Marc Kirchner
Improving Hotel Ad Performance
General Assembly Data Science sec. 4 – August 2013

1 Motivation

Many e-commerce sites depend on revenue that's based on advertising. Sites let external providers place ads on their pages. Many of these external providers are ad exchange platforms that allow advertisers bid in real time for an ad placement on the publisher (the site). One commonly used business model is based on cost-per-clic (CPC) in which the advertisers only pay the publisher when their ad ends up being clicked. In those cases the ad exchange platform also gets paid per click.

Therefore it's critical to place ads that are likely to be clicked. Ads that are relevant to the user who's interacting with the site have more chances of being clicked (and being actually useful to the consumer).

My project has been focused on a specific case with which I'm familiar with, as it's one of our client's in the company I work for, where we host an ad-exchange platform. We offer several products and I focused on a specific one: Sponsored Search Ads. One of our clients is travelocity.com and what we do is place hotel advertisements in the hotel search results of the site.

The current click-through-rate (CTR) of that product is 1.67%, which means that only that percentage of users click on an ad placed on the search results page.

One reason for that low CTR may be that the decision about which hotel ad should be shown to the user is based entirely on the highest advertiser bid. There is no logic to pick the ad that's most relevant to the user.

But how can we know what type of hotel is the user looking for? Is the user searching for a hotel for leisure purposes or for business? Would it be better to show a hotel, let's say, close to the airport and with conference rooms or one closer to the touristic attractions and family friendly? What about romantic spots?

One thing we know from the user are the parameters that define the intent. These are: destination, check-in date, checkout date, number of travelers and number of rooms.

The hypothesis made is that we can pair the user's intent with each of the candidate hotel advertisements and come up with a probability of click for each pair. If we can do that, given a user's intent we would pick the hotel ad that has highest probability of click and give the relevant ad priority in the auction that decides which ad to place.

To calculate the probabilities we will train a classifier using historical data. Each pair of features (user features and hotel features) along with the class for each pair (click vs not click) will represent an example in the training set.

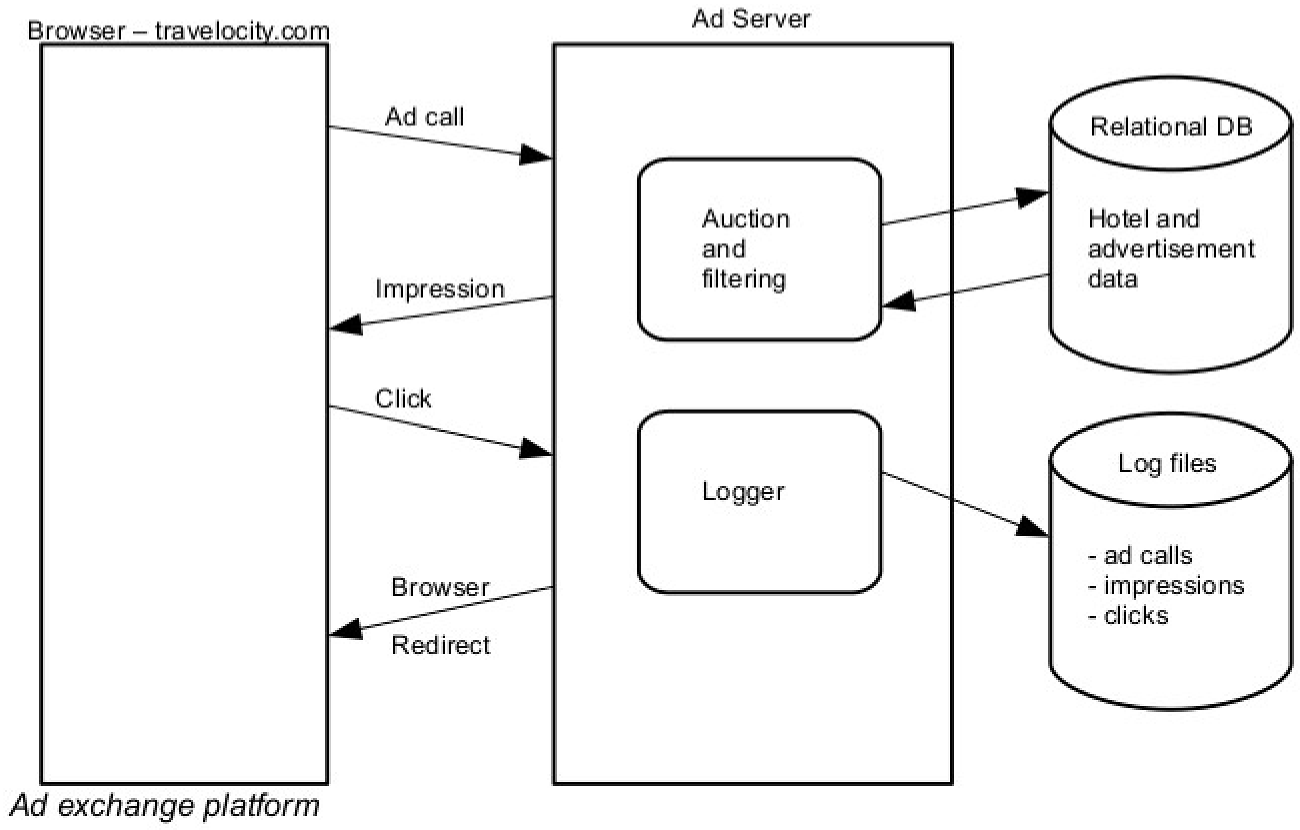
2 Data

The ad exchange platform that runs the auctions keeps a log of all the hotel ad requests, the ads being chosen to be displayed and the ads being clicked.

The ad call requests hold data about the user's intent (search parameters: destination, check-in date, checkout date, number of travelers, number of rooms) and other data carried on the HTTP request (HTTP headers, cookies, referral URL's, etc).

The ad chosen is also logged in the system as an impression record. An impression record holds data about the ad and the corresponding hotel. It's logged along with a unique ID of the corresponding ad call.

Lastly every click that comes to the system is also logged along with the corresponding ad call ID. This way we are able to link clicks with ad calls and impressions.



Data Extraction

The first challenge regarding the data was about the data extraction itself. The system log files are flat files hosted in Amazon S3. Each file contains much more data than the one relevant to our problem, or at least much more data than we would be able to process at once in our laptop.

As an example, one day's worth of Travelocity data held in our systems is in the order of Gb (ad calls: 2.5 Gb, impressions: 4Gb). We wanted to perform the experiment with data for all the year 2013 in order to have enough examples to extract useful patterns. Looking at the data fields we selected the ones that would be used as features for the model. That would narrow down the volume quite a bit, but we still had to deal with too much data. We decided to limit the experiment to searches for hotels in New York City. This way we would have a manageable data set and at the same time we would be focusing on a particular market. It's quite possible that different cities have different traveling patterns.

The actual data extraction process was implemented as a PIG script. We run a cluster of 20 nodes in Amazon WS using the Mortar Hawk interface. The PIG script had to be able to parse the log files. For that purpose I was able to use a library that my co-workers in the data team had written. The library defines a JsonLoader class that's plugged into the PIG script as a JAR extension. From my company's available code we also used PIG macros that had been defined to provide basic filtering of the data (remove data coming from blacklisted IP addresses, erroneous requests coming from web crawlers and other bots, etc.).

Loading data into PIG was the first step towards feature extraction. In the script we applied several projections and filters to discard ad calls that were not served and impressions that didn't correspond to the placement we were focusing. We also filtered ads that didn't correspond to New York searched. To do so we wrote a python UDF that would extract the city from the field that contained that data embedded in a URL.

Once we had the data filtered in each PIG relation: ad_calls, impressions and clicks, we first joined ad_calls and impressions in an INNER join. We are only interested in ad calls that have impressions. The result of that join was then joined with clicks in a LEFT JOIN. That's because not all impressions produced clicks and we are obviously interested in extracting data for both scenarios (there was a click in the impression vs there wasn't one).

After extracting the selected features and filtering the data to focus on the New York market we were able to have a manageable data set of 24Mb, for a period of time spanning from January 1st 2013 to August 7th.

3 Features

As mentioned earlier the training data for our model will be a set of examples composed of user / intent features and hotel / advertisement features, along with the label or class assigned (click vs non-click).

In the user features we included two features that are not related to the intent. Those are related to the operating system and web browser that the user is performing the search from. These features may give a relevant signal of clicking behavior in general.

The initial set of hotel features we could extract from our log files were pretty limited.

Here is the initial set:

User / intent features	Hotel / ad features
Number of travelers	Number of stars
Number of rooms	Does the ad contain the word “promotion”
Is the search performed on a week day?	Does the ad contain the word “discount”
Advance purchase range type *	Does the ad contain the word “free”
Operating system	
Browser family (IE, Firefox, Chrome)	

The “advance purchase type” feature represents a classification of the user intent based on check-in and checkout dates. The different classifications are:

1. Week day traveler searching for a hotel more than 21 days in advance of the trip
2. Week-end traveler searching for a hotel more than 21 days in advance of the trip
3. Week day traveler searching for a hotel less or equal to 21 days in advance of the trip
4. Week-end traveler searching for a hotel less or equal to 21 days in advance of the trip

We felt the need to add features related to the hotel. The problem is that we don't hold much of the characteristics of each hotel in our systems. So we decided to use the travelocity.com hotel details page to scrape details for each hotel relevant to the experiment.

Specifically, we decided to scrape the text that lists the amenities for each hotel: restaurant, swimming pool, conference center, etc.

The python script used for that had to perform a hotel search prior to hitting the hotel details page for each hotel. That was because the details page required an HTTP session already created, and that session seemed to be created during an initial search.

The python script would then just download the whole HTML page for further processing.

The number of hotels we downloaded the details page for was 321. The hotel IDs were obtained by dumping data from the relational database used to manage advertisers. That data was extracted as a CSV file. The python script processed each of the 321 rows by using the hotel ID to construct the hotel details page URL.

This way we could download 321 HTML pages but we noticed that many of them didn't have valid details data. It seemed that some hotels didn't produce a details page when requesting it. The reason for that may be stale data in our systems (hotels that used to be present in Travelocity's systems and they aren't anymore). However, bearing in mind that we are only interested in hotels that have been shown in ads during 2013, we assumed that most of those would have valid amenities information.

The process of adding amenities features to each example in the training set involved the use of a CountVectorizer. The vectorizer produced a set of unique words representing amenities across all hotels. Each training example would have then an additional feature for each possible token representing an amenity. That feature would be either true or false depending on the existence of the amenity in the corresponding hotel details information.

This way we could come up with an additional set of 190 features. Here are some examples:

```
'has_dining': 'n'

'has_airport': 'n'

'has_gym': 'y',

'has_wi': 'n'

'has_tv': 'y'

'has_children': 'n'

'has_conference': 'y'

'has_pool': 'n'

'has_tennis': 'n'
```

4 Model

The full data set for 2013 contained 75,000 of examples spread across 100 files, output of the Map/Reduce reducers. We decided to use that data split to separate 10% of the data as out-of-sample data. That data will be used to generate predictions on probabilities of clicks after training and testing the model. This way we will be able to explore the characteristics of those predictions.

Training / testing

We used a Logistic Regression model. The implementation chosen was the one provided in the `sklearn.linear_model` Python package. As a baseline we used the default model.

We fit and tested the model using the cross validation scoring functionality in the `sklearn.cross_validation` package.

Our positive examples (clicked ads) are quite sparse in our data (1.67%), so we were interested in the AUC (Area Under Curve) metric in the `sklearn.metrics`, which would provide a measure of how well the model is classifying the examples regardless of the ratio of positive vs negative examples.

The initial results were discouraging: 0.5 AUC across cross validations. This meant basically that the model predictions were as good as random predictions.

From this baseline we run the model using different configurations, like the regularization parameter C. But the configuration that made a real change in the model test results was the use of weights to over-sample or under-sample the classes in our training set. This meant that our sparse positive examples would get over-sampled when creating the different training sets in the cross-validation cycles. The specific setting for this configuration parameter was `class_weight='auto'`

Testing different settings for the regularization parameter we came up with this results in the cross validation:

```
==============================================================
Using model = LogisticRegression(C=0.014, class_weight='auto')


Cross validation results:
[ 0.58314185  0.56783342  0.59888981  0.57044815]
==============================================================
```

The model behaved better than a random model: approx 0.58 AUC.

Predictions on out-of-sample data

Running the model to produce click probabilities on out-of-sample data would tell us if there were significant differences of probabilities across examples. This is a summary of the probabilities predicted on oos data:

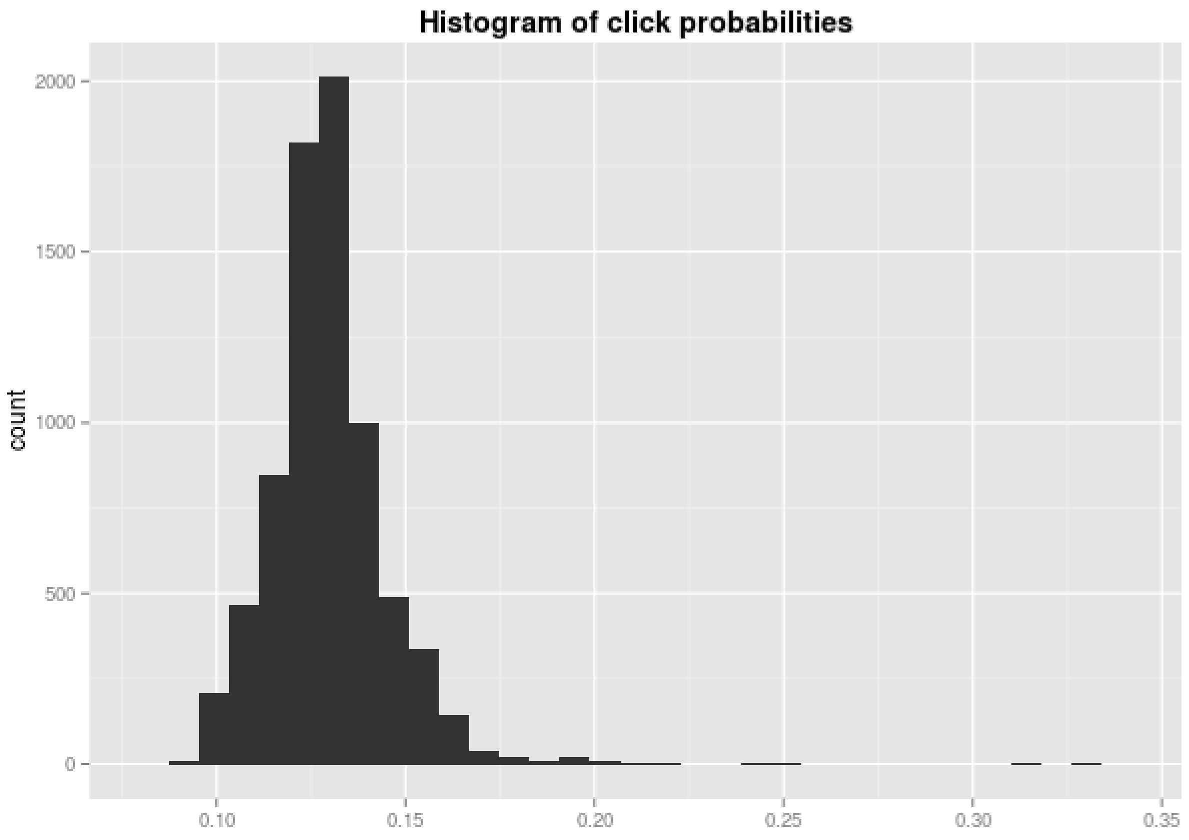
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.09245 0.12010 0.12740 0.12930 0.13540 0.33100
```

We appreciate a big difference between the minimum and the maximum probability but it seems that the higher probabilities are concentrated in the last quartile, which means that there may be only a few examples with higher probability than the mean.

To have a visualization of those probabilities we extracted the predictions into a CSV file and used the R `ggplot2` library to produce a histogram.

The histogram shows the number of occurrences of each probability data point, having them previously grouped in “bins” corresponding to probability ranges. That's because the probability numbers have many decimals and we don't want to see an occurrence of 1 for each number.

Looking at the graph we see that there are only a few examples having a significantly higher probability of click.



5 Conclusions / Note about OOS predictions.

The conclusion extracted is that we may be able to increase the probability of click for certain searches but those searches may be a very low number.

There are some improvements that could be made but first of all I want to highlight the fact that I should have performed the out-of-sample predictions differently. The probability histogram shown in the graph above is for predictions made on real user-hotel pairs, meaning, pairs of user searches and the hotel that was shown for each search. That's not how we would run the model in reality. As mentioned in the document introduction the intention was that once the model was trained we would predict probabilities for ***each possible hotel*** in the system. This means that the out-of-sample predictions should have done adding pairs to the data set, particularly, adding N-1 pairs for each original pair where N is the number of hotels relevant to the search in the system (New York hotels in our case). The added pairs would contain features for all the hotels that ***were not shown to the user***.