Polymorph - Unified Auctions

Final Report

codebase
software consulting

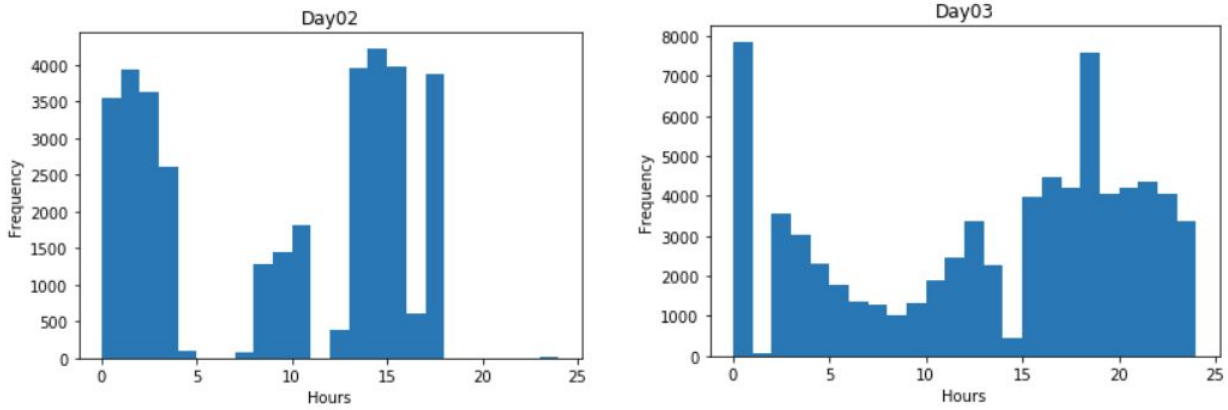
Data Processing Discussion

Exploratory Data Analysis

Keywords and URLs

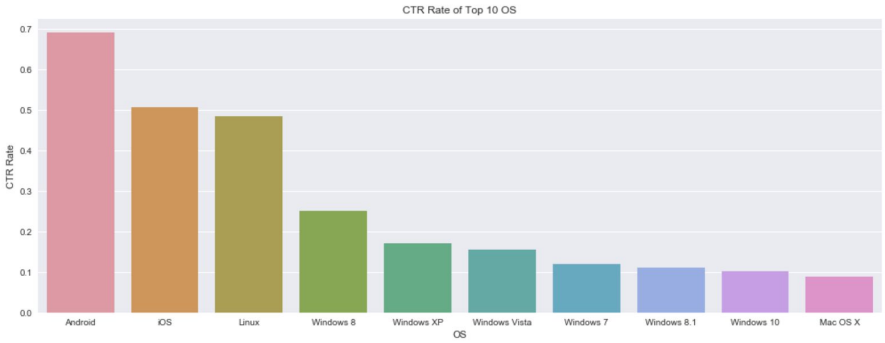
During EDA, we found that there was very little variety in the URL features. The top two referer domains, for example, constituted 80% of all referer values. The 'path' portion of these URLs (the portion after ".com"), however, had more interesting values, so we decided to extract those keywords and make them their own feature, which we one hot encoded to make trainable.

Click Time Distribution



We also found during EDA that the data we were working with was distributed very sporadically and inconsistently, as shown above. We speculate that this has to do with the data collection system and is not representative of when ads are requested throughout the day. But regardless of the cause, we learned to not emphasize time as one of our features since the inconsistent distribution may cause our model to skew in unwanted ways.

OS Type



The user's operating system was a good feature that we incorporated into our model. After calculating the CTR rate for the top 10 most common operating systems in our data, we found that Android and iOS (mobile) devices had significantly higher click-through rates than other systems, so we decided to include the user's OS as a feature. This is just one example of many ways in which we conducted exploratory data analysis and feature selection.

UUID

An initial look at the UUID column showed that it had a large number of unique values (as there are many unique users requesting ads), making a naive OHE technique impossible to use. Instead we analyzed this model by looking at the most common users and any trends we could find with these users. Interestingly enough, we found that about only 5,000 users made up 20% of the data, and even more surprising is that these users only made up 10% of the total clicks. This means that these frequent users are likely bots, or fake users; however, we were were unable to incorporate them into the models as they would overfit to certains days as the users didn't remain the same.

is_Bot

The is_Bot field is an exemplification of the null data tradeoff we had to make in our model. While this feature was very powerful, as when true, this meant the user wouldn't click, the field was null for most of the dataset. Additionally, we were unsure if it was data that we would presented with before the auction (or after). Overall, we dropped many columns like this that could lead to a much better prediction in a subset of the data, but could potentially add noise or be useless in predicting samples that they didn't have data for.

Data Modifications

Positive/Negative ratio modification

Before mid-semester deliverable, we trained on a positive and negative ratio proportional to the natural ratio in all the data. However, this was a problem because there were much more negatives than positives, so our model was only able to learn the features for the negative classes and was thus marking most things as negatives

To combat this problem, we train on a 50:50 split for positives and negatives. This allowed our model to be able to learn the features for both classes. We did this by collecting all the positive data on days 2-3, and then we downsampled the number of negative data points by randomly taking 0.2% of the negatives in those days.

Training on all auctions

We still tested on the unfiltered data from day one because we wanted the data to accurately reflect the data that we would see in the real world. Essentially, this means testing on a dataset that was between 2-4% click.

Revamping Preprocessing

Feature Selection - NaNs

Intuition

As machine learning models cannot train on NaN values, we needed some way to deal with these values in our data. A common practice is to either drop all samples that contain any NaN values, or to replace NaN values with some value. In numerical data, replacing with either 0 or the mean value of the feature works really well. However, for categorical data, switching out NaNs with the most frequent class or creating a new class altogether is very risky - we may overfit extremely hard to this most-frequent class or new class if there is a large composition of NaNs in the dataset. For example, if the most frequent class, class A, only has a 7% presence in the feature, and 20% of the feature are NaNs, then by replacing the NaNs with A would result in an artificially induced increase by 400%! We would be changing the data drastically, which is very dangerous.

Implementation

We took a look at 2 datasets - 1 exclusively for samples where the ad was clicked, and the other containing the overall data. We opted to drop all features that had more than 20% NaN composition, then drop all rows that contained any NaN values. This way, we only dropped around 10% of our overall dataset, whereas if we did not drop any features, we would have dropped more than 50% of our dataset when dropping rows.

Feature Selection - Timestamp

Intuition

Several of our features consisted of timestamps. Time can be considered a numerical feature, but this has its pitfalls - for example, if we represented time as hours 1-24, we introduce a leading bias into our later hours. Also, the fact that hour 1 comes after hour 24 cannot be represented numerically. As a result, we decided to one hot encode timestamps into time buckets.

We believed that dividing timestamps into minutes would cause severe overfitting - is there really that much a difference between 7:10 and 7:12? As a result, we divided our timestamps into hours.

Finally, due to the fact that timestamps were given from a centralized location despite the fact that ads were shown in different time zones all over the world, we utilized geographical data to convert these global timestamps to local timestamps, as this makes more sense.

Implementation

We used python's datetime library to convert global hours to local hours, and generated these hours by slicing the timezone data. We introduced this as its own feature - 'hour'.

Feature Selection - Keywords

Intuition/Implementation

We generated keywords by combining parsed words from the referer url and destination url features. However, this gave us a list of keywords, which our models cannot use. To deal with this, we one-hot encoded keywords into a different feature for each keyword.

Batch One-Hot Encoding

Intuition

Previously, with algorithms that only fit to the entire dataset at once, we ran into severe memory issues - we could not fully one-hot encode every sample, otherwise we would run out of memory! To deal with this, we introduced batch training for several models, allowing us to only load a tiny subset of the data into memory at once. This gave us much more freedom in one hot encoding. Whereas from before, we only one hot encoded the k-most frequent classes, we now encoded all classes that appeared more than k times throughout the data.

We still did not fully one hot encode the classes, but this is because of the dangers of overfitting. Classes that only appear a few times have high variation in the information

that can be learned from them, and fitting to them is dangerous, which is why we only limited our encoding to classes that appear at least k times.

Implementation

We found that setting k = 100 was a good value for batch one-hot encoding. This expanded our featureset from 39 features to 5300, which tells us much more information than before, where we would only expand to around 4-500 features.

Machine Learning Discussion

One New Metric: Log-loss

Log-loss will always penalize the classification unless it has 100% confidence in the correct class, which almost never happens. This means that the model will always be able to learn from every single prediction it makes. It also essentially has weights on the probabilities it predicts. The model will be penalized more for the confident incorrect classifications than unconfident incorrect classifications.

Log loss also gives probabilities, as compared to simply giving each class a 0 or 1. We thought that this would also be more useful for Polymorph, since you want to use these probabilities to decide how much revenue you would be expecting to generate from an ad.

Gradient Boosting

Motivation

After researching that so many Kaggle competition winners use gradient boosted trees to win CTR competitions, we decided to investigate XGBoost. Gradient boosted trees combine the simplicity and low overhead of decision tree ensembles with the optimizations introduced through gradient boosting algorithms. XGBoost is more powerful when compared to random forests, precisely due to its boosting algorithm, which allows it to support custom loss functions, giving it flexibility and generality to make better predictions. It is able to output probabilities, rank features, and performs automatic feature selection, all of which are very useful aspects that help improve our predictions.

Implementation

XGBoost follows gradient descent, which takes incremental step sizes in the correct direction. It calculates the derivative of the loss function and tries to bring it down to 0. This has been proven to work very well in many of the more recent Kaggle competitions.

XGBoost is also created from an ensemble of trees, which maximizes the probability that a selected feature will actually be useful. XGBoost also builds each ensemble off of the previous ensembles, which allows it to correct off the previous ensemble's error. This minimizes the error as well as decreases the training time, making XGBoost a more feasible option.

XGBoost offers two types of models, a native model and a scikit-learn API. We decided to use the native model because it is faster and outputs probabilities. The dataset is very imbalanced, so hyperparameter tuning had a large impact. XGBoost has very good documentation on which model parameters make it more conservative, so changing

parameters such as decreasing the learning rate to decrease overfitting, setting a maximum weight, scaling positive weights, and adding randomness to make training robust to noise, all improved precision (sometimes at the cost of recall). One nice feature of XGBoost is that we're able to do batch training, which is fast when we want to train on a million samples because it's less memory intensive due to the huge overhead of one hot encoding.

Results/Takeaways

Feature Dropping:

Precision and recall are very sensitive to hyperparameter changes for XGBoost. Our log-loss is relatively low, at 0.176, but this is likely due to the overwhelming amount of negative classes that our model got correct, evident by its high negative accuracy. Due to class imbalance, overfitting is very difficult to overcome even with thorough hyperparameter tuning.

No Feature Drop:

In the experimental no feature drop gradient boosting model (where no columns were dropped, but rather rows with NaN values), we managed to get an f1 score (0.139), 10 times higher than the rest of the models. This was mainly due to the very careful selection of positives it chose with it getting more than 50% of its predicted positives correct. While being incredibly accurate, this model does have the drawback of only applying to 50% of the data set (as opposed to 90% for the remaining data). In addition, due to the lack of filtering of columns such as UUID, it may have a tendency to overfit to a certain day's trends.

Field-Aware Factorization Machines

Motivation

FFMs is an effective model for classifying large sparse data, which makes it well suited for CTR prediction. In this setting, it is important to take the interactions between different features into consideration. For example, a male user is probably more likely to click on a Nike ad than a female user, so we need a way to take this into account during training. The model aims to solve this by learning two latent vectors of length k between each feature (i.e. two weight vectors for Nike-male interaction) and taking the dot product of the two. FFMs have been proven to work, as they have won numerous world-wide CTR competitions.

Implementation

We used xLearn, which is a high performance, easy-to-use and scalable machine learning package that is well suited for scenarios like CTR prediction. Upon hyperparameter tuning, we found that using a learning rate of 1 and a regularization parameter of 0.0001 resulted in the most optimal performance. In order to feed our data into the model, we were required to convert it into a text file with a specific format first. However, this conversion was very slow, so we used Dask to help parallelize the process. Even with the aid of this library, it still took around an hour to convert 600,000 samples into a text file on a 2016 MacBook Pro with an Intel Core i7 processor. Perhaps, future conversion can be done on a cloud computing service such as AWS to speed up this process.

Results/Takeaways

We achieved a positive accuracy/recall of 20%, a negative accuracy of 97.9%, and a log loss of 0.0644. Even with such a high negative accuracy, our model's precision was still 0.64%. The resulting F1 score was 0.0124. However, like the other models, the low log-loss implies our probability estimates are very good. Due to memory limitations from the text file conversions, we weren't able to train on as much data nor as many features as the other models. If this process can be streamlined with AWS, the FFMs could be trained on more data and potentially produce better results. Finally, the creator of the FFMs module is planning to add support for online learning, so there is definitely potential in the future to do this.

Naive Logistic Regression

Motivation

We took inspiration from a research paper published at Facebook which uses a combination of decision trees and logistic regression to predict CTR. The simplicity of logistic regression also allows us to minimize overfitting when there's a large and sparse feature set. Finally, we decided to pursue this model since logistic regression models the probability of the default class and is a commonly used tool for classification.

Implementation

We used SKLearn's LogisticRegression library. Upon hyperparameter tuning, we found that using a l2 penalty with alpha=0.001 resulted in the best performance.

Results/Takeaways

We achieved a positive accuracy of 10%, a negative accuracy of 99.55%, and a log loss of 0.03. Even with such a high negative accuracy, our precision was still 0.01376, which is to be

expected due to the heavy class imbalance in the testing data. This resulted in a f1 score of 0.0157. However, our extremely low log-loss implies our probability estimates are very good.

Logistic Regression with Stochastic Gradient Descent

Motivation

By using stochastic gradient descent, we could apply batch training to logistic regression, allowing us to apply batch one-hot encoding. This gave us more information, and we hoped that our models would be more effective from this.

Implementation

We used SKLearn's SGDClassifier, with loss function as log for logistic regression. Upon hyperparameter tuning, we found that using a l2 penalty with alpha=0.01 resulted in the best performance.

Results/Takeaways

We achieved a positive accuracy/recall of 22.5%, a negative accuracy of 98.21%, and a log-loss of 0.0625, which is very low (and good!). Even with such a high negative accuracy, our precision was still at a tiny 0.00813, which is to be expected due to the heavy class imbalance in the testing data. This resulted in a f1 score of 0.0157, which is not very good, but again, this is to be expected. However, our extremely low log-loss implies our probability estimates are very good. In conclusion, this model is very effective for its probability estimates.

Random Forests

Motivation:

Our motivation for initially looking into random forests is that they require relatively low amounts of preprocessing. This is because this model inherently selects which features it finds as valuable and is good at choosing a sparse array of features for prediction. This made it a good investigation tool before using other models for getting a baseline estimate.

Implementation:

We used the Sklearn Random implementation of random forest (both traditional and boosted). In this model, one of the key hyperparameters we found useful was balancing the tree depth and number of trees. Increasing these too much led to overfitting

(and taking too long to train), while having a low value led to underfitting and bad performance.

Results/Takeaways:

In the end, after tuning parameters, we managed to achieve a f1 score of 0.036 (precision 0.02 and recall 0.19) on the previous data we had. This score was very promising for this initial data set, but unfortunately had little room to grow. This is why after the mid-semester deliverable we chose to move to a more robust and faster model of the Random Forest in XgBoost.

Final Discussion and Takeaways

Feature Set Discussion

Our feature set is as follows:
_host, ad_network_id, advertiser_id, c_cnt, c_flag_cnt, campaign_id, campaign_type, f_cnt, geo_city_name, geo_country_code3, geo_region_name, geo_timezone, i_cnt, i_flag_cnt, i_timestamp, Pub_network_id, r_cnt, r_num_ads_requested, r_num_ads_returned, pub_network_id, r_cnt, r_num_ads_requested, r_num_ads_returned, r_timestamp, rate_metric, referer, session_id, site_id, token, ua, ua_device, ua_device_type, ua_major, ua_minor, ua_os_name, url, user_agent, uuid, vi_cnt, vi_flag_cnt

As seen, most features were kept, except for some notable exceptions, such as geo_country_code1/geo_country_code2 (as geo_country_code3 is more descriptive), is_bot, and uuid. Some features that were the most effective were the counts, such as i_cnt and vi_cnt, and ua_device_type. Other effective features were campaign_id and ad_network_id.

Optimal Model Discussion

	F1	Log-loss	Precision	Recall/ Pos. Accuracy	Neg. Accuracy	Sample #	Time to train
Naive Logistic	0.024194	0.031131	0.013761	0.1	0.995517	100,000	30 sec
Logistic w/ SGD	0.0157	0.0625	0.008137	0.22521	0.98214	**1,000,000**	3 hours
Gradient Boosting	0.01405	0.17648	0.00713	**0.47368**	0.95821	100,000	**10 sec**
Gradient Boosting - No Feature Drop	**0.139**	**0.01456**	**0.48**	0.1008	**0.9998**	25,000	30 sec
Ensemble of FFMs	0.012	0.0698	0.0062	0.21194	0.976	600,000	15 sec

To summarize the performance of our models when not dropping the count columns, above is a chart of all the important metrics, in addition to number of samples used to train, and time taken. Gradient boosting with no feature drop performs the best, as

the bolded numbers indicate that its f1, log-loss, precision, and negative accuracy are the highest. However, these results have to be taken with a grain of salt, as 50% of all the data given to us had to be dropped, since many rows have NaN values. If choosing from a model that drops columns instead of rows (in order to retain 90% of all rows), the logistic regression models are the best alternate options. This is because its f1 scores are higher and log-loss errors are lower than FFMs and naive gradient boosting. Naive logistic has better numbers, but logistic with SGD (stochastic gradient descent) has not been fully explored and has more potential. Further explorations with SGD include changing the k values to prevent overfitting, as it does better than naive logistic regression on validation data, suggesting that the lower scores result from overfitting.