A Large Scale Prediction Engine for App Install Clicks and Conversions

Narayan Bhamidipati, Ravi Kant, Shaunak Mishra
narayanb,rkant,shaunakm@yahoo-inc.com
Yahoo Research

ABSTRACT

Predicting the probability of users clicking on app install ads and installing those apps comes with its own specific challenges. In this paper, we describe (a) how we built a scalable machine learning pipeline from scratch to predict the probability of users clicking and installing apps in response to ad impressions, (b) the novel features we developed to improve our model performance, (c) the training and scoring pipelines that were put into production, (d) our A/B testing process along with the metrics used to determine significant improvements, and (e) the results of our experiments. Our algorithmic improvements resulted in a 3X improvement in satisfaction for app install advertisers on our ad platform. In addition, we dive into how sequential model training, deep learning, and transfer learning resulted in a further 7% lift in conversion rate and 11% lift in revenue. Finally, we share the scientific, data-related, and product-related challenges that we encountered – we expect others across the industry would greatly benefit from these considerations and our experiences when they kick-start similar efforts.

CCS CONCEPTS

•**Information systems** → *Online advertising;*

ACM Reference format:
Narayan Bhamidipati, Ravi Kant, Shaunak Mishra. 2017. A Large Scale Prediction Engine for App Install Clicks and Conversions. In *Proceedings of CIKM'17, November 6–10, 2017, Singapore.,* 9 pages.
DOI: https://doi.org/10.1145/3132847.3132868

1 INTRODUCTION

There are over 5 million apps available in the two major app stores, Google PlayStore and Apple App Store [18], today and with close to 100 thousand new apps published every month. Most of the apps do not attract millions of users organically, so app developers rely heavily on advertising to increase their user base. The cost they are willing to pay for each new user that ends up installing their app is proportional to the value that this user is expected to bring and depends on a number of direct and indirect factors. The direct factors include the amount the user is expected to spend on in-app purchases, and how engaged the user is likely to be. On the other hand, the indirect factors account for the value that the individual contributes to beyond just activity in the app, say, by getting more friends to install the app, or by simply adding to the volume of the user base to reach milestones that benefit the developer – for example, when the user base of the app crosses a certain threshold, the app may be featured in news articles and other lists driving organic growth, or the app valuation may increase super-linearly with the size of the active user base.

Needless to say, driving app-installs turned out to be a lucrative business, with the mobile app-install revenue in US alone from the major ad networks generating USD $5.7 Billion in 2016, and some startups (*e.g.*, Chartboost and Tapjoy) supporting app install campaigns exclusively [6]. Yahoo Gemini has an app-install offering, too, and serves app-install ads across various Yahoo apps (*e.g.*, Yahoo Mail, Tumblr, Yahoo Finance). The prediction engine for clicks and conversions that we describe in this paper was mainly developed to support Yahoo Gemini's app-install offering. At a high level, the goal of this engine was to increase advertiser satisfaction and lower advertiser churn while maintaining revenue metrics (*e.g.*, cost per impression). To achieve this goal, we built a scalable machine learning pipeline for better estimating the probability of users clicking on app install ads and the resulting conversions (*e.g*, installing the app, in-app purchases, and other in-app activity). In the context of accuracy and scoring latency challenges naturally associated with such a large scale as Yahoo Gemini, our major contributions can be summarized as follows:

(1) model training improvements (via sequential training),
(2) latency improvements (via sequential scoring),
(3) cross feature engineering (via deep learning and transfer learning).

The remainder of this paper is organized as follows. Section 2 covers the background and related work. Section 3 describes the high level considerations that went into modeling, feature engineering, training and testing. Section 4 deals with our contributions toward improving model training (sequential training) and scoring latency (sequential scoring); this is followed by Section 5 on engineering cross features, including those that were obtained using deep learning and transfer learning, and Section 6 on novel features synthesized from raw features. Section 7 covers our offline evaluation, as well as, the results of online experiments. Finally, in Section 8, we discuss some valuable learnings gained from this project, something that we expect to be of wider interest to industry and academia alike.

2 BACKGROUND

2.1 Online advertising

Yahoo Gemini is an ad marketplace that unifies search and native advertising and reaches 600 million monthly mobile users [21]. When these users launch Yahoo apps or other Gemini-affiliated apps, they are served relevant native ads along with the content.

CIKM'17, November 6–10, 2017, Singapore.
 ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: https://doi.org/10.1145/3132847.3132868

The ads themselves vary in their objectives, with some vying to just show up in front of the user, while others want the users to visit a particular website, and yet others aim for the user to install an app. Throughout this paper, we would restrict ourselves to app-install ads on Yahoo Gemini.

Here we provide more details about terminology that is fairly standard in the computational advertising [4] literature, but is critical for understanding this paper. We have summarized terms relevant to this paper in Table 1, and describe the same below in the context of Yahoo Gemini.

Publishers (e.g., Yahoo Mail app) care about revenue (advertisers' spend - any payouts), *CPM* (cost per mille), Yield-Rate ($CTR \times CVR$). Advertisers, on the other hand, measure the performance of their campaigns using metrics such as *Click-Through-Rate* (CTR), *Conversion-Rate* (CVR), and *Cost-Per-Install* (CPI) which is the app-install equivalent of *Cost-Per-Action* (CPA). As part of the campaign setup, advertisers create one or more adgroups, each of which includes one or more creatives (referred to as ads, for simplicity). For each adgroup they specify a *bid*, the maximum amount they are willing to pay per a certain action, with an action being an impression (or view), a click, a completed view, or a conversion. The *pricing types* supported by Gemini for app-install ads are *Cost-per-Click* (CPC), where advertisers pay only when users click on ads and *Cost-per-Completed-View*, where the payment is only when a video is watched to the end.

Advertisers also have the *oCPC* (optimized *CPC*) option where they pay for clicks, but specify a CPI target that Yahoo Gemini optimizes towards [22]. Consequently, campaign efficiency (η) is defined as $\frac{Actual\ CPI}{Target\ CPI}$, and the efficiency histogram is produced by grouping campaigns by efficiency bins. Finally, to capture a notion of satisfaction over all the app-install advertisers participating in the Gemini marketplace, we defined happy spend and happy campaigns as the spend and count, respectively, of the campaigns with efficiency below $1 + \epsilon$, where ϵ represents the tolerance relative to the Target CPI provided by advertisers.

At each opportunity, a unified auction is run as follows. The probability $P(action|impression\ of\ ad_i, user, context)$, denoted $pctr_i$, is computed using a machine learnt model for every eligible ad – be it app-install or not, and the $eCPM_i$ of ad_i is computed as $bid \times pctr_i$, *i.e.*, it is the (maximum) amount the publisher can expect if the ad is shown. Ads are ranked in descending order of eCPM, and only the top ad(s) get to be shown to the user. The actual cost to the advertiser is determined using a *Generalized Second Price*. Advertisers have the option to let Gemini optimize their CPC or CPCV campaigns towards a conversion goal, so instead of specifying a bid, they provide only an *ecpa_goal* per adgroup, which is the target CPI for the adgroup, and the pricing type is referred to as *oCPC* and *oCPCV*, respectively. The bid is then computed as $pcvr \times ecpa\_goal$, where $pCVR = P(install|click)$, is also as estimated using a machine learnt model. The eCPM is then computed as earlier.

2.2 Related Work

In the context of our click and conversion prediction setup, we describe below prior related work on click prediction models and deep learning.

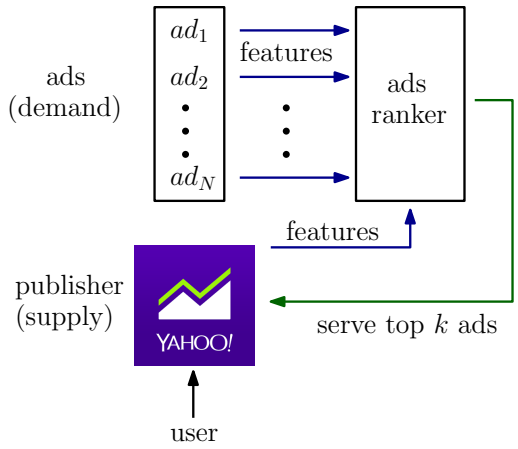

Figure 1: Ads ranker setup.

Table 1: Online advertising terms

revenue	$\triangleq$	advertisers' spend - any payouts
CPM	$\triangleq$	$\frac{spend}{impressions} \times 1000$
CTR	$\triangleq$	$\frac{clicks}{impressions}$
CVR	$\triangleq$	$\frac{conversions}{clicks}$
pCTR	$\triangleq$	$P(click\|impression)$
pCVR	$\triangleq$	$P(install\|click)$
Yield-Rate (YR)	$\triangleq$	$CTR \times CVR = \frac{installs}{impressions}$
bid	$\triangleq$	pCVR $\times$ eCPA_goal
eCPM	$\triangleq$	bid $\times$ pCTR
efficiency (η)	$\triangleq$	$\frac{Actual\ CPI}{Target\ CPI}$
Happy Spend	$\triangleq$	$\sum_{i:\eta_i \leq 1+\epsilon} spend_i$

Click prediction models: Accuracy of click prediction models plays an important role in auctions for ads [2, 3, 5, 7, 11, 14, 15, 19]. Training and scoring such prediction models at a large scale comes with its own set of challenges (*e.g.*, permissible model complexity). In this context, LR models have been successfully used in large scale click prediction setups [12, 20]. In particular, an FTRL-Proximal online learning algorithm for LR was used in [12], and in [20] boosted decision trees and LR were combined to get significant performance lifts. Motivated by the success of LR in such large scale setups, we considered an LR based pipeline for our click and conversion prediction models (with additional refinements like sequential training and scoring).

Deep learning and feature engineering: Feature engineering plays an important role in improving the accuracy of linear prediction models (*e.g.*, LR). In particular, hand crafted interaction features (based on domain knowledge) can be very effective. However, manually designing such interaction features requires considerable domain knowledge, and is not a scalable approach in our setup. Deep learning [9, 16] has been been successful in overcoming this challenge in multiple domains (*e.g.*, image recognition, speech); it

offers the potential of learning directly from raw features without a serious effort on feature engineering. In [17], the authors use a deep neural network to do feature *crossing* in an implicit fashion. They focus on sponsored search, a setup closely related to ours, and show significant performance lifts using their deep-crossing model. However, in our context, using such deep neural networks for online scoring is seriously constrained by scoring latency and hardware costs (*e.g.*, need for GPUs). Another approach for having cross features is via factorization machines [13]. For the scale in our setup, factorization machines are constrained by online scoring latency (our feature space is of the order of 10^6). In contrast to the works described above, we focus on a scalable approach for reverse engineering (via deep learning) a set of cross features for use in our LR based pipeline (within our latency constraints).

3 SYSTEM ARCHITECTURE

In this section, we give an overview of the modeling approach and feature engineering for our prediction engine.

3.1 Modeling

A model is a simplified representation that explains the phenomenon we are observing. In our context, the objective is to explain what makes a user more or less likely to click on app-install ads and go on to install and use the app. Given that app-install ads have CPC and CPCV pricing types, a CTR model is required for every ad serving opportunity. In addition, to enable optimization (oCPC and oCPCV), a CVR model is required. For the CTR model, the observable outcomes given the impression of the ad are clicks and completed views for Text and Video ads, respectively. The CVR model on the other hand, observes conversion notifications given a click.

Listed below are some high level considerations that went into modeling.

- *Monolithic models vs. vertical specific models*: The click and conversion behavior of users varied widely between different supply types. Some verticals that had predominantly basic style supply (*e.g.*, mail), had a higher accidental click rate, resulting in lower CVRs. The user bases were also very specific to the verticals. While there would certainly be signals that would generalize from one vertical to the other, we decided to build vertical specific models with each model being trained on relatively homogeneous data. This also helped us to scale better. Another advantage of having vertical specific models is that it acts as though every feature is crossed with the vertical. In addition, we refined our vertical specific models through transfer learning on the data pooled from all verticals (details in Section 5.2).
- *Choice of linear vs. non-linear models*: The CTR and CVR estimation problem could be solved using linear models like logistic regression (LR), as well as, non-linear models like GBDT [20] and neural networks. We picked LR as it is tried and tested, and scoring is extremely simple and easily scalable, all without sacrificing performance quality.
- *Two-step LR (sequential training and sequential scoring)*: As one of our novel contributions, we used a two-step logistic regression model, where the model was first trained on a set of primary features, and then secondary features were added to refine the residues from primary features (details in Section 4)). Such *sequential training* had two advantages: (i) noisy features could be added in the second step of training to make the model more robust, (ii) features which are expensive to compute online (*e.g.*, user-demand cross features) could also be added in the second step for reducing scoring latency (details in Section 4). In particular, the way we set up sequential training enabled a natural way to do *sequential scoring* for a set of candidate ads (*i.e.*, filtering some ads based on first step LR scores) and greatly improved our scoring latency (details in Section 4).
- *Whether to consider the position of the ad*: Unlike traditional stream ads and sponsored search, where impressions at different positions are to be treated differently, the influence of the rank of an ad on the click is far less in mobile ads due to the scrolling behavior on a smaller screen. So, while the chances of a lower ranked getting an impression may be lower, if the user does scroll down to it, the experience is very similar to that of the top-ranked ad. There could be other minor behavior differences between users that could introduce a bias (one example being that lower rank ads in the stream may appear near less interesting content, and hence the ad might be more interesting to the user), but given how they are not widely applicable in all the app experiences, we left such differences to be revisited after the big problems are solved.
- *Coarse vs. fine features*: Features for the model were picked based on their significance to app installs. Naturally, all available demand features, supply features, user features, and context features were utilized. The modeling question, though, was how the coarse and fine features were to be treated. Our logistic regression model is trained on both coarse (*e.g.*, country, app category, supply vertical, *etc.*), as well as, fine features (*e.g.*, state, campaign and creative ids, section ids, *etc.*) simultaneously, with the assumption that the model training would be smart enough to pick the finer features when there is enough signal in them, and discard them as noise otherwise.

3.2 Feature Engineering

One of our priorities was to develop the ability to rapidly add new features into the model, test them in bucket and promote to production if they perform well. A significant effort was required for feature engineering along the following directions:

(1) Cross features: Linear models do not learn feature interactions well, and there are many studies in the literature on how to add cross features to mitigate this weakness. We experimented with a few approaches (details in Section 5):
   (a) hand crafted cross features,
   (b) cross features from GBDT,
   (c) cross features from transfer learning with multilayer perceptrons (MLP).

(2) Synthesized features and binning: New user features were synthesized based on activity history (details in Section 6). These went a long way in improving our models. Also,

continuous and high arity features in their raw form could not be effectively consumed; so features needed to be discretized. We also needed to control the sparsity introduced by high arity features. We used binning (for continuous features) and clustering (for high arity features).

3.3 Training and Testing

To evaluate the training quality of CTR and CVR models, we compute:

(1) logloss and AUC (area under the ROC curve)
(2) overfit (difference between the train AUC and test AUC),
(3) bias (ratio of expected to actual number of positives).

Training process in our system involves the following steps: (1) generating the labeled data, (2) data pruning, (3) splitting the data to test and train sets for model evaluation, (4) training a two-step LR model, and (5) parameter tuning. We describe some of these steps below.

Data Pruning. During this step, we filter out models with insufficient examples (*e.g.*, if a particular vertical has very few examples). In addition, we filter out extremely rare features.

Splitting the data for model evaluation. Input data for each model is split into train and test subsets. Train data is used to generate the models, whereas test data is used for evaluation of the models and parameter tuning. We randomly select 75% of the data for training and the remaining 25% for testing. We split the data at the user level; this means that each user appears in only one of the train and test sets.

Parameter Tuning. Choosing hyperparameters that maximize the AUC on the test data is a common approach for model selection. Our experiments indicated that a large difference between the train and test AUC is a strong indicator of model overfitting. Therefore, we chose models with the best test AUC satisfying an overfit constraint (*i.e*, train AUC minus test AUC should not exceed a pre-determined threshold).

4 SEQUENTIAL TRAINING AND SCORING

To meet our prediction accuracy and scoring latency challenges, we extend vanilla LR to two-step LR, and introduce scoring optimizations on top of it for efficient scoring and ranking. In the remainder of this section, we first describe training and scoring for vanilla LR in our context (*i.e.*, ranking ad candidates). This is followed by sections on two-step LR (sequential training) and associated scoring optimizations (sequential scoring).

4.1 Training and scoring with vanilla LR

Vanilla LR with cross features can be described as follows:

$$s(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{(i,j)\in C} w_{ij} x_i x_j, \tag{4.1}$$

$$p(c = 1|x) = \frac{1}{1 + e^{-s(x)}}, \tag{4.2}$$

where $x = [x_1, x_2, \ldots, x_n]$ is the feature vector (one-hot encoded), $s(x)$ denotes the score of x, w_i denotes the LR weight associated with (primary) feature x_i, w_{ij} denotes the LR weight associated with cross feature $x_i * x_j$, and C denotes the set of cross features used for the LR model. The click (or conversion) probability given feature vector x, is denoted by $p(c = 1|x)$ in (4.2) (essentially the sigmoid function applied to score $s(x)$). To infer the weights w_i and w_{ij}, we minimize log loss over the training data set. Although we consider ℓ_1 and ℓ_2 regularization in our implementation, we omit regularization details here for brevity.

For online scoring of ads using the vanilla LR model described above, we first need to map raw features to one-hot encoded feature vector x (as in (4.1)). Non-demand features like user features, time features and supply features are mapped per request, while demand features (*e.g.*, app rating) are mapped per ad candidate. For example, for a single ad display opportunity, if we have to score and rank 1000 candidate ads, we generate 1000 feature vectors (with common user features but different demand features). Having generated the candidate feature vectors, we score the vectors (as shown in (4.1) and (4.2)) using the LR weights from trained CTR and CVR models to get the associated pCTR and pCVR.

4.1.1 Sum of demand features trick. In our context, since we have binary features (*i.e.*, x is one-hot encoded), we can pre-compute the sum of demand feature weights (SDFW) for every ad and simply plug in the SDFW while scoring feature vector x as in (4.1). This significantly reduces the time for looking up the weights for the demand features while scoring an ad. We will refer to this trick as SDFW optimization.

4.1.2 Problems with vanilla LR. Vanilla LR as described above suffers from the following problems in our context:

- *Oblivious to feature hierarchy.* From domain knowledge in our context, we have a sense of feature *hierarchy*, *i.e.*, a sense of features useful for refining an initial model but noisy in terms of training the initial model. When used for training the initial model, such noisy features can end up getting high weights in vanilla LR, dominate over other features and the trained model may not generalize well.
- *Oblivious to feature computation costs.* Many predictive cross features (*e.g.*, user-demand cross features) have to be computed online for every ad request, and contribute significantly to the scoring latency. Also, unlike the SDFW trick in Section 4.1.1, such cross features cannot be precomputed. Hence, in the vanilla LR model, we are constrained (latency wise) by the set of cross features we select in our model.

Both the problems mentioned above are related to the fact that vanilla LR considers all features *equally* while training and scoring. To overcome the above problems, we build on the intuition of partitioning our feature space, and refine vanilla LR training (described in Section 4.2) and scoring (described in Section 4.3) to simultaneously meet our accuracy and latency challenges.

4.2 Two-step logistic regression (sequential training)

To mitigate the problems described in Section 4.1.2, we partition the list of features (including both primary and cross features) into two classes: $\mathcal{P}_1$ and $\mathcal{P}_2$. We will first describe the two-step logistic regression framework with reference to the feature partitions, and

then revisit the criteria for partitioning the feature space into $\mathcal{P}_1$ and $\mathcal{P}_2$ in Section 4.4.

We start with training the step-1 LR model with just features from $\mathcal{P}_1$ as shown below:

$$s^{(1)}(x) = w_0^{(1)} + \sum_{i \in \mathcal{P}_1} w_i^{(1)} x_i + \sum_{(i,j) \in \mathcal{P}_1} w_{ij}^{(1)} x_i x_j, \quad (4.3)$$

$$p^{(1)}(c = 1|x) = \frac{1}{1 + e^{-s^{(1)}(x)}}, \quad (4.4)$$

where the superscript (1) is used to denote step-1. As in the vanilla LR model, the training (*i.e.*, learning weights $w_0^{(1)}$ and $\{w_i^{(1)}\}_{i \in \mathcal{P}_1}$) is done to minimize logloss for $p^{(1)}(c = 1|x)$. For the step-2 LR model, we *freeze* the weights for the features in $\mathcal{P}_1$, and train an LR model with the features in $\mathcal{P}_2$ as shown below:

$$s^{(2)}(x) = s^{(1)}(x) + w_0^{(2)} + \sum_{i \in \mathcal{P}_2} w_i^{(2)} x_i + \sum_{(i,j) \in \mathcal{P}_2} w_{ij}^{(2)} x_i x_j, \quad (4.5)$$

$$p^{(2)}(c = 1|x) = \frac{1}{1 + e^{-s^{(2)}(x)}}.$$

The step-2 model is trained to learn only the step-2 weights (*i.e.*, $w_0^{(2)}$ and $\{w_i^{(2)}\}_{i \in \mathcal{P}_2}$) to minimize the log loss for $p^{(2)}(c = 1|x)$. As shown in (4.5), for $s^{(2)}(x)$ we consider the weight of features in $\mathcal{P}_1$ (learnt in step-1) by adding $s^{(1)}(x)$, and learn only $w_0^{(2)}$ and $\{w_i^{(2)}\}_{i \in \mathcal{P}_2}$ in step-2. After training step-1 and step-2 LR models, for scoring a feature vector x, we combine the weights from the two models in the following manner:

$$s^{(1,2)}(x) = w_0^{(2)} + \sum_{i \in \mathcal{P}_2} w_i^{(2)} x_i + \sum_{(i,j) \in \mathcal{P}_2} w_{ij}^{(2)} x_i x_j + w_0^{(1)} + \sum_{i \in \mathcal{P}_1} w_i^{(1)} x_i + \sum_{(i,j) \in \mathcal{P}_1} w_{ij}^{(1)} x_i x_j, \quad (4.6)$$

$$p^{(1,2)}(c = 1|x) = \frac{1}{1 + e^{-s^{(1,2)}(x)}}$$

We will use the term two-step LR model to refer to the model described above. The two-step LR model is a natural extension of logistic regression along the lines of multistep linear regression, where the higher step features try to fit *residues* from the previous step features.

4.3 Two-step online scoring (sequential scoring)

Consider the problem of scoring and ranking N ads using the two-step LR model in (4.6). To score each candidate ad online for a display opportunity, one needs to look up weights of associated $\mathcal{P}_1$ and $\mathcal{P}_2$ features, and then compute the score. However, some feature values are costly (time intensive) to compute online and become the bottleneck for scoring latency requirements. For example, cross features involving user and demand features have to be computed online from the raw features. They cannot be pre-computed as aggregates like in the SDFW trick in Section 4.1. At the same time, such crosses can be very predictive, and dropping these costly features would lead to accuracy losses in pCTR and pCVR.

To address the challenge of having costly features in the model and meeting our latency requirements at the same time, we exploited the following observation. Typically, when we score N candidate ads, we rank them to get the top k ads. The value of k can be as low as one, if the request was made for a single display opportunity. Compared to the total number of candidate ads N, usually $k << N$. This observation led to the following idea: if we are able to select top N_1 ads (out of N) using our step-1 LR model, we we will need to score only N_1 ads in our two-step LR model. To be more precise, we compute the eCPM bids for N ads using the step-1 pCTR and pCVR models, get the top N_1 ads (by eCPM order), and then compute the eCPM bids for these N_1 ads using the two-step LR model. This can significantly reduce the overall scoring latency if N_1 is significantly lower than N. However, one needs to ensure that the N_1 ads selected for two-step LR model should include the top k ads resulting from scoring and ranking the full set of N ads with the two-step LR model. This requirement is closely linked to the accuracy of the step-1 model as well as the value of N_1. In empirical evaluations with our CTR and CVR models, we noticed that we could *safely* have N_1 as low as 20% of N without any loss in the final results for top k ads (details in Section 7). In the remainder of this paper, we will use the term *sequential scoring* for the above scoring technique. Figure 2 shows this scoring technique in the context of our ads ranker.

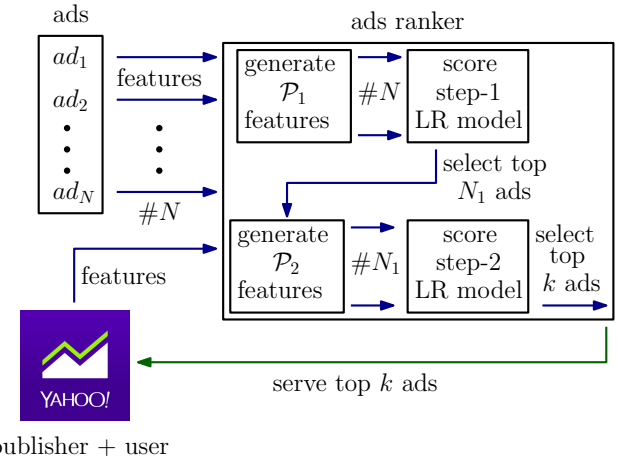

Figure 2: Ads ranker with sequential scoring with two-step LR model. The two-step model scores N_1 ads with the associated $\mathcal{P}_1$ and $\mathcal{P}_2$ features.

4.4 Feature partitioning criteria

In general, the partitioning can be done on basis of: (i) feature noise (ii) feature computation cost, *i.e.*, features which are expensive to compute online (*e.g.*, user-demand crosses) or are noisy can be included in $\mathcal{P}_2$. In our context, we considered the set of cross features and some handpicked (noisy) primary features (based on domain knowledge) $\mathcal{P}_2$, and the rest as $\mathcal{P}_1$. This was mainly motivated by scoring latency challenges as described in Section 4.3 above.

5 CROSS FEATURE ENGINEERING

We tried different approaches for incorporating interaction (cross) features into the LR model. Our feature space was of the order of 10^6, making it infeasible to consider all possible crosses (due to model complexity, scoring latency as well overfitting problems). Hence, we tried scalable approaches to come up with a reasonable list of predictive cross features which could be used within our stringent scoring latency constraints. The first approach was to use hand crafted cross features as well as cross features derived from gradient boosted decision trees (GBDT); the GBDT cross features were along the lines of work done in [20]. The second method, a novel contribution inspired from deep learning, was to reverse engineer cross features from a multilayer perceptron (MLP) trained with transfer learning.

5.1 Hand crafted and GBDT based cross features

We created a pool of hand crafted cross features that captured some interactions between: (i) user and ad features, and (ii) publisher and ad features. Inspired by the work done in [20] with GBDT, we derived additional cross features as follows. We trained vertical specific GBDT models (*e.g.*, separate models for mail and tumblr), and identified frequently occurring crosses (which we then used in our LR model). Unlike the approach in [20], we don't incur the overhead of (frequently) training and (online) scoring of GBDT models; we just update the cross feature pool periodically via offline experiments with GBDT models.

5.2 Cross features from transfer learning with MLP

In this section, we introduce a novel technique for cross feature engineering (inspired by techniques in deep learning [10, 17]). In short, we first train a deep MLP [8] on data across all verticals. With such a *global* model as the initial model, we train on vertical specific data to get a vertical specific model. After such transfer learning, we reverse-engineer predictive cross features from the MLP. This method was particularly useful for CVR models in our context since the conversion data for each vertical tends to be sparse and of low volume (compared to clicks). Similar to the work done in [17], the MLP in our case learns predictive crosses implicitly (plus some assistance from transfer learning), and then we try to infer the predictive crosses from the MLP. We describe the details of our cross feature engineering technique below (transfer learning with MLP followed by the method to reverse engineer cross features).

5.2.1 Transfer learning using MLP. MLPs are feed forward neural networks [8]. In a single hidden layer MLP, the first (hidden) layer processes the input (feature vector) as follows:

$$y_1 = \mathcal{F}(H_1 x + b_1), \tag{5.1}$$

where $x \in \mathbb{R}^n$ is the feature vector, $H_1 \in \mathbb{R}^{n \times l_1}$ is the hidden layer weights matrix, $b_1 \in \mathbb{R}^{l_1}$ is the hidden layer bias vector, $\mathcal{F}(\cdot)$ is a non-linear function (*e.g.*, softmax, ReLU) [8]. The outputs from this layer (*i.e.*, $y_1 \in l_1$) are further processed in the output layer as shown below:

$$y_{out} = H_{out} y_1 + b_{out},$$
$$[p(c = 1|x), p(c = 0|x)] = \sigma(y_{out}),$$

where $p(c = 1|x)$ denotes the conversion probability given feature vector x, and $\sigma(\cdot)$ is the softmax function [8]. Additional hidden layers can be stacked below the output layer to get deeper MLPs. Training MLPs is a non-convex optimization problem, and the choice of initial model used in training is crucial. A common technique to do transfer learning with MLPs is to first train an MLP on a related data set (whose size is usually larger than the target data set), and then use the trained model as the initial model for training another MLP on the target data set [10]. In addition, one can also treat the outputs of an intermediate layer of the MLP as additional features for the primary prediction task (*e.g.*, extra features for a logistic regression model for conversion prediction). We will use the term feature extraction layer for such an intermediate layer of the trained MLP.

In our context, we gathered conversion data across all verticals and trained MLP $\mathcal{M}_{global}$ with multiple hidden layers and ReLU as $\mathcal{F}(\cdot)$ (as shown in (5.1)). Next, we used $\mathcal{M}_{global}$ as the initial model for training MLP $\mathcal{M}_v$ for a particular vertical v (*i.e.*, using data only from vertical v). We then used the top most hidden layer of $\mathcal{M}_v$ as the feature extraction layer for the purpose of reverse engineering cross features (described below in Section 5.2.2).

5.2.2 Reverse engineering cross features from MLP. For each vertical v, let us denote the feature extraction layer of the trained MLP $\mathcal{M}_v$ (after transfer learning) as $\mathcal{L}_v^*$. We consider the outputs of $\mathcal{L}_v^*$ on the training data for vertical v, and do a sparse quadratic regression on the extracted feature values. In particular, we approximated the output of jth node of layer $\mathcal{L}^*$ in the following manner:

$$\hat{y}_j^*(x) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{i<j} a_{ij} x_i * x_j, \tag{5.2}$$

where $\hat{y}_j^*(x)$ is the approximation of $y_j^*(x)$, *i.e.*, output of jth node in $\mathcal{L}^*$ when x is fed to MLP $\mathcal{M}_v$. The task of regression is to find weights a_i and a_{ij} which minimize the mean square error between the approximation $\hat{y}_j^*$ and actual output y_j^*. We used ℓ_1 regularization to enforce sparsity in the inferred weights in (5.2). Finally, we rank the cross features according to the weights a_{ij} obtained from the sparse regression, and select the top k cross features accordingly. The process of reverse engineering cross features described above is summarized in Figure 3.

6 SYNTHESIZED FEATURES

In addition to cross features, we also synthesized new features from raw data. Listed below are some examples of new features that were synthesized and A/B tested:

(1) User interests from ad interactions: signals representing each user's interests learnt from past ad clicking, non-clicking, and conversion behavior. The signal looks at the interests represented by each ad and an aggregated interest vector for the user is generated. The interests represented by an ad are obtained by running the creative and landing page of the ad to obtain the interest vector. The interest

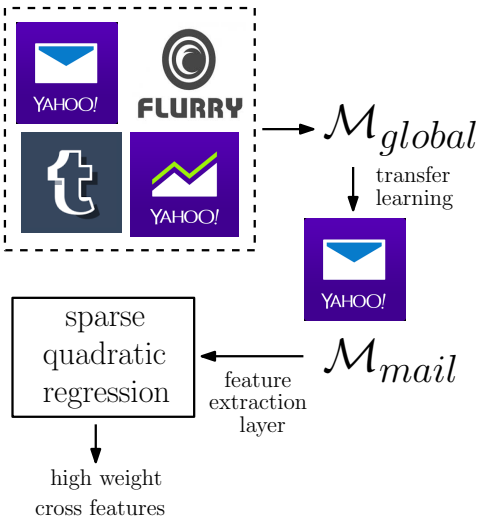

Figure 3: Reverse engineering cross features using transfer learning with MLP.

contribution from an ad-interaction decays as the time of the interaction goes further into the past.

(2) User interests from search history: the general idea is to capture the interests encapsulated in queries monetized by search advertising as well the interests expressed in the queries that were not immediately monetized.

(3) App install propensities: a score for each user is derived based on the organic install behavior. The number of new apps each user installed is retrieved by looking at the difference between the app list in the history and the current app list.

(4) Click feedback: historical aggregates of number of clicks, impression, conversion, CTRs, CVRs grouped by user, supply features, and demand features. To control the arity of this feature we hash it to a small number of buckets.

(5) User lookalike - features to identify lookalikes of good performers of mobile campaigns. These features include recency and frequency of usage in various app categories of user, segment qualifications.

(6) App activity: we count number of app activity events, categorizing users engagement metrics into various ios/andriod app categories.

7 RESULTS

In this section, we report results from our offline as well as online experiments for different verticals (split depending on the publisher, *e.g.*, Yahoo mail and tumblr are separate verticals). We first report offline results (sequential training and scoring, transfer learning and lifts from cross features) followed by online results related to key performance metrics like revenue, CTR, CVR and CPI.

Vertical	CTR AUC %	CVR AUC %
global	1.12	1.43
vertical 1	2.02	2.16
vertical 2	1.63	0.89
vertical 3	2.78	0.71
vertical 4	0.82	2.26
vertical 5	3.82	0.62

Table 2: CTR and CVR prediction AUC lifts (in %) for sequential training versus vanilla LR (with GBDT and hand crafted cross features).

7.1 Offline Evaluation

7.1.1 Sequential training and scoring. As described in Section 4, we use sequential training (2-step LR) as well as sequential scoring (*i.e.*, select N_1 out of N candidate ads for scoring with the 2-step LR model). The value of N_1 was obtained using empirical evaluations; N_1 was large enough such that all ads in the list of top k ads (after scoring N ads with 2-step LR model) were also present in the list of top N_1 ads returned after scoring with step-1 LR model. In our evaluations, we found that N_1 could be as low as 20% of N without any loss in results. This significantly reduced our scoring latency with cross features, and enhanced our scalability (in terms of number of candidate ads N that could be scored within the time constraints).

For sequential training with all cross features included in $\mathcal{P}_2$, we obtained positive offline results as shown in Table 2 for different verticals (and a global model covering all verticals). The table shows the percentage increase in area under curve (AUC) with sequential training (as described in Section 4) versus vanilla LR (with all primary features, cross features from GBDT and hand crafted cross features).

7.1.2 MLP transfer learning results. With the eventual goal of reverse engineering predictive cross features, we first trained MLPs[1] using transfer learning as described in Section 5.2. In particular, we gathered conversion data across all verticals and trained MLP $\mathcal{M}_{global}$ with multiple hidden layers and ReLU as $\mathcal{F}(\cdot)$ (as shown in (5.1)). Next, we used $\mathcal{M}_{global}$ as the initial model for training MLP $\mathcal{M}_v$ for a particular vertical $v \in \{1, 2, \ldots, 5\}$ (*i.e.*, using data only from vertical v). As shown in the second column of Table 3, we obtained significant (test) AUC lifts in multiple verticals. We went a step further, and used the outputs of the topmost hidden layer of $\mathcal{M}_v$ as additional features in a vanilla LR model for conversion prediction (*i.e.*, feature extraction). This further improved the AUC lifts in each vertical as shown in the third column of Table 3. The baseline for the results reported in Table 3 was the two-step LR model with primary features, handcrafted crosses and GBDT crosses (the MLPs used the same set of features as input). In the next section, we describe results for cross features that were reverse engineered from the trained MLPs described above.

7.1.3 Offline cross features. Using the regression technique described in Section 5.2, we reverse engineered 60 cross features from MLPs after transfer learning. After the addition of these 60 cross

[1]We used TensorFlow [1] for training MLPs.

CVR vertical	*TL* AUC %	*TL+LR* AUC %
vertical 1	0.97	1.10
vertical 2	1.23	1.31
vertical 3	0.77	0.88
vertical 4	2.45	2.53
vertical 5	0.44	0.44

Table 3: CVR prediction AUC lifts (in %) after transfer learning (TL) and feature extraction (TL + vanilla LR).

Vertical	CTR AUC %	CVR AUC %
global	0.89	1.73
vertical 1	0.52	−0.27
vertical 2	2.54	1.33
vertical 3	1.73	1.74
vertical 4	0.93	2.80
vertical 5	1.59	1.10

Table 4: CTR and CVR prediction AUC lifts after cross features (transfer learning + MLP crosses) with two-step LR model.

features, we obtained significant AUC lifts (with the two-step LR model) as shown in Table 4. The baseline for the reported results was the two-step LR model with handcrafted and GBDT cross features.

7.2 Online Experiments

After performing extensive offline evaluation as described above, we ran several controlled experiments on live traffic. Each experiment involved modeling changes, additional features, parameter tuning, etc., and was deemed successful only if it resulted in an improvement in advertiser satisfaction in addition to CPM and CVR. Our vertical specific models trained with vanilla LR using the primary mobile specific features resulted in over a 3X improvement in CPM, yield rate, and advertiser satisfaction over the existing baseline (a generic model for all Gemini native ads, not specific to app-install ads).

Subsequent online experiments with sequential training and the handcrafted and GBDT cross features nearly doubled the CVR on some verticals while increasing the overall spend, too. Further performance lifts were gained with the addition of crosses from MLPs with transfer learning, and are listed in Table 5. As shown, the spend (revenue) increased and CPI decreased significantly across all verticals. Overall, this led to a 11% increase in revenue and 7% increase in CVR with advertiser happiness close to the one without the additional cross features.

7.3 Discussion

Broadly speaking, the online results were along expected lines (see Table 5). The offline lifts in AUC correlated strongly with online business metrics:

- Advertiser's perspective: Each vertical saw a decrease in CPI (over 10% decrease on a couple of verticals). This fulfilled our primary goal of increasing advertiser happiness.
- Publisher's perspective: There was a significant increase in spend for all verticals. This fulfilled the primary goal of the advertiser (increased revenue).
- User's perspective: One indicator of user satisfaction with the ads is the yield rate. CTRs were flat overall but conversion rates were significantly higher for 3 of the 5 verticals and flat for the others. The overall yield was significantly higher (20-30% increase for 3 of the 5 verticals). There was one vertical where there was a small drop in yield which was attributable to certain traffic and marketplace quirks, the details of which would not be useful to a broad audience.

8 LEARNINGS

We had several interesting learnings and discoveries during the course of the project, which we share below so that other teams embarking on similar projects would benefit by either incorporating the learnings to either accelerate deployments or avoid the pitfalls.

- *Rapid iteration and focus on the right metrics:* Our initial focus was on CTR, CVR, CPM, and CPI, but we soon discovered that with drastically different selection, as well as, changes in the proportion of impressions and spend on oCPC vs. CPC, one could see improvements in one or more of the metrics and could still leave the advertisers unhappy. Eventually, we arrived at the conclusion that looking at the business metrics broken down by vertical, and computing the spend and campaign happiness was the key to promoting the right models. The gains reported in this paper were made via many experiments, and carefully selecting only the ones that improved advertiser satisfaction.
- *Domain knowledge matters:* One other learning, that is also part of the domain knowledge gained, is that there could be several hours between a serve and an impression attributed to it, some more hours from the impression to a click resulting from it, and several days from the click to the conversion. The typical attribution windows are 1 day for impression to click, and 7 days for click to conversion, but the conversion notifications could come many weeks after the click. This scenario arises when the user installs the app after clicking on the app, but does not open it immediately. As a consequence of this, spend from a given day will gradually accrue some more conversions on each of the following days, which can only result in a reduction in CPI, and consequently, an improvement in efficiency and happiness. The key takeaway here was to be aware that advertiser satisfaction metrics for any experiment will not be reliable the first few days after the experiment is started. Similarly, advertisers shouldn't be pausing their campaigns with high CPIs too quickly as many of their conversions could still very well be on the way.
- *Fast Exploration and tackling cold starts*: Cold start issues – wherein new ads, campaigns, or advertisers require exploration to estimate their CTRs and CVRs accurately – may be mitigated by using more metadata. In our case, we used many demand features like app category and ratings to both cut down exploration time and perform reasonably

vertical	spend %	CTR %	CVR %	yield %	CPM %	CPC %	CPI %
vertical 1	10.46	−4.67	30.02	23.63	8.69	13.63	−12.38
vertical 2	9.25	−5.64	0.02	−5.62	−7.26	−1.72	−1.73
vertical 3	8.61	2.70	26.04	28	6.28	3.84	−17.82
vertical 4	9.19	2.01	2.52	4.58	4.04	1.99	−0.51
vertical 5	20.83	1.89	21.12	23.47	9.37	11.89	−7.61

Table 5: Online performance lifts (in %) after adding cross features from MLPs with transfer learning.

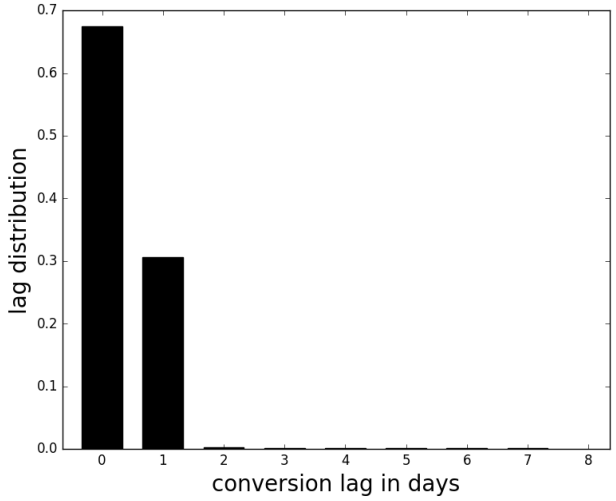

Figure 4: Conversion lag distribution for app installs (time between click and first use of the app). Majority of the conversions are received within 2 days of the click.

well during exploration. This was beneficial because advertisers look for high scale in delivery along with low CPIs, and usually expect good results very soon.

9 FUTURE WORK

We continue to experiment with more features, modeling improvements, and training. Some of the interesting next steps for us are (a) to have incremental model training with sequential training, (b) optimize training and scoring deep neural networks for a large scale setup like ours, and (c) incorporate long term value of app-installers in our modeling pipeline.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, and et al. Tensorflow: Large-scale machine learning on heterogeneous systems. *Software from tensorflow.org*, 2015.
[2] D. Agarwal, R. Agrawal, R. Khanna, and N. Kota. Estimating rates of rare events with multiple hierarchies through scalable log-linear models. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 213–222. ACM, 2010.
[3] D. Agarwal, A. Z. Broder, D. Chakrabarti, D. Diklic, V. Josifovski, and M. Sayyadian. Estimating rates of rare events at multiple resolutions. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–25. ACM, 2007.
[4] A. Z. Broder. Computational advertising. In *SODA*, volume 8, pages 992–992, 2008.
[5] B. Edelman, M. Ostrovsky, and M. Schwarz. Internet advertising and the generalized second price auction: Selling billions of dollars worth of keywords. In *American Economic Review*, 2007.
[6] eMarketer. US mobile app install ad spending, 2014-2016. *http://www.emarketer.com/Chart/US-Mobile-App-Install-Ad-Spending-2014-2016-billions-change-of-total-mobile-ad-spending/198131*.
[7] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in Microsoft's Bing search engine. ICML 2010.
[8] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.
[9] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 2006.
[10] Y. B. Jason Yosinski, Jeff Clune and H. Lipson. How transferable are features in deep neural networks? NIPS 2014.
[11] N. Kota and D. Agarwal. Temporal multi-hierarchy smoothing for estimating rates of rare events. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1361–1369. ACM, 2011.
[12] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. KDD 2013.
[13] S. Rendle. Factorization machines. In *IEEE International Conference on Data Mining*, ICDM 2010.
[14] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: Estimating the click-through rate for new ads. WWW 2007.
[15] R. Rosales, H. Cheng, and E. Manavoglu. Post-click conversion modeling and analysis for non-guaranteed delivery display advertising. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 293–302. ACM, 2012.
[16] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 2015.
[17] Y. Shan, T. R. Hoens, J. Jiao, H. Wang, D. Yu, and J. C. Mao. Deep crossing: Web-scale modeling without manually crafted combinatorial features. KDD 2016.
[18] Statista. Number of apps available in leading app stores as of March 2017. *https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/*.
[19] H. R. Varian. Position auctions. In *International Journal of Industrial Organization*, 2007.
[20] H. Xinran, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. Candela. Practical lessons from predicting clicks on ads at Facebook. *International Workshop on Data Mining for Online Advertising*, 2013.
[21] Yahoo Gemini. *https://gemini.yahoo.com/advertiser/home*.
[22] Yahoo Gemini. Drive app installs on mobile. *https://developer.yahoo.com/gemini/advertiser/guide/adcreation/drive-app-installs-mobile/*.