TEAM 5

Sameen Haroon | Abhishek Saha | Sharon Xu | Zeyu Zhu

Introduction

Online advertising is a form of marketing which delivers promotional messages to consumers via the Internet. Due to the volume of advertisements being published & huge cost associated with marketing campaigns, serving the right advertisement to the right consumer is extremely important. Hence, the ability to predict whether a consumer will click an advertisement on a search engine such as Google will help the firm determine the right advertisement to serve the right consumer. Our project intends to unravel this prediction by creating a classification model, utilizing data of 9 days (Oct 21st - 29th 2014) during which more than 30 million instances of advertisements were served to various consumers.

Initial Data Exploration

The entire training dataset has 31,991,090 observations and 24 variables. All the variables are present in each record (i.e. no missing values observed). Given the fact that this is a binary classification problem and the features in the dataset are all categorical, the first thing we wanted to understand is the number of levels and the distributions of levels for each variable. Looking into the target variable, its classes are strongly unbalanced (17% click vs 83% not click). However, based on our understanding, this imbalance is representative of the true proportion of the classes, and as such, the dataset will not be rebalanced.[1]

Next, we factorized all the features except id and got the number of levels for each variable. The number of levels for each variable are shown below. Variables with more than 30 levels are shown in the left table and variables with less than 30 levels are shown in the right.

Variable	Number of Levels
device_ip	5762925
device_id	2296165
app_id	8088
device_model	8058
site_domain	7341
site_id	4581
C14	2465
app_domain	526
C17	407
hour	216
C20	171
C19	66
C21	55
app_category	36

variable	Number of Levels
site_category	26
C16	9
C15	8
C1	7
banner_pos	7
device_type	5
device_conn_type	4
C18	4

Table with number of levels for categorical variables

[1] Per office hours, we learned that the test data follows a similar proportion split, and our goal is to make a prediction on data that is representative of the test

From the table above, we immediately noticed the two variables with over 1 million levels (device id, and device ip). From a business understanding, we felt that device_ip and device_id wouldn't be available at the time of prediction and would probably introduce data leakage when using them to predict the clicking behavior.

At the same time, to click deeper, we also looked at the number and proportion of observations associated with each variables level and tried to understand the general distributions of the variables' levels. For instance, for *device_id*, we saw that "one" device (probably the 'unknown' category) was actually used by 83% of the total observations, and this variable would probably add little value in our prediction model. For *device_ip*, the levels are very fragmented, with only one level taking up more than 0.5% of records, such that no level represents a sizeable part of our data. With this and our business perspective in mind, we decided to exclude *device_id* and *device_ip* from our predictive modeling.

	device_ip	freq	prop
1	6b9769f2	168598	0.0052701549
2	431b3174	109122	0.0034110123
3	2f323f36	76251	0.0023835074
4	af9205f9	75657	0.0023649397
5	930ec31d	75110	0.0023478412
6	af62faf4	73993	0.0023129253
7	285aa37d	73719	0.0023043604
8	009a7861	73686	0.0023033288
9	6394f6f6	71703	0.0022413428
10	d90a7774	71533	0.0022360288
11	c6563308	57766	0.0018056903
12	57cd4006	57093	0.0017846532
13	75bb1b58	56989	0.0017814023
14	1cf29716	56673	0.0017715245
15	ddd2926e	56616	0.0017697428
16	488a9a3e	56412	0.0017633660

Showing 1 to 17 of 5,762,925 entries

	device_id	freq	prop
1	a99f214a	26317545	8.226523e-01
2	c357dbff	17428	5.447767e-04
3	936e92fb	11075	3.461901e-04
4	0f7c61dc	10505	3.283727e-04
5	afeffc18	7529	2.353468e-04
6	28dc8687	3627	1.133753e-04
7	987552d1	3357	1.049355e-04
8	d857ffbb	3333	1.041853e-04
9	cef4c8cc	3055	9.549534e-05
10	b09da1c4	2931	9.161926e-05
11	03559b29	1937	6.054811e-05
12	02da5312	1775	5.548420e-05
13	d2e4c0ab	1277	3.991736e-05
14	f1d9c744	1246	3.894834e-05
15	abab24a7	1225	3.829191e-05
16	096a6f32	1224	3.826065e-05
17	73b81e30	1049	3.279038e-05

Showing 1 to 18 of 2,296,165 entries

Tables showing frequency and proportion of the various levels taken on by device_ip and device_id, ranked from largest to smallest

We also looked at the levels for each categorical variable to get a sense of the general distributions and found that for variables having a large number of levels (over 200), most of the levels have very few occurrences (less than 0.05% of the total). Examples of variable levels' frequency tables are shown below. The frequencies are sorted from the highest to the lowest, and we can see that for site_id, only 115 out of 4581 levels have more than 0.05% of the total observations.

	site_id	freq	prop
108	2d7bacb1	16899	0.0005282408
109	15941585	16841	0.0005264278
110	c74e86fb	16822	0.0005258339
111	7aec40a2	16671	0.0005211138
112	9a28a858	16594	0.0005187069
113	cd58172f	16488	0.0005153935
114	14c5faa4	16377	0.0005119238
115	686546d3	16055	0.0005018585
116	2c4ed2f7	15974	0.0004993265
117	d266097b	15728	0.0004916369
118	c21df24f	15617	0.0004881672
119	5014edea	15413	0.0004817904
120	e030b8dc	15396	0.0004812590
121	fa2fd0b3	15198	0.0004750698
122	d7714196	15107	0.0004722252
123	a831e579	15106	0.0004721940

Showing 110 to 127 of 4,581 entries

	C14	freq	prop
287	21694	16766	0.0005240834
288	22116	16744	0.0005233957
289	21697	16638	0.0005200823
290	23729	16630	0.0005198322
291	22987	16618	0.0005194571
292	17263	16564	0.0005177692
293	22120	16366	0.0005115799
294	23726	16305	0.0005096732
295	21676	16287	0.0005091105
296	20170	16265	0.0005084228
297	21675	15941	0.0004982950
298	21677	15829	0.0004947940
299	22598	15654	0.0004893237
300	23635	15597	0.0004875420
301	21691	15564	0.0004865105
302	9461	15564	0.0004865105
303	20128	15289	0.0004779143

Showing 290 to 307 of 2,465 entries

Table showing frequency and proportion of observations for site_id and C14

After initial data exploration, we split the data into 60% for training, 20% for tuning the hyperparameters for each algorithm and 20% for choosing the best model. We then cleaned data and grouped factor levels based on the training data.

Data Cleaning

Cleaning, Part 1

For data cleaning, we first dropped the variables *device_id* and *device_ip* because of previous observations. Then we transformed the *hours* variable because the original format (%y%m%d%H timestamp) would not be useful unless we extract some features related to users' behaviors (e.g. hours when people are at home/work, surfing on weekend etc.). Therefore we created 2 variables for the time feature: the hour of day as well as the day of week which are believed to be relevant for the likelihood of people clicking on an ad.

For categorical variables which have more than 30 levels[2], multiple methods are applied to reduce the dimensionality. First of all, we built a table containing the levels and their corresponding frequency for each high-dimension variable. Those low-occurrence levels with a

[2] 30 was used as the cutoff since most algorithms/packages in R can handle a maximum of 32 levels for a factor variable

frequency less than 0.05% are grouped into a factor level called 'other'. Levels with high frequency (represent 1% or above of total observations) are kept as unique levels.
Secondly, for those factor levels with a frequency between these two thresholds (0.05% - 1%), we fitted logistic regression on each variable individually and grouped levels by how they are associated with clicks. We first assigned those levels whose coefficient was not significantly different from 0 (i.e. p-value>0.1) into the 'other' category. For levels that were significant, they were first divided into subgroups based on the direction of the coefficient i.e. positive and negative coefficients were split out. Within each subgroup, we further grouped all levels into quartiles based on the magnitude of the coefficient, and assigned a new level to each quartile. As such, for significant levels of a categorical variable, we ended up with 8 new levels (Q1_Pos, Q2_Pos, Q3_Pos, Q4_Pos, Q1_Neg, Q2_Neg, Q3_Neg, Q4_Neg) and non-significant levels were added to the 'other' category.

This way, we successfully reduce dimensionality of all categorical variables under 32, which can be handled by our algorithms.

C1	**7**	**C14**	**32**
banner_pos	7	C15	8
site_id	**18**	**C16**	**9**
site_domain	18	C17	32
site_category	**26**	**C18**	**4**
app_id	18	C19	25
app_domain	**15**	**C20**	**25**
app_category	12	C21	27
devuce_model	**24**	**hour**	**24**
device_type	5	day_of_week	7
device_conn_type	**4**		

Table showing final number of levels associated with each categorical variable

We used the same level grouping that was created in the training data to update levels in the validation and test data. If the validation and test data had levels that didn't occur in the training data, these levels were assumed to be of low incidence and assigned to the 'other' category. This way, the three datasets shared the same grouped levels of factors.

Cleaning, Part 2

During model creation, we realized that for the techniques we were using, all levels need to exist and have records across training, validation and test data. To satisfy this rule, we found levels that didn't occur in all three datasets and dropped rows with these levels assuming they

have low incidences[3]. In effect, this had little impact on our data e.g. only one row was dropped from the training data in this process (such that nrow went from 19194654 to 19194653).

While we initially attempted to build models on the full training data (~19 million rows), we encountered long vector errors across multiple algorithms. Since this meant we couldn't use the full-size training data for model creation, we decided to use ensemble methods by splitting the training dataset into 10 chunks of roughly 2 million records.

Model selection

For model building, we opted to try the following five techniques. These are all appropriate choices for a binary classification problem.

- Lasso logistic regression
- Ridge logistic regression
- Classification Tree
- Bagging
- Random forests

We did consider using other approaches as well, but decided against them for various reasons. For instance, on attempting modeling using stepwise logistic regression, the compute times were extremely high which made this approach infeasible. We also explored implementing neural networks, but realized a substantial amount of research and adaptation was needed to use them for this problem. Moreover, given instructor feedback regarding project scope, we decided against this technique.

Modeling Approach

Our initial approach involved a three-way split of the data. In addition to the training data, we created an initial validation data that was to be used to optimize parameter values for each model, as well as the main validation data that would be used for model evaluation and selection.

For each technique, the following parameter optimizations were planned:

- Lasso and Ridge logistic regression: Optimize *lambda*
- Classification Tree: Optimize *minsplit*
- Bagging: Optimize *nodesize*
- Random forests: Optimize *mtry* and *nodesize*

[3] In practice, given earlier cleaning, this should only drop levels that occur in the training data but not in the validation sets. This is because our earlier cleaning assigned any unseen levels from the validation and test data to the 'other' category

While mini-runs of the code worked, we realized that our findings weren't scalable on the full data. For instance, we ran into size-limits for *mtry* optimization in random forests, and the computation for *nodesize* on random forests and bagging took an exceedingly large amount of time (such that it was infeasible). Given this, and additional challenges implementing even basic algorithms using our full data, we opted to switch to an ensemble approach. More specifically, we split our training data into 10 chunks (of ~1.9M each).

Based on our optimization attempts on both sampled and full data, we also decided to switch to using a single validation dataset instead. This approach made sense given what we learned for each technique, as detailed below

- Lasso and Ridge regression: The parameter we optimized was lambda. This optimization was performed via the cross-validation method that is built-into the package we were using *glmnet.* Since this creates mini-validation sets to find the best parameter value, there wasn't a need for two separate validation datasets.
- Classification Tree: For our classification tree modeling, we found that optimizing *minsplit* did improve performance (based on mini-runs on downsampled data). However, since having two validation datasets was not useful for any of the other algorithms, we instead implemented the parameter optimization by creating a mini-validation split using 2 of our training chunks instead.
- Bagging and Random forests:
    - The parameter we were trying to optimize for was nodesize. To find the best value of *nodesize* we looped over a list of possible values, but the runtimes were substantially high and didn't produce compelling results (e.g. for random forests, the default *nodesize* of 1 was the best according to the loop)
    - For random forests, we also tried to optimize *mtry* i.e. the number of features sampled for each split. To obtain this, we used 'tuneRF' function. But our testing results suggested the best *mtry* from 'tuneRF' was the default, so optimizing this on a separate validation data set didn't add much value. Moreover, tuneRf wasn't a feasible option, since the function has strict size limitations i.e. wouldn't run on even our chunks of 2 million records.

Per the above, our final modeling involved 10 sets of training data, and a single validation dataset (representing a 60-40 split overall).

Model Creation Details

Logistic Regression (Lasso and Ridge)

We mainly used two packages for fitting logistic regression and making predictions. First, we used matrix package *sparse.model.matrix* function to transform all levels of factors to a sparse

matrix. To make sure there was no dimension mismatch, we first combined all training chunks to produce a single sparse matrix and re-split it using the same sequence as the original training chunks for consistency across models. This way, all our X-variables were transformed to dummy variables in a sparse matrix. We opted to use a sparse matrix since it greatly reduces RAM usage versus a full matrix.

For model building, we used the glmnet package which fits logistic regression using a sequence of values for the regularization parameter lambda. The model essentially tests a series of lambda values ranging from 0 to 0.05, and gives us the opportunity to specify which lambda to use for prediction based on the associated error.

We set the input as our matrix, response variable as click, nfolds as 5 (i.e. 5-fold cross validation), lambda as the default value, family as binomial (i.e. for logistic regression). Alpha was set to 0 for Ridge, and 1 for lasso regression.

After fitting on the training data, we chose the lambda that gave the minimum mean cross-validated error as our best logistic model for prediction. This was then used to get the click predictions on our main validation dataset.

We repeated this approach across all 10 chunks and got 10 logistic models fitted on each chunk. We also got 10 sets of predictions for each of these models on our validation data. Taking an ensemble approach, we averaged the 10 sets of y-prediction values to get our overall prediction for the validation data, and evaluated it's log loss versus the actual.

Classification Tree

For coming up with our classification tree, we used the recursive partitioning technique. In this technique, we decided to tune the parameter *minsplit*, which is the minimum number of observations that must exist in a node for the algorithm to attempt a split. To perform the tuning, we needed a mini-validation dataset, hence we used 2 out of the 10 training chunks as mini-validation sets and only 8 out of 10 training chunks for creating tree models.

To obtain our final list of values of minsplit to test for each chunk, we used the first training chunk to determine a good range. We used this first chunk to build trees with 17 different *minsplit* values (2, 10, 25, 50, 100, 250, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000) and checked their performance over the mini-validation set. This gave our best parameter value as 6000 with log loss of 0.422. With this in mind, we created our final (shorter) list of values to tune on i.e. (20, 500, 2000, 5000, 6000, 7000).

We then looped over the 8 training chunks, each time getting the best parameter value from the list of 6 based on their performance on the mini-validation set. This gave us 8 lists of predictions on the main validation data. Finally, we ensembled the results by averaging out the predictions. These final predictions were then compared against the actual values to get the overall log loss value on the validation data.

Bagging and Random Forests

For both bagging and random forests, we used 100 trees to start and set other parameters as default to fit a model in each of the 2-million observations chunk. We then predicted probabilities on the validation set for each model and averaged the results across all 10 models to get the final log loss.

For random forests, we tried using the tuneRF function to find the model with best mtry and build the model directly but encountered the long vector problem despite only 2 million observations, so we decided to use the default mtry which is 4 for all the models.

When evaluating the results, we saw especially high variation in predictions for random forest, and realized that we probably needed to use a larger number of trees to control for this. As such, re-implemented random forest with *ntree*=250 for all 10 chunks, and saw improved performance versus the original model. However, overall performance for random forest was the worst from all our datasets on the validation data. This is likely since the "forcing" behavior of selecting variables in random forests didn't work well in this dataset, because it ignored feature importance when selecting the variables to build the trees while there are relatively many "noisy" variables in this dataset.

Model selection and validation results

The table below shows the log-loss results for each technique, as evaluated on the main validation dataset (40% of total training data).

Ensemble model (n=number of models)	**Log-Loss** (on Validation Data)
Lasso logistic regression (n=10)	0.40656460
Ridge logistic regression (n=10)	0.40684240
Trees (n=8)	0.42291560
Bagging (n=10, ntree=100)	0.84780370
Random Forest (n=10, ntree=250)	1.14117100
Random Forest (n=10, ntree=100)	1.26705600

Table showing results on validation data

Based on the results on the validation data, the logistic regression using the lasso penalty norm provided the best performance. This seems reasonable, since for models with a large number of sparse features, logistic regression is said to be faster to train and execute and less prone to overfitting.[4] It is also in line with the fact that Google's ad prediction system used an advanced

[4] Perols, J. (2011). *Financial statement fraud detection: An analysis of statistical and machine learning algorithms*. Auditing, 30(2), 19-50.

form of logistic regression[5], and hence, is a useful approach for advertisers seeking to predict ad click probability.

Test predictions

Since lasso logistic regression provided the best results on the validation data, we then obtained click predictions on the test data using this model.

Appendix: Details on Code file submitted

Code File	Contents
Team5-DataExploration	Contains initial exploration of data, and split of original training data into training and validation datasets
Team5-Cleaning-Part1	Contains initial stage of data cleaning for training, test, validation files. This includes creating the hour of day, weekday variables and grouping levels in each categorical variable such that none are above 32
Team5-Cleaning-Part2	Contains second round of data cleaning. Goes through training, validation, and test data and ensures variable levels are aligned across each. This file also splits the training data into 10 chunks for building multiple models to ensemble.
Team5-Model-Lasso	Contains code for running lasso logistic regression on training chunks, getting performance on validation data. Since this was the best model on validation, this file also includes the code to obtain test data predictions
Team5-Model-Ridge	Contains code for running ridge logistic regression on training chunks, getting performance on validation data.
Team5-Model-Trees	Contains code for running tree model on training chunks, getting performance on validation data
Team5-Model-Bagging	Contains code for running bagging model (with ntree=100) on training chunks, getting performance on validation data
Team5-Model-RandomForest	Contains code for running random forest model (with ntree=250) on training chunks, getting performance on validation data
Team5-InitialModel-ParameterTuning-Bagging	Example file that shows parameter tuning implemented for bagging on full data, prior to the use of the ensemble method. Similar approaches were run for random forest and trees as well, but ultimately technique was used via an ensemble method.

[5] McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., ... & Chikkerur, S. (2013, August). Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1222-1230). ACM.