A Stochastic Treatment of Learning to Rank Scoring Functions

Sebastian Bruch, Shuguang Han, Michael Bendersky, Marc Najork
Google Research
{bruch,hanshuguang,bemike,najork}@google.com

ABSTRACT

Learning to Rank, a central problem in information retrieval, is a class of machine learning algorithms that formulate ranking as an optimization task. The objective is to learn a function that produces an ordering of a set of documents in such a way that the utility of the entire ordered list is maximized. Learning-to-rank methods do so by learning a function that computes a score for each document in the set. A ranked list is then compiled by sorting documents according to their scores. While such a deterministic mapping of scores to permutations makes sense during inference where stability of ranked lists is required, we argue that its greedy nature during training leads to less robust models. This is particularly problematic when the loss function under optimization—in agreement with ranking metrics—largely penalizes incorrect rankings and does not take into account the distribution of raw scores. In this work, we present a stochastic framework where, instead of a deterministic derivation of permutations from raw scores, permutations are sampled from a distribution defined by raw scores. Our proposed sampling method is differentiable and works well with gradient descent optimizers. We analytically study our proposed method and demonstrate when and why it leads to model robustness. We also show empirically, through experiments on publicly available learning-to-rank datasets, that the application of our proposed method to a class of ranking loss functions leads to significant model quality improvements.

CCS CONCEPTS

• **Information systems → Learning to rank**;

KEYWORDS

Ranking Metric Optimization; Stochastic Scoring Functions for Learning to Rank; Learning to Rank

ACM Reference Format:
Sebastian Bruch, Shuguang Han, Michael Bendersky, Marc Najork. 2020. A Stochastic Treatment of Learning to Rank Scoring Functions. In *The Thirteenth ACM International Conference on Web Search and Data Mining (WSDM'20), February 3–7, 2020, Houston, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3336191.3371844

1 INTRODUCTION

At the heart of many Information Retrieval tasks lies a ranking problem where items in a set must be ordered so as to maximize a utility. In *ad hoc* document retrieval, for example, given a user query, the task is to retrieve and present a list of documents in decreasing order of relevance. Approaching ranking problems using supervised machine learning is what is known as Learning to Rank.

Learning-to-rank methods construct a parameterized function from sets of objects into permutations. An object may be a query-document pair represented by a feature vector and permutations are ordered documents. The space of all permutations, however, grows factorially in the size of the input set, thereby making the stated learning task intractable. Instead, learning-to-rank is reformulated as learning a *scoring* function that computes a relevance score for every document. By sorting documents in decreasing order, one may deterministically derive a ranked list.

Sorting is a deterministic way to construct a permutation from scores. Such a mapping makes sense during inference where the stability of ranked lists is often required. We argue, however, that this behavior may not in fact be desired during training.

Our intuition is that the greedy nature of a deterministic mapping makes for a model that is less robust to noise. This is particularly the case when the behavior of the loss function mimics that of ranking metrics. This includes a class of ranking losses that compute a penalty only for incorrectly ordered documents and, crucially, do not factor the distribution of raw scores into the total cost.

To elaborate, consider a query and two documents, one more relevant than the other, for which scores computed by a model are within a small neighborhood but that yield the correct ordering under a sort operation. Because of the correct ordering of the pair, it is possible that the model receives no penalty during training and as such has no incentive to push the scores of the two documents further apart. However, with the scores being close, it is possible that a minor change to the query or one of the documents could reverse the ordering and produce an incorrect ranked list, thereby reducing the robustness of the learned model. The behavior that is intuitively desired, on the other hand, is to not only encourage correct ordering of documents but also widen the margin between relevance classes.

The research question we consider in this work is then whether a randomized mapping of scores to permutations improves model robustness. In a stochastic setting, rather than sorting documents by scores to construct a permutation, permutations are *sampled* from a distribution defined by raw scores. By sampling from this distribution, the model from our example above is more likely to encounter a permutation in which the order of documents is reversed and, as a result, incur a non-zero penalty.

In this work, we examine the hypothesis above. We introduce a differentiable sampling strategy that can be integrated with gradient descent optimizers. We then present an analytical study of classes of loss functions and explain when and why our proposed method leads to model robustness. We also evaluate our proposed method on publicly available datasets. Experimental results confirm our intuition from above and agree with our analytical findings.

Our contributions can be summarized as follows:

- We present a stochastic treatment of the mapping from scores produced by a learning-to-rank function to permutations, which can be applied to arbitrary ranking loss functions;
- We analyze the proposed method and show when and why one may expect quality improvements; and,
- We demonstrate empirically that the produced models converge faster and exhibit greater generalization in some cases.

The remainder of this paper is organized as follows: Section 2 provides an overview of the literature. We review learning-to-rank in Section 3 and give a detailed description of our proposed method in Section 4. In Section 5, we study existing loss functions within our framework. Sections 6 and 7 present our evaluation setup and results. Finally, we conclude the paper in Section 8.

2 BACKGROUND AND RELATED WORK

Learning-to-rank algorithms generally address the ranking problem using a score-and-sort approach [4, 5, 7, 20, 21, 25, 40]. The goal is to learn a scoring function to compute relevance scores which, in turn, induce a ranking. In its most general form, the domain of learning-to-rank functions is a set rather than a single item. However, virtually all learning-to-rank methods with a few exceptions [1, 10, 33] simplify the problem further by learning a univariate function that produces a relevance score for a document independently of other documents in the input set.

It is true then that learning-to-rank can be formulated as classification or regression—in fact, many early learning-to-rank methods such as RankSVM [20] or RankNet [4] take a very similar approach. These algorithms reduce the ranking problem to one of correctly predicting relevance scores by optimizing a "pointwise" loss [13] or correctly classifying ordered pairs of documents by optimizing a "pairwise" loss [4, 5, 20]. These simplified reformulations of learning-to-rank are, however, misaligned with the ranking utilities.

Ranking utilities such as Normalized Discounted Cumulative Gain [19] or Expected Reciprocal Rank [9] work with permutations (i.e., ranked lists) which are discrete structures. As a result, ranking utilities, as a function of a set of input documents, are flat almost everywhere and discontinuous at some finite set of points.

The non-smoothness of ranking utilities pose a challenge that the learning-to-rank community has sought to study. The literature offers a range of methods from direct optimization of metrics using coordinate ascent over parameters of linear models [29], to optimizing an exponential upper-bound of ranking metrics using boosted weak learners [41], to optimizing a differentiable surrogate loss function [7, 32, 36, 38, 40]. Other methods, such as LambdaRank [6] and its gradient boosted regression tree-based [12] variant LambdaMART [39], assume the existence of an unknown loss function whose gradients are however designed based on some heuristic. The list of so-called "listwise" algorithms goes on but the individual methods fall into one of the above categories.

Despite these differences, existing listwise learning-to-rank algorithms agree on one element: scores computed by the learned scoring function are deterministically mapped to a ranked list by way of a sort operation. One exception is SoftRank [36]. Taylor et al. consider a score to be the mean of a Gaussian distribution. With scores being smooth in this way, they go on to estimate position distributions and ultimately define a smoothed version of ranking metrics. We note that while our work bears some superficial resemblance with SoftRank, our approach is fundamentally different: SoftRank considers each score to itself be a Gaussian distribution—an arbitrary choice—whereas in this work, we take a set of scores to define a distribution from which a permutation may be sampled. Furthermore, our method is efficient while in SoftRank, estimating position distributions given score distributions requires an inefficient construction.

Another work that uses additive noise is YetiRank [16]. In particular, YetiRank perturbs relevance scores by a noise sampled from the Logistic distribution, and uses the perturbed scores to weight document pairs. YetiRank is different from our work in the following ways: (a) while the authors demonstrated that additive noise results in an improved model, the use of Logistic distribution was not justified, whereas in this work we mathematically motivate the use of the Gumbel distribution; and (b) YetiRank uses noise to identify and re-weight document pairs, whereas our methodology could be used to sample from the space of permutations, thereby presenting a more general, ranking-appropriate framework.

Finally, another related work is the LambdaLoss framework [38]. Wang et al. propose a probabilistic framework to model ranking loss functions and show that existing ranking losses are instances of LambdaLoss. One term in LambdaLoss captures the probability of a permutation given a set of scores, $p(\pi|f(\boldsymbol{x}))$. In their work, however, Wang et al. use a degenerate distribution where this probability is 1 for a permutation (deterministically) obtained by sorting scores in decreasing order. Our proposed stochastic framework allows one to construct a non-degenerate distribution over permutations, from which a permutation may be sampled directly and efficiently.

3 PROBLEM FORMULATION

In this section, we formalize the problem and introduce our notation. Let $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X}^n \times \mathbb{R}_+^n$ be a training example where $\boldsymbol{x}$ is a vector of n objects x_i, $1 \leq i \leq n$, $\boldsymbol{y}$ is a vector of n nonnegative relevance labels y_i, $1 \leq i \leq n$, and $\mathcal{X}$ is the space of all objects. To simplify discussion, we refer to x_i as a "document" and $\boldsymbol{x} \in \mathcal{X}^n$ as a list of n documents, but note that x_i could itself be a d-dimensional vector representing a query-document pair. For every document $x_i \in \boldsymbol{x}$, we have a corresponding relevance label $y_i \in \boldsymbol{y}$. As such, $(\boldsymbol{x}, \boldsymbol{y})$ represents a single query with n documents and their corresponding relevance labels. Finally, let Ψ be a set of training examples.

As noted in earlier sections, the goal is to find a scoring function $f : \mathcal{X}^n \to \mathbb{R}^n$ that minimizes the empirical loss:

$$\mathcal{L}(f) = \frac{1}{|\Psi|} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \Psi} \ell(\boldsymbol{y}, f(\boldsymbol{x})), \tag{1}$$

where $\ell(\cdot)$, a local loss, is assumed to be differentiable in this work.

Most learning-to-rank algorithms simplify the task by learning a univariate function $u : \mathcal{X} \to \mathbb{R}$ that computes a score for each document independently of others: $f(\boldsymbol{x})|_i = u(x_i)$, $1 \leq i \leq n$, where $f(\cdot)|_i$ denotes the i^{th} dimension of $f(\cdot)$. Without loss of generality, we assume the scores $f(\boldsymbol{x})$ are the logarithmic transformation of a strictly positive measure. Formally, $f(\boldsymbol{x})|_i = \log(\alpha_i)$, where

$\alpha_i \in \mathbb{R}_+$. We note that the function f is often parameterized and written as $f(\cdot;\Theta)$ where Θ is the set of parameters. For a list of Θ typical in learning-to-rank methods we refer the reader to Section 2.

Turning to the loss function ℓ, we have from Section 2 that listwise functions ($\ell : \mathbb{R}^n_+ \times \mathbb{R}^n \rightarrow \mathbb{R}$) are more aligned with ranking objectives. For this reason, we focus on listwise losses in this study.

Some listwise loss functions such as ListNet [7] first project labels $\boldsymbol{y}$ and scores $f(\boldsymbol{x})$ onto the probability simplex Δ^{n-1} to form distributions $P_{\boldsymbol{y}}$ and P_f, respectively. The probability mass placed on each x_i indicates the likelihood of x_i appearing at position 1 of the final ranked list. They then compute the distance between the two distributions. In effect, this class of loss functions is a composition of $P_{\boldsymbol{y}}, P_f : \mathbb{R}^n \rightarrow \Delta^{n-1}$ and $\hat{\ell} : \Delta^{n-1} \times \Delta^{n-1} \rightarrow \mathbb{R}$.

Another class of losses derived from the approximation framework of Qin et al. [32] or SoftRank [36] are more aligned with ranking metrics. These loss functions form permutations from scores and subsequently compute a smooth approximation to a ranking metric. In effect, these functions can be decomposed to $\pi_f : \mathbb{R}^n \rightarrow \Pi^n$, where Π^n is the space of all permutations of size n, and a function $\hat{\ell} : \mathbb{R}^n_+ \times \Pi^n \rightarrow \mathbb{R}$.

Finally, we place losses such as LambdaMART's [6, 39] into a third class. These losses have one factor that acts on permutations (e.g., the change in ranking metric resulting from swapping two documents in the ranked list) and another factor that acts on raw scores (e.g., a sigmoid term).

One typical construction of $\pi_f(\cdot)$ that is commonly used by the last two classes of loss functions during training is the sort operation or a smooth approximation to it. Such a derivation of permutations from scores is a deterministic process: Given $f(\boldsymbol{x}) \in \mathbb{R}^n$, $\pi_f(\cdot)$ always produces the same permutation. While this is an often desired behavior during inference, in this paper, we are interested in studying a stochastic construction of $\pi_f(\cdot)$ during *training*.

4 PROPOSED METHODOLOGY

The core idea is to let scores $f(\boldsymbol{x})$ define a distribution over the space of permutations and subsequently sample permutations from said distribution. More formally, let us denote such a distribution by $\mathbb{P}_{\boldsymbol{\alpha}}$ with parameter $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ where, as before, $f(\boldsymbol{x})|_i = \log(\alpha_i)$ for all i. Let us assume that sampling from $\mathbb{P}_{\boldsymbol{\alpha}}$ can be *reparameterized* [27] as sampling from an *independent* source of noise $\mathbb{Q}$ and passing that noise through a *smooth* deterministic function $\hat{g}_{\boldsymbol{\alpha}}$—we will show how this can be achieved in the next section. Given this sampling process, we propose to optimize the *expected* empirical loss:

$$\mathcal{L}(f) = \frac{1}{|\Psi|} \sum_{\Psi} \mathop{\mathbb{E}}_{\boldsymbol{\pi}_f \sim \mathbb{P}_{\boldsymbol{\alpha}}} [\hat{\ell}(\boldsymbol{y}, \boldsymbol{\pi}_f)] = \frac{1}{|\Psi|} \sum_{\Psi} \mathop{\mathbb{E}}_{Z \sim \mathbb{Q}} [\hat{\ell}(\boldsymbol{y}, \hat{g}_{\boldsymbol{\alpha}}(Z))], \quad (2)$$

where $\hat{\ell} : \mathbb{R}^n_+ \times \Pi^n \rightarrow \mathbb{R}$ is smooth. Because $\hat{g}_{\boldsymbol{\alpha}}$ is smooth, Equation (2) can be optimized using gradient descent, and because we are sampling from an independent noise distribution, $\mathbb{Q}$, the expectation may be approximated using Monte Carlo [34]. Note that, without reparameterizing the sampling process, the gradient of the expectation with respect to parameters of the distribution cannot be estimated using Monte Carelo. In the next section, we will present our proposed method to draw samples from $\mathbb{P}_{\boldsymbol{\alpha}}$ without necessarily constructing the distribution.

4.1 Sampling Permutations

In this section, we consider the task of drawing a single permutation given n documents and their scores. A permutation may be constructed following the Plackett-Luce model. Conceptually, the process works as follows: Scores are projected onto the probability simplex and a single document is sampled from the resulting distribution. The sampled document is subsequently removed from the set and placed at position 1 of the permutation. The process restarts to fill the next position until the set of documents is empty.

More formally, given a set of documents and their scores $f(\boldsymbol{x})|_i = \log(\alpha_i)$, $1 \leq i \leq n$, define the following probability distribution:

$$P_f(x_i) \triangleq \frac{e^{\frac{1}{\beta} f(\boldsymbol{x})|_i}}{\sum_{j=1}^n e^{\frac{1}{\beta} f(\boldsymbol{x})|_j}}, \; 1 \leq i \leq n, \quad (3)$$

which is a temperature-controlled Softmax transformation where $\beta > 0$. Given the above distribution, draw a sample x_{π_1} and place it at the first position. Continue this sampling without replacement to fill subsequent positions until no document is left in the set. Note that, in each round, the distribution in Equation (3) may change.

Drawing a sample x_{π_k} can be achieved using the Gumbel-Max method [26–28], a known technique that meets our reparameterizability requirement above. Let us illustrate this by drawing the first sample, x_{π_1}; extending this to x_{π_k} for $k > 1$ is trivial. To do that, we draw n noise samples (one per document in the set) independently from Gumbel($\mu = 0, \beta$) (i.e., source of noise $\mathbb{Q}$ in Equation (2)), denoted by G_i for all $1 \leq i \leq n$. We then produce x_{π_1} where $\pi_1 = \mathop{\mathrm{argmax}}_{1 \leq i \leq n} \log(\alpha_i) + G_i$ (i.e., $\hat{g}_{\boldsymbol{\alpha}}$ of Equation (2)). It can be shown that the probability of sampling x_i in the first round is equal to Equation (3).

Drawing a sample G_i from the Gumbel distribution can itself be done very efficiently by further reparameterization: Sample $U_i \sim \text{Uniform}(0, 1)$ and set $G_i = -\beta \log(-\log U_i)$. It is easy to show that G_i is Gumbel($\mu = 0, \beta$) distributed; to see why, note that $-\beta \log(-\log(\cdot))$ is the Gumbel quantile function.

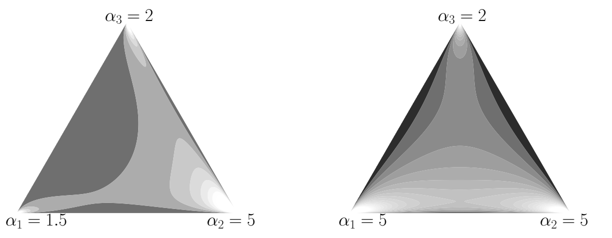
Unfortunately, the sampling procedure above is not differentiable as the random variable being sampled (i.e., a document) is discrete or, equivalently, because $\hat{g}_{\boldsymbol{\alpha}}$ = argmax is non-differentiable. In the next section, we address this non-differentiability.

4.2 Stochastic Scores

To address the non-differentiability of sampling from discrete distributions, Maddison et al. recently proposed the Concrete distribution [27]. The Concrete distribution is a continuous relaxation of the discrete space. That is, instead of strictly sampling from the vertices of the probability simplex (where samples are discrete), Maddison et al. propose to relax the discrete space into the interior of the simplex (i.e., if $\hat{\boldsymbol{y}}$ is a sample from Concrete($\boldsymbol{\alpha}$) then $\hat{\boldsymbol{y}} \in \Delta^{n-1}$). This is illustrated in Figure 1 for two example $\boldsymbol{\alpha}$ vectors.

Sampling from Concrete($\boldsymbol{\alpha}$) needs only a minor adjustment to the Gumbel-Max method: Instead of argmax, a temperature-controlled Softmax operator is used. As a result, the i^{th} coordinate of $\hat{\boldsymbol{y}}$ has the following form:

$$\hat{y}_i = \frac{e^{\log \alpha_i + G_i}}{\sum_{j=1}^n e^{\log \alpha_j + G_j}}. \quad (4)$$



(a) $\boldsymbol{\alpha} = (1.5, 5, 2)$ (b) $\boldsymbol{\alpha} = (5, 5, 2)$

Figure 1: Visualization of Concrete($\boldsymbol{\alpha}$) with $\beta = 1$. Brighter points indicate more likely samples. A point in the interior of the simplex defines a probability distribution over the three documents.

The Softmax above can optionally be controlled by a temperature such that, in the limit, it converges to the Gumbel-Max sampling method [27].

Sampling from the Concrete distribution is actually a more natural choice for constructing a permutation. Instead of an iterative sampling without replacement as laid out in Section 4.1, a single sample from the Concrete distribution suffices to derive an ordering. In effect, the continuous random variable $\hat{\boldsymbol{y}} \sim \text{Concrete}(\boldsymbol{\alpha})$ can be understood as encoding the likelihood of each $x_i \in \boldsymbol{x}$ appearing at position 1 of the permutation.

Given the properties of the Concrete distribution, we arrive at our proposed permutation sampling strategy: We first sample $\hat{\boldsymbol{y}}$ from the Concrete distribution parameterized by raw scores $\boldsymbol{\alpha}$ where $\hat{y}_i$'s are "stochastic scores" for elements of $\boldsymbol{x}$. We then sort elements of $\boldsymbol{x}$ by these scores to obtain a permutation.

What is interesting is that one may simply view $\hat{\boldsymbol{y}}$ as stochastic scores sampled from a distribution defined by raw scores $f(\boldsymbol{x})$. The advantage to this interpretation is that we can simply pass $\hat{\boldsymbol{y}}$ to any arbitrary listwise loss function $\ell : \mathbb{R}_+^n \times \mathbb{R}^n \to \mathbb{R}$ in lieu of $f(\boldsymbol{x})$ without constructing permutations. This is the approach we take in the remainder of this paper. Algorithm 1 summarizes our proposed method by showing how stochastic scores are sampled given raw scores. Note that, we apply a logarithmic transformation to $\hat{y}_i$'s for consistency (to operate in log-scale) and to simplify analysis.

5 ANALYSIS OF LOSS FUNCTIONS IN THE STOCHASTIC FRAMEWORK

In Section 4, we left open the choice of the local loss function ℓ but only required that it be differentiable. In this section, we consider three classes of listwise loss functions: cross entropy from ListNet [7], approximate ranking metrics [32], and LambdaMART [6, 39]. The cross entropy loss acts on the space of scores and does in fact incur a penalty for scores that are (incorrectly) too close to each other; our intuition is then that cross entropy is unlikely to benefit from our proposed method. We choose the approximate metric loss because it is less concerned with raw scores and instead operates on the space of permutations. As such it is an appropriate loss to help verify our hypothesis from Section 1. Finally, we select LambdaMART because its loss is a hybrid of score-based and permutation-based functions.

Throughout this section we set the Gumbel shape β to 1 to simplify notation, but the same analysis can be extended to arbitrary values of $\beta > 0$.

Algorithm 1: Stochastic scores for learning-to-rank

```
**Input:** Training batch $B_\Psi$; current parameters $\Theta$; number of samples to draw, $N$; Gumbel shape $\beta$
**Result:** Sampled batch $\hat{B}_\Psi$ of stochastic scores
1:  $\hat{B}_\Psi \leftarrow \emptyset$
2:  for $(\boldsymbol{x}, \boldsymbol{y}) \in B_\Psi$ do
3:      $\alpha_i \leftarrow e^{f(\boldsymbol{x};\Theta)|_i}$, $1 \leq i \leq n$
4:      for 1 to $N$ do
5:          $U_i \sim \text{Uniform}(0,1)$, $G_i = -\beta \log(-\log U_i)$, $\forall 1 \leq i \leq n$
6:          $\hat{y}_i = \frac{e^{\log(\alpha_i)+G_i}}{\sum_{j=1}^n e^{\log(\alpha_j)+G_j}}$, $1 \leq i \leq n$
7:          $\hat{\boldsymbol{y}} \leftarrow (\log \hat{y}_1, \log \hat{y}_2, \ldots, \log \hat{y}_n)$
8:          append$(\hat{B}_\Psi, (\hat{\boldsymbol{y}}, \boldsymbol{y}))$
9:      end for
10: end for
```

5.1 Cross Entropy

Cross entropy is at the heart of several learning-to-rank algorithms [7, 40] and was shown [2] to indirectly optimize NDCG in a binary relevance regime. The loss provides a notion of distance between two probability distributions:

$$\ell(\boldsymbol{y}, f(\boldsymbol{x})) = H(P_{\boldsymbol{y}}, P_f) \triangleq -\sum_{i=1}^{n} P_{\boldsymbol{y}}(x_i) \log P_f(x_i), \tag{5}$$

where $P_{\boldsymbol{y}}$ and P_f are probability distributions over $\boldsymbol{x}$ derived from labels and scores respectively, $n = |\boldsymbol{x}|$. There are many recipes for constructing $P_{\boldsymbol{y}}$ and P_f. In this work, we choose a general construction. In particular, in the deterministic case, we have that:

$$P_{\boldsymbol{y}}(x_i) = \frac{y_i}{\sum_{j=1}^n y_j}, \quad P_f(x_i) = \frac{e^{f(\boldsymbol{x})|_i}}{\sum_{j=1}^n e^{f(\boldsymbol{x})|_j}} = \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \tag{6}$$

Using the above, we can expand Equation (5) as follows:

$$H(P_{\boldsymbol{y}}, P_f) = -\sum_{i=1}^{n} P_{\boldsymbol{y}}(x_i) \log P_f(x_i) = -\sum_{i=1}^{n} \frac{y_i}{\sum y_j} \log \frac{\alpha_i}{\sum \alpha_j} \tag{7}$$

$$= -\sum_{i=1}^{n} \frac{y_i}{\sum y_j} \log \alpha_i + \log \sum_{j=1}^{n} \alpha_j. \tag{8}$$

Let us now consider this loss in the context of Algorithm 1. While $P_{\boldsymbol{y}}$ need not change, we let P_f be the sample $\hat{\boldsymbol{y}} \sim \text{Concrete}(\boldsymbol{\alpha})$:

$$P_f(x_i) = \frac{e^{\log \alpha_i + G_i}}{\sum_{j=1}^n e^{\log \alpha_j + G_j}}, \tag{9}$$

where $G_k \sim \text{Gumbel}(\mu = 0, \beta = 1)\, \forall 1 \leq k \leq n$. Plugging this into Equation (5) and taking the expectation leads to the following:

$$\underset{G_k \sim \text{Gumbel}\ \forall k}{\mathbb{E}}[H(P_{\boldsymbol{y}}, P_f)] = -\mathbb{E}[\sum_{i=1}^{n} P_{\boldsymbol{y}}(x_i) \log P_f(x_i)] \tag{10}$$

$$= -\mathbb{E}[\sum_{i=1}^{n} \frac{y_i}{\sum y_j}(\log \alpha_i + G_i - \log \sum_{j=1}^{n} \alpha_j e^{G_j})] \tag{11}$$

$$= -\sum_{i=1}^{n} \frac{y_i}{\sum y_j} \log \alpha_i - \gamma + \mathbb{E}[\log \sum_{j=1}^{n} \alpha_j e^{G_j}] \tag{12}$$

$$\leq -\sum_{i=1}^{n} \frac{y_i}{\sum y_j} \log \alpha_i + \log \mathbb{E}[\sum_{j=1}^{n} \alpha_j e^{G_j}] \quad (13)$$

$$= \underbrace{-\sum_{i=1}^{n} \frac{y_i}{\sum y_j} \log \alpha_i + \log \sum_{j=1}^{n} \alpha_j}_{H(P_{\boldsymbol{y}}, P_f)} + \log \mathbb{E}[e^{G}], \quad (14)$$

where (11) holds by definition, (12) is derived by linearity of expectation and the fact that $\mathbb{E}_{G \sim \text{Gumbel}(\mu=0,\beta=1)}(G) = \gamma$ (the Euler-Mascheroni constant), (13) is true by Jensen's inequality and concavity of $\log(\cdot)$, and (14) rearranges the expression.

In theory, the expectation in the last term of Equation (14) does not exist. In practice, however, we draw a Gumbel sample by drawing a noise not from Uniform$(0, 1)$ but from Uniform$(\epsilon, 1-\epsilon)$. With that change, the expectation in Equation (14) is a constant.

The analysis above shows that the performance of the cross entropy loss is invariant under the stochastic procedure of Algorithm 1: Equation (14) upper-bounds the optimization problem in Algorithm 1. We therefore do not expect a performance gain using the cross entropy loss function. This finding certainly agrees with our intuition. The cross entropy loss, being a function that operates on raw scores and not permutations induced by raw scores, already penalizes a model for producing scores that are incorrectly too close.

5.2 Approximate Ranking Metric

We now turn to a second class of loss functions that are much closer to ranking metrics: approximate ranking metrics by Qin et al. [32]. In [32], the authors provide a general framework where, given a ranking metric, one can derive a smooth approximation. More recently, Bruch et al. [3] showed that using neural networks to optimize this approximate surrogates results in models that are comparable with the state-of-the-art methods.

To understand this family of loss functions better, we take NDCG [19] as an example metric and show how we may derive an approximate NDCG, dubbed ApproxNDCG. Subsequently, we examine the effect of Algorithm 1 on this framework.

5.2.1 Overview of ApproxNDCG. We first begin by a definition of NDCG:

$$\text{NDCG}(\boldsymbol{y}, \boldsymbol{\pi}_f) = \frac{\text{DCG}(\boldsymbol{\pi}_f, \boldsymbol{y})}{\text{DCG}(\boldsymbol{\pi}_{\boldsymbol{y}}, \boldsymbol{y})}, \quad (15)$$

where $\boldsymbol{\pi}_f$ is a ranked list induced by f on $\boldsymbol{x}$, $\boldsymbol{\pi}_{\boldsymbol{y}}$ is the ideal ranked list (where $\boldsymbol{x}$ is sorted by $\boldsymbol{y}$), and DCG is defined as follows:

$$\text{DCG}(\boldsymbol{y}, \boldsymbol{\pi}) = \sum_{i=1}^{n} \frac{2^{y_i} - 1}{\log_2(1 + \boldsymbol{\pi}[i])}, \quad (16)$$

where $\boldsymbol{\pi}[i]$ is the rank of x_i.

As shown in Equations (15) and (16), all that is needed to compute DCG is $\boldsymbol{\pi}_f$, which may be calculated as follows:

$$\boldsymbol{\pi}_f[i] \triangleq 1 + \sum_{j \neq i} \mathbb{I}_{f(\boldsymbol{x})|_i < f(\boldsymbol{x})|_j}, \quad (17)$$

where $\mathbb{I}_{s<t}$ is an indicator which is 1 if $s < t$ and 0 otherwise.

The main idea in [32] is to construct a smooth variant of DCG by approximating the indicator function $\mathbb{I}$ using a sigmoid function:

$$\mathbb{I}_{s<t} = \mathbb{I}_{t-s>0} \approx \sigma(t-s) \triangleq \frac{1}{1 + e^{-\eta(t-s)}}, \quad (18)$$

where $\eta > 0$ controls how tightly the sigmoid fits the indicator function. A small η leads to a curve loosely related to the indicator function, and a large η leads to gradients vanishing not far from the origin, making them uninteresting.

The approximation in Equation (18) is smooth and plugging it into Equations (17) and (16) yields ApproxNDCG. Because NDCG is a utility, we define the loss ℓ to be negative ApproxNDCG.

Since we choose to parameterize our function f with neural networks when optimizing this loss in later sections, it will help to paint a mental picture of what this approximation does. Conceptually, one can replicate the model n times (one per every input $x_i \in \boldsymbol{x}$) to obtain n output nodes. These output nodes together represent $f(\boldsymbol{x})$. What Equation (18) does is that it adds a sigmoid activation layer over pairs of output nodes where the input to the sigmoid is $f(\boldsymbol{x})|_i - f(\boldsymbol{x})|_j$. DCG can then be thought of as a loss over the output of these sigmoid units. This is illustrated in Figure 2a.

5.2.2 Effect of Algorithm 1. Let us now consider the effect of Algorithm 1 on ApproxNDCG. In that setting, instead of $f(\boldsymbol{x})$ the loss is computed using a log-transformed sample $\hat{\boldsymbol{y}}$ from Concrete$(\boldsymbol{\alpha})$:

$$\log \hat{y}_k \triangleq \log \frac{e^{\log \alpha_k + G_k}}{\sum_{j=1}^{n} e^{\log \alpha_j + G_j}}, \quad (19)$$

where $G_k \sim \text{Gumbel}(\mu = 0, \beta = 1) \forall 1 \leq k \leq n$, and $\log(\alpha_k) = f(\boldsymbol{x})|_k$ as before. The sigmoid approximation of the indicator function in Equation (18) then becomes: $\mathbb{I}_{\log \hat{y}_j < \log \hat{y}_i} \approx \sigma(\log \hat{y}_i - \log \hat{y}_j)$. We have that:

$$\log \hat{y}_i - \log \hat{y}_j = \log \frac{e^{\log \alpha_i + G_i}}{\sum e^{\log \alpha_k + G_k}} - \log \frac{e^{\log \alpha_j + G_j}}{\sum e^{\log \alpha_k + G_k}} \quad (20)$$

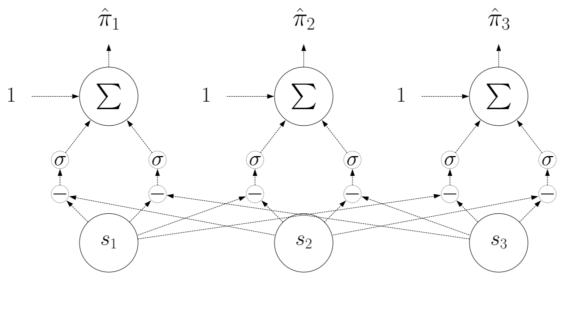
$$= (\log \alpha_i - \log \alpha_j) + (G_i - G_j). \quad (21)$$

In other words, the difference of logs of stochastic scores ($\log \hat{y}_i$'s) is equal to the difference of raw scores ($\log \alpha_i$'s) plus the difference of two independently-drawn Gumbel samples. The sigmoid can thus be rewritten as follows:

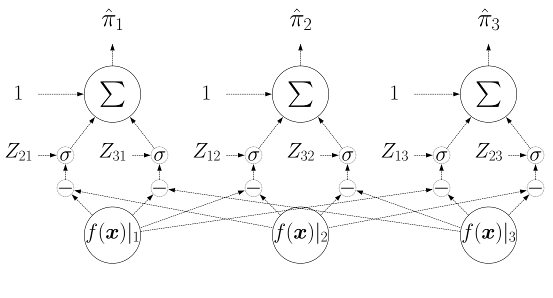
$$\sigma(\log \hat{y}_i - \log \hat{y}_j) = \sigma((f(\boldsymbol{x})|_i - f(\boldsymbol{x})|_j) + Z_{ij}), \quad (22)$$

where $Z_{ij} = G_i - G_j$. Moreover, it can be shown that Z_{ij} is a zero-mean Logistic noise: $Z_{ij} \sim \text{Logistic}(\mu = 0, s = \beta)$.

This finding is encouraging. We have just shown that computing ApproxNDCG (or more generally, any loss derived using the approximation framework of [32]) using stochastic scores is equivalent to using raw scores to compute the loss but where a Logistic noise is injected to the input of the conceptual sigmoid layer placed over pairs of output nodes. This is illustrated in Figure 2b. Deep learning research suggests that such a combination of nonlinearities and stochastic perturbations can improve generalization. Dropout [14, 17, 24], masking noise in denoising auto-encoders [37], semantic hashing [35], and noisy activation functions [15] are a few such examples.



(a)



(b)

Figure 2: Construction of the rank approximation function $\hat{\pi}_f$ for a list with 3 documents. In (a), $s_k = f(\boldsymbol{x})|_k$ in the deterministic case and in the stochastic framework $s_k = \hat{y}_k$ where $\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \hat{y}_3) \sim$ Concrete($e^{f(\boldsymbol{x})|_1}, e^{f(\boldsymbol{x})|_2}, e^{f(\boldsymbol{x})|_3}$). (b) illustrates an alternative view of the stochastic setting which is equivalent to deriving rank approximations from raw scores $f(\boldsymbol{x})$ but where $Z_{ij} \sim$ Logistic($\mu = 0$), $\forall i \neq j$ are injected to the sigmoid activation functions.

5.3 LambdaMART

We also consider LambdaMART [39] in this framework. The objective optimized by LambdaMART is a function of two factors: One measures the absolute change in NDCG by swapping documents i and j in a ranked list denoted by $|\Delta\text{NDCG}_{ij}|$, and another capturing the probability of one document appearing higher than the other in the ranked list given their raw scores. Formally, the objective is a sum over the following pairwise terms [4]:

$$\ell_{ij} = |\Delta\text{NDCG}_{ij}| \log\left(1 + e^{-\sigma(\log\alpha_i - \log\alpha_j)}\right), \tag{23}$$

where σ is a hyperparameter and $f(\boldsymbol{x})|_i = \log\alpha_i$. Instead of working with this loss function, LambdaMART directly models its gradients with respect to raw scores, $\lambda_i = \frac{\partial\ell}{\partial f(\boldsymbol{x})|_i}$.

In the context of Algorithm 1, raw scores are substituted with stochastic scores $\hat{\boldsymbol{y}}$ as in Equation (19). Furthermore, by sorting documents by stochastic scores, we may derive a permutation from which ΔNDCG_{ij} can be calculated. The updated utility will be a sum of the following pairwise terms:

$$\ell_{ij} = \mathop{\mathbb{E}}_{G_k\sim\text{Gumbel}}[|\Delta\text{NDCG}_{ij}| \log\left(1 + e^{-\sigma(\log\hat{y}_i - \log\hat{y}_j)}\right)]. \tag{24}$$

Note that, because stochastic scores $\hat{\boldsymbol{y}}$ are differentiable with respect to raw scores $f(\boldsymbol{x})$, we can update λ_i's (using the chain rule) and use the same gradient boosting algorithm as LambdaMART.

We leave a detailed analysis of LambdaMART under our stochastic regime to future work, but we believe there are two factors that need a closer examination. First, we believe stochastic scores allow the model to explore the gradient space by taking into consideration likely permutations from which ΔNDCG_{ij}'s may be computed. This is in contrast to raw scores which expose the model only to a greedy estimate of the gradient. Second, because the gradients of Equation (24) (i.e., λ_i's) comprise of a sigmoid term as in Equation (22), we expect to observe a similar effect to Section 5.2.

6 EXPERIMENTAL SETUP

The theoretical properties demonstrated in earlier sections are encouraging but whether they translate into empirical gains remains to be tested. In the remainder of this paper we set out to do just that. We start by a description of our experimental setup, including details of model hyperparameters. To invite the research community to reproduce our experiments and to encourage further research in this direction, we open source our code.

6.1 Datasets

We conduct our experiments on two publicly available datasets: MSLR Web30K [31] (Fold 1) and Yahoo! Learning to Rank Challenge [8] (Set 1). Both datasets contain roughly 30,000 queries. Web30K (Yahoo!) has on average 120 (24) documents per query, where query-document pairs are represented by 136 (519) numeric features. Documents are labeled with graded relevance from 0 to 4 with larger labels indicating a higher relevance.

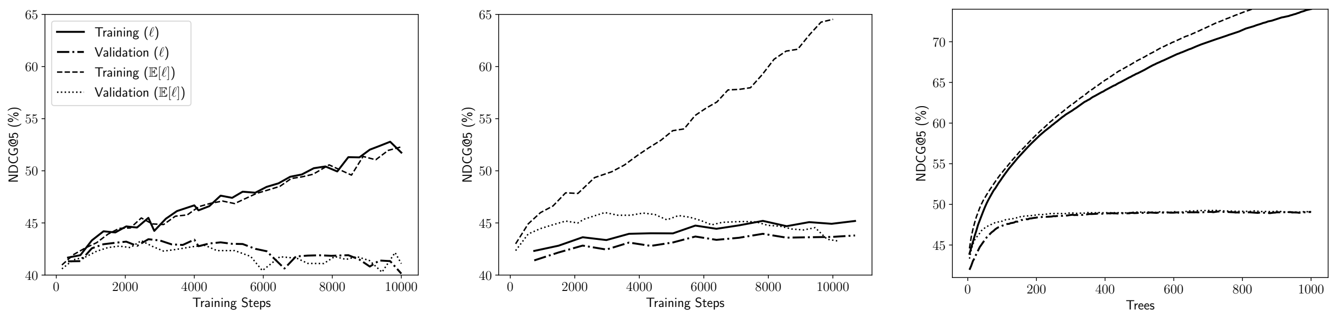
Both datasets contain a training, a validation, and a test set. We conduct hyperparameter tuning strictly on the validation set. Final model evaluation is performed on the held-out test set. In all of our experiments throughout this paper, we run 10 trials of each experiment and report mean metrics and 95% confidence intervals.

Finally, we discard queries with no relevant documents during evaluation. There are 189 (248) such queries out of 6,306 (6,983) queries in the Web30K (Yahoo!) test set. The reason for ignoring these queries is that ranking metrics can be arbitrarily 0 or 1 for such queries. That arbitrary choice makes a comparison between different implementations unfair. The astute reader will notice that, because of this clean-up, the measurements reported in this work may be lower than what is reported in the literature.

6.2 Models

We conduct experiments using three baseline models and compare their performance in the context of Algorithm 1. These include: LambdaMART [39], ApproxNDCG [32], and ListNet [7] henceforth denoted by CrossEntropy. We note that we also compared the above with other existing methods such as ListMLE [40] and RankNet [5] but found these to be relatively weak and, as such, exclude them from this report. For brevity, we use the name of a loss function to refer to a model that optimizes that loss.

We train LambdaMART models (denoted by λMART) using LightGBM [22]. The hyperparameters are guided by previous work



(a) **(b)** **(c)**

Figure 3: NDCG@5 on Web30K during training of a model optimizing a baseline loss ℓ and another (same hyperparameters) optimizing our proposed expected loss $\mathbb{E}[\ell]$ where ℓ is CrossEntropy in (a), ApproxNDCG in (b), and λMART in (c).

(e.g., [22, 38]) and are further fine-tuned on the validation set. For Web30K (Yahoo!): learning rate is 0.05 (0.05), num_leaves is 400 (400), min_data_in_leaf is 50 (100), and min_sum_hessian_in_leaf is set to 200 (10), σ of Equation (23) is 2, and lambdamart_norm is set to false. We use NDCG@5 to select the best models on validation sets by fixing early stopping round to 200 up to 1000 trees.

We use Tensorflow Ranking [30] (v0.1.3 with Tensorflow v1.14.0-rc1) to train models with ApproxNDCG and CrossEntropy. The model configurations for these two are similar with only the loss functions being different. The model itself is a fully-connected feed-forward network with ReLU activation ($\text{ReLU}(t) = \max(t, 0)$). The network has 7 hidden layers with 1024, 512, 256, 128, 64, 32, and 16 nodes each. Batch Normalization [18] (with momentum set to 0.99 for Yahoo! and 0.8 for Web30K) is applied between consecutive layers, including over the input layer. Similar to λMART models, the hyperparameters are selected based on NDCG@5 on the validation set; training batch size is set to 128; and learning rate to 0.005. Lastly, we use AdaGrad [11] to optimize the objective. We found empirically that setting the loss of the mini-batch to be the sum (rather than mean) of sample losses speeds up AdaGrad's convergence in this setup—this amounts to adjusting the effective learning rate. For ApproxNDCG, we set the hyperparameter η to 10 in Equation (18).

Finally, we apply Algorithm 1 to the losses above and denote them with $\mathbb{E}$[CrossEntropy], $\mathbb{E}$[ApproxNDCG], and $\mathbb{E}[\lambda$MART]. To that end, we implemented Algorithm 1 in Tensorflow Ranking and LightGBM. We set the number of samples to 8 and Gumbel shape β to 1 and 0.25 for the neural networks and decision tree-based models, based on a grid search on validation sets.

In a first set of experiments, we use the same baseline hyperparameters to train $\mathbb{E}[\cdot]$ models so as to strictly measure the impact of substituting deterministic ordering with stochastic scores.

In a second set of experiments, we further fine-tune $\mathbb{E}[\cdot]$ hyperparameters on validation sets. This leads to a few adjustments as follows: The refined $\mathbb{E}$[CrossEntropy] and $\mathbb{E}$[ApproxNDCG] models consist of 3 hidden layers, each with 1024, 512, and 256 nodes respectively. We set Batch Normalization's momentum to 0.4 and add dropout [17] between layers with a keep rate of 0.5. We use Adam [23] to minimize the losses, where the loss of a mini-batch is the mean of the sample losses. The hyperparameters for the fine-tuned $\mathbb{E}[\lambda$MART] model do not change.

7 EXPERIMENTAL RESULTS

In this section, we present our experiments and discuss the results. In particular, we compare the convergence of the baseline models with our proposed method during training and draw insights from the observed behavior. We then examine the generalizability of the proposed method by evaluating a tuned model on held-out test data and comparing the results with similarly tuned baseline models. Through both sets of experiments, we also demonstrate that the intuition and analysis in Section 5 hold empirically.

7.1 Convergence

We have hypothesized that the stochastic nature of our proposed method leads to a more robust model when applied to an appropriate loss function such as ApproxNDCG. We have also speculated that the same strategy does not lead to any significant change when the loss function under consideration is cross entropy-like. In a first set of experiments, we examine those claims empirically.

We train our baseline models where one optimizes the cross entropy loss, another ApproxNDCG, and a third λMART. We then fix all hyperparameters but substitute the loss with the expectation of each loss using Algorithm 1 and train new models. In this way, we study the effect of our method in isolation.

Figure 3a plots the performance of CrossEntropy and the proposed $\mathbb{E}$[CrossEntropy] models as measured by NDCG@5 on the Web30K training and validation sets as training progresses. Figures 3b and 3c illustrate a similar setup for ApproxNDCG and λMART and their $\mathbb{E}[\cdot]$ variants. In all plots, the horizontal axis represents the training iteration (steps for neural networks and trees for λMART). Each curve represents a single trial but that regardless of which trial we choose to plot, the comparison between the models holds. We further note that we observe a similar trend on the Yahoo! dataset which we have omitted due to space constraints.

As shown in Figure 3a, it is clear that CrossEntropy does not benefit from Algorithm 1. This observation validates our intuitive understanding of when Algorithm 1 may prove unhelpful: If the loss function already encourages a wide margin between scores, the application of our proposed method will have no effect. This observation also confirms the analysis in Section 5.1.

Figure 3b paints a different picture. The performance gap between ApproxNDCG and $\mathbb{E}$[ApproxNDCG] is significant and grows

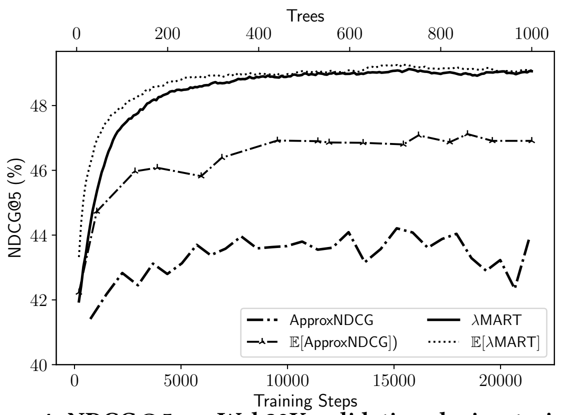


Figure 4: NDCG@5 on Web30K validation during training. Each model is configured with its own fine-tuned hyperparameters, a different setting from Figure 3 where hyperparameters were fixed across baseline and $\mathbb{E}[\cdot]$ models.

wider as training progresses. The $\mathbb{E}$[ApproxNDCG] model, however, shows signs of overfitting earlier because it is trained with the baseline model's hyperparameters; further fine-tuning, as discussed below, addresses the early overfitting problem. We also observe that the application of Algorithm 1 to ApproxNDCG results in a faster convergence to a better optimum—as we will show shortly, training the ApproxNDCG model for more steps does not yield an NDCG better than the largest NDCG achieved by $\mathbb{E}$[ApproxNDCG].

Training λMART with stochastic scores leads to a higher convergence rate as depicted in Figure 3c. This comes at no cost to effectiveness: $\mathbb{E}[\lambda$MART] consistently outperforms λMART during training. We believe this outcome is due to exposing the model to a larger number of permutations. This is similar to the effect of other noisy methods applied to λMART, such as bagging, but we leave a detailed analysis of this topic to future work.

We have just established that our proposed method when applied to ApproxNDCG and λMART and in isolation of other factors leads to a higher convergence rate without sacrificing effectiveness. In the next set of experiments, we train new $\mathbb{E}[\cdot]$ models with fine-tuned hyperparameters. Figure 4 illustrates NDCG@5 on the Web30K validation set as training progresses. We observe that in addition to an increased convergence rate, the gap between ApproxNDCG and $\mathbb{E}$[ApproxNDCG] widens substantially. Note that because the $\mathbb{E}[\lambda$MART] hyperparameters did not change after fine-tuning (see Section 6), the curves in this figure are the same as Figure 3c but we illustrate them again to allow comparison with other models.

7.2 Generalization

We now turn to a second set of experiments where we compare the various models under consideration on held-out test sets. We report the results of our experiments on both the Web30K and Yahoo! datasets in Table 1.

A comparison of the ApproxNDCG model with $\mathbb{E}$[ApproxNDCG] again confirms the analysis from earlier sections. The latter consistently improves upon the former significantly on both datasets and by a wide margin on Web30K.

We did not find the results from CrossEntropy insightful as it performs poorly overall. However, as expected, a comparison between CrossEntropy and $\mathbb{E}$[CrossEntropy] shows no statistically significant performance difference. We have excluded those from the Yahoo! result set.

Finally, a comparison between the λMART models is in alignment with Section 7.1. On both datasets, $\mathbb{E}[\lambda$MART] performs better than λMART across rank cutoffs. This difference becomes more pronounced if one must use fewer number of trees.

Table 1: Evaluation on test sets in mean NDCG (95% confidence intervals). Bold-faced entries are best column-wise.

	Web30K		
Model	NDCG@1	NDCG@5	NDCG@10
CrossEntropy	42.66 (±0.36)	43.06 (±0.16)	45.44 (±0.16)
$\mathbb{E}$[CrossEntropy]	42.98 (±0.75)	43.28 (±0.65)	45.63 (±0.62)
ApproxNDCG	46.64 (±0.22)	45.38 (±0.11)	47.31 (±0.10)
$\mathbb{E}$[ApproxNDCG]	48.81 (±0.25)	47.48 (±0.10)	49.33 (±0.10)
λMART	50.06 (±0.27)	49.39 (±0.08)	51.26 (±0.04)
$\mathbb{E}[\lambda$MART]	**50.22** (±0.18)	**49.44** (±0.06)	**51.40** (±0.04)
	Yahoo!		
ApproxNDCG	69.63 (±0.17)	72.32 (±0.10)	76.77 (±0.06)
$\mathbb{E}$[ApproxNDCG]	70.10 (±0.12)	72.69 (±0.13)	77.11 (±0.09)
λMART]	72.47 (±0.15)	74.97 (±0.09)	79.16 (±0.06)
$\mathbb{E}[\lambda$MART]	**72.58** (±0.09)	**75.16** (±0.04)	**79.34** (±0.05)

8 CONCLUDING REMARKS

In this work, we examined how learning-to-rank loss functions utilize the scores produced by a scoring function during training. We asked whether a deterministic mapping from scores to permutations could prove suboptimal for losses derived from ranking metrics. As an alternative, we introduced a stochastic framework where a list of scores defines a distribution over permutations and from which permutations may be sampled. We then found through analysis and experiments on benchmark datasets that our framework improves model convergence and generalization when applied to an appropriate class of ranking losses.

Given our initial findings, we believe there are a number of promising extensions to this work. One direct application of our proposed method is in connection with the LambdaLoss framework [38], as noted in Section 2.

We also observed a connection between our proposed method and regularization as discussed in Section 5.2. More generally, our approach can be understood as perturbing scores by Gumbel noise. This finding is exciting and we are therefore interested in studying other distributions—sampling from the Concrete distribution as done in this work is simply one way to do so—that result in a different and more appropriate noise distribution. Particularly, we are interested in an examination of stochastic learning-to-rank functions in an efficiency-effectiveness trade-off framework.

9 ACKNOWLEDGEMENTS

This work would not be possible without the support provided by the TF-Ranking team. We thank Donald Metzler for his input on an early draft of this work, and the anonymous reviewers for their feedback.

REFERENCES

[1] Qingyao Ai, Xuanhui Wang, Sebastian Bruch, Nadav Golbandi, Mike Bendersky, and Marc Najork. 2019. Learning Groupwise Multivariate Scoring Functions Using Deep Neural Networks. In *Proc. of the 5th ACM SIGIR International Conference on the Theory of Information Retrieval*. 85–92.

[2] Sebastian Bruch, Xuanhui Wang, Mike Bendersky, and Marc Najork. 2019. An Analysis of the Softmax Cross Entropy Loss for Learning-to-Rank with Binary Relevance. In *Proc. of the 2019 ACM SIGIR International Conference on the Theory of Information Retrieval*.

[3] Sebastian Bruch, Masrour Zoghi, Mike Bendersky, and Marc Najork. 2019. Revisiting Approximate Metric Optimization in the Age of Deep Neural Networks. In *Proc. of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[4] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proc. of the 22nd International Conference on Machine Learning*. 89–96.

[5] Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report Technical Report MSR-TR-2010-82. Microsoft Research.

[6] Christopher J. C. Burges, Robert Ragno, and Quoc Viet Le. 2006. Learning to Rank with Nonsmooth Cost Functions. In *Advances in Neural Information Processing Systems*. 193–200.

[7] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proc. of the 24th International Conference on Machine Learning*. 129–136.

[8] Olivier Chapelle and Yi Chang. 2011. Yahoo! learning to rank challenge overview. In *Proc. of the Learning to Rank Challenge*. 1–24.

[9] Olivier Chapelle, Donald Metzler, Ya Zhang, and Pierre Grinspan. 2009. Expected Reciprocal Rank for Graded Relevance. In *Proc. of the 18th ACM Conference on Information and Knowledge Management*. 621–630.

[10] Mostafa Dehghani, Hamed Zamani, Aliaksei Severyn, Jaap Kamps, and W. Bruce Croft. 2017. Neural Ranking Models with Weak Supervision. In *Proc. of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 65–74.

[11] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159.

[12] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* 29, 5 (2001), 1189–1232.

[13] Fredric C. Gey. 1994. Inferring Probability of Relevance Using the Method of Logistic Regression. In *Proc. of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 222–231.

[14] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. 2013. Maxout Networks. In *Proc. of the 30th International Conference on Machine Learning*. 1319–1327.

[15] Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. 2016. Noisy Activation Functions. In *Proc. of the 33rd International Conference on Machine Learning*. 3059–3068.

[16] Andrey Gulin, Igor Kuralenok, and Dmitry Pavlov. 2010. Winning the Transfer Learning Track of Yahoo!'s Learning to Rank Challenge with YetiRank. In *Proc. of the 2010 Yahoo! Learning to Rank Challenge*. 63–76.

[17] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. (2012). arXiv:1207.0580

[18] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. of the 32nd International Conference on Machine Learning*. 448–456.

[19] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems* 20, 4 (2002), 422–446.

[20] Thorsten Joachims. 2006. Training linear SVMs in linear time. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 217–226.

[21] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. 2017. Unbiased Learning-to-Rank with Biased Feedback. In *Proc. of the 10th ACM International Conference on Web Search and Data Mining*. 781–789.

[22] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*. 3146–3154.

[23] Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90.

[25] Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

[26] R. Duncan Luce. 1959. *Individual Choice Behavior: A Theoretical Analysis*. New York: Wiley.

[27] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *International Conference on Learning Representations*.

[28] Chris J Maddison, Daniel Tarlow, and Tom Minka. 2014. A* Sampling. In *Advances in Neural Information Processing Systems*. 3086–3094.

[29] Donald A Metzler, W Bruce Croft, and Andrew Mccallum. 2005. *Direct maximization of rank-based metrics for information retrieval*. CIIR report 429. University of Massachusetts.

[30] Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *Proc. of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2970–2978.

[31] Tao Qin and Tie-Yan Liu. 2013. Introducing LETOR 4.0 Datasets. (2013). arXiv:1306.2597

[32] Tao Qin, Tie-Yan Liu, and Hang Li. 2010. A general approximation framework for direct optimization of information retrieval measures. *Information Retrieval* 13, 4 (2010), 375–397.

[33] Tao Qin, Tie-Yan Liu, Xu-Dong Zhang, De-Sheng Wang, and Hang Li. 2008. Global ranking using continuous conditional random fields. In *Advances in Neural Information Processing Systems*. 1281–1288.

[34] Christian P. Robert and George Casella. 2005. *Monte Carlo Statistical Methods*. Springer-Verlag.

[35] Ruslan Salakhutdinov and Geoffrey Hinton. 2009. Semantic Hashing. *Int. J. Approx. Reasoning* 50, 7 (July 2009), 969–978.

[36] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. 2008. SoftRank: Optimizing Non-smooth Rank Metrics. In *Proc. of the 1st International Conference on Web Search and Data Mining*. 77–86.

[37] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proc. of the 25th International Conference on Machine Learning*. 1096–1103.

[38] Xuanhui Wang, Cheng Li, Nadav Golbandi, Michael Bendersky, and Marc Najork. 2018. The LambdaLoss Framework for Ranking Metric Optimization. In *Proc. of the 27th ACM International Conference on Information and Knowledge Management*. 1313–1322.

[39] Qiang Wu, Christopher JC Burges, Krysta M Svore, and Jianfeng Gao. 2010. Adapting boosting for information retrieval measures. *Information Retrieval* 13, 3 (2010), 254–270.

[40] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise approach to learning to rank: theory and algorithm. In *Proc. of the 25th International Conference on Machine Learning*. 1192–1199.

[41] Jun Xu and Hang Li. 2007. AdaRank: A Boosting Algorithm for Information Retrieval. In *Proc. of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 391–398.