# Project: AdTracking Fraud Detection

Janit Bidhan (jbidhan@gmu.edu) Sreenivasa Rayaprolu (srayapr@gmu.edu)

## INTRODUCTION:

In this project we aim to solve a problem of classification. We are worked on a ML engine that should identify fraudulent clicks while considering the click time, IP address, and other factors. Less than 1% of the clicks were fraudulent, hence the data was severely skewed. So, to give a gist we are predicting if an IP address that are generating fraudulent clicks and it is challenging since the data is massive and skewed towards non fraudulent IP addresses.

## MOTIVATION:

Click fraud affects businesses that advertise online alarmingly resulting in wasted money and increase in the fake click numbers. Click fraud may be extremely costly for internet advertising firms. With the proliferation of smart phones and Internet gadgets, it is critical to keep an eye on this, as Advertising networks may hike pricing if just many individuals click on the advertisement.

## DATASET:

The dataset being utilized in this project from a big data service platform(TalkingData) from China that has tracked these frauds by recording the number of clicks, click time and their IP Addresses over an app and checking if they end up installing the advertised application. The dataset (train.csv) is a huge (7.54GB) file with more than 1900 Million records having just 4 Million fraudulent labels which is way too less.

Description of each column:

- **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

## EXPLORATORY DATA ANALYSIS | PREPROCESSING

On running data analysis over the dataset, we found that:

- There were a lot of null values in the 2 timestamp columns i.e., "attributed_time","click_time", so we dropped them. Also, “attributed_time” has direct correlation with the label. So, the model will not learn anything if we keep that.
- We also found that there were a lot of duplicate rows in the train file provided by the dataset. So, taking unique made sense.
- All the columns except the label (“is_attributed”) are categorical columns which are replaced with integer id for the same entity.
- The right-side shows summary of data.
- The ratio on the right side, means for each 1 count Fraud row there are 403 Not Fraud rows present. This is very high and show how the data is skewed.

Number of unique values in each column:

```
+------+---+------+---+-------+-------------+
|    ip|app|device| os|channel|is_attributed|
+------+---+------+---+-------+-------------+
|277396|706|  3475|800|    202|            2|
+------+---+------+---+-------+-------------+
```

Summary of 7+GB data:
Label Counts Count Fraud: 456846,
Count Not Fraud: 184447044 >>> Ratio: 403
Summary of 7+GB data for unique rows:

```
Total:  54025901
Left After Dropping: 54025901
% Of Drops:  0.0
```

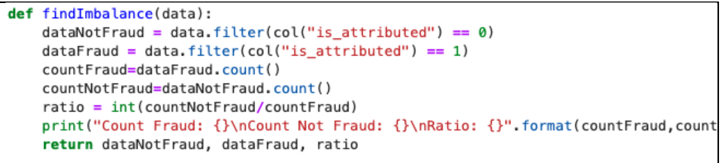
Label Counts: Count Fraud: 423712,
Count Not Fraud: 53602189 >>> Ratio: 126

We did data preprocessing to just take unique non-null rows. Now, once the ratio was dropped from 403 to 126, which is better, but we still need to manage the data imbalance. So, next we discuss the sampling techniques that were used next to balance the data further.

## SAMPLING TECHNIQUES

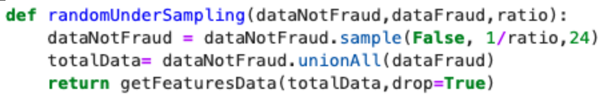
One of the bigger tasks for us was to create functions for various sampling techniques without using outside libraries. We used various sampling techniques to balance the imbalanced data. We have custom functions that we used to balance the data. We are making use of DataFrame.sample method when need to randomly sample. Each of the functions are explained below:

We used this common method to find imbalance ratio between the two class labels. The ratio was a key factor in deciding sampling techniques to be used.



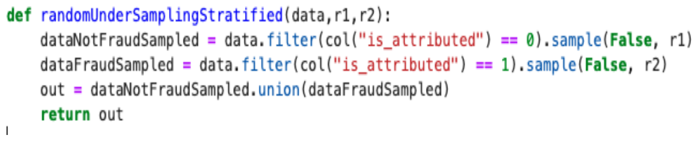
```python
def findImbalance(data):
    dataNotFraud = data.filter(col("is_attributed") == 0)
    dataFraud = data.filter(col("is_attributed") == 1)
    countFraud=dataFraud.count()
    countNotFraud=dataNotFraud.count()
    ratio = int(countNotFraud/countFraud)
    print("Count Fraud: {}\nCount Not Fraud: {}\nRatio: {}".format(countFraud,count
    return dataNotFraud, dataFraud, ratio
```

1.**Random Undersampling** means to randomly delete examples in the majority class. So, we drop off randomly rows as per the ratio.



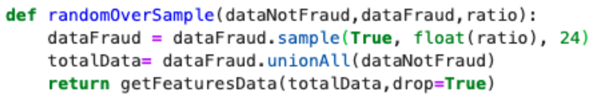
```python
def randomUnderSampling(dataNotFraud,dataFraud,ratio):
    dataNotFraud = dataNotFraud.sample(False, 1/ratio,24)
    totalData= dataNotFraud.unionAll(dataFraud)
    return getFeaturesData(totalData,drop=True)
```

2.**Stratified Sampling** is a method for sampling from a population whereby the population is divided into subgroups and units are randomly selected from the subgroups.



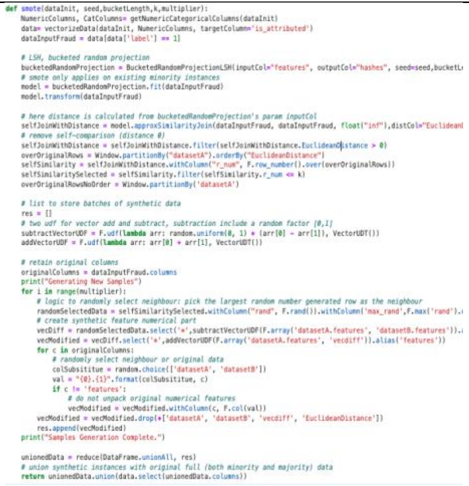
```python
def randomUnderSamplingStratified(data,r1,r2):
    dataNotFraudSampled = data.filter(col("is_attributed") == 0).sample(False, r1)
    dataFraudSampled = data.filter(col("is_attributed") == 1).sample(False, r2)
    out = dataNotFraudSampled.union(dataFraudSampled)
    return out
```

3.**Random Oversampling** means to randomly duplicate examples in the minority class. So, we randomly add rows as per the ratio.



```python
def randomOverSample(dataNotFraud,dataFraud,ratio):
    dataFraud = dataFraud.sample(True, float(ratio), 24)
    totalData= dataFraud.unionAll(dataNotFraud)
    return getFeaturesData(totalData,drop=True)
```

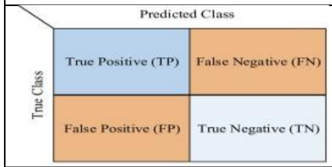
4.**Synthetic Minority Oversampling Technique (SMOTE)** is a technique to up-sample the minority classes while avoiding overfitting. It does this by generating new synthetic examples close to the other points (belonging to the minority class) in feature space. We wrote our custom logic to find out the nearest points and generate synthetic examples. The following algorithm was taken from the research paper mentioned in the references. The algorithm translates to code on the right-side. (See code for exact implementation.)



```python
def smote(dataInit, seed,bucketLength,k,multiplier):
    NumericColumns, CatColumns= getNumericCategoricalColumns(dataInit)
    data= vectorizeData(dataInit, NumericColumns, targetColumn='is_attributed')
    dataInputFraud = data[data['label'] == 1]

    # LSH, bucketed random projection
    bucketedRandomProjection = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", seed=seed,bucketL
    # smote only applies on existing minority instances
    model = bucketedRandomProjection.fit(dataInputFraud)
    model.transform(dataInputFraud)

    # here distance is calculated from bucketedRandomProjection's param inputCol
    selfJoinWithDistance = model.approxSimilarityJoin(dataInputFraud, dataInputFraud, float("inf"),distCol="Euclidean
    # remove self-comparison (distance 0)
    selfJoinWithDistance = selfJoinWithDistance.filter(selfJoinWithDistance.EuclideanDistance > 0)
    overOriginalRows = Window.partitionBy("datasetA").orderBy("EuclideanDistance")
    selfSimilarity = selfJoinWithDistance.withColumn("r_num", F.row_number().over(overOriginalRows))
    selfSimilaritySelected = selfSimilarity.filter(selfSimilarity.r_num <= k)
    overOriginalRowsNoOrder = Window.partitionBy('datasetA')

    # list to store batches of synthetic data
    res = []
    # two udf for vector add and subtract, subtraction include a random factor [0,1]
    subtractVectorUDF = F.udf(lambda arr: random.uniform(0, 1) * (arr[0] - arr[1]), VectorUDT())
    addVectorUDF = F.udf(lambda arr: arr[0] + arr[1], VectorUDT())

    # retain original columns
    originalColumns = dataInputFraud.columns
    print("Generating New Samples")
    for i in range(multiplier):
        # logic to randomly select neighbour: pick the largest random number generated row as the neighbour
        randomSelectedData = selfSimilaritySelected.withColumn("rand", F.rand()).withColumn('max_rand',F.max('rand').
        # create synthetic feature numerical part
        vecDiff = randomSelectedData.select('*',subtractVectorUDF(F.array('datasetA.features', 'datasetB.features')).
        vecModified = vecDiff.select('*',addVectorUDF(F.array('datasetA.features', 'vecdiff')).alias('features'))
        for c in originalColumns:
            # randomly select neighbour or original data
            colSubsititue = random.choice(['datasetA', 'datasetB'])
            val = "{0}.{1}".format(colSubsititue, c)
            if c != 'features':
                # do not unpack original numerical features
                vecModified = vecModified.withColumn(c, F.col(val))
        vecModified = vecModified.drop(*['datasetA', 'datasetB', 'vecdiff', 'EuclideanDistance'])
        res.append(vecModified)
    print("Samples Generation Complete.")

    unionedData = reduce(DataFrame.unionAll, res)
    # union synthetic instances with original full (both minority and majority) data
    return unionedData.union(data.select(unionedData.columns))
```

## EVALUATION METRICS:

We used the following these evaluation metrics for comparing the results:

| ROC | Precision | Recall |
|---|---|---|
| An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. | Precision refers to the number of documents correctly assigned to class A compared to the total number of documents predicted as belonging to class A. (total predicted positive) $\text{Precision} = \frac{tp}{tp + fp}$ | The recall refers to the number of documents correctly assigned to class A compared to the total number of documents belonging to class A (total true positive). $\text{Recall} = \frac{tp}{tp + fn}$ |
| Confusion Matrix | Accuracy | F1 Score |
|   Predicted Class<br>True Class<br>True Positive (TP)<br>False Negative (FN)<br>False Positive (FP)<br>True Negative (TN) | Accuracy allows you to know the proportion of good predictions compared to all predictions. $\text{Accuracy} = \frac{tn + tp}{tn + fp + fn + tp}$ | The F1-Score subtly combines precision and recall. $\text{F1-Score} = 2 \frac{\text{precision} * \text{recall}}{(\text{precision} + \text{recall})}$ |

## TASKS [ SAMPLING EXPERIMENTS | DATA CLEANING]

In the Preprocessing of the data: The dataset being used for this assignment is enormous and requires a bit of preprocessing. We did:

a. **Removing Unwanted Columns:** We dropped the unwanted timestamp columns from the train.csv as it does not provide any help in the predictions and converted to parquet files for better readability.

b. **Applying Under sampling to whole data:** We start with applying random under-sampling to the non-fraud labels(0) to bring the ratio from 126 to 10. So, we could utilize various sampling techniques over the data. Once done, we split the data into train-test split. We further under sampled the data to have even better ratio.

   i. **Low Imbalanced Set of 26M Record:** Reduce the ratio to 5. This is our main dataset to check the accuracy of the model. It is now imbalanced with 21M label 0 records and 4M Label 1 records.

   ii. **Low Imbalanced Set of 6M Record:** Reduce the ratio to 4 and under-sample both labels to form a smaller dataset with 5M label 0 and 1M label 1.

c. **Stratified Train Test Split:** To maintain the ratio between the two classes we split the data into train and test set using stratified random Train-Test Split with 80:20 respectively.

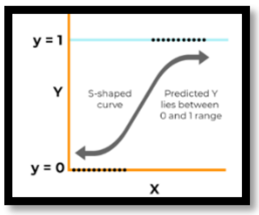
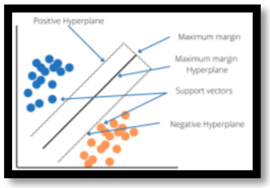
d. **Balance the Data with the different sampling techniques**: Post these, we keep the Test-Set for validation, while the trainset created is now passed on to create different sets like under sampled-trainset, oversampled-trainset, and SMOTE-trainset. Doing so, the Low Imbalanced Sets are balanced, and different machine learning techniques would be applied, and then results will be compared. This is our hybrid approach to solve the problem.

e. **Apply Machine Learning:** The two Low Imbalanced Sets each with 3 types of balanced sets are compared over the different machine learning model. Each of the model is cross validated over 10 folds and the results of the best models were recorded.

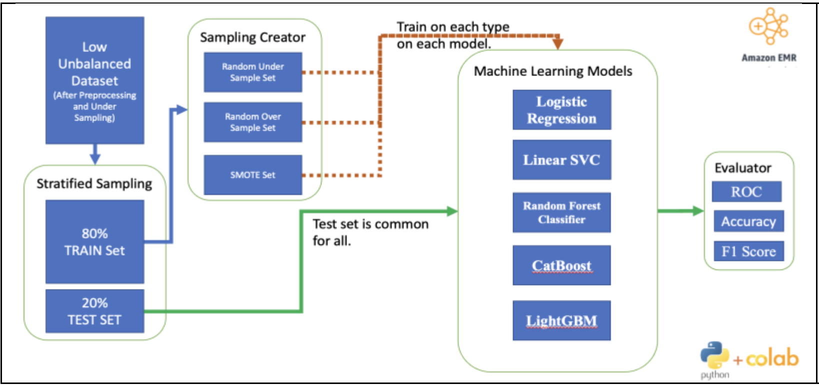
## INFRASTRUCTURE | MACHINE LEARNING MODELS

We were able to run all except LightGBM on **Local Spark Cluster, Perseus Cluster and Google Colab**. **For LightGBM we used AWS's ElasticMapReduce (EMR) Cluster running on m5.2xlarge (32GB RAM) with (Spark 3.0.3 and Hadoop 2.7), and used S3 bucket for storage.** These are the models evaluated using the BinaryClassificationEvaluator.

Amazon EMR

| | |
|---|---|
| **LogisticRegression:** Logistic regression estimates the probability of an event occurring, such as fraud or not fraud, based on a given dataset of independent variables. Since the outcome is a probability, the dependent variable is bounded between 0 and 1.   y = 1<br>Y<br>S-shaped curve<br>Predicted Y lies between 0 and 1 range<br>y = 0<br>X | **LinearSVC:** Linear Support Vector Machine (Linear SVC) is an algorithm that attempts to find a hyperplane to maximize the distance between classified samples. Our dataset is linearly separable data as it can be classified into two classes using a single straight line. (Assumption)   Positive Hyperplane<br>Maximum margin<br>Maximum margin Hyperplane<br>Support vectors<br>Negative Hyperplane |
| **RandomForestClassifier:** Random forests or random decision forests is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. This model should probably learn the pattern better than other previous models. | **CatBoost:** The CatBoost (Categorical Boosting) algorithm is one of the gradient boosting algorithms. This algorithm is designed to work with categorical features. The algorithm applies its own version of the label encoding process to all categorical data. The CatBoost algorithm creates several binary decision trees each time trying to reduce the error. Our dataset has all categorical features and this model should perform better. |
| **LightGBM:** LightGBM is a gradient boosting framework on decision trees that improves model performance while using memory efficiently. To solve the problems efficiency and scalability this technique employs Gradient-based One Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS excludes a significant data instance with negligible gradients and use the ones which give good estimate of information gain. EFB is used in bundling mutually exclusive features to reduce the number of features. | |

## PIPELINE



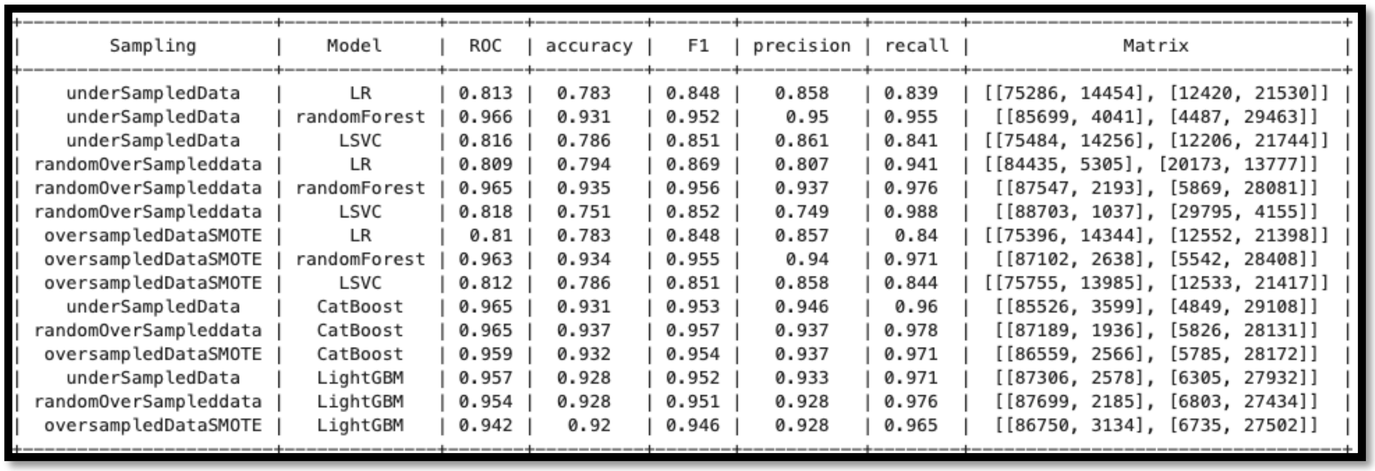
> This is the pipeline that was used for each of the low imbalanced dataset to balance it using different sampling techniques. Keeping the Test Set same across. All the sets were processed using Feature Vectorizer to create features and labels using custom function.

> In the results section we see comparison between the two datasets on using different Sampling Techniques with different machine learning models.

## RESULTS

Now we compare the two Low Imbalanced Sets using different Sampling Techniques to have created 6 different balanced sets. These sets are now run over different machine Learning Models discussed in previous section.

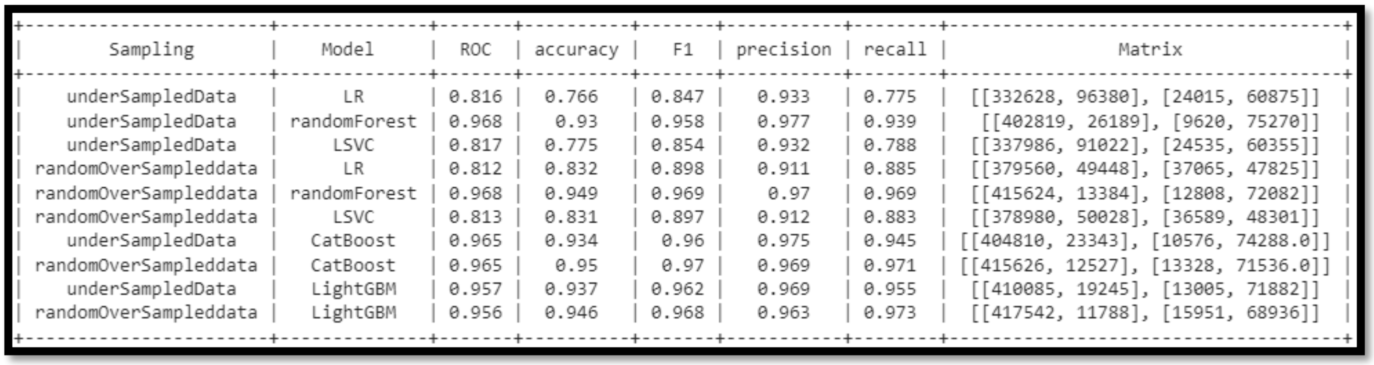
**Balanced Set of 6M Record using different Sampling Techniques over different models**:



```
+-----------------------+-------------+-------+----------+-------+-----------+--------+----------------------------------+
|        Sampling       |    Model    |  ROC  | accuracy |   F1  | precision | recall |              Matrix              |
+-----------------------+-------------+-------+----------+-------+-----------+--------+----------------------------------+
|    underSampledData   |      LR     | 0.813 |  0.783   | 0.848 |   0.858   | 0.839  | [[75286, 14454], [12420, 21530]] |
|    underSampledData   | randomForest| 0.966 |  0.931   | 0.952 |    0.95   | 0.955  |  [[85699, 4041], [4487, 29463]]  |
|    underSampledData   |     LSVC    | 0.816 |  0.786   | 0.851 |   0.861   | 0.841  | [[75484, 14256], [12206, 21744]] |
| randomOverSampleddata |      LR     | 0.809 |  0.794   | 0.869 |   0.807   | 0.941  | [[84435, 5305], [20173, 13777]]  |
| randomOverSampleddata | randomForest| 0.965 |  0.935   | 0.956 |   0.937   | 0.976  |  [[87547, 2193], [5869, 28081]]  |
| randomOverSampleddata |     LSVC    | 0.818 |  0.751   | 0.852 |   0.749   | 0.988  |  [[88703, 1037], [29795, 4155]]  |
|  oversampledDataSMOTE |      LR     |  0.81 |  0.783   | 0.848 |   0.857   |  0.84  | [[75396, 14344], [12552, 21398]] |
|  oversampledDataSMOTE | randomForest| 0.963 |  0.934   | 0.955 |    0.94   | 0.971  |  [[87102, 2638], [5542, 28408]]  |
|  oversampledDataSMOTE |     LSVC    | 0.812 |  0.786   | 0.851 |   0.858   | 0.844  | [[75755, 13985], [12533, 21417]] |
|    underSampledData   |   CatBoost  | 0.965 |  0.931   | 0.953 |   0.946   |  0.96  |  [[85526, 3599], [4849, 29108]]  |
| randomOverSampleddata |   CatBoost  | 0.965 |  0.937   | 0.957 |   0.937   | 0.978  |  [[87189, 1936], [5826, 28131]]  |
|  oversampledDataSMOTE |   CatBoost  | 0.959 |  0.932   | 0.954 |   0.937   | 0.971  |  [[86559, 2566], [5785, 28172]]  |
|    underSampledData   |   LightGBM  | 0.957 |  0.928   | 0.952 |   0.933   | 0.971  |  [[87306, 2578], [6305, 27932]]  |
| randomOverSampleddata |   LightGBM  | 0.954 |  0.928   | 0.951 |   0.928   | 0.976  |  [[87699, 2185], [6803, 27434]]  |
|  oversampledDataSMOTE |   LightGBM  | 0.942 |   0.92   | 0.946 |   0.928   | 0.965  |  [[86750, 3134], [6735, 27502]]  |
+-----------------------+-------------+-------+----------+-------+-----------+--------+----------------------------------+
```

While using SMOTE improved the accuracy for few models, but mostly the results of SMOTE were like Random Over Sampling results, so we did not apply this sampling technique over the 26M database. (NOTE: SMOTE is compute expensive as it uses a Nearest Similar Neighbors to oversample rather than just duplicating).

**Balanced Set of 26M Record using different Sampling Techniques over different models**:



```
+-----------------------+--------------+-------+----------+-------+-----------+--------+-----------------------------------------+
|        Sampling       |    Model     |  ROC  | accuracy |   F1  | precision | recall |                  Matrix                 |
+-----------------------+--------------+-------+----------+-------+-----------+--------+-----------------------------------------+
|    underSampledData   |      LR      | 0.816 |  0.766   | 0.847 |   0.933   | 0.775  |   [[332628, 96380], [24015, 60875]]     |
|    underSampledData   | randomForest | 0.968 |   0.93   | 0.958 |   0.977   | 0.939  |    [[402819, 26189], [9620, 75270]]     |
|    underSampledData   |     LSVC     | 0.817 |  0.775   | 0.854 |   0.932   | 0.788  |   [[337986, 91022], [24535, 60355]]     |
| randomOverSampleddata |      LR      | 0.812 |  0.832   | 0.898 |   0.911   | 0.885  |   [[379560, 49448], [37065, 47825]]     |
| randomOverSampleddata | randomForest | 0.968 |  0.949   | 0.969 |    0.97   | 0.969  |   [[415624, 13384], [12808, 72082]]     |
| randomOverSampleddata |     LSVC     | 0.813 |  0.831   | 0.897 |   0.912   | 0.883  |   [[378980, 50028], [36589, 48301]]     |
|    underSampledData   |   CatBoost   | 0.965 |  0.934   |  0.96 |   0.975   | 0.945  | [[404810, 23343], [10576, 74288.0]]     |
| randomOverSampleddata |   CatBoost   | 0.965 |   0.95   |  0.97 |   0.969   | 0.971  | [[415626, 12527], [13328, 71536.0]]     |
|    underSampledData   |   LightGBM   | 0.957 |  0.937   | 0.962 |   0.969   | 0.955  |   [[410085, 19245], [13005, 71882]]     |
| randomOverSampleddata |   LightGBM   | 0.956 |  0.946   | 0.968 |   0.963   | 0.973  |   [[417542, 11788], [15951, 68936]]     |
+-----------------------+--------------+-------+----------+-------+-----------+--------+-----------------------------------------+
```

The results could be described as Random Forest, CatBoost, LightGBM perform better than LogisticRegression and

LinearSVC. While the best sampling technique turns out to be combination of both (underSampling + overSampling). Both the oversampling techniques i.e., Random Over Sampling and Synthetic Minority Oversampling Technique (SMOTE) preform equally good on this dataset. Maybe if we have some other dataset SMOTE would have done much better. So, to conclude we can say that using under sampling to convert highly imbalanced data to low imbalanced data and then applying oversampling to give balanced data to do classification gives us the best results.

## LEARNINGS | NEW IMPLEMENTATIONS

- Using column oriented datafiles like parquet file which is an efficient file format for large data in storing the preprocessed and sampled data.
- Using different Sampling methods and implementing algorithms such as SMOTE which are not present in the spark library.
- Deploying Machine Learning Models on a Large-Scale data (>7GB) creating a custom pipeline to preprocess and run modeling.
- Integrating Machine learning models originally written in Java with PySpark by adding external Jar files and python wrappers.
- Configuring and using AWS ElasticMapReduce clusters for machine learning tasks with high compute requirements.

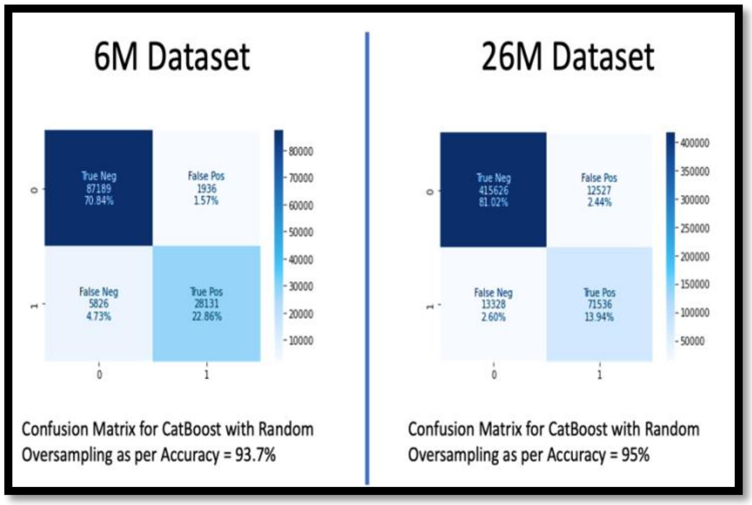
## FUTURE WORK

Since, our pipeline is ready, we want to try to use predictions from all the models and apply ensemble technique. Each of the 5 different model's predictions column could be merged, and majority voting to be used to predict the label.

## CONCLUSION

We acquired knowledge about how to manage enormous datasets, perform analysis, and make classification. We created custom functions to under sample and over sample the data. We did not intend to use any outside library, so we implemented the algorithm given in the research paper for Synthetic Minority Oversampling Technique (SMOTE). Moreover, CatBoost and LightGBM Classification models are not present in Spark Library. We had to integrate these with pySpark, using external Jar files and python wrappers.

So, we can conclude that using under sampling to convert highly imbalanced data to low imbalanced data and then applying oversampling to give balanced data gives us the best result of accuracy of 93.7% on 6M Dataset and 95% on 26M Dataset. The Confusion Matrix is shown for same on the right.



## REFERENCES:

1. Spark Documentation: https://spark.apache.org/docs/latest/api/python/
2. Blogs: https://towardsdatascience.com/comprehensive-guide-on-item-based-recommendation-systems-d67e40e2b75d
3. SMOTE PAPER: https://arxiv.org/pdf/1106.1813.pdf
4. Dataset: https://www.kaggle.com/competitions/talkingdata-adtracking-fraud-detection/overview
5. Some examples for learning: https://sparkbyexamples.com/pyspark-tutorial/
6. LightGBM : A Highly Efficient Gradient Boosting Decision Tree. (https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf)
7. CatBoost: Unbiased Boosting with Categorical Features (https://arxiv.org/pdf/1706.09516.pdf)