Online Advertisement Click-Through Prediction

Table of Contents

Abstract

Our report analyzed the performance of various machine learning methods on their ability to correctly predict an Internet user's propensity to click on an online advertisement. To identify the method with the optimal performance, we split our data into 60% train and 40% validation. Then, we compared the minimal log-loss according to the formula below across many different binary class estimation methods, such as decision trees, randomForests, neural nets, XGBoost and more:

$$Log\ Loss\ =\ -\frac{1}{n}\sum_{i=1}^{n}(y_i \log p_i +\ (1-y_i)\log(1-p_i))$$

We ran each model on random samples of 10K, 100K, 1M and 3M data points. We found these values of n to be more than sufficient, given the limitations of the memory capacity of our available technology. Our results showed that the XGBoost model had the smallest log loss value across all values of n: 0.4062232 (using the 1M training dataset). Our most important insight gained from this data analysis is realizing that since our tree is not very deep, there are few significantly informative variables in our dataset, despite its width after being encoded. Not only does this make it easy to check with the business side if it makes sense ("sanity check"), but it also makes it easily interpretable.

Introduction

Our dataset contained 31,991,091 rows and 24 columns. Of the 24 columns, 23 were explanatory variables containing information about the online ad, the site the ad was displayed, the app used to visit the site and the device used to access the app, and a response variable, denoting whether a user had clicked the ad or not. This response variable was ‘click’ a binary indicator that took on a value of ‘1’ when users clicked on an ad or a value of ‘0’ otherwise. In addition, we had 9 anonymized categorical explanatory variables.

Based on how we intended to use the training data to inform our model building, we assumed that the training data was representative of the population, especially with such a large dataset.

Exploratory Data Analysis & Data Prep

Based on our initial variable observations, the overwhelmingly large size of our dataset prompted us to decompose our data into smaller and easier to use subsets, which we assumed were representative of our entire training dataset. Once the data was subsetted in R, we began preparing the data for modeling. Each step of this process was critical to ensure that our models ran smoothly and our results were easily interpretable.

The first step in cleaning our data was to explore each variable in our dataset. Using our subsetted data containing 6M records, we examined the class of each of our 24 variables. This initial first step helped us to understand what kind of data we were working with. This step informed us how we should most efficiently manipulate the data in order to properly format it for subsetting and variable creation later on.

Then, using the 6M subset, we examined the length of unique levels of each variable, and ultimately dropped the ‘id’, ‘device_id’, and ‘device_ip’ columns. ‘id’ had a length of unique levels which was close to the number of observations, so while it was not a true id, we found that since it was mostly unique in our dataset, it would not be extremely informative to any model building process. ‘device_id and ‘device_ip were dropped as well because they were essentially proxies for internet users, which should not be used in our data because this uber-specific user information will likely not be found in unseen data.

The only variable we modified was the ‘hour’ column. In our dataset, this column came in the format YYMMDDHH. For instance, an entry 19121907 is equivalent to December 19th, 2019 from 7am to 8am. Based on knowledge of this format, we broke ‘hours’ into a ‘Weekdays’ column, which contained the day of the week, and an ‘Hours’ column, which contained the hour value according to the 24-hour military time clock.

The next major step in our data preparation phase was to recode our variables. While many of our variables took on less than 10 unique values, some variables took on

hundreds or thousands of unique values. We sorted the length of each unique value for all categorical X variables in our dataset.

An example is shown in the figure below, which displays a frequency table for unique values of the variable ‘C16’ - an anonymized explanatory variable in a randomly sampled subset of 200K values from the original training data.

```
> table(Subset_160$C16)

    20     36     50     90    250    320    480    768   1024
    21   1593 188375    393   9014      8    519     15      6
```

In the figure, the top row of numbers indicates the value of the ‘C16’ variable, which seemingly takes on one of eight possible values: {20, 36, 50, 90, 250, 320, 480, 768, 1024}. The bottom row of numbers indicates the number of times that a particular value was present in the dataset.

In order to decrease the number of categories in each column, using common sense based on the proportion each category took up for each column to determine which top categories would be made into dummy variables. For instance, according to the figure above, since 50 appears in nearly 190K of the 200K rows, it is the top category of ‘C16’ and will be codified later as a dummy variable.

The process described above was sequentially repeated until either all categories had been codified or a set limit for the number of categories had been reached. For a column similar to ‘C16’, which only has eight unique categories, a limit was not reached. However, larger categories, such as ‘C17’, which contained over 400 unique categories, the top X number of categories was easily computed using the code shown below:

```
DT <- data.table(DT)
DT <- DT[ , -'click']
DT <- DT[ , !c('id', 'device_id',
'device_ip')]
DT <- apply(DT, 2, as.factor)
DT <- data.table(DT)
col_names <- colnames(DT)
top_cat <- list()
for(i in col_names){
 tmp <-
data.table(sort(table(DT[[i]]),decreasing =
T))
 top_cat <- c(top_cat, list(tmp$V1))
}
names(top_cat) <- col_names
n.obs <- sapply(top_cat, length)
seq.max <- seq_len(max(n.obs))
mat <- sapply(top_cat, "[", i = seq.max)
top_cat <- data.table(mat)
top_cat_cutoff <- top_cat[1:X, ]
```

With these adjustments, the number of categories for each original explanatory variable was computed to create One Hot Encoding. We decided to create enough top categories across all explanatory variables to have datasets with 50 factors. After careful crafting, our dummy variable creation process is appendable to any test dataset, such that any category or value that our model does not recognize was easily aggregated into an ‘other’ dummy variable. Based on our knowledge of the data, we believe it is improbable that any unseen category or value could significant impact our

category frequency rankings, given that we assume our training dataset follows a distribution that represents that of the true population.

Transformations, Scaling, and PCA

We did not apply transformations to any variables. There were no numerical variables for us to have to worry about skewness or scale that could throw off our algorithms. Because of this, we also did not have to worry about standardizing our inputs. However, we still needed to do scaling for NN of 0,1, which does not affect our data anyways because all our variables are already binary dummy variables.

Because all our variables were categorical, we could not use PCA to reduce the dimensions. Even if we assigned binary dummy variables or ordinal values, these would still be numbers that are not meaningful to do PCA on.

Models: Training and Validation

We chose to run variations on 8 types of models with adjusted parameters and different sized training sets. We started with 10K, 100K, and 1M/3M observations in each round of model building. We also created different dummy creating functions to deal with the limitations of certain models with how many factor levels can be present.

We used 10K to optimize the parameters of the models, then 100k to confirm how each model was doing based on those optimized parameters. We ran 1M on the top 3 models for a final round to see which model performed the best based on log loss. We use the model from the 1M round with the least log loss to generate our predictions on the test data. We chose to split our model building into this process because building on smaller datasets should give us a good idea of how good a model will be without having to waste as much time and computational power, and running the larger datasets only on models we think may be a contender for the best model would be the most efficient use of time and computational power.

Additional notes for some models

Logistic Regression:

With Logistic Regression, we implemented lasso regularization to prevent overfitting as our model is incredibly complex due to the number of variables. By using lasso, we should be able to essentially cancel out variables that are assigned a weight of zero.

Neural Network Binary Classifier with Keras:

For neural networks, we ran 4 versions of the model, as we cannot use grid search to determine how wide or deep to make the model. A 4x4, 4x10, and a 10x4 neural net. We used a sigmoid activation as our output layer because sigmoid is for binary classifications specifically while softmax is for multiclass classifications.

Log Loss Validation Results

	$Log\ Loss = -\frac{1}{n}\sum_{i=1}^{n} (y_i \log p_i + (1-y) \log(1-p_i))$		
Model	**10K**	**100K**	**1M**
Naive Bayes	**3.600513**	**5.484711**	**N/A**
k-NN Classification	**0.7535227**	**3.493225**	**N/A**
Logistic Regression	**2.031548**	**2.051319**	**N/A**
Decision Trees	**0.4258803**	**0.4409933**	**0.4369219**
RandomForest	**0.4724798**	**0.5888852**	**N/A**
Bagging	**0.4812438**	**0.4858376**	**N/A**
Neural Network 4x4	**0.5994437**	**0.5451024**	**N/A**
Neural Network 4x10	**0.6129936**	**0.5464956**	**N/A**
Neural Network 10x4	**1.478176**	**1.567287**	**N/A**
XGBoost	**0.4019809**	**0.4134947**	**0.4062232**

*Maxdepth = 10, nrounds = 30

Legend

Poor Model	Good Model	Great Model	Best Overall Model

Insights

Our best model was XGBoost, which is a decision-tree based Machine Learning algorithm that uses a gradient boosting framework.[1] We do not think it is a coincidence that our second best model was based on a decision-tree framework as well. In fact, each of these two models surpassed more complex models, such as the neural net model variations we tried.

We noticed that even though we have a lot of data, our trained trees were not as deep as one might think with the plethora of variables that are present in the dataset. Therefore, despite all of the many factors and levels present in our data,this still may suggest that there are only a few variables in this particular dataset that provide significant insight into predicting whether an advertisement will be clicked or not - our target variable. We believe that the short depth of our best performing model may help narrow down future studies in the size of their net, which may potentially save in costs on data storage, computer processing, and data collection.

Further Work

We acknowledge that both our model building and data testing phases each faced computational limitations based on our limited processing power. Obviously the more data we can train our models on, the better the performance. While we are confident that 1 million observations is sufficient to represent the population distribution, we would have liked to run 3 million observations to just ensure that we capture more of the true population distribution. We were limited by our computer's memory capacity and vector memory capacity.

In the future, we would also like to run cross validation testing using k-folds, to help improve the model and prevent overfitting because we will be able to account for every observation being represented in the model building. In the same idea, we would also like to resample to balance the skewed distribution of clicks and no clicks in order to understand our model's actual accuracy.

While we were not able to do PCA because this is categorical data, we would like to learn and apply MCA (Multiple Correspondence Analysis) in order to reduce the complexity of our model and reduce multicollinearity.

1 **XGBoost Algorithm: Long May She Reign!** "Xgboost Algorithm: Long May She Reign!." *Medium*. N. p., 2019. Web. 20 Dec. 2019.

Appendix

Bagging.R

- This code is the template we used to run our bagging models

BinaryClassificationNN.r

- This code reads in the output from the NN models in order to calculate the Log Loss from the NN

BinaryNN_4by4.py

- This code runs a NN of 4x4 using the sigmoid activation function and outputs csvs to be read into R

BinaryNN_4by10.py

- This code runs a NN of 4x10 using the sigmoid activation function and outputs csvs to be read into R

BinaryNN_10by4.py

- This code runs a NN of 10x4 using the sigmoid activation function and outputs csvs to be read into R

BinaryNN_10by10.py

- This code runs a NN of 10x10 using the sigmoid activation function and outputs csvs to be read into R

Decision_Tree.R

- This code is the template we used to run our decision tree models

final_prediction.R

- This code is where we ran our final XGBoost model on the test data to generate predictions

k-NN_Model.R

- This code is the template we used to run our k-NN models

LogLoss.R

- This is our wrapping function to calculate the Log-Loss evaluation metric for each model

Naive_Bayes_Model.R

- This code is the template we used to run our naive bayes models

Random_Forest.R

- This code is the template we used to our random forest models

to_dummy.R

- This code builds a function that creates all the dummy variables for any data based on the list of the top 50 categories of each variable in our sampled 6M data, and transformed the dummy variables to ensure they would be in an exactly consistent format across any circumstance (e.g.: fixing some special formatting issues that might arise when R makes model formulas automatically from the column names of dataset)

trim_dummy.R

- This code was used to fix issues with our output dummy variable dataset (some forced whitespace from R and forces whole dummy columns to all 0s if not present in any observations)

XGBoost_Model.R

- This code is the template for our XGBoost models, and how our research into the XGBoost model, since we did not learn this model in class.