# Machine Learning Final Project

PREDICTING AD-CLICKS USING MACHING LEARNIING

|Team 7 |
| Harshit Arora, Patrick He, Barbara Wu, Jie Zhu |
| December 19, 2019 |

## Project Summary

In online advertising, achieving a high click-through rate (CTR) is essential to the success of the advertisement. A high CTR is a good indication that the users find the ad is helpful and relevant. For the final project, we focus on using a large volume of data ("features") to predict whether an ad is likely to be clicked or not. Thus, this is a binary classification problem. This report is aimed to provide details on our team's approach for data cleaning, model induction and comparison process.

## Data Description

The training dataset provides a large volume of historical click-through results (*31.9 million records of 24 variables that relate to the user's attributes as well as whether they clicked on the ad or not*) from October 21, 2014 to October 29, 2014 (*9 days in total*). The following table contains all the variables in the training dataset:

**Table 1: Original Training Set Variables**

| Variable Types | Variable Names | Description |
|---|---|---|
| Identifiers | id | The identifier of the ad |
| | hour | The identifier of the time when the ad was displayed (in form of YYMMDDHH) |
| | site-id | An identifier for the website |
| | site-domain | An identifier for the site domain |
| | app-id | An identifier for the application showing the ad |
| | app-domain | An identifier for the app's domain |
| | device-id | An identifier for the device used |
| Categorical Variables | C1, C14-C21 | Anonymized categorical variables |
| Code-Related Variables | site-category | A code for the site's category |
| | app-category | A code for the app's category |
| | device-ip | A code for the device's IP |
| Device-Related Variables | device-model | The model of the device |
| | device-type | The type of the device |
| | device-conn-type | The type of the device's connection |
| Banner | banner-pos | The position in the banner |
| Target Variable | Click | Binary variable: 1 as the ad was clicked on and 0 as not |

**** Note: the bolded box highlights the target variable, Click, in this binary classification problem.*

## Data Exploration and Data Cleaning

The initial training dataset has about 31,991,090 data entries and 24 variables *(23 features and 1 target variable, listed in table 1*). The dataset does not contain any missing values in any of the columns. Knowing the fact that this is a supervised learning problem where all the features are categorical (unordered), our first instinct was to check the target variable distribution as well as the levels (factors) in each of the features.

## Target Variable: Ad-Clicked? (Binary Variable)

Looking into the target variable distribution, we noticed that the distribution of the classes is unbalanced (*83% unclicked ads vs. 17% clicked ads*). However, given that we expect this distribution to represent the real-world scenario, where most people do not click ads, we decided that up-sampling should not be performed. (*Our hypothesis is that the test data should also represent a similar distribution, given it contains observations of the near future*)

## Categorical Feature Levels

The first step to analyze the levels (factors) of each categorical feature was to define each of them as factors. Next, we checked the number of levels in each of the 23 features.

**Table 2: Original Training Set Number of Levels**

| Variable Names | Number of Levels |
|---|---|
| hour | 216 |
| site-id | 4,581 |
| site-domain | 7,341 |
| app-id | 8,088 |
| app-domain | 526 |
| device-id | 2,296,165 |
| C1 | 7 |
| C14 | 2,465 |
| C15 | 8 |
| C16 | 9 |
| C17 | 407 |
| C18 | 4 |
| C19 | 66 |
| C20 | 171 |
| C21 | 55 |
| site-category | 26 |
| app-category | 36 |
| device-ip | 5,762,925 |
| device-model | 8,058 |
| device-type | 5 |
| device-conn-type | 4 |
| banner-pos | 7 |

From the table above, we noticed that majority of the categorical features consist of a very high number of levels (*10 features have over 200 levels each*). Since these are all factors, using dummy binary variables (*e.g. One Hot Encoding*) is necessary, which, to say the least, is computationally infeasible. We approached this *problem* by seeing the cumulative frequency distribution of each of the features. Per Pareto's 80/20 rule, we did expect that majority of the observations would contain only a small percentage of the overall number of levels. In other words, it seemed that we could significantly reduce the number of dimensions by using some criteria to group relatively infrequent levels together (a new category called "Others"), such as:

1. For the variables with the number of levels less than 25, we decided to keep all the original number of levels. Thus, we did not modify 7 of the 23 features.
2. For the variable with the number of levels at or more than 25, we aimed to follow the Pareto Rule and include the Top-N levels of each category that can capture at least 80% of the observations in their original levels (Non-Others)
   ***Note**: *Majority of the variables will follow this rule. The other few variables will follow the rule in step 3.*
3. For 3 variables, we decided to stop the Top-N criteria around 50% (*shown in the table 3 below*). This is because we realized that the incremental number of levels to go any higher (to 80%) was significantly high.
4. For device IP and device ID, we decided to exclude the variables from the modeling process. More details in the following section as well as table 3.

## Data Manipulation and Feature Selection

### Device IP and ID Exclusion

Table 2 shows that "device_id" and "device_ip" have significantly more levels than any other listed variables. In the specific case of "device_id", we noticed that only one of the levels, or id, contain 83% of the observations, while the others are present in very low proportions. For "device_ip", the frequency distribution was much more balanced, which, to some extent, indicates that the two variables tend to be more unique and personalized compared to other features. To confirm our hypothesis, we also performed a quick data exploration on the test dataset. Not surprised, the result aligns with our assumption: *more than 40% of the observations in the test dataset have a new ID or IP*. With this information, we decided our model may not benefit from having these two features (and the dimensions they would have added to the model)

### Time-Related Variables

We believe that the time-related variable (hour, in YYMMDDHH format) will be an important component in this business context. Our expectation was that click-through rate will vary at different time of the day (*e.g. less clicks at night*) as well as vary from day to day (*e.g. more clicks on weekend days*). Furthermore, since the data is only for 9 days, having the year and the month do not add any extra value. We segmented the data based on days of the week (*7 levels: Monday through Sunday*) and then by hours of the day (*24 hour*) and did see changes in the click-through-rate.
With such an insight in mind, we decided to extract the day of the week and the hour of the day from the original dataset into 2 new columns: Day (*ranging from 1 to 7*) and Hour (*ranging from 1 to 24*) and add them as new categorical variables in the model, and removing the original "hour" variable

**Summary: Final Training Dataset**

After all the data manipulation, we now have approximately 400 cleaned dummy variables for the 22 categorical features (shown in table 3 below).

**Table 3: Revised Training Set Features and Number of Levels**

| Variable Names | Number of Levels (Revised) | % of Data Captured in *Non-Other* Levels |
|---|---|---|
| weekday | 7 | 100% |
| day_hour | 24 | 100% |
| site-id | 31 | 81.40% |
| site-domain | 25 | 83.12% |
| app-id | 25 | 87.66% |
| app-domain | 25 | 99.74% |
| C1 | 7 | 100% |
| C14 | 51 | 50.64% |
| C15 | 8 | 100% |
| C16 | 9 | 100% |
| C17 | 31 | 60.63% |
| C18 | 4 | 100% |
| C19 | 25 | 93.98% |
| C20 | 25 | 91.09% |
| C21 | 25 | 95.70% |
| site-category | 25 | 99.99% |
| app-category | 25 | 99.99% |
| device-model | 51 | 52.28% |
| device-type | 5 | 100% |
| device-conn-type | 4 | 100% |
| banner-pos | 7 | 100% |
| **device-ip | *REMOVED: Highly personalized variable. Even with even 50 levels, would only contain a minority of the data in "Non-Others"* | |
| **device-id | *REMOVED: 83% of the data has the same device IP, while others are very personalized* | |

## Additional Data Nuances

We noticed that one of the anonymized variables, C21, has a huge number of observations in the "-1" category. Our initial "guess" was that these could be missing observations, but given that we did not have the adequate knowledge of the variable definition, and the fact that this level was present in a large amount of rows, we decided to keep this variable.

We also realized that we need to have a consistent number of levels in features across Train, Validation and Test data (*3 subsets of the original, cleaned data*). Thus, we decided to ensure that any “new” levels encountered in the test data would be recategorized as “Others”. Doing so allows us to ensure the consistency of our datasets. Moreover, for models that require a matrix input (*e.g. sparse matrix*), the ordering of the levels should especially be consistent across the 3 datasets. Therefore, we made sure that, for each model, we did sanity check on the number of levels for each categorical variable. We only performed a consistent level re-ordering process when it was necessary.

Finally, it is important to note that although all the cleaning and exploration process was performed on the entire training dataset (*about 32M rows*), only a subset (*based on random sampling*) of the entire dataset was used in the actual training and validation process while building the predictive models (*primarily due to computational difficulties and limited marginal benefit on performance*).

## Model Induction

For our modelling process, we decided to try several techniques which are widely used for classification problems:

- Logistic Regression (*With Lasso & Ridge regularization*)
- Classification Trees (*on a modified dataset that capped under 31 levels per categorical variable, due to R’s tree package restrictions*)
- XGBoost Tree Booster
- XGBoost Linear Booster
- Neural Nets
- Random Forests

During the model induction process, we decided to go with a sample size of 10 million data entries (randomly sampled from the cleaned dataset, about 32 million of data entries). Then, before choosing the best model(s) which will use the 13M test set, we split the selected 10M sample: Training Set (about 5 million), Validation Set (about 2.5 million), and Test Set (about 2.5 million). Not all procedures used the test set, since we disqualified models that did not come close to the benchmark performance set by the *“simpler”* models such as Logistic Regression.

Finally, it is noteworthy to mention that these are not all the approaches we considered. Only few of the good ones were brought into the picture as we felt that the selected ones would provide improvement in the logloss measure. For instance, the performance of Classification Trees allowed us to further improve the model by using Boosting. Thus, we decided to use XGBoost in our final model. Similarly, we also considered Stepwise Logistic Regression; unfortunately, we had to drop this idea owing to the time constraint which makes this technique not feasible.

## Logistic Regression

We started with Logistic Regression due to a mix of simplicity and high utility of this model. To reduce the extent of overfitting, regularization via Lasso (*alpha*=1) and Ridge (*alpha*=0) was used from the *glmnet* package. A set of 200 lambda values were used to train the model and make predictions on the validation set. For each set of predictions, the corresponding log loss was calculated. The optimal lambda, which came out to be 7.5 x $10^{-5}$ for lasso, for e.g, gave the least log loss on the validation data. We observed a log loss of 0.412 in the case of Lasso and 0.415 in the case of ridge. Although the difference was not significantly different, it seemed reasonable that Lasso would have a slightly better performance as it tends to reduce model complexity more than ridge does by essentially making some feature coefficients zero (which it did.).

Finally, to make sure that the Lasso Logistic Regression model worked on a dataset it hasn't been tuned on, the log loss was calculated on a final test set, and a log loss of 0.416 was observed (only slightly higher, as expected, when compared to the validation performance) which ensured confidence that the model is capable of performing well without overfitting on new datasets.

## Neural Nets

For Neural Net, we used both the NeuralNet package in r and the Keras package in python to create a binary classification model. In python, we initially trained the model with 100K data (any higher would make computation time very long) and used 3 layers of 4 nodes each, with the activation function as ReLu. The final layer, since the outcome variable is binary, was set as Sigmoid.

In terms of data pre-processing for Python's Keras, we encoded the cleaned data using pandas' *getdummies()* function to convert them into dummy variables before passing the data to the model.  In R, we changed the parameter *hidden* = c(3,4)  to set the layers and nodes and feed the neural net with the training data.  After fitting on the model, we tested the performance of the neural net on the validation data and resulted in a LogLoss of 0.4036. The biggest concern about using Neural Nets was its computation time, which did not allow us to "tune" the model by modifying the width and the depth of the NN.

**Figure 1: Neural Network architecture with 3 layers and 4 nodes**

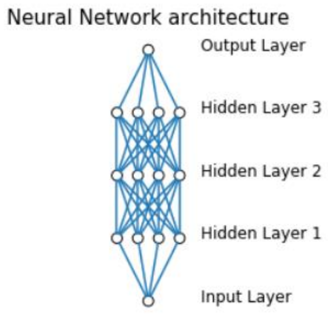
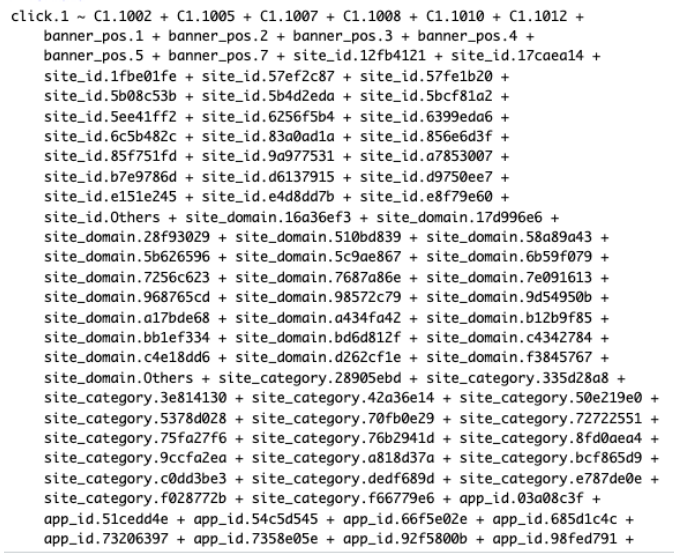


**Figure 2: Encoded categorical variables**



```
click.1 ~ C1.1002 + C1.1005 + C1.1007 + C1.1008 + C1.1010 + C1.1012 +
    banner_pos.1 + banner_pos.2 + banner_pos.3 + banner_pos.4 +
    banner_pos.5 + banner_pos.7 + site_id.12fb4121 + site_id.17caea14 +
    site_id.1fbe01fe + site_id.57ef2c87 + site_id.57fe1b20 +
    site_id.5b08c53b + site_id.5b4d2eda + site_id.5bcf81a2 +
    site_id.5ee41ff2 + site_id.6256f5b4 + site_id.6399eda6 +
    site_id.6c5b482c + site_id.83a0ad1a + site_id.856e6d3f +
    site_id.85f751fd + site_id.9a977531 + site_id.a7853007 +
    site_id.b7e9786d + site_id.d6137915 + site_id.d9750ee7 +
    site_id.e151e245 + site_id.e4d8dd7b + site_id.e8f79e60 +
    site_id.Others + site_domain.16a36ef3 + site_domain.17d996e6 +
    site_domain.28f93029 + site_domain.510bd839 + site_domain.58a89a43 +
    site_domain.5b626596 + site_domain.5c9ae867 + site_domain.6b59f079 +
    site_domain.7256c623 + site_domain.7687a86e + site_domain.7e091613 +
    site_domain.968765cd + site_domain.98572c79 + site_domain.9d54950b +
    site_domain.a17bde68 + site_domain.a434fa42 + site_domain.b12b9f85 +
    site_domain.bb1ef334 + site_domain.bd6d812f + site_domain.c4342784 +
    site_domain.c4e18dd6 + site_domain.d262cf1e + site_domain.f3845767 +
    site_domain.Others + site_category.28905ebd + site_category.335d28a8 +
    site_category.3e814130 + site_category.42a36e14 + site_category.50e219e0 +
    site_category.5378d028 + site_category.70fb0e29 + site_category.72722551 +
    site_category.75fa27f6 + site_category.76b2941d + site_category.8fd0aea4 +
    site_category.9ccfa2ea + site_category.a818d37a + site_category.bcf865d9 +
    site_category.c0dd3be3 + site_category.dedf689d + site_category.e787de0e +
    site_category.f028772b + site_category.f66779e6 + app_id.03a08c3f +
    app_id.51cedd4e + app_id.54c5d545 + app_id.66f5e02e + app_id.685d1c4c +
    app_id.73206397 + app_id.7358e05e + app_id.92f5800b + app_id.98fed791 +
```

## Decision Tree

The next model we tried was the decision tree, which is fast to compute, simple and easy to comprehend. R's tree package imposes a limit of 32 levels in each feature, which we realized when we tried running the model with our cleaned training data. To counter this situation, we went back to further modify the datasets and cap the maximum number of levels to 25 for all features (which did not change cumulative frequency capture in the Top-N features much, as only 4 features were affected).

The classification tree splits by considering each categorical feature in the training dataset and stops when each set of data reaches an end classification. We started the tree with *minsize* 5000 and *mincut* as 2500, as any lower would generate a tree with huge depth, leading to "maximum depth reached" error. We then increased the *minsize* and *mincut* in equal intervals to optimize the ideal size. The ideal *minsize* observed was 20,000 with *mincut* as 5,000.

To perform pruning for the classification tree, we used the fitted tree with number of end nodes ranged from 20 to 171 end nodes (Terminal size). From the tuning, we observed the LogLoss performance got relatively stable after 160 nodes and result in a value of 0.4106211. Finally, we tested our finalized tree with 171 terminal nodes (best size) and observed a LogLoss of 0.4126 on the test data.

**Figure 3: LogLoss Performance against Number of End Nodes**

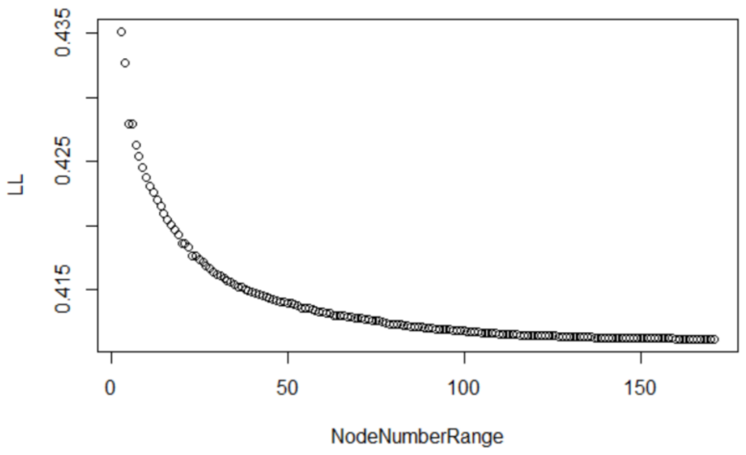
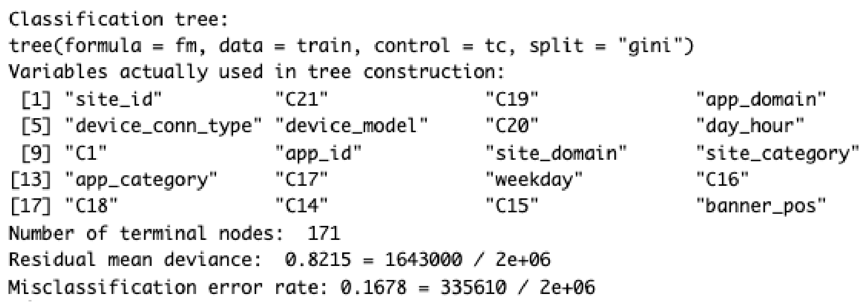


**Figure 4: Summary of Decision Tree**


```
Classification tree:
tree(formula = fm, data = train, control = tc, split = "gini")
Variables actually used in tree construction:
 [1] "site_id"          "C21"              "C19"              "app_domain"
 [5] "device_conn_type" "device_model"     "C20"              "day_hour"
 [9] "C1"               "app_id"           "site_domain"      "site_category"
[13] "app_category"     "C17"              "weekday"          "C16"
[17] "C18"              "C14"              "C15"              "banner_pos"
Number of terminal nodes:  171
Residual mean deviance:  0.8215 = 1643000 / 2e+06
Misclassification error rate: 0.1678 = 335610 / 2e+06
```

## Bagging & Random Forest

For both random forest and bagging, we initially set the tree size as 500, and then reduced the tree size from 250 to 100 for computational efficiency. The performance the on validation data, however, was far from ideal, so we further made a *for* loop to find the best *mtry* (number of features to be considered per split) From the result, we found the best performance appeared with a *mtry* at 11, but the LogLoss performance was still above 1 for both random forest and bagging. Due to the bad performance and low efficiency, we decided to turn to XGBoost, a gradient boosted algorithm.

**XGBoost**

XGBoost is a gradient boosted algorithm, which uses ensemble models to combine many single models to deliver better performance. The XGBoost package comes with two approaches: the tree booster (*gbtree*) and the linear booster (*gblinear*). Since we were dealing with a binary classification and the results are evaluated using the Log Loss metric, we set *objective* and *eval_metric* of both models to be “binary:logistic” and “logloss”, accordingly.

In addition, in order to save computing time, we set the *early_stopping_rounds* to be 10, so that XGBoost will stop if the result doesn’t improve within 10 iterations; in order to find the best model based on Log Loss for the validation dataset, we set a *watchlist* of both train set performance and validation set performance; to fully utilize the computing power provided, we set *nthreads* to be 0, so that all CPU cores will be used.

*Tree Booster*

The parameters used for tuning the tree booster are shown below:

**Table 4: The Parameters Used for Tree Booster Tuning**

| Parameters | Explanation |
|---|---|
| min_child_weight | Minimum sum of instance weight needed in a child |
| colsample_bytree | Subsample ratio of columns when constructing each tree |
| subsample | Subsample ratio of rows when constructing each tree |
| eta | Control the learning rate |
| gamma | Minimum loss reduction to make a further leaf node partition |
| max_depth | Maximum depth of a tree |

The idea here is to use *gamma, max_depth, and min_child_weight* to control the complexity of the tree, while using *colsample_bytree* and *subsample* to further reduce the risk of overfitting and accelerate the computing process. Our strategy for tuning *eta* was first using *eta* = 0.5 to explore which settings of other parameters are optimal for the validation set performance and then using *eta* = 0.1 to run the algorithm again with optimal parameters to further improve the model performance.

We used *eta* = 0.1, *gamma* = 3, *max_depth* = 17, *min_child_weight* = 5, *colsample_bytree* = 0.8, *subsample* = 0.75 to find our best model. The results were 0.393930 in Log Loss for train data and 0.398988 in Log Loss for validation data. When applied on test data, the tree model yielded a result of 0.397281 in Log Loss.

*Linear Booster*

The parameters used for tuning the tree booster are:

**Table 5: The Parameters Used for Linear Booster Tuning**

| **Parameters** | **Explanation** |
|---|---|
| lambda | L2 regularization term on weights |
| lambda_bias | L2 regularization term on bias |
| alpha | L1 regularization term on weights |

After tuning these parameters, we found that the linear booster model worked best using default settings, which are 0 for *lambda*, 0 for *lambda_bias*, and 0 for *alpha*. We used *eta* = 0.1 to re-run the linear booster again, and the results were 0.413218 in Log Loss for train data and 0.413386 in Log Loss for validation data. When applied on test data, the linear model yielded 0.413454 in Log Loss. Compared to the tree booster, the linear booster didn't perform equally well, and we decided to use the tree booster as our final model for the XGBoost approach.

## Choosing the best model(s) for predictions:

Looking at the model performance, XGBoost (tree booster) had the overall best performance with a LogLoss of around 0.398 on both validation and test data. That said, logistic regression and decision trees also performed reasonably well on large datasets, at 0.41 each. We considered an ensemble of the two models as well and decided to see if an ensemble between logistic regression and XGBoost would lead to better predictions. This was done by averaging the predictions of the two models, and then computing the corresponding LogLoss on multiple test datasets.

Through this process, we realized that XGBoost by itself was making better predictions than an ensemble of the two. It made sense from a generalization perspective that XGBoost by itself should be able to make good predictions on a new test dataset due to the gradient boosting framework it follows, which makes it fast to compute  tune, and helps reduce the likelihood of overfitting.

To conclude, our submitted predictions are based on the XGBoost model, trained on 5M randomly sampled data, which has a LogLoss of 0.398 on our test data.

## Appendix: Details on Code File Submitted

| Code File | Contents |
|---|---|
| Project-Code1-Team7.R | Contains code for data exploration, data cleaning (both train and test sets) and sampling the data |
| Project-Code2-Team7.R | Contains code for Logistic regression using Lasso and Ridge regularization |
| Project-Code3-Team7.R | Contains code for XGBoost, our final model, and making predictions on the final test data. |
| Project-Code4-Team7.R | Contains code for Decision Tree model |
| Project-Code5-Team7.py | Contains code for Neural Network trained on 100k data |
| Project-Code6-Team7.R | Contains code for Random Forests and Bagging |

## Appendix: Model Selection and Validation Results

| Supervised Learning Model | Log-Loss (on Validation Data) |
|---|---|
| Ridge Logistic Regression | 0.4120 |
| Lasso Logistic Regression | 0.4150 |
| Decision Trees | 0.4106 |
| **XGBoost** | **0.3989** |
| NeuralNets | 0.4036 |
| Random Forest | 1.2037 |