

Machine Learning Engineer Nanodegree

Capstone Project

Samrat Pandiri

April 22nd 2018

TalkingData AdTracking Fraud Detection Challenge

Definition

Project Overview

"TalkingData AdTracking Fraud Detection Challenge" is a classification problem that deals with detecting fraud in Ad traffic. As we know that one of the major areas of "Click Fraud" is in the area of mobile ad channels where automated scripts may click mobile Ads or download a mobile app without a real reason. The problem that we will be solving here is to predict if a user click is genuine or fraudulent.

Problem Statement

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 the app.

So, this is a classification problem where we have to build an algorithm that predicts whether a user will download an app after clicking a mobile app ad.

To build an algorithm that predicts whether a user will download an app after clicking a mobile app ad TalkingData has provided us a generous dataset covering approximately 200 million clicks over 4 days.

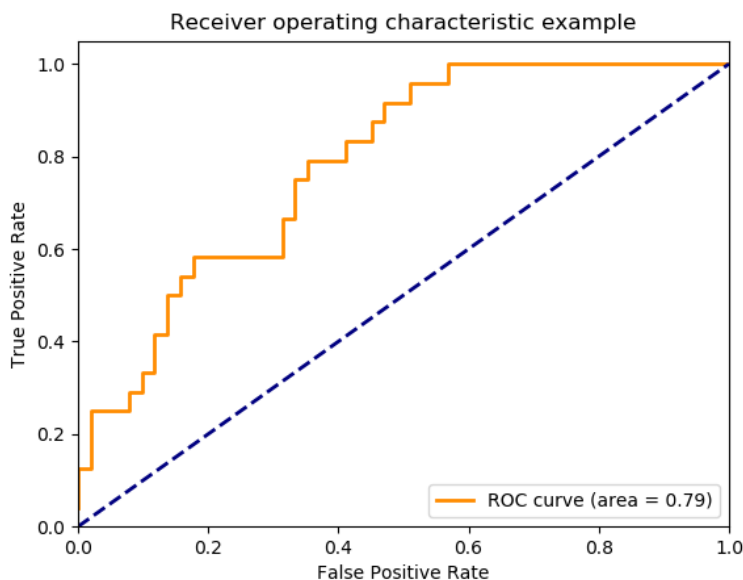
We will build a solution for the above stated problem and try to identify click frauds using Machine Learning. As we have an imbalanced data set, we will use a robust algorithm that is not sensitive to imbalance data and one good option is LightGBM.

Metrics

As the submissions are evaluated on area under the ROC curve between the predicted probability and the observed target we will also use the same metric.

A receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the positives (TPR = true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity, and FPR is one minus the specificity or true negative rate.

We will use the ***roc_auc_score*** in sklearn that computes the area under the receiver operating characteristic (ROC) curve, which is also denoted by AUC or AUROC. By computing the area under the roc curve, the curve information is summarized in one number.



Analysis

Data Exploration and Visualization

As we have been given some details about the files and the features, we will start with that.

File descriptions

- train.csv - The training set
- test.csv - the testing set

Data fields

Each row of the training data contains a click record, with the following features.

- 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 user download the app for 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

The train and test files are provided as zip files. So, we will first download them and use Python 'zipfile' module to get some basic idea.

Details of train.csv.zip

- File Name -> mnt/ssd/kaggle-talkingdata2/competition_files/train.csv
- Compressed Size -> 1238.43 MB
- UnCompressed Size -> 7188.46 MB

We can see that the actual size of the train dataset is around 7.2GB.

Details of test.csv.zip

- File Name -> test.csv
- Compressed Size -> 161.93 MB
- UnCompressed Size -> 823.28 MB

After uncompressing the train and test dataset we can count the records present in both of them.

- Records in train.csv => 184903891
- Records in test.csv => 18790470

As can be seen above, there are around 185 Million records in the train data and around 18.8 Million records in the test set.

Sample Records From train.csv

	ip	app	device	os	channel	click_time	attributed_time	is_attributed
0	33748	18	1	10	134	2017-11-09 08:15:22	NaN	0
1	26810	3	1	25	489	2017-11-09 08:15:22	NaN	0
2	105587	21	1	13	128	2017-11-09 08:15:22	NaN	0
3	3542	2	1	13	435	2017-11-09 08:15:22	NaN	0
4	124339	3	1	41	211	2017-11-09 08:15:22	NaN	0

Sample Records From test.csv

	click_id	ip	app	device	os	channel	click_time
0	0	5744	9	1	3	107	2017-11-10 04:00:00
1	1	119901	9	1	3	466	2017-11-10 04:00:00
2	2	72287	21	1	19	128	2017-11-10 04:00:00
3	3	78477	15	1	13	111	2017-11-10 04:00:00
4	4	123080	12	1	13	328	2017-11-10 04:00:00

As can be seen from the above table, the train dataset contains details about each click including the 'IP Address', 'App ID', 'Device ID', 'Operating System', 'Channel ID', 'Click

Time', 'Attributed Time' and 'Is Attributed' Flag. 'Is Attributed' Flag is True when the user who has clicked the as actually downloaded the App and the 'Attributed Time' tells us the time when the App was downloaded.

As the dataset is very huge, we could use the **sqlite3** library to create a database from the csv file and do some quick analysis. Below is the Python code snippet that can be used to create a new database file from train.csv.

```
# Opens file if exists, else creates file
con = sqlite3.connect("talkingdata_train.db")

# This object lets us send messages to our DB and receive results
cur = con.cursor()
sql = "SELECT sql FROM sqlite_master WHERE name='test_data'"
cur.execute(sql)

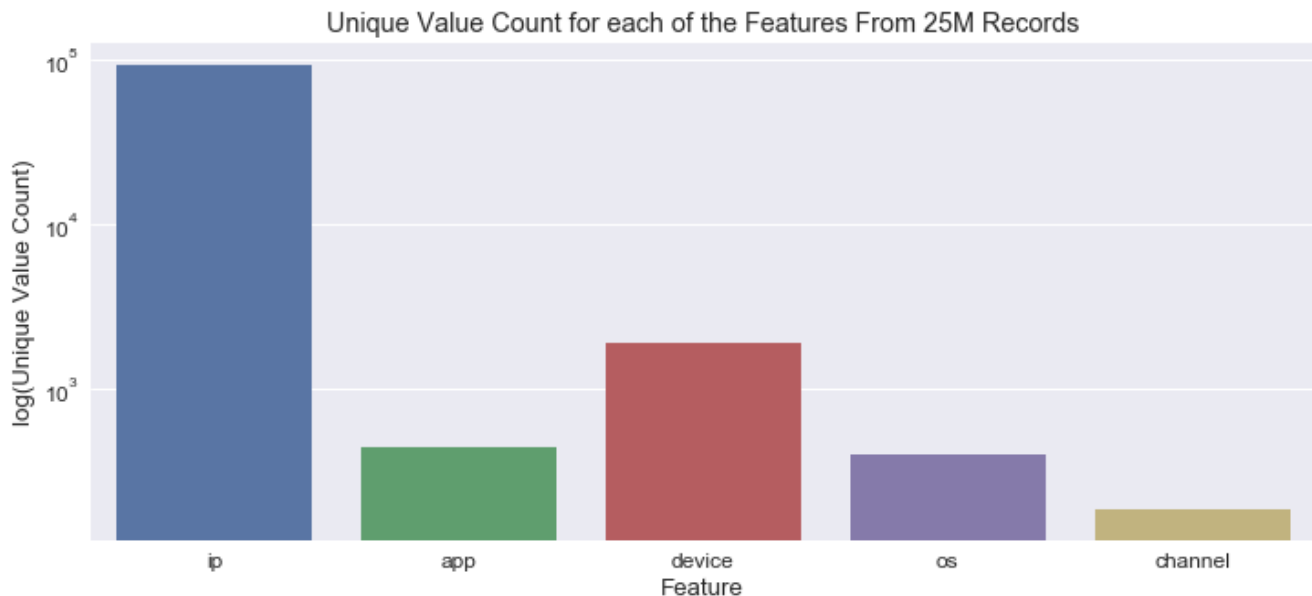
if not cur.fetchall():
    for chunk in pd.read_csv("train.csv", chunksize=5000):
        chunk.to_sql(name="train_data", con=con, if_exists="append",
index=False)  #"name" is name of table
        gc.collect()
```

Once the database file is created, we can execute SQL queries to get the required information. For example, we can use the below SQL statement to get the range of IP Addresses present in the train dataset.

```
sql = "select min(ip), max(ip) from train_data"
cur.execute(sql)
cur.fetchall()
```

The above SQL query returns (1, 364778) which will give us an idea on the range of values present for the 'IP Address'.

Below is a visualization of the unique values present for each of the features in the training data (Only the last 25 Million rows are considered for this project as it is very difficult to do feature engineering and model building on the complete dataset. We choose the last 25 Million rows as that would better represent this time series information.)



Algorithms and Techniques

As we have an imbalanced dataset with many records, using simple classification algorithms like `DecisionTreeClassifier` may not help us to get a better score. So, we will try to use an Ensemble Method that will do a better prediction.

There are many Ensemble methods including Bagging and Boosting and for our problem we will go with Boosting Ensemble. Boosting basically involves incrementally building an ensemble by training each new model instance to emphasize the training instances that previous models misclassified. The specific variation of the Boosting algorithm that we will be using for this problem is the Gradient Boosting method where the algorithm produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees.

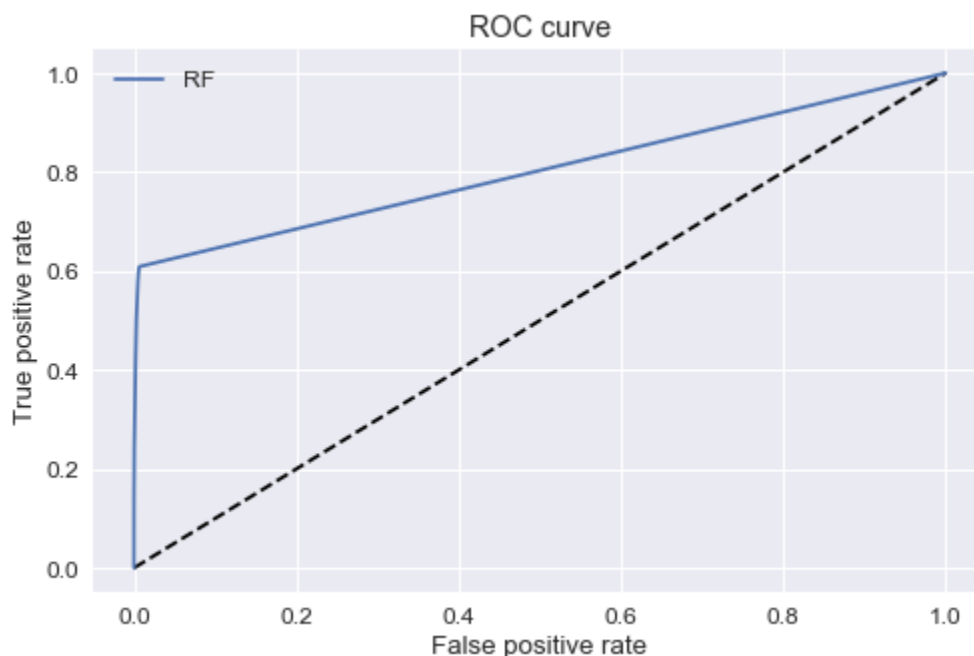
There are many Gradient Boosting algorithms like AdaBoost, xgboost, LightGBM and CatBoost. For this problem we will use the LightGBM library as it uses a lower memory footprint while working with large datasets and also does the model building faster than the other algorithms.

Advantages of LightGBM











- LightGBM uses histogram based algorithm for faster training speed and higher accuracy.
- LightGBM replaces continuous values to discrete bins which result in lower memory usage.
- LightGBM produces complex trees by following leaf wise split approach rather than a level-wise approach to achieve higher accuracy than other algorithms.
- LightGBM is capable of performing equally good with large datasets with a significant reduction in training time.
- Support for parallel learning.

Benchmark

We will build a simple RandomForestClassifier () on the training data without any feature engineering as a Benchmark Model. Later we will use the same training data to generate new features with Feature Engineering and then build a model using LightGBM. We should be getting a much better score with the LightGBM model compared to the naive RandomForestClassifier ().



Also, we can use the benchmark score provided by Kaggle as a reference. The model that we are going to build is finally used to predict the target for the test data provided for this competition. The submission file is then uploaded on to Kaggle to see that we do much better than the benchmark score.

2874	new	MatsudaRyo		0.9129	2	2d
2875	▼ 343	MohanManivannan		0.9128	51	6d
2876	▼ 342	Samira Karimzadeh		0.9126	1	10d
2877	▼ 341	ZZ87		0.9121	2	16d
2878	▼ 226	Jose A Gomez		0.9120	21	10h
	random_forest_benchmark.csv			0.9117		
2879	▼ 342	aqeelmistry		0.9112	5	2mo
2880	▼ 342	bobson		0.9102	7	6d
2881	▼ 342	Shyam Valsan		0.9099	7	1mo
2882	▼ 342	Mehdi Karoui		0.9095	2	1mo

Methodology

Data Preprocessing

From the train data we could see that only 'attributed_time' has NaN values. But as this attribute is related to the target - 'is_attributed', we can safely delete this feature from the training data. Also, as it is related to the target feature it won't be present in the test data and we are fine there.

One other feature that we are interested in preprocessing is the 'click_time'. As the data is spread across just few days, the day, month and year in 'click_time' doesn't add much value and can also cause an overfitting while building the model. So, we will just extract 'hour' and 'minute' from 'click_time' and use them for building our model.

Also, as we have very limited feature, we will try to use some Group By Transformation to create new Features. For example, we could Group By on 'ip' and 'os' and then get a count value for the same. This preprocessing to generate new features will help the algorithm to generate better predictions.

Implementation

1. Train and Test File Detail

As the train and test files are in zip format we will first use the zipfile library to get a glimpse of the dataset.

```
# Print details of the Train csv file.
for info in zf_train.infolist():
    print("File Name      -> {}".format(info.filename))
    print("Compressed Size -> {:.2f} {}".format(info.compress_size/(1024*1024), "MB"))
    print("UnCompressed Size -> {:.2f} {}".format(info.file_size/(1024*1024), "MB"))

File Name      -> mnt/ssd/kaggle-talkingdata2/competition_files/train.csv
Compressed Size -> 1238.43 MB
UnCompressed Size -> 7188.46 MB
```

```
# Print details of the Test csv file.
for info in zf_test.infolist():
    print("File Name      -> {}".format(info.filename))
    print("Compressed Size -> {:.2f} {}".format(info.compress_size/(1024*1024), "MB"))
    print("UnCompressed Size -> {:.2f} {}".format(info.file_size/(1024*1024), "MB"))

File Name      -> test.csv
Compressed Size -> 161.93 MB
UnCompressed Size -> 823.28 MB
```

Later, these train and test files are extracted for further processing. As the train file is very huge, we used sqlite3 library to create a database file. Later SQL queries are used to analyze the data.

Also, to lower the memory usage while feature engineering and model building, the range of values for each of the features are analyzed using SQL queries and the datatypes for each of the features are set manually.

```
# Data types of each of the features are chosen based on the value range instead of relying on the python interpreter.
```

```
dtypes = {
    'ip'           : 'uint32',
    'app'          : 'uint16',
    'device'       : 'uint16',
    'os'           : 'uint16',
    'channel'      : 'uint16',
    'is_attributed' : 'uint8',
    'click_id'     : 'uint32'
}
```

2. Read Train and Test Datasets

As we have huge records, processing them on a Laptop/Desktop with 8GB RAM will be impossible as we would need roughly 5x to 10x of memory to place the whole dataframe in memory during feature engineering or model building. So, we resorted to using the last 25 Million rows for our analysis, feature engineering and model building.

```
# Read the Last 25 Million records from training data
train_df = pd.read_csv("train.csv", skiprows=range(1,159903891), nrows=25000000, dtype=dtypes)
train_df.head()
```

	ip	app	device	os	channel	click_time	attributed_time	is_attributed
0	33748	18	1	10	134	2017-11-09 08:15:22	NaN	0
1	26810	3	1	25	489	2017-11-09 08:15:22	NaN	0
2	105587	21	1	13	128	2017-11-09 08:15:22	NaN	0
3	3542	2	1	13	435	2017-11-09 08:15:22	NaN	0
4	124339	3	1	41	211	2017-11-09 08:15:22	NaN	0

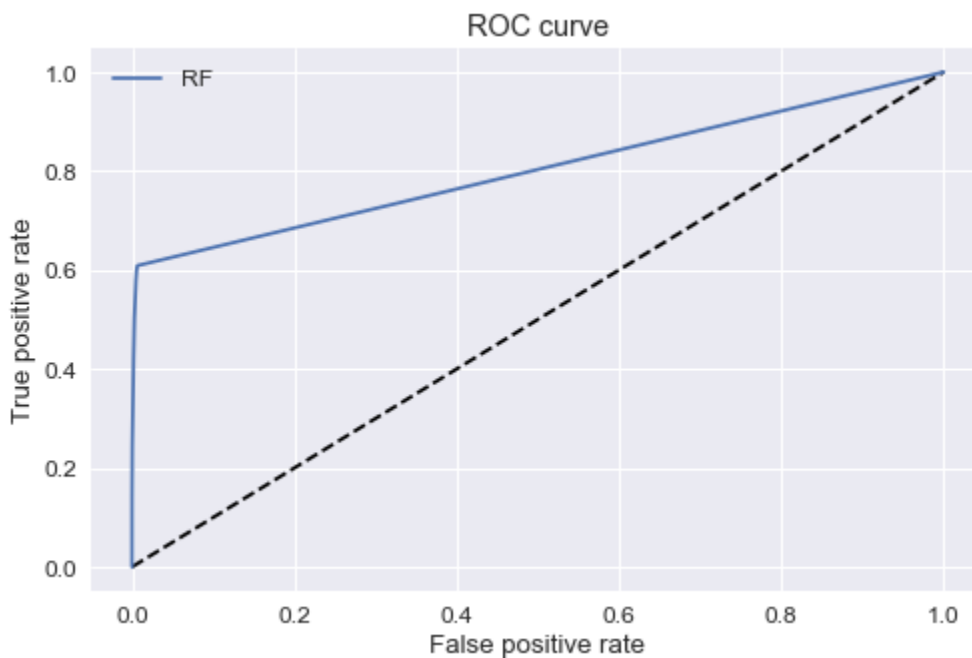
3. Naive Model

Before any feature engineering was done a naive model is built on the train data using which will serve as our benchmark along with the benchmark score on Kaggle.

```
rf = RandomForestClassifier()  
rf.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
                        max_depth=None, max_features='auto', max_leaf_nodes=None,  
                        min_impurity_split=1e-07, min_samples_leaf=1,  
                        min_samples_split=2, min_weight_fraction_leaf=0.0,  
                        n_estimators=10, n_jobs=1, oob_score=False, random_state=None,  
                        verbose=0, warm_start=False)
```

ROC AUC for the Naive Model



4. EDA and Feature Engineering

All the features present in the train dataset are explored to get a glimpse of the data. Below are few observations:

- There are a total of 91,647 different IP addresses present in the last 25M records. This means that there are multiple clicks from each of the IP addresses.

- From IP Address `5348` there are around `240k clicks`. Also, the top 20 IP addresses have more than 50k clicks from each one of them. One reasoning could be that many network providers dynamically assign the IP Addresses and it may not be the same user who is holding the IP address.
- There are a total of 435 different App ID's present in the last 25M records. This means that there will be huge number of clicks from each of the APP ID.
- From App ID `9` there are around `3M clicks`. Also, the top 7 App ID's each have more than 1M clicks.
- There are a total of 1905 different Devices present in the last 25M records.
- We can see that from Device `1` there are around `23M clicks`. Also, the top 5 Devices take the major share. It could be quite possible that the Android Devices are marked for Device ID - 1 as we know that Android has around 86% market share in the Mobile OS market.
- There are a total of 396 different Operating Systems present in the last 25M records.
- We can see that from `OS - 19` there are around `5.7M clicks` followed by `OS - 13` with around `5.2M Clicks`. Also, the top 5 Operating Systems take the major share with around 14M Clicks.

As the data is spread across just 3 days, we removed 'day', 'month' and 'year' from the 'click_time' and just retained the 'hour' and 'minute'. Including the 'year' and 'month' may not add much value for model building but including 'day' might cause an overfitting.

Also, as we have very minimal set of features, Group By Transformations are used to create new features.

```
# Use Group By Transformation to create new Features
print('grouping by ip-app combination [count]')
gp = train_df[['ip', 'app', 'channel']].groupby(by=['ip',
'app'])[['channel']].count().reset_index().rename(index=str,
columns={'channel': 'ip_app_count'})
train_df = train_df.merge(gp, on=['ip', 'app'], how='left')
del gp
gc.collect()
```

Refinement

During the initial analysis only the last 20 Million rows were used for analysis as there are frequent 'Out of Memory' issues. Later, instead of leaving it to the Python interpreter to determine the data types of each of the feature, SQL was used to get the range of values. With this approach of choosing the correct data type, helped in saving a lot of memory and we could use 25 Million rows without any issue.

From the train and test data it was understood that the data is spread across just few days. So, only the necessary properties like 'hour' and 'minute' are retained in 'click_time'. This has helped to get a better validation score and also a better score on the test data.

As we have very few features, ideas from related research work has helped to create new features using Group By Transformations. As the validation accuracy was increased with a single new feature, multiple of them were added in stages to improve the accuracy further.

Documentation of LightGBM has helped to fine tune the parameters like 'num_leaves', 'max_depth' and 'learning_rate' to get a better score.

Methodology

Model Evaluation and Validation

For the model building, we have used the last 2.5 Million rows from the training data as a validation set. This validation set will be used to get an understanding of the model and could be used to fine tune the model. Instead of building a model and submitting the result on Kaggle every time, we can use the validation score to see if there is an improvement. Once we see a better validation score, we can predict the target on the actual test data and submit the results on Kaggle to get a final score.

```
# Do the Train and Validation Split
VALIDATION_SIZE=2500000 #2.5M
TRAIN_SIZE = 25000000 #25M

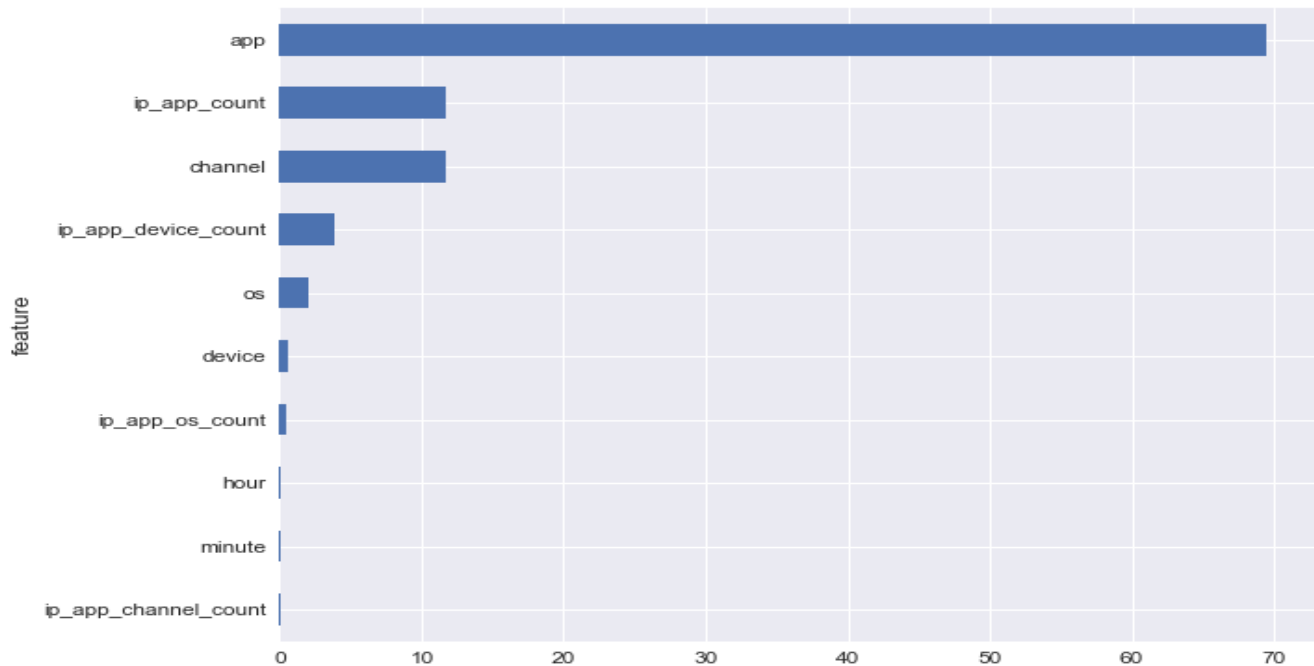
train = train_df[:(TRAIN_SIZE-VALIDATION_SIZE)]
val = train_df.iloc[(TRAIN_SIZE-VALIDATION_SIZE):TRAIN_SIZE]

X_train = train.drop(['is_attributed'], axis=1)
y_train = train['is_attributed']

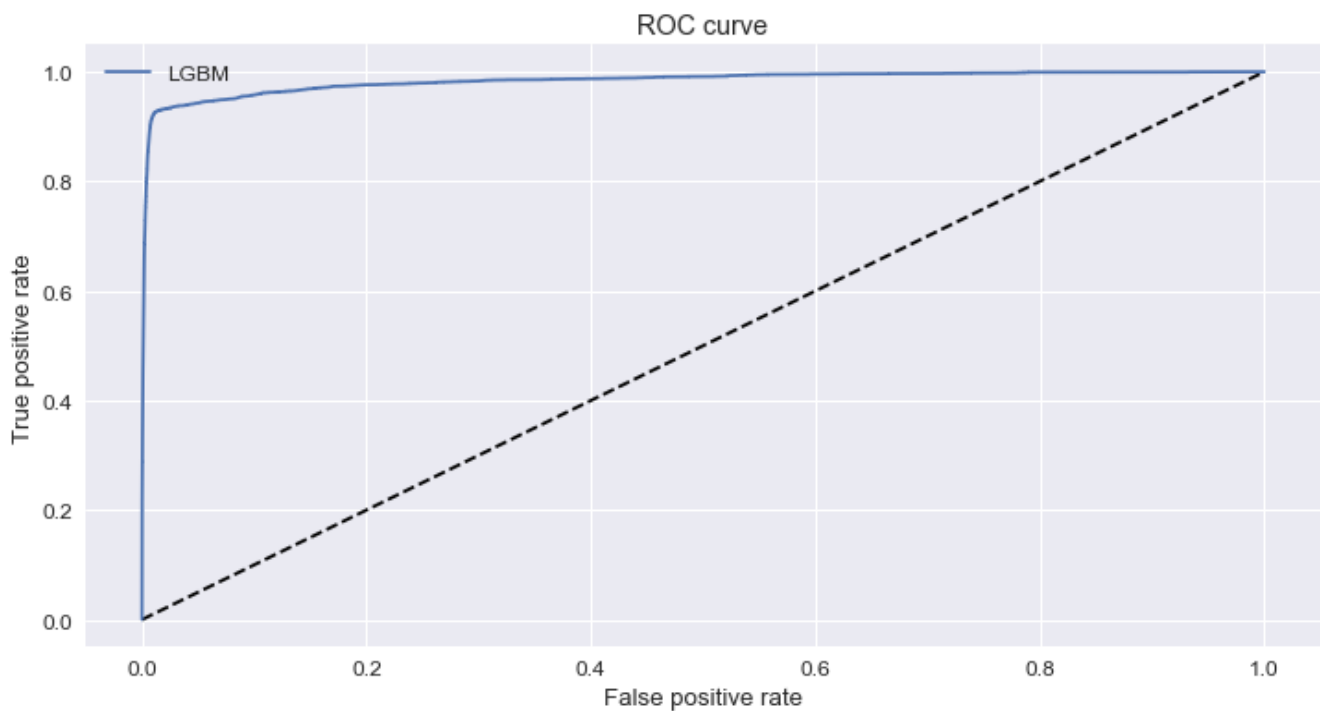
X_val = val.drop(['is_attributed'], axis=1)
y_val = val['is_attributed']
```

Based on the LightGBM documentation, the suggested parameter values are used in building the model. Based on the validation accuracy, parameters like 'num_leaves', 'max_depth' etc are changed to get a better score.

Feature importance is plotted so as to get a better understanding of the new features that are created during feature engineering. For example, a new feature like 'ip_app_os_count' was added to see if the validation score improves and also its importance in model building. As the validation score was increased and also the feature came top in the feature importance list, a submission on Kaggle was made to see the results. When the submission showed a better results, similar new feature was again added to the model and tested.



Finally, the ROC Curve is plotted to show how well our current is predicting the target compared to the naive model. We can also see this from the Validation Accuracy that is around 0.9837.



Justification

To check if our assumptions of better model is generated after adding new feature or after changing a parameter, the validation score improvement is checked. Also, the predictions were made on the test data and the results are submitted to Kaggle.

As can be seen below the current model has got a Public Score of **0.9635** which is much better than the benchmark score of **0.9117**.

All	Successful	Selected
Submission and Description		Public Score
submission.csv.gz 4 hours ago by Samrat Submission For Udacity Project		0.9635

Conclusion

Reflection

As we can see this is a very interesting and complex problem where we are trying to solve. With Millions of records just for 3 days we could image the amount of work that should go in to building a model on Billions of records. Apart from the amount of work, we should also think about the computing power that is required to process this massive data sets.

With the Public Score of 0.9635, which is much higher than the benchmark score of 0.9117, we should have done a good job in building a generalized model given the computing resources that we have at hand.

Improvement

One thing that would have improved our score is the high computing resources. A rough calculation of the data size reveals that we need around 100GB of memory to process all the records. Using all the records for feature engineering and model building will definitely help us to get a better score. Also, we could use automated Hyper Parameter libraries to get the best values for the LightGBM if we have had a computer with better processing capabilities as the model building on combination of parameters is very CPU intensive.

References:

- <https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection>
- https://en.wikipedia.org/wiki/Click_fraud
- <https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/data>
- <https://github.com/Microsoft/LightGBM>
- <http://xgboost.readthedocs.io/en/latest/>
- <https://mlwave.com/kaggle-ensembling-guide/>
- <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html>
- https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- <https://www.youtube.com/watch?v=OAl6eAyP-yo>
- <https://www.coursera.org/learn/machine-learning-projects>
- https://en.wikipedia.org/wiki/Training,_test,_and_validation_sets
- https://en.wikipedia.org/wiki/Gradient_boosting
- https://en.wikipedia.org/wiki/Ensemble_learning
- <https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/>
- <https://www.analyticsvidhya.com/blog/2017/03/imbalanced-classification-problem/>