Ad Click Prediction: a View from the Trenches

H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young,
Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov,
Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg,
Arnar Mar Hrafnkelsson, Tom Boulos, Jeremy Kubica
Google, Inc.
mcmahan@google.com, gholt@google.com, dsculley@google.com

ABSTRACT

Predicting ad click–through rates (CTR) is a massive-scale learning problem that is central to the multi-billion dollar online advertising industry. We present a selection of case studies and topics drawn from recent experiments in the setting of a deployed CTR prediction system. These include improvements in the context of traditional supervised learning based on an FTRL-Proximal online learning algorithm (which has excellent sparsity and convergence properties) and the use of per-coordinate learning rates.

We also explore some of the challenges that arise in a real-world system that may appear at first to be outside the domain of traditional machine learning research. These include useful tricks for memory savings, methods for assessing and visualizing performance, practical methods for providing confidence estimates for predicted probabilities, calibration methods, and methods for automated management of features. Finally, we also detail several directions that did not turn out to be beneficial for us, despite promising results elsewhere in the literature. The goal of this paper is to highlight the close relationship between theoretical advances and practical engineering in this industrial setting, and to show the depth of challenges that appear when applying traditional machine learning methods in a complex dynamic system.

Categories and Subject Descriptors

I.5.4 [**Computing Methodologies**]: Pattern Recognition—*Applications*

Keywords

online advertising, data mining, large-scale learning

1. INTRODUCTION

Online advertising is a multi-billion dollar industry that has served as one of the great success stories for machine learning. Sponsored search advertising, contextual advertising, display advertising, and real-time bidding auctions have all relied heavily on the ability of learned models to predict ad click–through rates accurately, quickly, and reliably [28, 15, 33, 1, 16]. This problem setting has also pushed the field to address issues of scale that even a decade ago would have been almost inconceivable. A typical industrial model may provide predictions on billions of events per day, using a correspondingly large feature space, and then learn from the resulting mass of data.

In this paper, we present a series of case studies drawn from recent experiments in the setting of the deployed system used at Google to predict ad click–through rates for sponsored search advertising. Because this problem setting is now well studied, we choose to focus on a series of topics that have received less attention but are equally important in a working system. Thus, we explore issues of memory savings, performance analysis, confidence in predictions, calibration, and feature management with the same rigor that is traditionally given to the problem of designing an effective learning algorithm. The goal of this paper is to give the reader a sense of the depth of challenges that arise in real industrial settings, as well as to share tricks and insights that may be applied to other large-scale problem areas.

2. BRIEF SYSTEM OVERVIEW

When a user does a search $\mathbf{q}$, an initial set of candidate ads is matched to the query $\mathbf{q}$ based on advertiser-chosen keywords. An auction mechanism then determines whether these ads are shown to the user, what order they are shown in, and what prices the advertisers pay if their ad is clicked. In addition to the advertiser bids, an important input to the auction is, for each ad $\mathbf{a}$, an estimate of $P(\text{click} \mid \mathbf{q}, \mathbf{a})$, the probability that the ad will be clicked if it is shown.

The features used in our system are drawn from a variety of sources, including the query, the text of the ad creative, and various ad-related metadata. Data tends to be extremely sparse, with typically only a tiny fraction of non-zero feature values per example.

Methods such as regularized logistic regression are a natural fit for this problem setting. It is necessary to make predictions many billions of times per day and to quickly update the model as new clicks and non-clicks are observed. Of course, this data rate means that training data sets are enormous. Data is provided by a streaming service based on the Photon system – see [2] for a full discussion.

Because large-scale learning has been so well studied in recent years (see [3], for example) we do not devote signif-

KDD'13, August 11–14, 2013, Chicago, Illinois, USA.

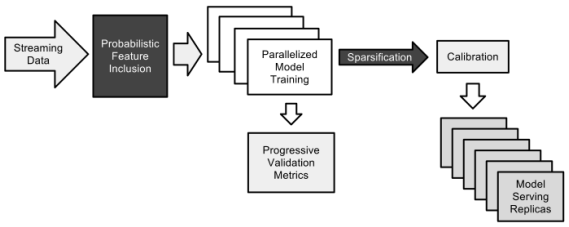

Figure 1: High-level system overview. Sparsification is covered in Section 3, probabilistic feature inclusion in Section 4, progressive validation in Section 5, and calibration methods in Section 7.

icant space in this paper to describing our system architecture in detail. We will note, however, that the training methods bear resemblance to the Downpour SGD method described by the Google Brain team [8], with the difference that we train a single-layer model rather than a deep network of many layers. This allows us to handle significantly larger data sets and larger models than have been reported elsewhere to our knowledge, with billions of coefficients. Because trained models are replicated to many data centers for serving (see Figure 1), we are much more concerned with sparsification at serving time rather than during training.

3. ONLINE LEARNING AND SPARSITY

For learning at massive scale, online algorithms for generalized linear models (e.g., logistic regression) have many advantages. Although the feature vector $\mathbf{x}$ might have billions of dimensions, typically each instance will have only hundreds of nonzero values. This enables efficient training on large data sets by streaming examples from disk or over the network [3], since each training example only needs to be considered once.

To present the algorithm precisely, we need to establish some notation. We denote vectors like $\mathbf{g}_t \in \mathbb{R}^d$ with boldface type, where t indexes the current training instance; the i^{th} entry in a vector $\mathbf{g}_t$ is denoted $g_{t,i}$. We also use the compressed summation notation $\mathbf{g}_{1:t} = \sum_{s=1}^{t} \mathbf{g}_s$.

If we wish to model our problem using logistic regression, we can use the following online framework. On round t, we are asked to predict on an instance described by feature vector $\mathbf{x}_t \in \mathbb{R}^d$; given model parameters $\mathbf{w}_t$, we predict $p_t = \sigma(\mathbf{w}_t \cdot \mathbf{x}_t)$, where $\sigma(a) = 1/(1+\exp(-a))$ is the sigmoid function. Then, we observe the label $y_t \in \{0, 1\}$, and suffer the resulting LogLoss (logistic loss), given as

$$\ell_t(\mathbf{w}_t) = -y_t \log p_t - (1 - y_t) \log(1 - p_t), \qquad (1)$$

the negative log-likelihood of y_t given p. It is straightforward to show $\nabla \ell_t(\mathbf{w}) = (\sigma(\mathbf{w} \cdot \mathbf{x}_t) - y_t)\mathbf{x}_t = (p_t - y_t)\mathbf{x}_t$, and this gradient is all we will need for optimization purposes.

Online gradient descent[1] (OGD) has proved very effective for these kinds of problems, producing excellent prediction accuracy with a minimum of computing resources. However, in practice another key consideration is the size of the final model; since models can be stored sparsely, the number of non-zero coefficients in $\mathbf{w}$ is the determining factor of memory usage.

[1]OGD is essentially the same as stochastic gradient descent; the name online emphasizes we are not solving a batch problem, but rather predicting on a sequence of examples that need not be IID.

Algorithm 1 Per-Coordinate FTRL-Proximal with L_1 and L_2 Regularization for Logistic Regression

#With per-coordinate learning rates of Eq. (2).
Input: parameters α, β, λ_1, λ_2
$(\forall i \in \{1, \ldots, d\})$, initialize $z_i = 0$ and $n_i = 0$
for $t = 1$ **to** T **do**
  Receive feature vector $\mathbf{x}_t$ and let $I = \{i \mid x_i \neq 0\}$
  For $i \in I$ compute

$$w_{t,i} = \begin{cases} 0 & \text{if } |z_i| \leq \lambda_1 \\ -\left(\frac{\beta + \sqrt{n_i}}{\alpha} + \lambda_2\right)^{-1} (z_i - \text{sgn}(z_i)\lambda_1) & \text{otherwise.} \end{cases}$$

  Predict $p_t = \sigma(\mathbf{x}_t \cdot \mathbf{w})$ using the $w_{t,i}$ computed above
  Observe label $y_t \in \{0, 1\}$
  for all $i \in I$ **do**
    $g_i = (p_t - y_t)x_i$ *#gradient of loss w.r.t.* w_i
    $\sigma_i = \frac{1}{\alpha}\left(\sqrt{n_i + g_i^2} - \sqrt{n_i}\right)$ *#equals* $\frac{1}{\eta_{t,i}} - \frac{1}{\eta_{t-1,i}}$
    $z_i \leftarrow z_i + g_i - \sigma_i w_{t,i}$
    $n_i \leftarrow n_i + g_i^2$
  end for
end for

Unfortunately, OGD is not particularly effective at producing sparse models. In fact, simply adding a subgradient of the L_1 penalty to the gradient of the loss ($\nabla_w \ell_t(\mathbf{w})$) will essentially never produce coefficients that are exactly zero. More sophisticated approaches such as FOBOS and truncated gradient do succeed in introducing sparsity [11, 20]. The Regularized Dual Averaging (RDA) algorithm produces even better accuracy vs sparsity tradeoffs than FOBOS [32]. However, we have observed the gradient-descent style methods can produce better accuracy than RDA on our datasets [24]. The question, then, is can we get both the sparsity provided by RDA and the improved accuracy of OGD? The answer is yes, using the "Follow The (Proximally) Regularized Leader" algorithm, or FTRL-Proximal. Without regularization, this algorithm is identical to standard online gradient descent, but because it uses an alternative lazy representation of the model coefficients w, L_1 regularization can be implemented much more effectively.

The FTRL-Proximal algorithm has previously been framed in a way that makes theoretical analysis convenient [24]. Here, we focus on describing a practical implementation. Given a sequence of gradients $\mathbf{g}_t \in \mathbb{R}^d$, OGD performs the update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}_t,$$

where η_t is a non-increasing learning-rate schedule, e.g., $\eta_t = \frac{1}{\sqrt{t}}$. The FTRL-Proximal algorithm instead uses the update

$$\mathbf{w}_{t+1} = \arg\min_{\mathbf{w}} \left(\mathbf{g}_{1:t} \cdot \mathbf{w} + \frac{1}{2}\sum_{s=1}^{t} \sigma_s \|\mathbf{w} - \mathbf{w}_s\|_2^2 + \lambda_1 \|\mathbf{w}\|_1\right),$$

where we define σ_s in terms of the learning-rate schedule such that $\sigma_{1:t} = \frac{1}{\eta_t}$. On the surface, these updates look very different, but in fact when we take $\lambda_1 = 0$, they produce an *identical* sequence of coefficient vectors. However, the FTRL-Proximal update with $\lambda_1 > 0$ does an excellent job of inducing sparsity (see experimental results below).

On quick inspection, one might think the FTRL-Proximal update is harder to implement than gradient descent, or requires storing all the past coefficients. In fact, however, only one number per coefficient needs to be stored, since we

	Num. Non-Zero's	AucLoss Detriment
FTRL-Proximal	baseline	baseline
RDA	+3%	0.6%
FOBOS	+38%	0.0%
OGD-Count	+216%	0.0%

Table 1: FTRL results, showing the relative number of non-zero coefficient values and AucLoss (1−AUC)) for competing approaches (smaller numbers are better for both). Overall, FTRL gives better sparsity for the same or better accuracy (a detriment of 0.6% is significant for our application). RDA and FOBOS were compared to FTRL on a smaller prototyping dataset with millions of examples, while OGD-Count was compared to FTRL on a full-scale data set.

can re-write the update as the argmin over $\mathbf{w} \in \mathbb{R}^d$ of

$$\left(\mathbf{g}_{1:t} - \sum_{s=1}^{t} \sigma_s \mathbf{w}_s\right) \cdot \mathbf{w} + \frac{1}{\eta_t}\|\mathbf{w}\|_2^2 + \lambda_1 \|\mathbf{w}\|_1 + (\text{const}).$$

Thus, if we have stored $\mathbf{z}_{t-1} = \mathbf{g}_{1:t-1} - \sum_{s=1}^{t-1} \sigma_s \mathbf{w}_s$, at the beginning of round t we update by letting $\mathbf{z}_t = \mathbf{z}_{t-1} + \mathbf{g}_t + (\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}})\mathbf{w}_t$, and solve for $\mathbf{w}_{t+1}$ in closed form on a per-coordinate bases by

$$w_{t+1,i} = \begin{cases} 0 & \text{if } |z_{t,i}| \leq \lambda_1 \\ -\eta_t(z_{t,i} - \text{sgn}(z_{t,i})\lambda_1) & \text{otherwise.} \end{cases}$$

Thus, FTRL-Proximal stores $\mathbf{z} \in \mathbb{R}^d$ in memory, whereas OGD stores $\mathbf{w} \in \mathbb{R}^d$. Algorithm 1 takes this approach, but also adds a per-coordinate learning rate schedule (discussed next), and supports L_2 regularization of strength λ_2. Alternatively, we could store $-\eta_t \mathbf{z}_t$ instead of storing $\mathbf{z}_t$ directly; then, when $\lambda_1 = 0$, we are storing exactly the normal gradient descent coefficient. Note that when η_t is a constant value η and $\lambda_1 = 0$, it is easy to see the equivalence to online gradient descent, since we have $\mathbf{w}_{t+1} = -\eta \mathbf{z}_t = -\eta \sum_{s=1}^{t} \mathbf{g}_s$, exactly the point played by gradient descent.

Experimental Results. In earlier experiments on smaller prototyping versions of our data, McMahan [24] showed that FTRL-Proximal with L_1 regularization significantly outperformed both RDA and FOBOS in terms of the size-versus-accuracy tradeoffs produced; these previous results are summarized in Table 1, rows 2 and 3.

In many instances, a simple heuristic works almost as well as the more principled approach, but this is not one of those cases. Our straw-man algorithm, OGD-Count, simply maintains a count of the number of times it has seen a feature; until that count passes a threshold k, the coefficient is fixed at zero, but after the count passes k, online gradient descent (without any L_1 regularization) proceeds as usual. To test FTRL-Proximal against this simpler heuristic we ran on a very large data set. We tuned k to produce equal accuracy to FTRL-Proximal; using a larger k leads to worse AucLoss. Results are given in Table 1, row 4.

Overall, these results show that FTRL-Proximal gives significantly improved sparsity with the same or better prediction accuracy.

3.1 Per-Coordinate Learning Rates

The standard theory for online gradient descent suggests using a global learning rate schedule $\eta_t = \frac{1}{\sqrt{t}}$ that is common for all coordinates [34]. A simple thought experiment shows that this may not be ideal: suppose we are estimating $\Pr(\text{heads} \mid \text{coin}_i)$ for 10 coins using logistic regression. Each round t, a single coin i is flipped, and we see a feature vector $\mathbf{x} \in \mathbb{R}^{10}$ with $x_i = 1$ and $x_j = 0$ for $j \neq i$. Thus, we are essentially solving 10 independent logistic regression problems, packaged up into a single problem.

We could run 10 independent copies of online gradient descent, where the algorithm instance for problem i would use a learning rate like $\eta_{t,i} = \frac{1}{\sqrt{n_{t,i}}}$ where $n_{t,i}$ is the number of times coin i has been flipped so far. If coin i is flipped much more often than coin j, then the learning rate for coin i will decrease more quickly, reflecting the fact we have more data; the learning rate will stay high for coin j, since we have less confidence in our current estimate, and so need to react more quickly to new data.

On the other hand, if we look at this as a single learning problem, the standard learning rate schedule $\eta_t = \frac{1}{\sqrt{t}}$ is applied to all coordinates: that is, we decrease the learning rate for coin i even when it is not being flipped. This is clearly not the optimal behavior. In fact, Streeter and McMahan [29] have shown a family of problems where the performance for the standard algorithm is asymptotically much worse than running independent copies.[2] Thus, at least for some problems, per-coordinate learning rates can offer a substantial advantage.

Recall that $g_{s,i}$ is the i^{th} coordinate of the gradient $\mathbf{g}_s = \nabla \ell_s(\mathbf{w}_s)$. Then, a careful analysis shows the per-coordinate rate

$$\eta_{t,i} = \frac{\alpha}{\beta + \sqrt{\sum_{s=1}^{t} g_{s,i}^2}}, \tag{2}$$

is near-optimal in a certain sense.[3] In practice, we use a learning rate where α and β are chosen to yield good performance under progressive validation (see Section 5.1). We have also experimented with using a power on the counter $n_{t,i}$ other than 0.5. The optimal value of α can vary a fair bit depending on the features and dataset, and $\beta = 1$ is usually good enough; this simply ensures that early learning rates are not too high.

As stated, this algorithm requires us to keep track of both the sum of the gradients and the sum of the squares of the gradients for each feature. Section 4.5 presents an alternative memory-saving formulation where the sum of the squares of the gradients is amortized over many models.

A relatively simple analysis of per-coordinate learning rates appears in [29], as well as experimental results on small Google datasets; this work builds directly on the approach of Zinkevich [34]. A more theoretical treatment for FTRL-Proximal appears in [26]. Duchi *et al.* [10] analyze RDA and mirror-descent versions, and also give numerous experimental results.

Experimental Results. We assessed the impact of per-coordinate learning rates by testing two identical models,

[2]Formally, regret (see, e.g., [34]) is $\Omega(T^{\frac{2}{3}})$ for standard gradient descent, while independent copies yields regret $\mathcal{O}(T^{\frac{1}{2}})$.

[3]For a fixed sequence of gradients, if we take α to be twice the maximum allowed magnitude for $\mathbf{w}_i$, and $\beta = 0$, we bound our regret within a factor of $\sqrt{2}$ of the best possible regret bound (not regret) for any non-increasing per-coordinate learning rate schedule [29].

Method	RAM Saved	AucLoss Detriment
Bloom ($n = 2$)	66%	0.008%
Bloom ($n = 1$)	55%	0.003%
Poisson ($p = 0.03$)	60%	0.020%
Poisson ($p = 0.1$)	40%	0.006%

Table 2: Effect Probabilistic Feature Inclusion. Both methods are effective, but the bloom filtering approach gives better tradeoffs between RAM savings and prediction accuracy.

one using a single global learning rate and one using per-coordinate learning rates. The base parameter α was tuned separately for each model. We ran on a representative data set, and used AucLoss as our evaluation metric (see Section 5). The results showed that using a per-coordinate learning rates reduced AucLoss by 11.2% compared to the global-learning-rate baseline. To put this result in context, in our setting AucLoss reductions of 1% are considered large.

4. SAVING MEMORY AT MASSIVE SCALE

As described above, we use L_1 regularization to save memory at prediction time. In this section, we describe additional tricks for saving memory during training.

4.1 Probabilistic Feature Inclusion

In many domains with high dimensional data, the vast majority of features are extremely rare. In fact, in some of our models, half the unique features occur only once in the entire training set of billions of examples.[4]

It is expensive to track statistics for such rare features which can never be of any real use. Unfortunately, we do not know in advance which features will be rare. Pre-processing the data to remove rare features is problematic in an on-line setting: an extra read and then write of the data is very expensive, and if some features are dropped (say, because they occur fewer than k times), it is no longer possible to try models that use those feature to estimate the cost of the pre-processing in terms of accuracy.

One family of methods achieves sparsity in training via an implementation of L_1 regularization that doesn't need to track any statistics for features with a coefficient of zero (e.g.,[20]). This allows less informative features to be removed as training progresses. However, we found that this style of sparsification leads to an unacceptable loss in accuracy compared to methods (like FTRL-Proximal) that track more features in training and sparsify only for serving. Another common solution to this problem, hashing with collisions, also did not give useful benefit (see Section 9.1).

Another family of methods we explored is probabilistic feature inclusion, in which new features are included in the model probabilistically as they first occur. This achieves the effect of pre-processing the data, but can be executed in an online setting.

We tested two methods for this approach.

- **Poisson Inclusion.** When we encounter a feature that is not already in our model, we only add it to the model with probability p. Once a feature has been added, in subsequent observations we update its coefficient values and related statistics used by OGD as per usual. The number of times a feature needs to be seen before it is added to the model follows a geometric distribution with expected value $\frac{1}{p}$.
- **Bloom Filter Inclusion.** We use a rolling set of counting Bloom filters [4, 12] to detect the first n times a feature is encountered in training. Once a feature has occurred more than n times (according to the filter), we add it to the model and use it for training in subsequent observations as above. Note that this method is also probabilistic, because a counting bloom filter is capable of false positives (but not false negatives). That is, we will sometimes include a feature that has actually occurred less than n times.

Experimental Results. The effect of these methods is seen in Table 2, and shows that both methods work well, but the Bloom filter approach gives a better set of tradeoffs for RAM savings against loss in predictive quality.

4.2 Encoding Values with Fewer Bits

Naive implementations of OGD use 32 or 64 bit floating point encodings to store coefficient values. Floating point encodings are often attractive because of their large dynamic range and fine-grained precision; however, for the coefficients of our regularized logistic regression models this turns out to be overkill. Nearly all of the coefficient values lie within the range $(-2, +2)$. Analysis shows that the fine-grained precision is also not needed [14], motivating us to explore the use of a fixed-point `q2.13` encoding rather than floating point.

In `q2.13` encoding, we reserve two bits to the left of the binary decimal point, thirteen bits to the right of the binary decimal point, and a bit for the sign, for a total of 16 bits used per value.

This reduced precision could create a problem with accumulated roundoff error in an OGD setting, which requires the accumulation of a large number of tiny steps. (In fact, we have even seen serious roundoff problems using 32 bit floats rather than 64.) However, a simple randomized rounding strategy corrects for this at the cost of a small added regret term [14]. The key is that by explicitly rounding, we can ensure the discretization error has zero mean.

In particular, if we are storing the coefficient w, we set

$$w_{i,\text{rounded}} = 2^{-13} \left\lfloor 2^{13} w_i + R \right\rfloor \quad (3)$$

where R is a random deviate uniformly distributed between 0 and 1. $g_{i,\text{rounded}}$ is then stored in the `q2.13` fixed point format; values outside the range $[-4, 4)$ are clipped. For FTRL-Proximal, we can store $\eta_t z_t$ in this manner, which has similar magnitude to w_t.

Experimental Results. In practice, we observe no measurable loss comparing results from a model using `q2.13` encoding instead of 64-bit floating point values, and we save 75% of the RAM for coefficient storage.

4.3 Training Many Similar Models

When testing changes to hyper-parameter settings or features, it is often useful to evaluate many slight variants of one form or another. This common use-case allows for efficient training strategies. One interesting piece of work in

[4]Because we deal exclusively with extremely sparse data in this paper, we say a feature "occurs" when it appears with a non-zero value in an example.

this line is [19], which used a fixed model as a prior and allowed variations to be evaluated against residual error. This approach is very cheap, but does not easily allow the evaluation of feature removals or alternate learning settings.

Our main approach relies on the observation that each coordinate relies on some data that can be efficiently shared across model variants, while other data (such as the coefficient value itself) is specific to each model variant and cannot be shared. If we store model coefficients in a hash table, we can use a single table for all of the variants, amortizing the cost of storing the key (either a string or a many-byte hash). In the next section (4.5), we show how the per-model learning-rate counters n_i can be replaced by statistics shared by all the variants, which also decreases storage.

Any variants which do not have a particular feature will store the coefficient for that feature as 0, wasting a small amount of space. (We enforce that by setting the learning rate for those features to 0.) Since we train together only highly similar models, the memory savings from not representing the key and the counts per model is much larger than the loss from features not in common.

When several models are trained together, the amortized cost is driven down for all per-coordinate metadata such as the counts needed for per-coordinate learning rates, and the incremental cost of additional models depends only on the additional coefficient values that need to be stored. This saves not only memory, but also network bandwidth (the values are communicated across the network in the same way, and we read the training data only once), CPU (only one hash table lookup rather than many, and features are generated from the training data only once rather than once per model), and disk space. This bundled architecture increases our training capacity significantly.

4.4 A Single Value Structure

Sometimes we wish to evaluate very large sets of model variants together that differ only by the addition or removal of small groups of features. Here, we can employ an even more compressed data structure that is both lossy and *ad hoc* but in practice gives remarkably useful results. This Single Value Structure stores just one coefficient value for each coordinate which is shared by all model variants that include that feature, rather than storing separate coefficient values for each model variant. A bit-field is used to track which model variants include the given coordinate. Note that this is similar in spirit to the method of [19], but also allows the evaluation of feature removals as well as additions. The RAM cost grows much more slowly with additional model variants than the method of Section 4.3.

Learning proceeds as follows. For a given update in OGD, each model variant computes its prediction and loss using the subset of coordinates that it includes, drawing on the stored single value for each coefficient. For each feature i, each model that uses i computes a new desired value for the given coefficient. The resulting values are averaged and stored as a single value that will then be shared by all variants on the next step.

We evaluated this heuristic by comparing large groups of model variants trained with the single value structure against the same variants trained exactly with the set up from Section 4.3. The results showed nearly identical relative performance across variants, but the single value structure saved an order of magnitude in RAM.

4.5 Computing Learning Rates with Counts

As presented in Section 3.1, we need to store for each feature both the sum of the gradients and the sum of the squares of the gradients. It is important that the gradient calculation be correct, but gross approximations may be made for the learning rate calculation.

Suppose that all events containing a given feature have the same probability. (In general, this is a terrible approximation, but it works for this purpose.) Suppose further that the model has accurately learned the probability. If there are N negative events, and P positive events, then the probability is $p = P/(N+P)$. If we use logistic regression, the gradient for positive events is $p-1$ and the gradient for negative events is p, the sum of the gradients needed for the learning rate Eq. (2) is

$$\begin{aligned}\sum g_{t,i}^2 &= \sum_{\text{positive events}} (1-p_t)^2 + \sum_{\text{negative events}} p_t^2 \\ &\approx P\left(1 - \frac{P}{N+P}\right)^2 + N\left(\frac{P}{N+P}\right)^2 \\ &= \frac{PN}{N+P}.\end{aligned}$$

This ruthless approximation allows us to keep track of only the counts N and P, and dispense with storing $\sum g_{t,i}^2$. Empirically, learning rates calculated with this approximation work just as well for us as learning rates calculated with the full sum. Using the framework of Section 4.3, total storage costs are lower since all variant models have the same counts, so the storage cost for N and P is amortized. The counts can be stored with variable length bit encodings, and the vast majority of the features do not require many bits.

4.6 Subsampling Training Data

Typical CTRs are much lower than 50%, which means that positive examples (clicks) are relatively rare. Thus, simple statistical calculations indicate that clicks are relatively more valuable in learning CTR estimates. We can take advantage of this to significantly reduce the training data size with minimal impact on accuracy. We create subsampled training data by including in our sample:

- Any query for which at least one of the ads was clicked.
- A fraction $r \in (0, 1]$ of the queries where none of the ads were clicked.

Sampling at the query level is desirable, since computing many features requires common processing on the query phrase. Of course, naively training on this subsampled data would lead to significantly biased predictions. This problem is easily addressed by assigning an importance weight ω_t to each example, where

$$\omega_t = \begin{cases} 1 & \text{event } t \text{ is in a clicked query} \\ \frac{1}{r} & \text{event } t \text{ is in a query with no clicks.} \end{cases}$$

Since we control the sampling distribution, we do not have to estimate the weights ω as in general sample selection [7]. The importance weight simply scales up the loss on each event, Eq. (1), and hence also scales the gradients. To see that this has the intended effect, consider the expected contribution of a randomly chosen event t in the unsampled data to the sub-sampled objective function. Let s_t be the

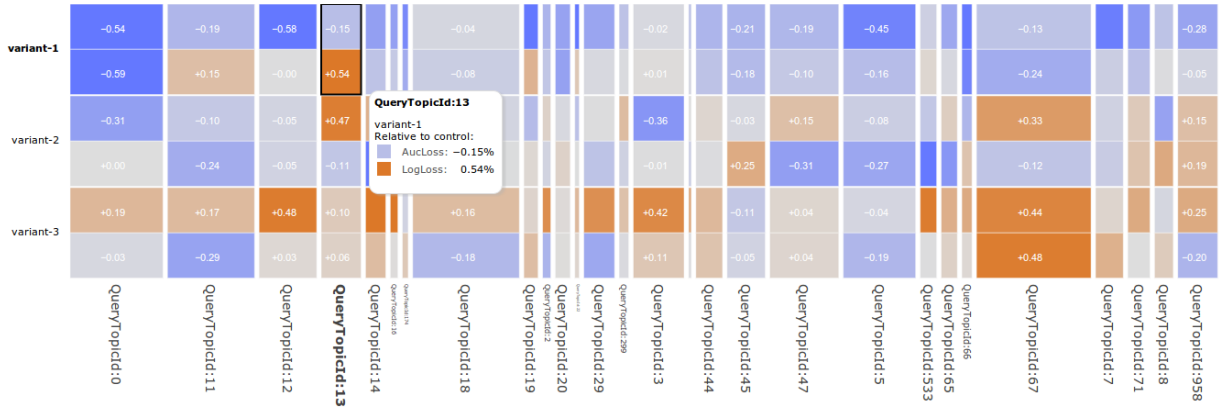

Figure 2: Screen shot of the high-dimensional analysis visualization. Here, three variants are compared with a control model, with results for AucLoss and LogLoss computed across a range of query topics. Column width reflects impression count. Detailed information pops up for a specific breakdown on mouse-over. The user interface allows for selection of multiple metrics and several possible breakdowns, including breakdowns by topic, country, match type, and page layouts. This allows fast scanning for anomalies and deep understanding of model performance. Best viewed in color.

probability with which event t is sampled (either 1 or r), and so by definition $s_t = \frac{1}{\omega_t}$. Thus, we have

$$\mathbb{E}[\ell_t(\mathbf{w}_t)] = s_t \omega_t \ell_t(\mathbf{w}_t) + (1 - s_t)0 = s_t \frac{1}{s_t} \ell_t(\mathbf{w}_t) = \ell_t(\mathbf{w}_t).$$

Linearity of expectation then implies the expected weighted objective on the subsampled training data equals the objective function on the original data set. Experiments have verified that even fairly aggressive sub-sampling of unclicked queries has a very mild impact on accuracy, and that predictive performance is not especially impacted by the specific value of r.

5. EVALUATING MODEL PERFORMANCE

Evaluating the quality of our models is done most cheaply through the use of logged historical data. (Evaluating models on portions of live traffic is an important, but more expensive, piece of evaluation; see, for example, [30].)

Because the different metrics respond in different ways to model changes, we find that it is generally useful to evaluate model changes across a plurality of possible performance metrics. We compute metrics such as AucLoss (that is, 1 − AUC, where AUC is the standard area under the ROC curve metric [13]), LogLoss (see Eq. (1)), and SquaredError. For consistency, we also design our metrics so that smaller values are always better.

5.1 Progressive Validation

We generally use progressive validation (sometimes called online loss) [5] rather than cross-validation or evaluation on a held out dataset. Because computing a gradient for learning requires computing a prediction anyway, we can cheaply stream those predictions out for subsequent analysis, aggregated hourly. We also compute these metrics on a variety of sub-slices of the data, such as breakdowns by country, query topic, and layout.

The online loss is a good proxy for our accuracy in serving queries, because it measures the performance only on the most recent data before we train on it—exactly analogous to what happens when the model serves queries. The online loss also has considerably better statistics than a held-out validation set, because we can use 100% of our data for both training and testing. This is important because small improvements can have meaningful impact at scale and need large amounts of data to be observed with high confidence.

Absolute metric values are often misleading. Even if predictions are perfect, the LogLoss and other metrics vary depending on the difficulty of the problem (that is, the Bayes risk). If the click rate is closer to 50%, the best achievable LogLoss is much higher than if the click rate is closer to 2%. This is important because click rates vary from country to country and from query to query, and therefore the averages change over the course of a single day.

We therefore always look at *relative* changes, usually expressed as a percent change in the metric relative to a baseline model. In our experience, relative changes are much more stable over time. We also take care only to compare metrics computed from exactly the same data; for example, loss metrics computed on a model over one time range are not comparable to the same loss metrics computed on another model over a different time range.

5.2 Deep Understanding through Visualization

One potential pitfall in massive scale learning is that aggregate performance metrics may hide effects that are specific to certain sub-populations of the data. For example, a small aggregate accuracy win on one metric may in fact be caused by a mix of positive and negative changes in distinct countries, or for particular query topics. This makes it critical to provide performance metrics not only on the aggregate data, but also on various slicings of the data, such as a per-country basis or a per-topic basis.

Because there are hundreds of ways to slice the data meaningfully, it is essential that we be able to examine a visual summary of the data effectively. To this end, we have developed a high-dimensional interactive visualization called GridViz to allow comprehensive understanding of model performance.

A screen-shot of one view from GridViz is shown in Fig-

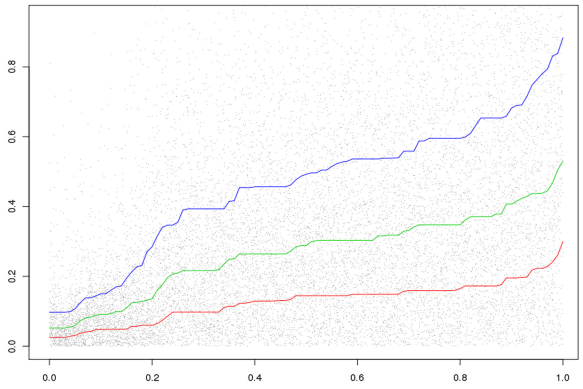

Figure 3: Visualizing Uncertainty Scores. Log-odds errors $|\sigma^{-1}(p_t) - \sigma^{-1}(p_t^*)|$ plotted versus the uncertainty score, a measure of confidence. The x-axis is normalized so the density of the individual estimates (gray points) is uniform across the domain. Lines give the estimated 25%, 50%, 75% error percentiles. High uncertainties are well correlated with larger prediction errors.

ure 2, showing a set of slicings by query topic for two models in comparison to a control model. Metric values are represented by colored cells, with rows corresponding to the model name and the columns corresponding to each unique slicing of the data. The column width connotes the importance of the slicing, and may be set to reflect quantities such as number of impressions or number of clicks. The color of the cell reflects the value of the metric compared to a chosen baseline, which enables fast scanning for outliers and areas of interest, as well as visual understanding of the overall performance. When the columns are wide enough the numeric value of the selected metrics are shown. Multiple metrics may be selected; these are shown together in each row. A detailed report for a given cell pops up when the user mouse overs over the cell.

Because there are hundreds of possible slicings, we have designed an interactive interface that allows the user to select different slicing groups via a dropdown menu, or via a regular expression on the slicing name. Columns may be sorted and the dynamic range of the color scale modified to suite the data at hand. Overall, this tool has enabled us to dramatically increase the depth of our understanding for model performance on a wide variety of subsets of the data, and to identify high impact areas for improvement.

6. CONFIDENCE ESTIMATES

For many applications, it is important to not only estimate the CTR of the ad, but also to quantify the expected accuracy of the prediction. In particular, such estimates can be used to measure and control explore/exploit tradeoffs: in order to make accurate predictions, the system must sometimes show ads for which it has little data, but this should be balanced against the benefit of showing ads which are known to be good [21, 22].

Confidence intervals capture the notion of uncertainty, but for both practical and statistical reasons, they are inappropriate for our application. Standard methods would assess the confidence of predictions of a fully-converged batch model without regularization; our models are online, do not assume IID data (so convergence is not even well defined), and heavily regularized. Standard statistical methods (e.g., [18], Sec. 2.5) also require inverting a $n \times n$ matrix; when n is in the billions, this is a non-starter.

Further, it is essential that any confidence estimate can be computed extremely cheaply at prediction time — say in about as much time as making the prediction itself.

We propose a heuristic we call the *uncertainty score*, which is computationally tractable and empirically does a good job of quantifying prediction accuracy. The essential observation is that the learning algorithm itself maintains a notion of uncertainty in the per-feature counters $n_{t,i}$ used for learning rate control. Features for which n_i is large get a smaller learning rate, precisely because we believe the current coefficient values are more likely to be accurate. The gradient of logistic loss with respect to the log-odds score is $(p_t - y_t)$ and hence has absolute value bounded by 1. Thus, if we assume feature vectors are normalized so $|x_{t,i}| \leq 1$, we can bound the change in the log-odds prediction due to observing a single training example $(\mathbf{x}, y)$. For simplicity, consider $\lambda_1 = \lambda_2 = 0$, so FTRL-Proximal is equivalent to online gradient descent. Letting $n_{t,i} = \beta + \sum_{s=1}^{t} g_{s,i}^2$ and following Eq. (2), we have

$$|\mathbf{x} \cdot \mathbf{w}_t - \mathbf{x} \cdot \mathbf{w}_{t+1}| = \sum_{i:|x_i|>0} \eta_{t,i} |g_{t,i}|$$
$$\leq \alpha \sum_{i:|x_i|>0} \frac{x_{t,i}}{\sqrt{n_{t,i}}} = \alpha \boldsymbol{\eta} \cdot \mathbf{x} \equiv u(\mathbf{x})$$

where $\boldsymbol{\eta}$ to is the vector of learning rates. We define the uncertainty score to be the upper bound $u(\mathbf{x}) \equiv \alpha \boldsymbol{\eta} \cdot \mathbf{x}$; it can be computed with a single sparse dot product, just like the prediction $p = \sigma(\mathbf{w} \cdot \mathbf{x})$.

Experimental Results. We validated this methodology as follows. First, we trained a "ground truth" model on real data, but using slightly different features than usual. Then, we discarded the real click labels, and sampled new labels taking the predictions of the ground-truth model as the true CTRs. This is necessary, as assessing the validity of a confidence procedure requires knowing the true labels. We then ran FTRL-Proximal on the re-labeled data, recording predictions p_t, which allows us to compare the accuracy of the predictions in log-odds space, $e_t = |\sigma^{-1}(p_t) - \sigma^{-1}(p_t^*)|$ where p_t^* was the true CTR (given by the ground truth model). Figure 3 plots the errors e_t as a function of the uncertainty score $u_t = u(\mathbf{x}_t)$; there is a high degree of correlation.

Additional experiments showed the uncertainty scores performed comparably (under the above evaluation regime) to the much more expensive estimates obtained via a bootstrap of 32 models trained on random subsamples of data.

7. CALIBRATING PREDICTIONS

Accurate and well-calibrated predictions are not only essential to run the auction, they also allow for a loosely coupled overall system design separating concerns of optimizations in the auction from the machine learning machinery.

Systematic bias (the difference between the average predicted and observed CTR on some slice of data) can be caused by a variety of factors, e.g., inaccurate modeling assumptions, deficiencies in the learning algorithm, or hidden features not available at training and/or serving time. To

address this, we can use a calibration layer to match predicted CTRs to observed click–through rates.

Our predictions are calibrated on a slice of data d if on average when we predict p, the actual observed CTR was near p. We can improve calibration by applying correction functions $\tau_d(p)$ where p is the predicted CTR and d is an element of a partition of the training data. We define success as giving well calibrated predictions across a wide range of possible partitions of the data.

A simple way of modeling τ is to fit a function $\tau(p) = \gamma p^\kappa$ to the data. We can learn γ and κ using Poisson regression on aggregated data. A slightly more general approach that is able to cope with more complicated shapes in bias curves is to use a piecewise linear or piecewise constant correction function. The only restriction is that the mapping function τ should be isotonic (monotonically increasing). We can find such a mapping using isotonic regression, which computes a weighted least-squares fit to the input data subject to that constraint (see, e.g., [27, 23]). This piecewise-linear approach significantly reduced bias for predictions at both the high and low ends of the range, compared to the reasonable baseline method above.

It is worth noting that, without strong additional assumptions, the inherent feedback loop in the system makes it impossible to provide theoretical guarantees for the impact of calibration [25].

8. AUTOMATED FEATURE MANAGEMENT

An important aspect of scalable machine learning is managing the scale of the *installation*, encompassing all of the configuration, developers, code, and computing resources that make up a machine learning system. An installation comprised of several teams modeling dozens of domain specific problems requires some overhead. A particularly interesting case is the management of the feature space for machine learning.

We can characterize the feature space as a set of contextual and semantic *signals*, where each signal (e.g., 'words in the advertisement', 'country of origin', etc.) can be translated to a set of real-valued features for learning. In a large installation, many developers may work asynchronously on signal development. A signal may have many versions corresponding to configuration changes, improvements, and alternative implementations. An engineering team may *consume* signals which they do not directly develop. Signals may be consumed on multiple distinct learning platforms and applied to differing learning problems (e.g. predicting search vs. display ad CTR). To handle the combinatorial growth of use cases, we have deployed a metadata index for managing consumption of thousands of input signals by hundreds of active models.

Indexed signals are annotated both manually and automatically for a variety of concerns; examples include deprecation, platform-specific availability, and domain-specific applicability. Signals consumed by new and active models are vetted by an automatic system of alerts. Different learning platforms share a common interface for reporting signal consumption to a central index. When a signal is deprecated (such as when a newer version is made available), we can quickly identify all consumers of the signal and track replacement efforts. When an improved version of a signal is made available, consumers can be alerted to experiment with the new version.

New signals can be vetted by automatic testing and white-listed for inclusion. White-lists can be used both for ensuring correctness of production systems, and for learning systems using automated feature selection. Old signals which are no longer consumed are automatically earmarked for code cleanup, and for deletion of any associated data.

Effective automated signal consumption management ensures that more learning is done correctly the first time. This cuts down on wasted and duplicate engineering effort, saving many engineering hours. Validating configurations for correctness before running learning algorithms eliminates many cases where an unusable model might result, saving significant potential resource waste.

9. UNSUCCESSFUL EXPERIMENTS

In this final section, we report briefly on a few directions that (perhaps surprisingly) did not yield significant benefit.

9.1 Aggressive Feature Hashing

In recent years, there has been a flurry of activity around the use of feature hashing to reduce RAM cost of large-scale learning. Notably, [31] report excellent results using the hashing trick to project a feature space capable of learning personalized spam filtering model down to a space of only 2^{24} features, resulting in a model small enough to fit easily in RAM on one machine. Similarly, Chapelle reported using the hashing trick with 2^{24} resultant features for modeling display-advertisement data [6].

We tested this approach but found that we were unable to project down lower than several billion features without observable loss. This did not provide significant savings for us, and we have preferred to maintain interpretable (non-hashed) feature vectors instead.

9.2 Dropout

Recent work has placed interest around the novel technique of randomized "dropout" in training, especially in the deep belief network community [17]. The main idea is to randomly remove features from input example vectors independently with probability p, and compensate for this by scaling the resulting weight vector by a factor of $(1 - p)$ at test time. This is seen as a form of regularization that emulates bagging over possible feature subsets.

We have experimented with a range of dropout rates from 0.1 to 0.5, each with an accompanying grid search for learning rate settings, including varying the number of passes over the data. In all cases, we have found that dropout training does not give a benefit in predictive accuracy metrics or generalization ability, and most often produces detriment.

We believe the source of difference between these negative results and the promising results from the vision community lie in the differences in feature distribution. In vision tasks, input features are commonly dense, while in our task input features are sparse and labels are noisy. In the dense setting, dropout serves to separate effects from strongly correlated features, resulting in a more robust classifier. But in our sparse, noisy setting adding in dropout appears to simply reduce the amount of data available for learning.

9.3 Feature Bagging

Another training variant along the lines of dropout that we investigated was that of feature bagging, in which k models are trained independently on k overlapping subsets of the

feature space. The outputs of the models are averaged for a final prediction. This approach has been used extensively in the data mining community, most notably with ensembles of decision trees [9], offering a potentially useful way of managing the bias-variance tradeoff. We were also interested in this as a potentially useful way to further parallelize training. However, we found that feature bagging actually slightly reduced predictive quality, by between 0.1% and 0.6% AucLoss depending on the bagging scheme.

9.4 Feature Vector Normalization

In our models the number of non-zero features per event can vary significantly, causing different examples $\mathbf{x}$ to have different magnitudes $\|\mathbf{x}\|$. We worried that this variability may slow convergence or impact prediction accuracy. We explored several flavors of normalizing by training with $\frac{\mathbf{x}}{\|\mathbf{x}\|}$ with a variety of norms, with the goal of reducing the variance in magnitude across example vectors. Despite some early results showing small accuracy gains we were unable to translate these into overall positive metrics. In fact, our experiments looked somewhat detrimental, possibly due to interaction with per-coordinate learning rates and regularization.

10. ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of the following: Vinay Chaudhary, Jean-Francois Crespo, Jonathan Feinberg, Mike Hochberg, Philip Henderson, Sridhar Ramaswamy, Ricky Shan, Sajid Siddiqi, and Matthew Streeter.

11. REFERENCES

[1] D. Agarwal, B.-C. Chen, and P. Elango. Spatio-temporal models for estimating click-through rate. In *Proceedings of the 18th international conference on World wide web*, pages 21–30. ACM, 2009.

[2] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD Conference*, 2013. To appear.

[3] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up machine learning: Parallel and distributed approaches*. 2011.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), July 1970.

[5] A. Blum, A. Kalai, and J. Langford. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *COLT*, 1999.

[6] O. Chapelle. Click modeling for display advertising. In *AdML: 2012 ICML Workshop on Online Advertising*, 2012.

[7] C. Cortes, M. Mohri, M. Riley, and A. Rostamizadeh. Sample selection bias correction theory. In *ALT*, 2008.

[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[9] T. G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[10] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *COLT*, 2010.

[11] J. Duchi and Y. Singer. Efficient learning using forward-backward splitting. In *Advances in Neural Information Processing Systems 22*, pages 495–503. 2009.

[12] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), jun 2000.

[13] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[14] D. Golovin, D. Sculley, H. B. McMahan, and M. Young. Large-scale learning with a small-scale footprint. In *ICML*, 2013. To appear.

[15] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale Bayesian click-through rate prediction for sponsored search advertising in microsofts bing search engine. In *Proc. 27th Internat. Conf. on Machine Learning*, 2010.

[16] D. Hillard, S. Schroedl, E. Manavoglu, H. Raghavan, and C. Leggetter. Improving ad relevance in sponsored search. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 361–370, 2010.

[17] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[18] D. W. Hosmer and S. Lemeshow. *Applied logistic regression*. Wiley-Interscience Publication, 2000.

[19] H. A. Koepke and M. Bilenko. Fast prediction of new feature utility. In *ICML*, 2012.

[20] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. *JMLR*, 10, 2009.

[21] S.-M. Li, M. Mahdian, and R. P. McAfee. Value of learning in sponsored search auctions. In *WINE*, 2010.

[22] W. Li, X. Wang, R. Zhang, Y. Cui, J. Mao, and R. Jin. Exploitation and exploration in a performance based contextual advertising system. In *KDD*, 2010.

[23] R. Luss, S. Rosset, and M. Shahar. Efficient regularized isotonic regression with application to gene–gene interaction search. *Ann. Appl. Stat.*, 6(1), 2012.

[24] H. B. McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and L1 regularization. In *AISTATS*, 2011.

[25] H. B. McMahan and O. Muralidharan. On calibrated predictions for auction selection mechanisms. *CoRR*, abs/1211.3955, 2012.

[26] H. B. McMahan and M. Streeter. Adaptive bound optimization for online convex optimization. In *COLT*, 2010.

[27] A. Niculescu-Mizil and R. Caruana. Predicting good probabilities with supervised learning. In *ICML*, ICML '05, 2005.

[28] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pages 521–530. ACM, 2007.

[29] M. J. Streeter and H. B. McMahan. Less regret via online conditioning. *CoRR*, abs/1002.4862, 2010.

[30] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer. Overlapping experiment infrastructure: more, better, faster experimentation. In *KDD*, pages 17–26, 2010.

[31] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *ICML*, pages 1113–1120. ACM, 2009.

[32] L. Xiao. Dual averaging method for regularized stochastic learning and online optimization. In *NIPS*, 2009.

[33] Z. A. Zhu, W. Chen, T. Minka, C. Zhu, and Z. Chen. A novel click model and its applications to online advertising. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 321–330. ACM, 2010.

[34] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *ICML*, 2003.