Machine Learning Engineer Nanodegree

Capstone Project

Danko Komlen

January 13th, 2019

I. Definition

Project Overview

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.

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. While successful, they want to always be one step ahead of fraudsters and are searching for solution via Kaggle community to help in further developing their solution.

My interest in the particular problem is related to predictions on time series data. This is something close to my daily job and I'm interested to learn more about.

An interesting research done on this problem is available in paper Detecting click fraud in online advertising: a data mining approach[1]. Authors have organized a Fraud Detection in Mobile Advertising competition, opening the opportunity for participants to work on real-world fraud data. The task was to identify fraudulent publishers who generate illegitimate clicks, and distinguish them from normal publishers. Main findings were that features derived from fine-grained time series analysis are crucial for accurate

[1] Detecting Click Fraud in Online Advertising: A Data Mining Approach: http://www.jmlr.org/papers/volume15/oentaryo14a/oentaryo14a.pdf

fraud detection, and that ensemble methods offer promising solutions to highly-imbalanced nonlinear classification tasks. Also from the winner of the competition we can learn that detecting large and/or duplicate clicks and distinguishing between morning and night traffics are important indicators for fraudulent acts. It was shown that high click fractions from top 10 high-risk countries provide strong signals for click fraud.

Another novel approach on this problem was done in a paper A Novel Ensemble Learning-based Approach for Click Fraud Detection in Mobile Advertising[2]. The proposed model is evaluated in terms of the resulting precision, recall and the area under the ROC curve. A final ensemble model based on 6 different learning algorithms proved to be stable with respect to all 3 performance indicators.

Problem Statement

In the AdTracking Fraud Detection challenge[3], the goal is to build an algorithm that predicts whether a user will download an app after clicking on a mobile app ad. For every click on ad, TalkingData collects additional metadata information about the user, like IP and device. Using this data we want to build a system that will tell us the probability that the user will download the advertised app after looking through the ad. TalkingData would then be able to use such a system to detect click fraud and pay advertisers only for legitimate clicks. Approach that will be taken consists of traditional data science steps for modeling classifier systems, which include data exploration and visualisation, data preprocessing, model training and model evaluation. These steps will be iterated through as more knowledge of the problem is gained. For example if we find that our initial models are not performing well enough, we would go back to preprocessing step and try to engineer better additional features.

Metrics

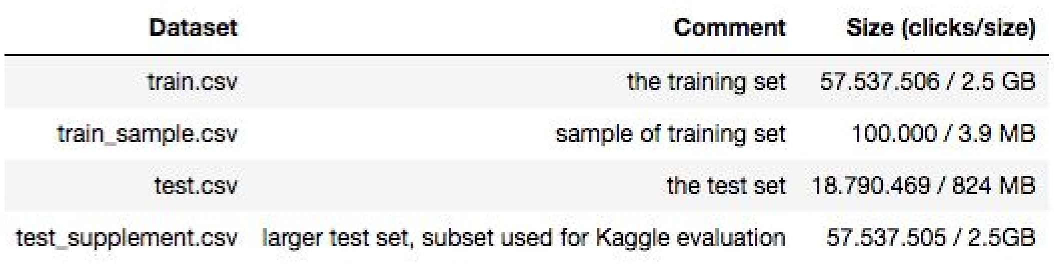
For the evaluation metric we will use area under ROC curve (AUC). This metric is suitable because the data is unbalanced in favour of app not downloaded class. AUC represents the probability that a random positive example has higher prediction than a random negative example.[4] AUC ranges in value from 0 to 1.

[2] A Novel Ensemble Learning-based Approach for Click Fraud Detection in Mobile Advertising: https://pdfs.semanticscholar.org/1518/6bd13f3f5a33598be9930d2e093055bb7c93.pdf
[3] Kaggle competition: https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection
[4] AUC: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

II. Analysis

Data Exploration

The data used in the project is the one provided by the TalkingData as part of their challenge in CSV file format[5]. Files contain time series of click records for the TalkingData ads.



Dataset	Comment	Size (clicks/size)
train.csv	the training set	57.537.506 / 2.5 GB
train_sample.csv	sample of training set	100.000 / 3.9 MB
test.csv	the test set	18.790.469 / 824 MB
test_supplement.csv	larger test set, subset used for Kaggle evaluation	57.537.505 / 2.5GB

Table 1 - Datasets overview

Every click record has following associated fields:

- click_time: timestamp of click (UTC)
- ip: ip address of click
- device: device type of user mobile phone
- os: os version id of user mobile phone
- channel: channel id of mobile ad publisher
- app: app id for marketing

Training data contains two additional fields:

- is_attributed: was app downloaded, to be predicted
- attributed_time: time of app download

We can see that train sample dataset (train_sample.csv) is considerably smaller than the full training data (train.csv) which is the main reason I decided to use it for exploration and training of the models. I also used sample from full training dataset as the benchmark test set by taking first 1.000.000 clicks from train.csv. It was done in same manner like in paper describing the benchmark model[6]. That way I could easily compare performance with my approach on same test data.

[5] Datasets on Kaggle: https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/data
[6] Benchmark model datasets: https://rpubs.com/el16/410747

	ip	app	device	os	channel	is_attributed
count	100000.000000	100000.00000	100000.000000	100000.000000	100000.000000	100000.000000
mean	91255.879670	12.04788	21.771250	22.818280	268.832460	0.002270
std	69835.553661	14.94150	259.667767	55.943136	129.724248	0.047591
min	9.000000	1.00000	0.000000	0.000000	3.000000	0.000000
25%	40552.000000	3.00000	1.000000	13.000000	145.000000	0.000000
50%	79827.000000	12.00000	1.000000	18.000000	258.000000	0.000000
75%	118252.000000	15.00000	1.000000	19.000000	379.000000	0.000000
max	364757.000000	551.00000	3867.000000	866.000000	498.000000	1.000000

Table 2 - Dataset attributes overview on training set

By looking at the mean value of the is_attributed field, we can see that our training set is heavily unbalanced in favour of non attributed clicks (99.8% of data). That is, most of the clicks don't result in downloading the app. Since all of the features are categorical, we can also check number of unique values for each of them (table 3).

attribute	unique values
ip	34857
app	161
device	100
os	130
channel	161
attributed_time	227
is_attributed	2

Table 3 - Attributes cardinality on training set

As it can be seen ip has highest cardinality while the rest of attributes are much smaller and similar in number of possible values.

	ip	app	device	os	channel	is_attributed
ip	1	0.0103998	-0.00149891	-0.000469915	0.00756287	0.0549551
app	0.0103998	1	0.248376	0.24716	-0.0282377	0.064426
device	-0.00149891	0.248376	1	0.924456	-0.0353978	-0.000695177
os	-0.000469915	0.24716	0.924456	1	-0.0331755	0.00618346
channel	0.00756287	-0.0282377	-0.0353978	-0.0331755	1	-0.0233364
is_attributed	0.0549551	0.064426	-0.000695177	0.00618346	-0.0233364	1

Table 4 - Correlation matrix done on training set

Simple exploration is also done by computing correlation matrix, which can easily show if any of attributes is correlated with target class. As it can be seen this is not the case, none of the attributes directly contributes to target class. We can also see that device and OS have high degree of correlation. This is expected as most devices usually come with same OS, like iPhone for example.

Exploratory Visualization

First approach taken in visualisation of the training data was making the histograms for each feature. These graphs show already seen imbalance of target is_attributed class. They also show similar imbalance in app, device and os features. On the other side channel and ip features have more balanced distribution across all values.

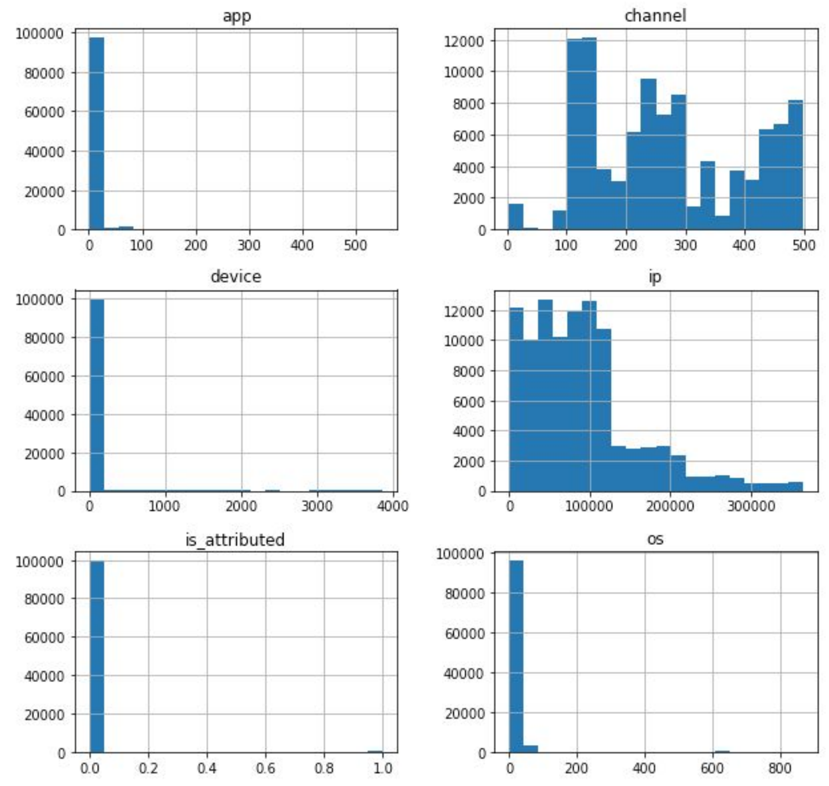


Image 1 - Attribute histograms done on training data

Similar histogram visualisation was done for attributed events, and as it was expected from the correlation matrix seen before, there are no features that clearly distinguish positive vs negative samples.

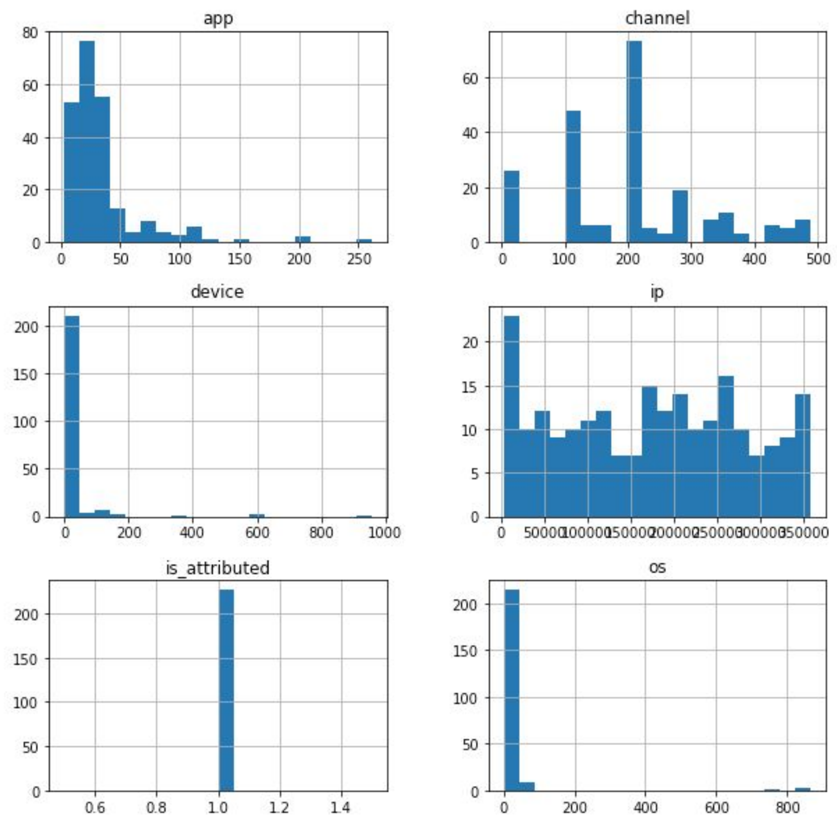


Image 2 - Attribute histograms for positive events

Next visualisation done was distribution of attributed events over time. As it can be seen there is a noticeable pattern when attributed events occur, which indicates that time of day can be a good additional feature to be extracted in preprocessing.

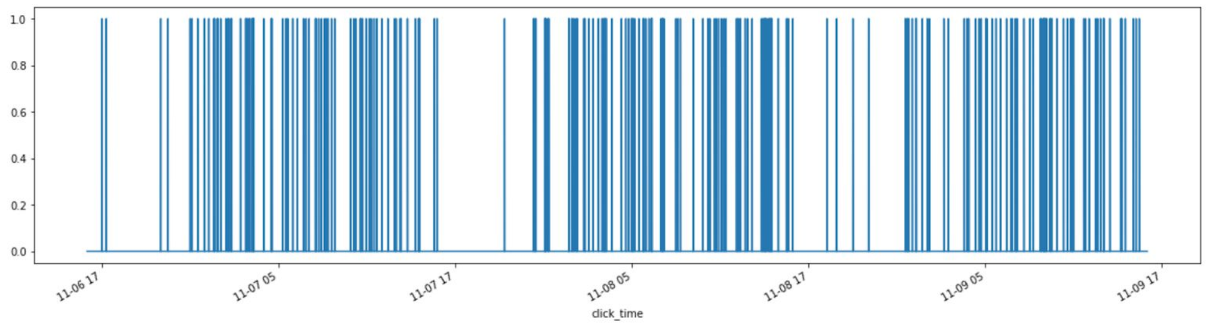


Image 3 - is_attributed class shown over time

Algorithms and Techniques

Algorithms applicable for the problem are any supervised learning classification algorithms. Algorithms used in the project are: Logistic Regression, Gaussian Naive Bayes, Decision Tree, Gradient Boosting and Random Forest. Default parameters used and implementation details can be seen in the table below.

Classifier algorithm	**Implementation**	**Default parameters**
Logistic Regression	sklearn.linear_model.LogisticRegression	C=1.0, max_iter=100, penalty='l2', random_state=42, solver='liblinear', tol=0.0001
Gaussian Naive Bayes	sklearn.naive_bayes.GaussianNB	priors=None
Decision Tree	sklearn.tree.DecisionTreeClassifier	class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=42,splitter='best'
Gradient Boosting	sklearn.ensemble.GradientBoostingClassifier	criterion='friedman_mse', learning_rate=0.1, loss='deviance', max_depth=3, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, presort='auto', random_state=42, subsample=1.0
Random Forest	sklearn.ensemble.RandomForestClassifier	class_weight=None, criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, 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=42

Table 5 - algorithms and default parameters

Logistic regression uses logistic function to predict a value from [0,1] range given input features x and parameters Θ:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}},$$

It is usually faster in execution compared to SVM, which is why I chose it for this problem. Liblinear implementation used here is also very efficient on large-scale datasets[7], like the one in this problem. For Gaussian Naive Bayes classifier the likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters are estimated using maximum likelihood. Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. Next, decision trees are a non-parametric supervised learning method where the goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. One of the advantages of DTs is ability to handle both numerical and categorical data, which are both present in our datasets. But on the other hand they tend to create biased trees if some classes dominate, which is also true in our case. Gradient boosting classifier is a generalization of boosting to arbitrary differentiable loss functions. It uses n_estimators number of shallow regression trees as weak learners. On the other hand, random forest classifier uses fully grown decision trees[8]. Gradient boosted trees generally perform better than a random forest, although there is a price for that: GBT have a few hyper params to tune, while random forest is practically tuning-free.[9]

Every algorithm will be trained on the preprocessed sample train set (train_sample.csv). All algorithm parameters will be set to implementation defaults and random state will be fixed so that the research can be easily reproduced. After initial training phase, every model will be evaluated on the benchmark test set using the AUC evaluation metric. Model with best score will be further optimised by hyperparameter tuning using grid search, 5-fold stratified cross validation and AUC metric. Optimised model will then be evaluated on benchmark test set (1.000.000 rows from train.csv) and results will be compared to benchmark model. Additionally model will be evaluated on larger test set (test.csv) and results will be compared on Kaggle leaderboard.

[7] LIBLINEAR: A Library for Large Linear Classification:
https://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf
[8] Gradient Boosting Tree vs Random Forest:
https://stats.stackexchange.com/questions/173390/gradient-boosting-tree-vs-random-forest
[9]What is better: gradient-boosted trees, or a random forest?
http://fastml.com/what-is-better-gradient-boosted-trees-or-random-forest/

Benchmark

Constant model that predicts no app download scores 0.5 on Kaggle evaluator. Random model that uses uniform distribution scores 0.4995 as private score (82% of test data) and 0.5003 as public score (18% of the test data). One approach to this problem is done by Elior Tal and his solution scores 0.825 on a subset of the test set[10]. In his work he compares performance of random forest and boosting algorithms. We will use this as a benchmark model as it is well documented and scores better than our initial baseline models. Another evaluation of the approach will be done against existing Kaggle solutions on the leaderboard[11].

III. Methodology

Data Preprocessing

In preprocessing part of the project unnecessary features were removed and new features were extracted in order to provide more information to classifier systems. Additionally data is also sorted by click_time first, because events are meant to be processed in time order.

Removed features are:

- **attributed_time**,
- **click_time** and
- **is_attributed**.

is_attributed is removed as it is the target class to be predicted, attributed_time was not considered as it is only available for attributed events in training data. Click_time was removed as it has high cardinality and instead I decided to extract just the hour of the day from it to capture the information about the periodical distribution of attributed events over time.

New features that were added as part of the preprocessing are:

- **hour** - Hour of the day

[10] Benchmark model: https://rpubs.com/el16/410747
[11] Kaggle leaderboard: https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/leaderboard

- **last_[attr]** - duration since last click with that attribute in seconds. attr is ip, app, os, channel and device

Last attributes were used to capture information about uniqueness of particular feature value. Duration was capped at maximum 1 day (86400 seconds), to reduce the cardinality of values since data is spanning over 1 week. 1 day was also used as a value for values that were first seen. As a last step all last_attr features are scaled to [0,1] interval.

ip	app	device	os	channel	hour	last_ip	last_app	last_os	last_channel	last_device
85592	12	1	13	145	15	0.130498	0.000220	0.000035	0.000035	0.000012
91779	18	1	41	379	15	1.000000	0.000104	0.001424	0.000556	0.000012
81374	14	1	25	118	15	1.000000	0.000764	0.000613	0.007731	0.000012
11911	1	1	22	115	15	1.000000	0.002373	0.000382	0.019086	0.000023
44018	13	1	19	477	15	0.387813	0.002211	0.000336	0.001377	0.000058

Table 6 - sample preprocessed events

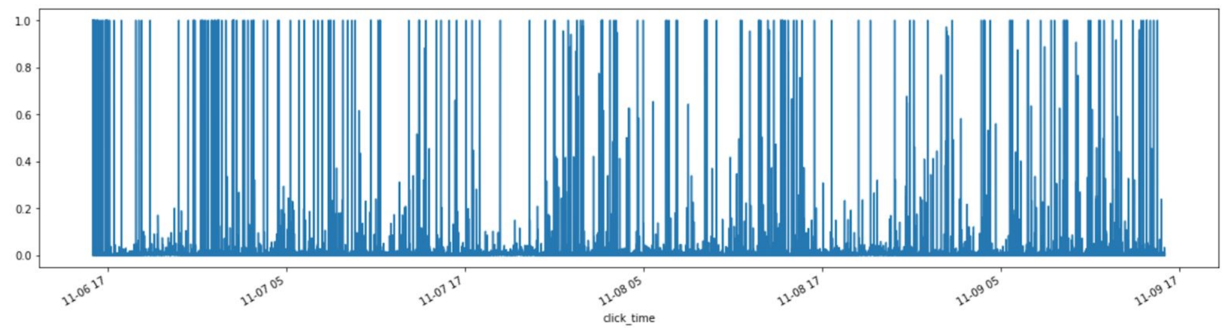
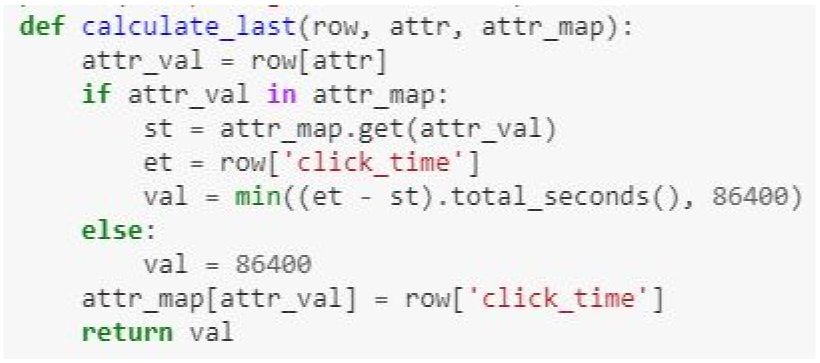


Image 4 - last_app attribute from train dataset

Implementation

Implementation was done using Jupyter notebook with Python 3.7 kernel. After data exploration and visualisation work, first part was preprocessing data and generating datasets for model training. This was done in a generic way with prepare_dataset() function that is used for preparing both training and all test datasets. The function extracts hour of the day attribute, removes unnecessary attributes: is_attributed, attributed_time and click_time and calculates new _last attributes by calculating duration since last click with same ip, app, os, channel and device attributes. This can be seen in more details in the code snippet below:



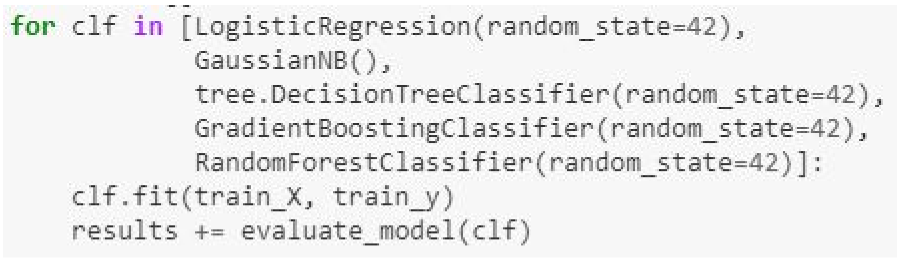
```python
def calculate_last(row, attr, attr_map):
    attr_val = row[attr]
    if attr_val in attr_map:
        st = attr_map.get(attr_val)
        et = row['click_time']
        val = min((et - st).total_seconds(), 86400)
    else:
        val = 86400
    attr_map[attr_val] = row['click_time']
    return val
```

Image 5 - code for calculating _last attributes

One of the challenges was getting preprocessing part be more performant and have an easy way to track progress. I found a solution by using Python's multiprocessing package for creating job pools[12] and tqdm[13] open source package for showing progress bars inside Jupyter notebooks. In the evaluation part main complication was training and evaluating multiple different models and being able to iterate fast when making changes on preprocessing part. To overcome this I found it useful to have generic functions for main parts of the pipeline, which were following:

- **def** read_dataset(path, nrows=**None**):
- **def** prepare_dataset(df):
- **def** evaluate_model(clf, tstX=test_X, tsty=test_y, verbose=**True**):
- **def** optimize_model(estimator, params, cv):

This way I could easily evaluate optimized model on several different test sets and trying different algorithms by simply adding them in following place:



```python
for clf in [LogisticRegression(random_state=42),
            GaussianNB(),
            tree.DecisionTreeClassifier(random_state=42),
            GradientBoostingClassifier(random_state=42),
            RandomForestClassifier(random_state=42)]:
    clf.fit(train_X, train_y)
    results += evaluate_model(clf)
```

Image 6 - code for evaluating different classifiers

[12] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing.pool
[13] https://github.com/tqdm/tqdm

After preparation of training and test datasets, all five models with default parameters were trained on the training set and evaluated using AUC score on test set. Results and confusion matrices for each can be seen below.

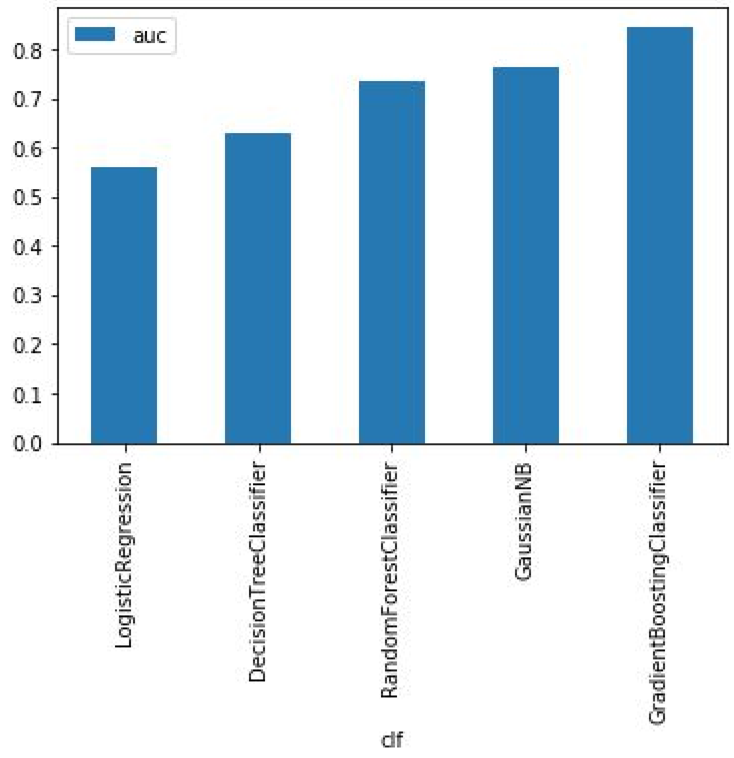
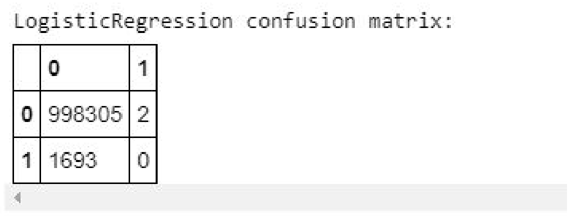
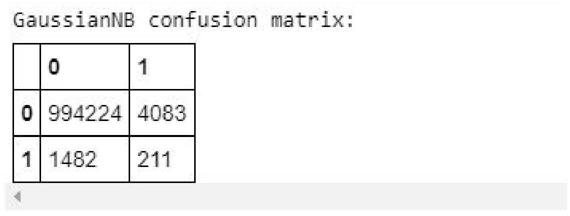


Image 7 - AUC scores from initial model evaluation



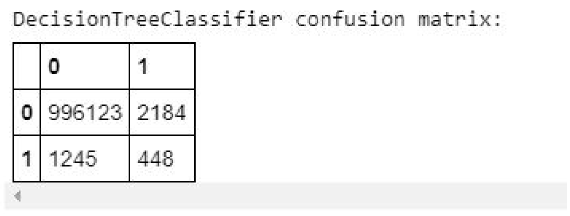
```
LogisticRegression confusion matrix:
```

	0	1
0	998305	2
1	1693	0



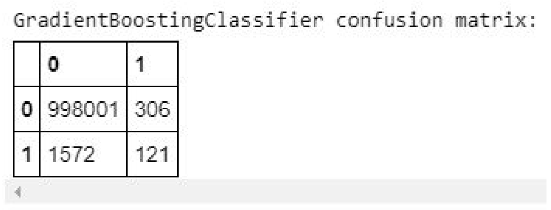
```
GaussianNB confusion matrix:
```

	0	1
0	994224	4083
1	1482	211



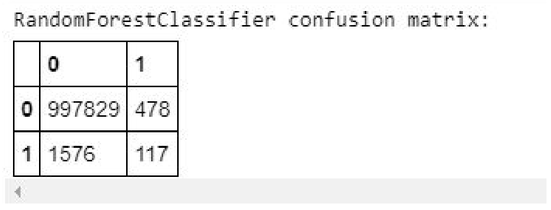
```
DecisionTreeClassifier confusion matrix:
```

	0	1
0	996123	2184
1	1245	448



```
GradientBoostingClassifier confusion matrix:
```

	0	1
0	998001	306
1	1572	121



```
RandomForestClassifier confusion matrix:
```

	0	1
0	997829	478
1	1576	117

Image 8 - Confusion matrices from initial model evaluation

Best model was Gradient boosting classifier with 0.84 AUC score.

Refinement

Based on the initial AUC score Gradient boosting classifier was then taken to next step of hyperparameter tuning using grid search with stratified 5-fold cross validation. Following parameters were tuned:

- 'loss' : ['deviance', 'exponential'],
- 'learning_rate': [0.1, 0.2, 0.3],
- 'n_estimators': [50, 100, 200],

After optimizing the parameters, trained model was evaluated again on test set and has achieved AUC score of 0.87 with parameters: learning_rate=0.1, loss='deviance' and n_estimators=100.

Biggest runtime bottleneck for training models was related to preprocessing part and generation of new attributes. Improvement was done by introducing parallelisation to preprocessing step, improvements are shown in table below.

	Train dataset	**Test dataset**
Sequential implementation	1min 3s	9min 39s
Parallel implementation	21.7 s	3min 58s

Table 7 - Preprocessing runtime improvements

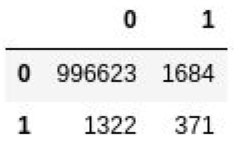
Another improvement was parallelising grid search which reduced parameter search time from 10.2 minutes to 8.7 minutes.

IV. Results

Model Evaluation and Validation

Final model was Gradient Boosting classifier, chosen after initial evaluation of five different classifier types and it was derived by doing hyperparameter tuning using grid search as explained in previous sections. Final parameters are reasonable and appropriate: criterion='friedman_mse', learning_rate=0.1, loss='deviance', max_depth=3, min_impurity_decrease=0.0, min_samples_leaf=1, min_samples_split=2, n_estimators=100.

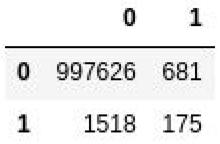
Model scored 0.87 AUC on benchmark test set, 0.892 on Kaggle private test set and 0.889 on Kaggle public test set.



	0	1
0	996623	1684
1	1322	371

Image 9 - Final model confusion matrix on benchmark test set

To test robustness of the model, new test dataset was prepared by modifying click time on original benchmark test dataset. Click times were moved by adding time delta from [-120, 120] integer minutes using uniform distribution. Model scored 0.870602 AUC score on modified datasets which shows good robustness on event ordering and its results can be trusted.



	0	1
0	997626	681
1	1518	175

Image 10 - Final model confusion matrix on modified benchmark test set

Justification

Final model results are better than simple constant and random benchmarks and also better then Elior Tal benchmark model.

Test set	**Bench. const**	**Bench. rand**	**Bench. Elior Tal**	**Final solution**
Benchmark	N/A	N/A	0.825	0.874531
Kaggle private	0.5	0.4995	N/A	0.8919713
Kaggle public	0.5	0.5003	N/A	0.8888370

Table 8 - Result comparison against benchmark models

Significance of the solution was tested using bootstrap confidence intervals[14]. Model scored 95% confidence interval of 86.52% and 88.49% using 1000 bootstraps with resampling and replacement on benchmark test set.

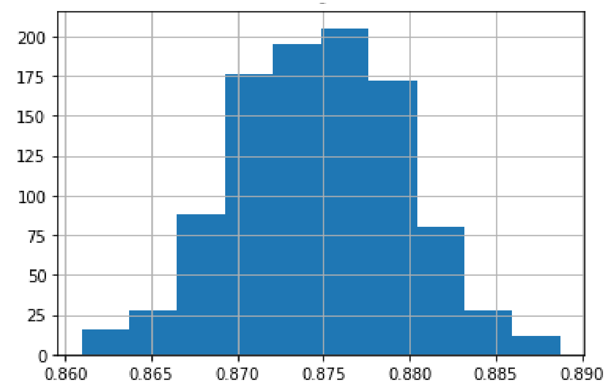


Image 11 - Histogram of AUC bootstrap scores

V. Conclusion

Free-Form Visualization

Quality of the project can be discussed first in aspect to benchmark solutions. Having better score on same test set indicates that preprocessed features and chosen model are a good starting point for further research in this area. Good part is also the robustness and significance of final model which is something that was not evaluated in the project for the benchmark model.

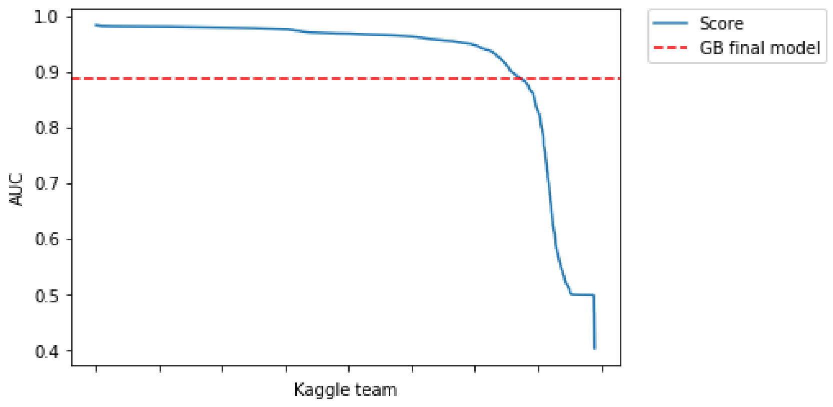


Image 12 - Final model comparison against Kaggle scores

[14] Bootstrap confidence intervals approach: https://machinelearningmastery.com/calculate-bootstrap-confidence-intervals-machine-learning-results-python/

We can also evaluate the quality of the solution against all public scores on related Kaggle competition (image 11). Although the score of 0.89 is pretty high we can see that majority of teams (over 85%) achieved better score. Model ranks 3360 of 3951 total public scores. So there is still a lot of room for improvement. Best public score achieved is 0.9834933 by team bestfitting.

Reflection

Process used in solving the problem consisted of several steps. First was data exploration step, where main features of the data were observed. This lead into data preprocessing part, where additional features were constructed from time series events. After that five different models were trained and AUC scores were compared. Preprocessing step was iterated on several times until ideas for new features started giving better results. After initial model training Gradient Boosting classifier was picked for hyperparameter tuning using grid search approach which resulted in final model. This model was evaluated by comparing with benchmark models, by testing on modified input data and by calculating bootstrap confidence intervals.

Most interesting part of the project was preprocessing part and evaluating ideas for new features to be inferred from the data. This was also the difficult part as it involved writing custom code and optimisations to be more performant. Final model and solution fits my expectations with small changes that I didn’t account for. Architecture of the solution has to be modified from my first proposal idea. Preprocessing step is something I didn’t account for in the beginning, marked green on bellow image 12. It can be used in general setting as long as the preprocessing scales to number of incoming events.

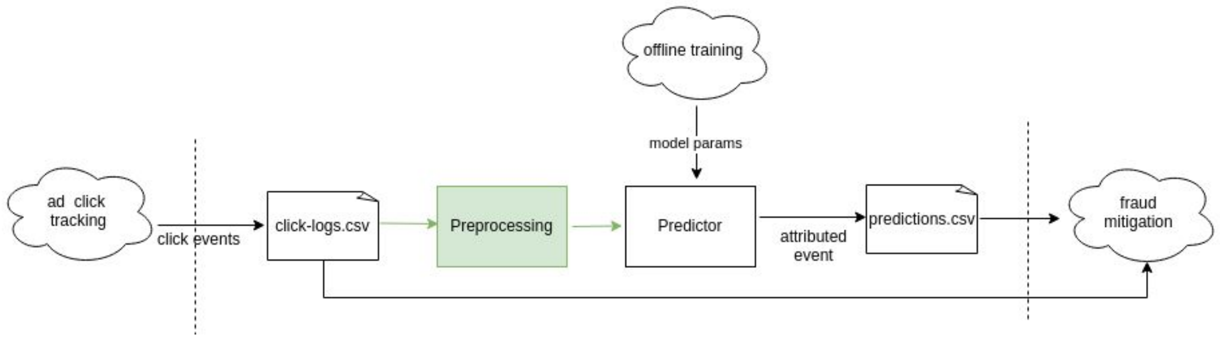


Image 13 - End-to-end project solution

Improvement

Improvements that can be done on showed solution include more exhaustive hyper parameter search. More model parameters can be added to search as well as wider range of possible values. All with a cost of longer search runtime. Execution time of different steps can be improved by using cloud based solutions like AWS SageMaker [15] and GPU accelerated processing.

Better model scores could also be achieved by using larger training set which can be processed with higher computing power. Additionally boosting method can be tried to get even better performance by combining multiple models.

From image 11 we can see that there are plenty of solutions that achieve higher scores on Kaggle. Some of them are available to public through Kaggle kernels, so this is also something that can be used to get even more ideas for improvement of the solution.

[15] AWS SageMaker https://aws.amazon.com/sagemaker/