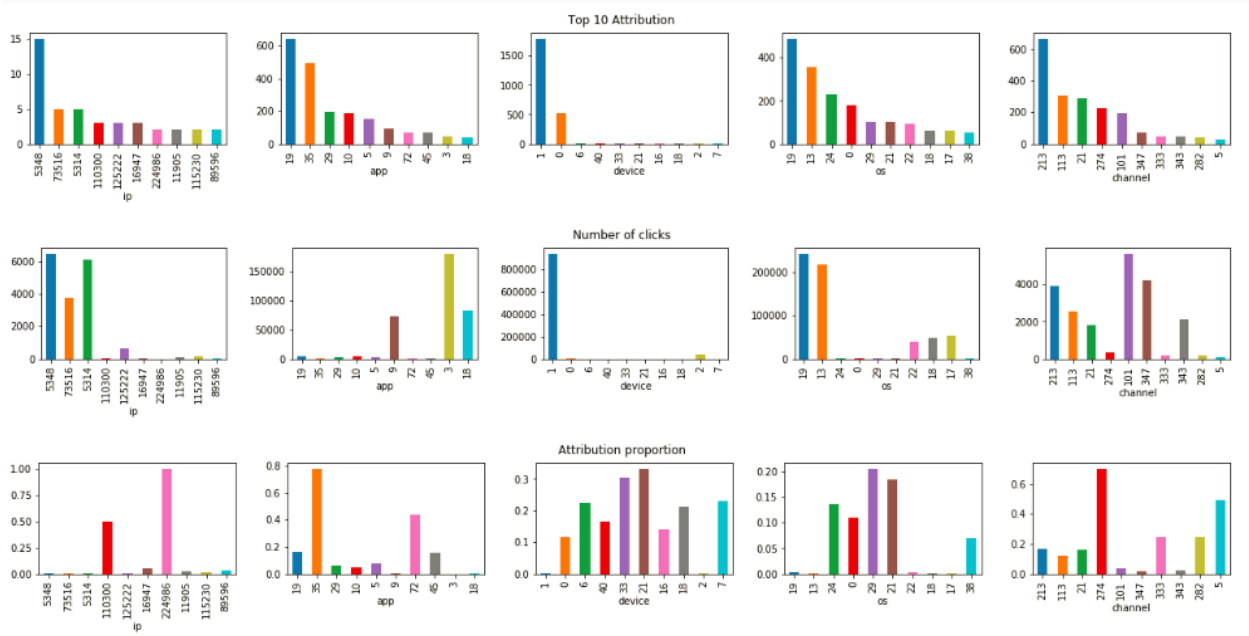
Again, the patterns are as expected. Since there isn't anything out of the ordinary, we exclude this column from further analysis.

Next, we take a look at the 'click_hour' variable:



As expected, the click frequency drops during night time in China (UTC+7); from the third plot we see that the attribution rates are relatively higher during the day compared to at night.

Top 10 values by attribution:



For the top 10 values by attribution numbers for each feature:
ip: There's a gradual decline in the attribution numbers for these ip's; 5348, 5314, 73516 stand out for click traffic, and 15195 for a very high attribution proportion.
app: The top attributed apps have very few clicks, again indicating the popularity of apps and a predisposition to convert easily. Two of them are above average.
device: there is a steep decline for attribution numbers for device. However, the proportions are higher than average for most.
os: Most of the attributions come from 4 OS' and OS 29 has the highest attribution proportion.
channel: Most of the top attributed channels also have a reasonably sized click traffic, but the most striking finding is the potency(attribution proportion) of channel 274 and 5 for attribution.

Algorithms and Techniques

We will first expand the data set through feature engineering, and then shrink it down after taking into consideration correlations and features potentially responsible for overfitting. Once we have the final feature set, we will create a balanced data set using SMOTE. We then create a benchmark model. Following this, we create a few more models using different algorithms to see which ones perform best in their most basic form.

The algorithms we will be using in the project are – Adaboost for feature selection, Random forest as the benchmark model, followed by Adaboost again with the final modeling feature set, logistic regression, XGBoost and Light Gradient Boosting model.

We will discuss them briefly here. Note: The justification for the use of each is *italicized*.

Adaboost:

Since we have a highly imbalanced data set, it is imperative that we are able to extract the most powerful features out that would generalize well on the test data. For this reason, we go a step further than simply removing uncorrelated features, and build an adaboost model to get the final feature set.

Adaptive ***boost***ing is an ensemble technique wherein the base estimator (in our case a decision tree classifier with max tree depth of 3) which a weak learner is built for the entire training data set. The same procedure is carried out 'n_estimators' (50) number of times, the only difference being that the weights for each data point in the model are set/adjusted based on whether they were classified correctly or incorrectly previously, so that the current model is built with an emphasis on the misclassified points and de-emphasis on the correctly classified data points. The final model or strong learner is a combination of the weak learners. This model can be thought of as a sequential ensemble model, where output from one model affects the input of the subsequent models. *Our data set has continuous numeric dependent variables, and a binary target variable and correctly classifying both classes is equally important. This makes AdaBoost a good candidate for this problem.*

Random Forest:

Random Forest has been a popular ensemble algorithm for a long time due to its simplicity. Here, multiple decision trees are built independently of each other on a sample that is the same size as the training data set, however the points are sampled with replacement thus ending up being different across the decision trees. The final prediction from the random forest model is the average of the set of decision trees built. Unlike AdaBoost, this technique takes more of a parallel approach to building the random forest model. *It is a simple ensemble model, which is applicable to use in this case because of binary targets and also because it synthetically adds variance in the overall training data through sampling with replacement. This would help with generalization when it came to predicting using data it has not seen before.*

Logistic Regression:

The base estimators above were decision trees that come up with multiple linear decision boundaries parallel to the axes, and if the data is not necessarily separated that way between the two classes, the model would unnecessarily over-fit. This is the main reason we also try the logistic regression algorithm for this data set, and also because we are dealing with a dense dataset with numeric variables and not categorical. A logistic

regression model looks for a single linear separator for the data points and not necessarily parallel to any of the feature axes.

Logistic regression is a parametric model in that it comes with a functional form whose parameters (coefficients) need to be estimated during training. Typically parametric models are more biased compared to non-parametric models (like trees) since they make assumptions about the underlying data. In this case the assumption is that the function is linear, which may not be the case for our data set. But these models are also faster and take less time even during making predictions. In our case however, since time is not a constraint and since we have a lot of data, non-parametric models can be relied upon more to be able to capture the true underlying mapping function, than us imposing a functional form. Hence we do not use any other parametric models in this project.

XGBoost:

E**x**treme **G**radient **Boost**ing is a newer form of gradient boosting which is much faster and more powerful by virtue of the way it uses computational resources. *It works well with large datasets such as the one used here. It also predicts for a binary class.*

Gradient boosting is much like AdaBoost in that it is an ensemble of weak learners (most commonly trees). The difference is that while AdaBoost iteratively modifies the weights of the samples as discussed earlier, in gradient boosting, the weak learners iteratively train on the residuals from the fitting of the previous weak learner until a certain threshold. They iteratively look for patterns in the (pseudo-)residual errors, and the weak learner keeps getting added to the previous model. The learners are built with the objective of minimizing the loss function using the gradient descent algorithm. (In short, in gradient descent, the objective function coefficients are optimized by taking a derivative of the function to understand the direction of the slope and then updated accordingly.) In our case of classification, the objective would be to minimize the loss from logistic regression.

The advantages of XGBoost are that it has a host of parameters that can be optimized to get the best classifiers and it still will not take as long as traditional gradient boosting models because it pushes to the extreme, the computational power it uses through parallelization, cache optimization etc. *Our sample data set is large with 1 million rows (much larger if we took a larger sample from TalkingData), and it would be computationally intensive to build a complex model. XGBoost allows for a good tradeoff.*

Light Gradient Boosting:

LGB is another boosting algorithm that is popular because it is fast (light). It works in a similar manner to XGBoost, the difference being that the tree-growth is level-wise in XGBoost whereas leaf-wise in LGB. *It uses less memory because it converts continuous values into bins before splits, thus works well for huge data sets like ours. It predicts for a binary class.* It is also fast because it enables parallel learning. However, since it uses leaf-wise split, it is more prone to overfitting. *We try this as an alternative to XGBoost to see if the leaf-wise split ends up capturing additional patterns that weren't earlier.*

Once we have identified the algorithm that is working best, we then follow it up with a grid search with cross validation to perform hyper parameter tuning to optimize for ROC AUC. Finally, we conclude with the model results.

Feature engineering:

Reference: https://www.kaggle.com/nanomathias/feature-engineering-importance-testing

We create the following 4 categories of new features to begin with:

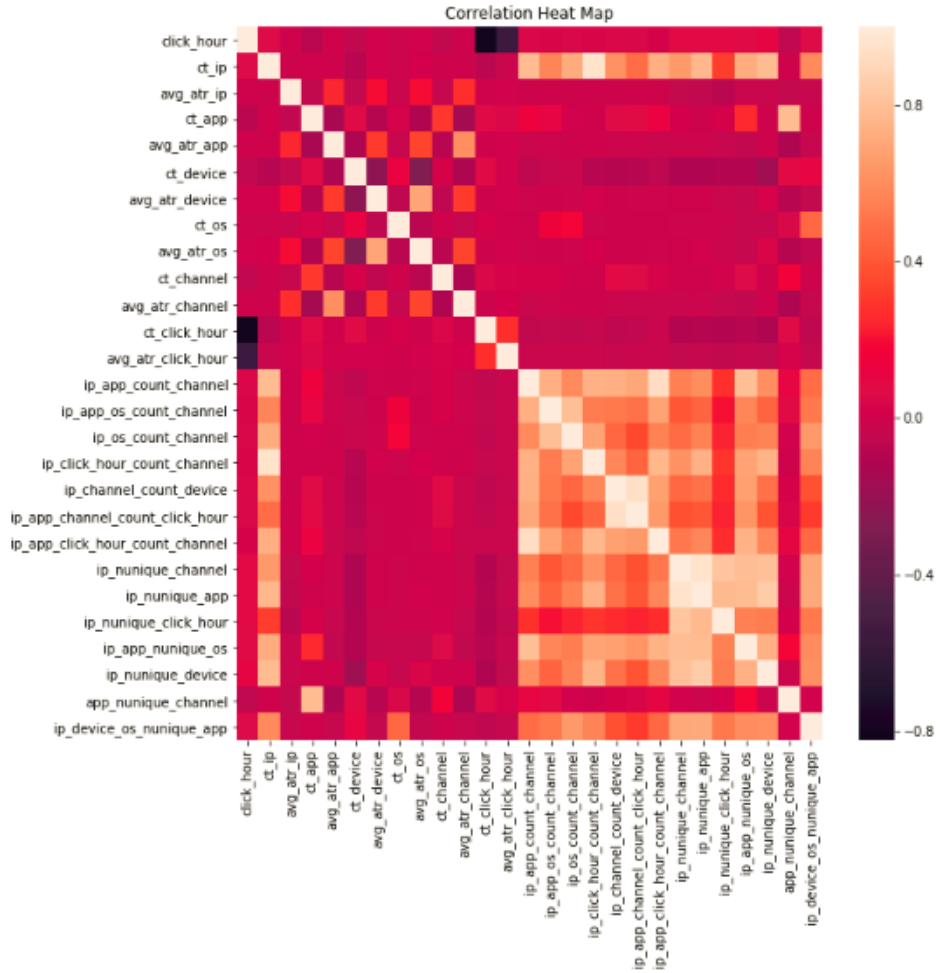
1. Counts for each of 5 categorical features and click_hour
2. Proportion of the clicks for each of the unique values in each feature which are attributed, i.e., mean of 'is_attributed' feature. (For the test data, this would be populated through relevant joins, and where there are missing values, filled with the column mean.
3. Counts for unique combinations of the base attributes.
4. Number of distinct values for a unique combinations of base attributes

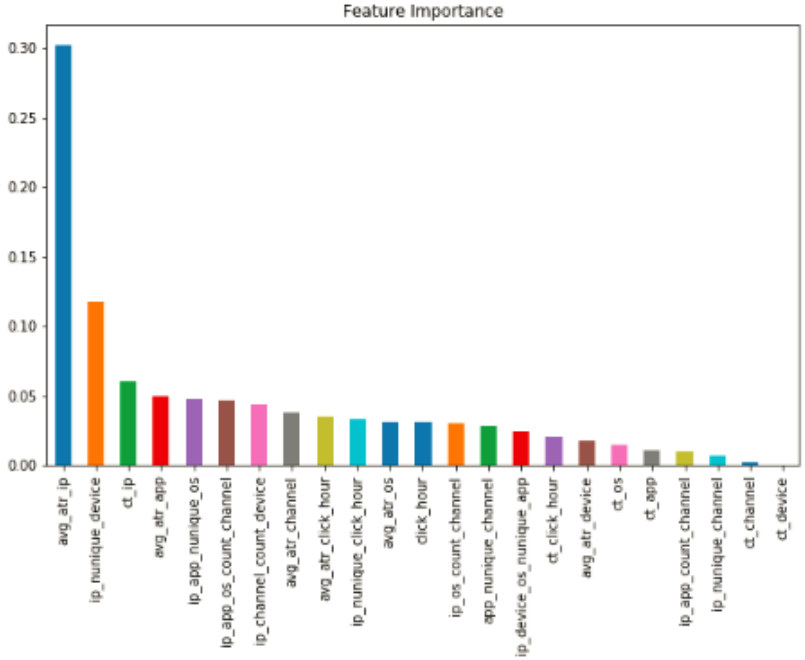
We drop all those columns that have more than 0.9 correlation coefficient.

Feature Selection

After we have removed the highly correlated features, we build an adaboost model to go one step further to analyze feature importance. The graph below shows the same.

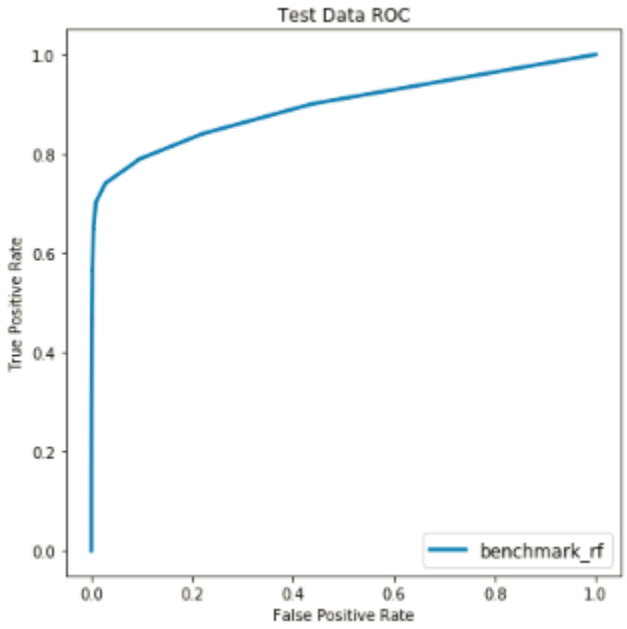
On looking at the feature importance we notice almost half of the features contribute less than 0.025 weight to the model. We will remove these as we want the final model to generalize well on new data. We also notice that there is one feature ('avg_atr_ip') chosen by the model which is overpowering, contributing to more than twice the weight of the second most important feature. This could possibly be associated with overfitting and so we remove this feature in addition to the earlier low-weight features.


Correlation Heat Map
click_hour
ct_ip
avg_atr_ip
ct_app
avg_atr_app
ct_device
avg_atr_device
ct_os
avg_atr_os
ct_channel
avg_atr_channel
ct_click_hour
avg_atr_click_hour
ip_app_count_channel
ip_app_os_count_channel
ip_os_count_channel
ip_click_hour_count_channel
ip_channel_count_device
ip_channel_count_device
ip_app_channel_count_click_hour
ip_app_click_hour_count_channel
ip_nunique_channel
ip_nunique_app
ip_nunique_click_hour
ip_app_nunique_os
ip_nunique_device
app_nunique_channel
ip_device_os_nunique_app
0.8
0.4
0.0
−0.4
−0.8


Feature Importance
0.30
0.25
0.20
0.15
0.10
0.05
0.00
avg_atr_ip
ip_nunique_device
ct_ip
avg_atr_app
ip_app_nunique_os
ip_app_os_count_channel
ip_channel_count_device
avg_atr_channel
avg_atr_click_hour
ip_nunique_click_hour
avg_atr_os
click_hour
ip_os_count_channel
app_nunique_channel
ip_device_os_nunique_app
ct_click_hour
avg_atr_device
ct_os
ct_app
ip_app_count_channel
ip_nunique_channel
ct_channel
ct_device

Benchmark

A basic Random Forest classifier is chosen as the benchmark model for the purposes of this project. After fitting this model on the training set and subsequently predicting for the test data set, we get an **ROC AUC score of 0.823**. The ROC curve is shown below:



III. Methodology

This project is executed using a python 3.0 kernel in Jupyter Notebook. The main packages we use in the notebook are as follows:

- pandas and numpy for data manipulation
- imblearn for oversampling from the imbalanced data set
- packages under sklearn for modeling, cross validation, hyper parameter tuning, computing and reporting metrics
- lightgbm and xgboost for modeling with their respective algorithms
- pickle to store the models created
- scipy to find optimal threshold
- matplotlib and seaborn for data visualization

Data Preprocessing

After the feature correlation exercise, and before we do any modeling, we perform over-sampling using the SMOTE (Synthetic Minority Over-sampling Technique from the imblearn.over_sampling package) on the highly imbalanced data set. This gives us a dataset with number of rows twice as many as the majority class. We then again take a random sample of the data to keep a final 1 million rows for training models.

After applying SMOTE, the target class ('is_attributed') distribution is as follows:

```
0    500321
1    499679
```

Implementation

After building the benchmark model, we now move on to build a few more using a variety of algorithms, including AdaBoost, logistic regression, XGBoost and LightGBM.

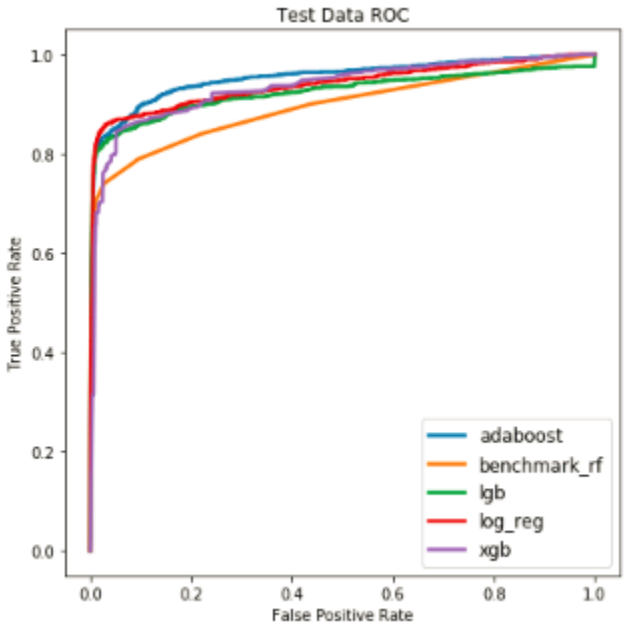
The procedure we follow is as follows:

- Instantiate a classifier object for the algorithm of interest
- Create a fit for the training data with the final feature set of 13 features.
- Save the model as a .pkl file
- Use the model to make predictions for the test data and compute relevant metrics
- Plot the ROC curve

The main functions we define to help us with these steps are:

- **save_model(clf, algo):** Takes as input the classifier and a string value for the model to dump a .pkl file in the working directory
- **pred_metrics(y_test, y_pred, algo):** Takes as input the actual and predicted values for the test data set's target variable to compute AUC, F1, F2, and F.5 and store in the metrics dict with key as the string value representing the model; prints the ROC AUC score.
- **plot_roc(clf, algo):** Takes as input the classifier and a string value for the model and outputs a plot with all ROC curves as stored in the roc data frame.
- **feat_importance(clf):** Takes as input a tree classifier and plots the feature weights in decreasing order.

After training on all these models, the final set of ROCs are plotted below:



And the metrics are tabulated below:

	AdaBoost	Random Forest (benchmark)	Light GBM	Logistic Regression	XGBoost
AUC	0.899	0.823	0.925	0.908	**0.935**
F.5	0.150	0.321	0.037	0.150	0.006
F1	0.216	0.396	0.058	0.216	0.009
F2	0.386	0.517	0.131	0.389	0.023

We see that the XGB model has performed the best (ROC AUC score of 0.935) for the given data and feature set, with the highest ROC AUC score. So we choose this to perform model tuning.

The main challenges through the implementation process was that of feature engineering and imbalanced class. We had to come up with features that would work well with test data. Given that most features were categorical and not exhaustive in the categories, we had to resort to a lot of 'group by' aggregations, which seem to have worked well. We solved the imbalanced class problem using SMOTE. Other than that, there were not too many noteworthy issues, mainly attributing to the complete and clean data set, with few missing values.

Refinement

For the hyper-parameter tuning of the XGB model we use sklearn's GridSearchCV. GridSearchCV searches for the best model amongst a number of them built according to a 'grid' of parameters provided with respect to a specific scoring criterion.

The arguments we pass to GridSearchCV are:

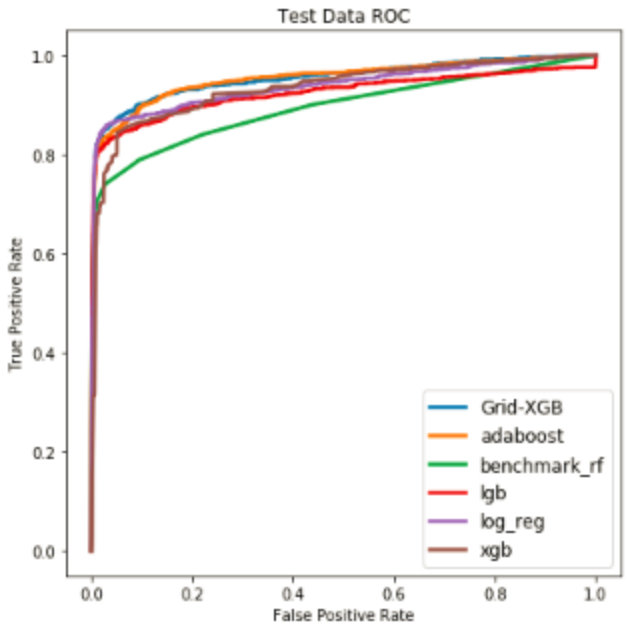
- The kind of classifier we want to build – base estimator in this case is the XGBClassifier()
- The set of parameters over which the grid search would be conducted
- The number of parallel jobs to run
- Number of folds to perform cross validation in each model run
- The scoring criterion (ROC AUC) based on which to pick the best estimator
- The refit value as True to refit with best parameters on the entire data set
- Verbose as 2 for limited messaging

The grid parameters we want to optimize on are:

- Learning rate – how big or small of a step is taken during gradient descent optimization
- Max depth – max depth of the XGBoost tree
- Min child weight – minimum number of samples in a leaf node below which the leaf will not be split further
- Subsample – proportion of the training data set that is used by XGBoost every time to build a model
- Col sample by tree – proportion of columns used to construct the tree

Other single value parameters we set for all of the grid searches are:

- Nthread – number of parallel threads used to run XGBoost
- Objective – the objective function that needs to be optimized in every iteration of the XGBoost algorithm
- Silent – to not print running messages during execution
- N_Estimators – Number of estimators /weak learners for each XGBoost model
- Missing – value to replace missing values with
- Random state – to ensure consistency of model results across multiple runs



We get a final ROC AUC score of **0.954.**

To make the final class predictions, we will see if there is a better threshold than the default 0.5 using scipy's optimize package. On optimizing for the objective function of F1 score, we find that a **threshold of 0.735** is best, which we will use for final class prediction.

IV. Results

Model Evaluation and Validation

The best estimator from the previous exercise of grid search has the following form:

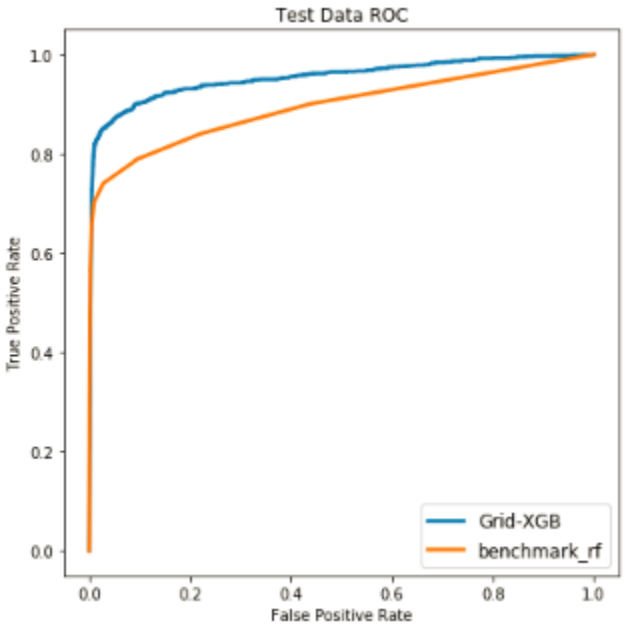
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.6, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=7, min_child_weight=10, missing=-999, n_estimators=10,
       n_jobs=1, nthread=4, objective='binary:logistic', random_state=1337,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None, silent=1,
       subsample=0.9)
```

Grid Search Parameter	Values searched	Optimal parameter value
Learning Rate	[0.05, 0.1]	0.1
Max Depth	[3, 7]	7
Min Child Weight	[5, 10]	10
Subsample	[0.7, 0.9]	0.9
Col Sample by Tree	[0.6, 1.0]	0.6

We are confident about the model because we have done multiple levels of validation to get to this point – 3-fold cross validation as well as running the final model on test data which it has never seen before. On evaluating this model, we see it has performed very well with an ROC AUC score of 0.954. And with the optimal threshold of 0.735, it has an accuracy of 99.53% with an F1 measure of 0.43.

Justification

Comparing the final model with the benchmark model, we see a considerable improvement:



Let us compare some of the metrics for the benchmark model and the final model with predicted class using optimal threshold when run on the test data set:

Metric	Benchmark Model	Final XGBoost Model
AUC	0.823	**0.954**
Accuracy	99.50%	**99.53%**
F1	0.396	**0.427**

The final XGBoost model does better than the benchmark model on all three metrics.

Confusion matrices:

Benchmark Model

Predicted / Actual	0	1	All
0	496685	2057	498742
1	440	818	1258
All	497125	2875	500000

Final XGBoost Model

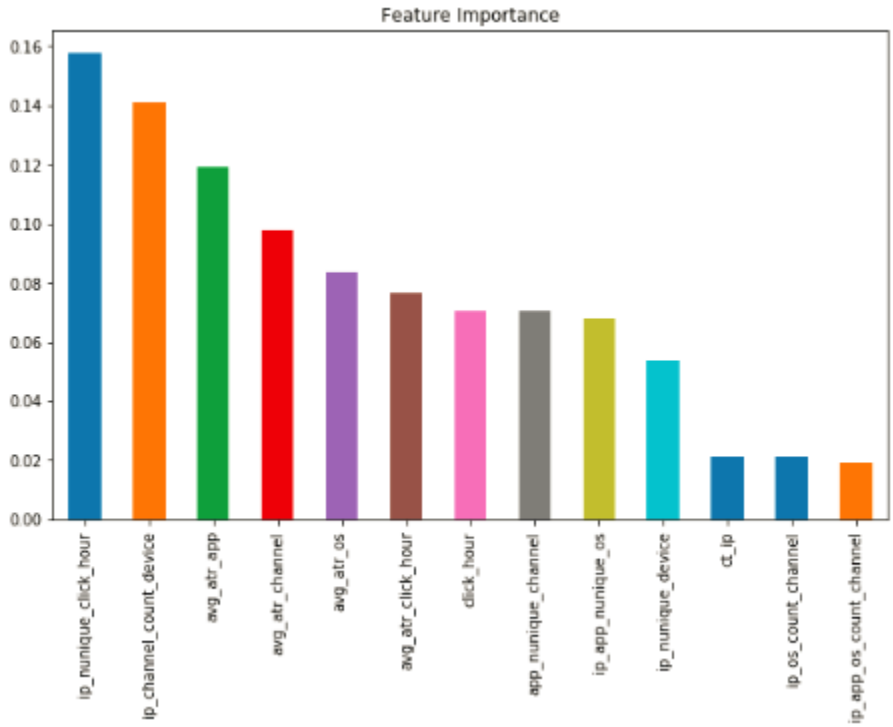
Predicted / Actual	0	1	All
0	496752	1990	498742
1	377	881	1258
All	497129	2871	500000

We that the final model does better than the benchmark – it has reduced both the false positives and the false negatives (a total of 130 samples). Other than the high false positives resulting in low precision and hence F1 score, I am satisfied with the results.

V. Conclusion

Free form visualization

The final set of feature importance for the refined XGBoost model is given below:



Based on this graph, we are better able to understand the underlying data and under what circumstances it is more likely for an ad click to result in an app download. The top ones based on the graph above are mentioned below:

- 'ip_nunique_click_hour': The hour at which an app was clicked on from an ip address is a strong indicator for classifying attribution.
- 'ip_channel_count_device': Channels within an ip address through which ads are catered are also strong indicators.
- 'avg_atr_app': An app itself inherently can be used to determine the chances of attribution, since the proportion of clicks for a particular app is also a strong feature in this predictive model.
- 'avg_atr_channel': Attribution proportions for channels. i.e., some channels are more likely to have clicks end up attributed than others.
- 'avg_atr_os': Attribution proportions for os.

Reflection

The project started with loading the necessary packages and doing a preliminary data sanitation check. We then explored the features in the data set through visualization. By the end of this exercise, we had a good understanding of the data as well as had removed feature – 'attributed_time', and created a 'click_hour' feature. We then moved on to feature engineering, followed by removing correlated features. We further removed a second set of features based on feature importance from an AdaBoost model that was built using a balanced data set. This completed the pre-processing phase.

Once we had the final feature set, we built a number of models using different algorithms mentioned earlier, starting with the benchmark model. We plot the ROC curves for comparison and also look at the different metrics to select which model we want to further refine. This completes the implementation phase.

Finally, we refine the XGBoost model using grid search over a set of parameter values. The best estimator from this step is our final model. We plot out feature importance to see which features show up on top. A final step before we predict the final target class for the test data is to pick the optimal threshold. We do this by optimizing for the F1 score. This completes the refinement phase. And we finish with discussing the final model performance and its comparison with the benchmark model.

The predictive modeling problem that we addressed as part of this project came with some challenges. Firstly, the highly imbalanced data set led us to not have as good a representation of the positive class as we did the negative, with only 0.3% being

positive. Secondly, the set of initial features were also a challenge which was two-fold – one, there were only 8 features, and two, they were mostly all categorical and each with too many unique values to have one-hot-encoding make sense.

The way we overcame these challenges was by applying SMOTE for over-sampling, and through feature creation and selection. The feature set did not seem rich and diverse enough but I decided to move along to see if they were good enough and they were. In hindsight, I could have tried out models using counts, average attribution, and unique values for an exhaustive set of combinations of base features to see if it would have provided a better lift.

One significant observation I made was that it is quite important that one get the initial feature set somewhat right. If you don't then no amount of model improvement can get you as close to a good model as it would with a stronger initial feature set. The inclusion of the feature selection section by examining feature importance from AdaBoost, and dropping some more features, played a major role in leading me to high performing models subsequently, even though the marginal increase in performance between models was small.

Although I feel that I did not get as much of a lift in the model performance (F1 score particularly) as I was expecting after building the more powerful boosting models, this model can definitely be used for this problem, while we attempt to build a richer feature set along with a deeper and more comprehensive grid search. It must be noted that I worked with only a subset of the data that was available from Talking Data, due to computational limitations. So it is possible that the model would generalize better if trained on a larger data set with more positive class samples.

Improvement

If I considered the current solution as benchmark, I am quite certain that a better model exists. However, it would require more resources and much longer to train those models. The top three ways I can immediately see to improve the model performance, as dwelled upon lightly in the previous section are:

- **Increase the sample size:** TalkingData had close to 190 million records, but I only sampled 1.5 million records from the top 10 million rows and eventually used 1 million records for training and 0.5 million for testing. Given more powerful resources, I could have tried extracting more positive class records rather than use SMOTE.
- **Increase feature set:** Create features for count, attribution proportion, unique values for all combinations of ip, app, os, device, channel, click_hour. This would

make the data set quite sparse but it would be worth a try given that algorithms like XGBoost can handle sparse data sets well.

- **More comprehensive grid search:** Currently grid search is going over 32 (param values) x 3 (cv) = 96 fits. I could expand the grid params dict to have a broader search. I could also optimize for F1 instead of ROC AUC to check for improvement in precision without compromising too much on recall.

References:

- https://www.kaggle.com/nanomathias/feature-engineering-importance-testing
- https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection#description
- https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf
- https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/
- https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/
