# ISOM 674
# Machine Learning I

## Final Project Report

Team 3
Jake Arendsen
Feifan Gu
Sidi Liu
Yidi Zhang

19 December 2019

## I. INTRODUCTION & DATA DESCRIPTION

Our objective for the project was to replicate something of the nature of Homework 4. We wanted to build a bunch of different models, optimize them to a reasonable degree, and then compare log-loss across the models using validation data. This process obviously ended up being more intensive than that would make it sound, but that was our goal. Throughout the journey, we looked to optimize models in any way that we could while also avoided the common pitfalls of Machine Learning, such as overfitting.

We were given two separate datasets: "ProjectTrainingData.csv" and "ProjectTestData.csv". The training set has 30 million entries with 24 different variables. The first variable ('id') is the identifier of the ad and the second variable ('click') is our target variable. These two variables are not our predictors and we will only use the remaining 22 categorical variables to build our model. The test set contains 13 million entries with 23 different variables. The target variable 'click' is not included. After a quick exploration of the data, we found that most of our variables are highly-skewed (see appendix). Our goal is to build the best model in terms of log-loss using the training set we were given, apply the model on the test set, and eventually generate the predicted 'click' for each row of the test set.

## II. DATA CLEANING/FEATURE ENGINEERING

To start with, the first obvious hurdle to overcome was simply parsing the data to both make it into a memory efficient format and give us validation data for later in the journey. To do that, we relied greatly on the tips provided to us encouraging random sampling of the data in Linux. We first shuffled the data in R, using previous scripts provided to us as a basis. Then, almost directly following the hint, we divided the dataset into 31 chunks of 1,000,000 data entries using Linux. The line of Linux code we used is as follows: “split –l 1000000 –d –a 3 --additional-suffix=.csv ProjectTrainingData.csv Train-”. Because the data was already shuffled, we could simply use the first split as training data and the second split as validation data. As we developed the models further and began to discover which were truly worth considering, we used better split of training and validation data encompassing more of the overall data to be as thorough as possible.

Our next problem was with the data itself; it’s gross, messy, and computationally intensive (almost like that’s how the real world works). This dataset has almost entirely categorical variables, with lots of them being stored as character vectors. The character vectors were an immediate no-go, as almost every model we have used this year relies on numerical data or factors to work properly. The leftover variables that come in as numerical are actually anonymized categorical variables, so we had to make sure that was accounted for instead of being treated like a numerical variable. The one upside to this is that very little feature engineering could actually be done, so that was relatively quick. The only thing we did do was extract the hour from the hours variable. We figured just having the hour would actually be more useful than having the full date, as the hour of the day could be usefully applied to things outside of the date range of the training data, while the rest likely would not be. With that, we started to tackle how the categorical variables could best be handled.

Considering all of the predictor variables are categorical and many algorithms requires numerical variables, we knew we had to create dummy variables. However, after EDA, we found that many of the variables had numerous categories, like thousands or even ten thousands, which could generate millions of dummy variables, wasting computational efficiency when running models. So we had to divide the categories into several groups to decrease the numbers of categories in all.

The basic ideas here is to merge those categories with small frequency into a large group: “others”. We ordered the categories of each variable that had too many categories by their frequency. Then, using cumsum() in data.table, we easily added a column showing the cumulative percentage of categories’ count. Most of the time, we kept the categories that appeared the most and account for 75% of all counts, and merged the rest into a large category represented by ‘-1’, since ‘-1’ is not in the range of any variables that we grouped. What is worth mentioning here is we find that the device_ips are actually ipv6 which are hex numbers. And the first 4 digits represent the country and the city. So we only keep the first four digits and used the same grouped strategy on this variable. We transformed ‘device_ip’, ‘site_id’, ‘app_id’ and ‘site_domain’ according to the “75%” criteria.

Since there is one dominant category in ‘device_id’,’a99f214a’, which account for more than 80% counts, we only divided ‘device_id’ into 2 categories. And the count distribution of ‘device_model’, ‘C14’ and ‘C17’ is rather uniform, we lower the criteria of 75% to 30% to avoid too many categories left. Also, raising the criteria to 99% for app_domain was reasonable, which only kept 15 categories.

As we progressed through the problem, we realized that models contained in R might not be sufficient for optimizing our model. Therefore, turning to Python was necessary to at least ensure we covered all our options. Thankfully, Python has a very useful tool for doing much of the same cleaning. The tool, OneHotEncoder, transforms any variable in a pandas dataframe indicated to be categorical into an array of dummy variables. For most of the modeling we needed to accomplish, this was sufficient for creating necessary variables. We did have to run this with combined training and test data to create proper categories, but we ensured test data was in no way involved with actually training the data.

## III. DECISION TREE

The first model we ran was a decision tree. This relatively simple algorithm let us test some things out and see what kind of numbers we can begin to expect, as well as get comfortable with the data. We were able to run the algorithm before we really even factored the data, as the tree algorithm just drops character vectors. This results in an admittedly imperfect model that got us a log-loss of .4283, after finding what the best node value was.

We found the best node value just as we had in class, by searching a wide range and narrowing it down. The size of the initial tree was 283, so we checked 10 trees from 20 to 260 nodes initially. When that found a best k of 30 trees, we could then narrow our search range to 20 to 40 nodes and search every node. This found the exact optimal node value, which was k=26. The other thing we discovered this early was that we could potentially get predictions of a true 0 or 1, both of which could result in problems in the log-loss function. We solved this by using an algorithm to scale our predictions between .005 and .995, as these values were extreme enough to still result in very small log loss when accurate while not causing errors when wrong. Overall, we viewed our log-loss of .4283 as a great start that we seeked to improve further.

## IV. RANDOM FOREST

Our first attempt at improvement was with a Random Forest model. While we thought this would be a simple step up, this ended up being the step where we realized all of the factoring was necessary. That being said, once all the factoring was done, this wasn't much more complicated than it was during Homework 4.

We ran two different Random Forest models, one with 100 trees and one with 500 trees. Both of these get log-loss of .71, so we were able to quickly conclude that Random Forests were not the best fit for this problem. (as a fun fact, in terms of log-loss, this is worse than just assigning p of .17 to everything, which achieved a log-loss of .4558 on our validation set). The reason we conclude for this is the confidence with which Random Forests predicts; a lot of the p-values for Random Forest models were between .01 and .05. To speak like an economist, the marginal benefit of being this exactly right is far outweighed by the marginal cost of being this severely wrong, the log-loss equation is very harsh here on every wrong prediction. That, combined with the fact that there are a nontrivial amount of wrong predictions because almost everything was predicted to be not a click, results in a more extreme log-loss value than expected.

## V. NEURAL NETWORK

Since we could not further improve our log loss using random forest, we decided to move on and build a neural network. We have to first convert all the categorical variables into dummies using one hot encoding since neural network does not take in any categorical variables. When building the model, we tried adjusting both the width and the depth of the neural network. We also tried adjusting the batch size and epochs when fitting the model without any significant improvement. In the end, the model that generates the best performance has one hidden layer with 8 nodes and one output layer with 1 node. The resulting log loss is 0.4223, which is only slightly better than the decision tree.

## VI. LOGISTIC REGRESSION

In logistic regression model building, we chose  the glmnet function over the glm function in order to avoid overfitting problem. Glmnet penalizes the size of estimated coefficients via Lasso or Ridge penalties. The glm function by default does not do any regularization. We can see from the table that the bigger lambda is, the better performance that model has on validation data.

Glmnet applies an elastic net as regularization term, which attempts to serve as a median between lasso and ridge regression. This allows it to better serve as a trade-off between bias and variance then either lasso or ridge regression. Whether it's "better" than ridge or lasso regression, however, depends on the ultimate level of tradeoff we desire between bias and variance. For us, it's all about minimizing that log-loss function, so we ultimately ran a range of lambda to determine which lowered this the most. Here, we ran a range of lambda from -.001 to 1000 with length of 200 and the lowest log loss is 0.447 on lasso logistic regression.

Since the glmnet function supports sparse matrices, a more memory efficient format of R's matrices, to be able to perform the regression, we first needed to turn all 23 variables into dummies. The number of dummy variables depends on the method that we use to subgroup each feature. After data exploration, we applied different methods on different variables. For example, for site_id,  we split everything above the third quartile (75%) of cumulative distribution as individual value and those below assigned as one group. So we have in total 18 groups of site_id feature. We then applied the same transformation procedure on training, validation and test data to make sure all features have the same transforming criteria.

Another trick we found during the process of creating sparse matrices is that we typically had different numbers of dummies in training and validation dataset, so we needed to adjust so we'd have the same amount of dummies in both. The way we solved this was to append the training data with new dummies that do not exist in training but in validation data and assign those new dummies as zero values before making sparse matrices. The same procedure needs to be done on validation data. Then we turn all dummy matrices into sparse matrices to use the glmnet function.

## VII. XGBOOST

Next, we used XGBoost on a grouped 3 million training dataset which we created earlier. Since XGBoost cannot recognize categorical variables, we had to dummy all those categorical variables before we could utilize XGBoost. We had to make fewer categories as well, or else millions of columns could be generated. Using the grouped data is a good solution.

We used OneHotEncoder to dummy the x-variables and trained the model with a 70-30 train-test split. With parameters set according to experience, the log-loss of the model on test set is 0.407.

Since XGBoost runs quite slowly, we did not use gridsearchCV here to do hyperparameter tuning, which acquires a lot of time considering time of each running. So we just used a simple 2-layer for-loop to try to optimize 2 hyperparameters every time. Unfortunately, it does not work that well, as the best log-loss I got was still 0.404 – slightly improved from our earlier result. So we decided to try another ensemble method, LightGBM, which is much faster than XGBoost.

## VIII. LIGHTGBM

Higher speed is not the only merit of LightGBM. Instead of using level(depth)-wise tree growth, which is how XGBoost builds trees, LightGBM adopts leaf-wise tree growth, which could decrease more loss. Additionally, LightGBM can directly handle categorical variables using its algorithm based on histograms, which could create efficiency and improve accuracy as we no longer need to use grouped data which could lead to loss of accuracy.

Here, we just used the original 3 million training dataset only with the ‘hour’ variable transformed. We still, however, needed to factorize all those categorical variables and turn the number which represent different factors to category type, so that it can perfectly fit in the LightGBM model.

We split the training set to training and validation set. This time we used grid search, which did not require much time to runusing LightGBM. After tuning learning_rate, num_leaves, max_depth, etc., we got our best model, saved in the code as ‘gsearch’. It had a log-loss of 0.394 of cross validation, and a log-loss of 0.391 on the hold-out test set. So we decided to use LightGBM with such hyperparameters as our final model.

To use the model to predict the final result for the true test dataset, we first need to combine the training set with the test set by columns, so that we could factorize them with same number indicating same categories. After turning numbers into categories, we split them back into train and test. Fit the model on the train set again, we use the fitted model on the test set and get an array of probabilities, which is our final answer.

## IX. SUMMARY OF RESULTS

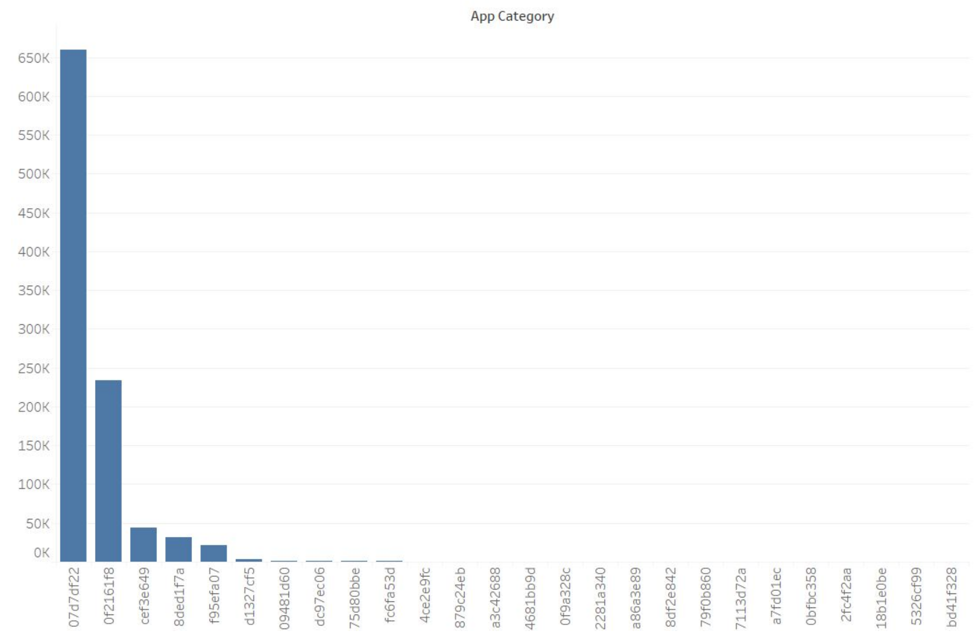
| Model | Decision Tree | Random Forest | Neural Network | Logistic Regression | XGBoost | **LightGBM** |
|---|---|---|---|---|---|---|
| Best log loss | 0.428 | 0.456 | 0.422 | 0.447 | 0.404 | **0.391** |

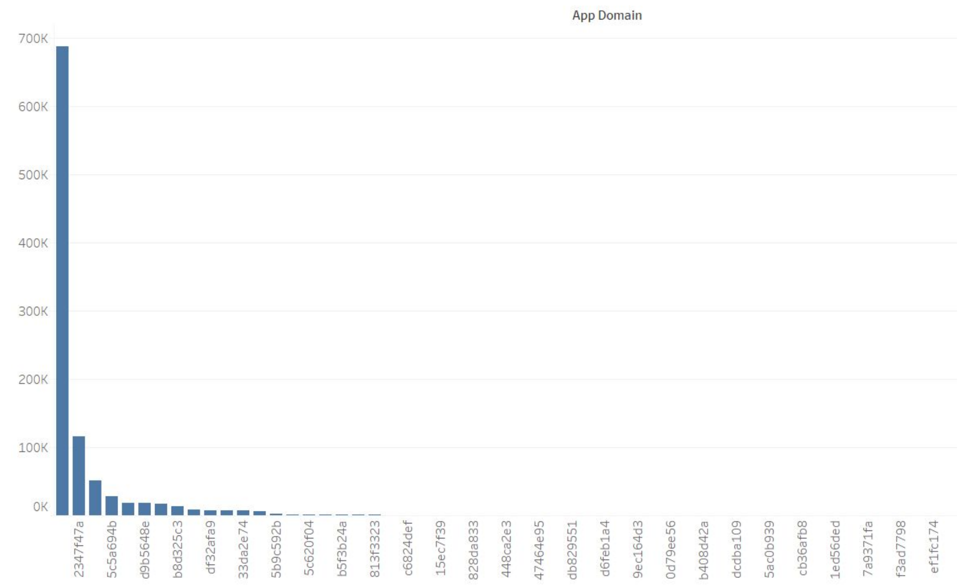
## X. APPENDIX

**CODE FILES (You can easily understand what they do based on the file name)**

- EDA & FeatureEngineering.R
- Tree.R (including decision tree and random forest)
- NeuralNetwork.ipynb
- LogisticRegression.r
- Lightgbm.ipynb
- XGBoost.ipynb (including final prediction)

## Examples of the highly-skewed distribution of some of the categories





Other variables, like App Id and Device Model, have very similar distributions, they have not been included for space reasons.