Combining Decision Trees and Neural Networks for Learning-to-Rank in Personal Search

Pan Li[*]
UIUC
panli2@illinois.edu

Zhen Qin
Google Inc.
zhenqin@google.com

Xuanhui Wang
Google Inc.
xuanhui@google.com

Donald Metzler
Google Inc.
metzler@google.com

ABSTRACT

Decision Trees (DTs) like LambdaMART have been one of the most effective types of learning-to-rank algorithms in the past decade. They typically work well with hand-crafted dense features (e.g., BM25 scores). Recently, Neural Networks (NNs) have shown impressive results in leveraging sparse and complex features (e.g., query and document keywords) directly when a large amount of training data is available. While there is a large body of work on how to use NNs for semantic matching between queries and documents, relatively less work has been conducted to compare NNs with DTs for general learning-to-rank tasks, where dense features are also available and DTs can achieve state-of-the-art performance. In this paper, we study how to combine DTs and NNs to effectively bring the benefits from both sides in the learning-to-rank setting. Specifically, we focus our study on personal search where clicks are used as the primary labels with unbiased learning-to-rank algorithms and a significantly large amount of training data is easily available. Our combination methods are based on ensemble learning. We design 12 variants and compare them based on two aspects, ranking effectiveness and ease-of-deployment, using two of the largest personal search services: Gmail search and Google Drive search. We show that direct application of existing ensemble methods can not achieve both aspects. We thus design a novel method that uses NNs to compensate DTs via boosting. We show that such a method is not only easier to deploy, but also gives comparable or better ranking accuracy.

KEYWORDS

learning to rank, personal search, decision trees, neural networks

ACM Reference Format:
Pan Li, Zhen Qin, Xuanhui Wang, and Donald Metzler. 2019. Combining Decision Trees and Neural Networks for Learning-to-Rank in Personal Search. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3292500.3330676

[*]This work was done while Pan Li was an intern in Google.

1 INTRODUCTION

How to design effective ranking functions is an important research topic in Information Retrieval (IR). While traditional IR models such as BM25 [12] and language models [36] estimate relevance between queries and documents based on their textual features, many other features such as PageRank scores of web pages [34], recency of emails [30], and historical user interactions [1] have been shown to be useful in ranking functions. Learning-to-rank techniques, which were introduced more than a decade ago, typically combine all these dense features into a single ranking function with the objective of optimizing retrieval accuracy based on labeled training data [28]. In the traditional learning-to-rank setting, training data is human labeled and usually has tens of thousands of queries in total [29]. Each document associated with a given query has hundreds of hand-crafted dense features. Based on this data, many different machine learning algorithms, ranging from Support Vector Machines (SVMs) [21] to Neural Networks (NNs) [5] to Decision Trees (DTs) [26], have been studied in the past. Among them, DT-based models (more precisely Gradient-Boosted Decision Trees) [6] have become the most competitive ones. For example, they took first place in the Yahoo! Learning-to-Rank Challenges [9] and are used in commercial search engines [43].

The DT-based models are empirically successful for several reasons. (1) The ranking functions learned by DTs can be highly nonlinear, which can fit the complex query-document relationship better than simple linear models. (2) DTs are good at handling dense numerical features, and are less sensitive to the dynamic ranges and uneven distributions of features [41]. (3) The characteristic of splitting data into different branches in DTs is a natural fit of non-smooth ranking metrics such as Normalized Discounted Cumulative Gain (NDCG) [20]. This is reflected in the representative algorithms LambdaMART [41] and RandomForest-Hybrid [19].

In the traditional learning-to-rank setting with tens of thousands of queries, NN-based models such as RankNet [5] and LambdaRank [8] are less effective than DT-based models. However, in recent years, there has been a resurgence of interest in NNs given the availability of large-scale training data that have millions of queries and significantly more computational resources [24]. One appealing property of NNs for search ranking is that they can now effectively learn from highly sparse and complex features such as query keywords and document sentences. For example, NNs have been successfully used to extract semantic representations from query and document text, which can be further used for relevance computation [18, 32]. Such representations achieve semantic matching by bridging the vocabulary gap between queries and documents based on a large scale data set. As opposed to NNs, DTs are best suited to handle dense features. They are much less efficient at learning from raw texts directly as they do not scale to the very large datasets that are necessary to extract meaningful semantic representations [37].

Despite the promise, relatively little work has compared NNs with DTs in the learning-to-rank setting where both dense and sparse features are present. The first reason is lack of large-scale learning-to-rank training data. Existing work [13] tried to bypass this problem using weak supervision to increase the training data size. However, such data is not fully validated for learning-to-rank tasks. The second reason is the deployment complexity. DTs and NNs have different internal structures and thus have fundamentally different properties. For example, NNs can be easily sped up thousands of times via parallelization using specialized hardware such as graphics processing units (GPUs). However DTs can be hardly sped up by tens of times on GPUs even with sophisticated parallelization strategies [25]. Thus, the two models would most likely be hosted by separate services with different types of devices in reality. A deeply coupled model is not friendly for practical deployment since separate services may be updated independently. Hence a loosely coupled model is more desirable.

In this paper, we study how to combine DTs and NNs in the learning-to-rank setting from two aspects: ranking effectiveness and ease-of-deployment. To fully exploit the ranking power of both DTs and NNs, we conduct our study in the setting where both *sparse* and *dense* features are present and where large-scale labeled data is available. Although our idea could naturally carry over to a broad range of learning-to-rank scenarios as long as these conditions are satisfied, in this work, we choose personal search to evaluate our idea. In personal search, clicks are used as the primary training data and it is easy to collect millions of queries that is much larger than the traditional LETOR data sets [29]. Though clicks are known to be biased, recent progress in unbiased learning-to-rank [22, 39] addresses this concern. In the past few years, both DTs and NNs models have been studied for personal search ranking [3, 40, 44]. Our work is a continuation of this line on how to effectively combine DTs and NNs in a deployment friendly way.

Our combination methods are based on *ensemble learning* techniques [33]. We design 12 different variants and study them from the ranking accuracy and deployment aspects. We find that a simple adaption of ensemble learning methods such as linear combination or standard stacking cannot achieve good results in both aspects at the same time. We thus propose a novel boosted-stacking approach where NNs relay the boosting from DTs. In this way, thanks to their power in capturing complex correlations, NNs are able to compensate DTs for difficult queries that DTs cannot work well. Meanwhile, NNs are decoupled from DTs to a large extent. Furthermore, to reduce the dependency of DTs and NNs and make our proposed model more friendly for deployment, we propose another structure by adding a lightweight *adapter* model that can further decouple DTs and NNs. Such an adapter allows us to update DTs and NNs independently.

We conduct our experiments using click data from two of the world-wide large personal search engines — Gmail search and Google Drive search. Our results show that it is highly beneficial to combine DTs and NNs, as significantly better ranking accuracy can be achieved when comparing DT-only and NN-only models. Our proposed boosted-stacking methods not only provide loosely-coupled structures that are easy to deploy, but also achieve comparable or even better ranking accuracy than the standard stacking methods. Comparing across different variants, our experiments also demonstrate that using the output scores of DTs achieves much better results than using ranks and may provide guidance in building a commercial search engine using both DTs and NNs.

The rest of this paper is organized as follows. In Section 2, we review some related works and introduce basic concepts of ensemble learning. In Section 3, we describe the methods of combining DTs and NNs and also analyze their pros and cons in terms of model deployment. Section 4 presents the extensive experimental results of the proposed models in terms of their ranking accuracy. We finally conclude our work in Section 5.

2 RELATED WORK

Our work is mostly related to ensemble learning, which combines multiple models to improve the overall learning performance [33]. The techniques can be broadly classified into bagging, boosting, and stacking. Such techniques are widely used in learning-to-rank tasks, notably in the well-known "Yahoo! Learning to Rank Challenge" [9]. We review them in this section.

Bagging, which stands for Bootstrap Aggregating, allows one to train base models independently via boostrapping and further aggregates the models by averaging their outputs. The most widely-used bagging approach is RandomForest, of which each base model corresponds to a single DT [4]. Recent work shows that RandomForest-based learning-to-rank algorithms may achieve competitive performance while these approaches typically require many more iterations (trees) than models based on the boosting strategy [19].

Boosting works in a more sophisticated way than bagging, which requires the base models to be trained sequentially and expects the models added in later iterations to complement the ones previously added. Boosting algorithms typically focus on how to adjust the weights of different instances so as to properly emphasize the ones that former learners fail to produce good predictions for. A vast number of learning-to-rank approaches fall into this category [15, 16, 26, 42]. Among them, LambdaMART [41], based on additive gradient boosting trees, is relatively robust and demonstrates competitive performance over multiple benchmark datasets. All of these belong to the DT family of methods.

Note that most bagging- and boosting-based ensembles consist of multiple base models that are of the same type. Such ensembles are called *homogeneous* ensembles. In our case, the ensemble integrates two different types of base models—DTs and NNs. These types of ensembles are often called *heterogeneous* ensembles.

Stacking, similar to boosting, also imposes some order to train models. In contrast to boosting, stacking allows the latter learners to use the output of former learners as their input features. The winner of the Yahoo Ranking Challenge [6] conducted an empirical study of stacking a neural network (LambdaRank [8]) over the outputs of a collection of independently trained models (MART [16], LambdaMART [41] and LambdaRank [8]). However, the results did not show clear advantage over the simple average.

Recently Yin et al. [43] and Ai et al. [2] investigated a ranking - re-ranking procedure that essentially stacks two base ranking models. Different from our setting, the ranking - re-ranking scheme contains a default order of two base models so both of these approaches use standard stacking techniques.

Another recent study also considered combining NNs and DTs [27] while the target was to improve ads CTR instead of ranking performance in search. Besides the difference in set-ups, the combining strategies based on NNs-boosting-DTs were missed, which are demonstrated to be optimal according to our study. Some related evidence and more subtle difference are discussed in Section 4.2.

3 COMBINATION METHODS

In this section, we discuss different combination methods based on ensemble learning. Our ensemble model consists of DTs and a single NN. As these two base models are heterogeneous, our combination methods can be broadly thought of as stacking while we may leverage some idea of boosting to achieve the goal of easier model deployment. Note that the methodology discussed in this section is applicable to generic learning-to-rank tasks. We first introduce some preliminaries for learning-to-rank problems and then describe our proposed methods.

3.1 Preliminaries

Suppose we have a set of queries $Q = \{q_i\}_{i=1}^{|Q|}$. Each query q_i is associated with a set of documents $\mathbf{d}_i = \{d_{i,1}, d_{i,2}, ..., d_{i,n_i}\}$ where n_i is the total number of documents corresponding to q_i. Let $x_{i,k}$ denote the training features for the query-document pair $\langle q_i, d_{i,k}\rangle$. Each query-document pair $\langle q_i, d_{i,k}\rangle$ is associated with a label $y_{i,k}$ that denotes the level of relevance of such a pair. Typically, for click-through data, $y_{i,k}$ ($k \in [n_i]$) is a binary indicator of whether the users has clicked document $d_{i,k}$. The learning-to-rank problem is to learn a function f that takes $x_{i,k}$ as the input and outputs a relevance score between q_i and $d_{i,k}$ to optimize a predefined ranking objective

$$\max_f \frac{1}{|Q|} \sum_{i=1}^{|Q|} m\left(\{y_{i,k}, f(x_{i,k})\}_{k=1}^{n_i}\right) \tag{1}$$

where the metric $m(\cdot)$ is an evaluation metric like NDCG [11]. Directly optimizing the above objective is difficult because the metric $m(\cdot)$ is not smooth. Hence, typical learning-to-rank algorithms use smooth surrogate objectives such as logistic or hinge loss [28] that are easier to optimize. Doing so converts (1) into the following minimization problem.

$$\min_f \bar{\ell} :\triangleq \frac{1}{|Q|} \sum_{i=1}^{|Q|} \ell\left(\{y_{i,k}, f(x_{i,k})\}_{k=1}^{n_i}\right) \tag{2}$$

where $\ell(\cdot)$ is the loss function that can be pointwise, pairwise or listwise [28].

In our case, we define f in (1) as an ensemble function $f(g_1, g_2, x_{i,k})$ that takes DT-based model $g_1(\cdot)$, NN-based model $g_2(\cdot)$, and features $x_{i,k}$ as input and outputs the ensemble score. In the following, we describe how to define f based on ensemble learning techniques. For easy reference, our learning methods are summarized in Table 1.

Table 1: Training Strategies for the combinations of DTs and the NN; $h(\cdot)$ refers to the mappings in (8) and (11).

Models	$h(\cdot)$	Training Procedure
LIN	–	1) Train the NN and DTs independently; 2) Combine the outputs optimally linearly.
T-S-N	–	1) Train DTs first; 2) Train the NN with outputs of DTs as inputs.
N-S-T	–	1) Train the NN first; 2) Train DTs with outputs of NN as inputs.
N-B-T	–	1) Train the NN first; 2) Use DTs to boost the NN.
T-B-N	lin	1) Train DTs first;
	pow	2) Use the NN to boost DTs after mapping (8).
	sig	
T-B-NT	lin	1) Train DTs first;
	pow	2) Use the NN to boost DTs after the mapping (8).
	sig	3) Train an adapter h' to transform outputs of the NN.
R-B-N	lin	1) Train DTs first and view DTs as a ranker
	jum	2) Use the NN to boost DTs after the mapping (11).

3.2 Linear Combination

The most straightforward method is the commonly-used linear combination. In this method, both DTs and the NN are trained independently first to obtain $g_1(x_{i,k})$ and $g_2(x_{i,k})$ respectively. Then we use the approach proposed in [41] to obtain an optimally linear model over DTs and the NN. We use LIN to denote this method. Specifically, the final scoring function f becomes

$$\textbf{LIN}: \quad f(g_1, g_2, x_{i,k}) = \alpha \cdot g_1(x_{i,k}) + (1-\alpha) \cdot g_2(x_{i,k}) \tag{3}$$

where α is determined by optimally searching in the interval $[0, 1]$ for a given ranking metric.

LIN can be thought as the simplest ensemble model. It is the most efficient in both training and inference. Also, it allows us to find the α that is optimal with respect to a ranking metric without introducing any surrogate loss functions. While it is possible to use other methods to combine two scores $g_1(x_{i,k})$ and $g_2(x_{i,k})$, we found that LIN can in general give optimal results while maintaining simplicity. What's more, the DTs and the NN can be updated independently. What we need to do is to retrain the parameter α any time the DTs or NN are updated.

3.3 Standard Stacking

Following the idea of standard stacking, we stack one model over the other. Since we have two models, we have the following two options: (1) train DTs from scratch and then stack its output into the input of the NN, denoted as T-S-N; (2) train the NN from scratch and then stack its output into the input of DTs, denoted as N-S-T.

$$\textbf{T-S-N}: \quad f(g_1, g_2, x_{i,k}) = g_2(g_1(x_{i,k}), x_{i,k}) \tag{4}$$

$$\textbf{N-S-T}: \quad f(g_1, g_2, x_{i,k}) = g_1(g_2(x_{i,k}), x_{i,k}) \tag{5}$$

Since both DTs and the NN have more complex structures than linear models, function g_1 and g_2 are deeply coupled. Both models are expected to perform better than LIN in terms of ranking metrics, but the coupled structures reduce the ease-of-deployment. For example, in N-S-T, we need to update the DTs whenever the NN is updated. Such a method is difficult to deploy when the DTs and the NN are hosted on two different services that are difficult to update at the same time.

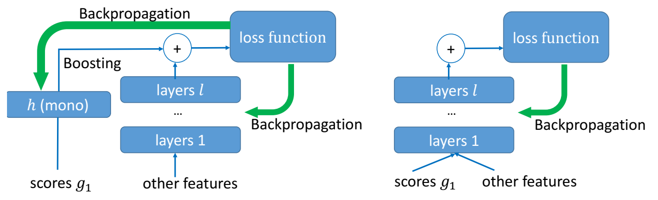

Figure 1: Training of the NNs: T-B-N (left) vs T-S-N (right)

3.4 Boosted Stacking

To reduce the degree of coupling in standard stacking, we design our combination methods by borrowing from boosting techniques.

3.4.1 The NN boosted by DTs: In this method, the NN is first trained from scratch to obtain $g_2(x_{i,k})$. Then instead of starting from a null base function, DTs start from $g_2(x_{i,k})$ for the boosting procedure. For example, LambdaMART [7] can be used to obtain function g_1. We use N-B-T to denote this method. The final prediction in this case can be obtained simply as

$$\textbf{N-B-T}: \quad f(g_1, g_2, x_{i,k}) = g_2(x_{i,k}) + g_1(x_{i,k}). \tag{6}$$

3.4.2 DTs boosted by the NN:. In this method, DTs are trained first, then we use the NN to relay the boosting from DTs. The intuition is to let the NN compensate where DTs work poorly. We use T-B-N to denote this method.

Traditional boosting is usually an iterative procedure. In each iteration, an additive model is obtained by a weak learner. For example, RankBoost [15] and AdaRank [42] use sample reweighting that gives higher weights for poorly predicted examples by previous models. GBDT [16] lets the next learner to approximate the functional gradient of the current objective and essentially makes the boosting procedure work like gradient descent in functional spaces. These paradigms are guaranteed to asymptotically achieve the optimal learning performance with a large number of learners [31]. However, in our case, we only have a single step boosting with a strong learner, the NN. So we adopt a strategy other than sample re-weighting.

Similar to (6) in the N-B-T case, we expect to have a simple combination of the outputs of DTs and NNs in T-B-N as the final output. However, since NNs are more sensitive to the scaling of their input features, it is better to introduce a mapping function that can rescale the outputs of DTs when using NNs for boosting DTs. Note that, to keep the ranking order output by DTs unchanged, the rescaling mapping should be monotonously increasing. Specifically, we do as the follows. During the training of NNs, we first compute a mapping of the outputs of DTs, $h(g_1(x_{i,k}))$, where $h(\cdot)$ is monotonously increasing. Then, we add $h(g_1(x_{i,k}))$ to the output score of NNs, i.e., $g_2(x_{i,k})$, and then feed the obtained sum into the loss function. Mathematically, the loss to train NNs follows the substitution:

$$\ell\left(\left\{y_{i,k}, g_2(x_{i,k})\right\}_{k=1}^{n_i}\right) \to \ell\left(\left\{y_{i,k}, h(g_1(x_{i,k})) + g_2(x_{i,k})\right\}_{k=1}^{n_i}\right)$$

This manner imposed on the back-propagation during training the NN helps to optimize both parameters in h and those in the network itself, respectively. The final inference of T-B-N can be obtained via

$$\textbf{T-B-N}: \quad f(g_1, g_2, x_{i,k}) = h(g_1(x_{i,k})) + g_2(x_{i,k}), \tag{7}$$

Note that the main difference between boosted-stacking T-B-N and standard stacking T-S-N, as shown in Figure 1, is that in T-B-N g_1 is only combined with the NN after the final layer. T-B-N lets g_1 and g_2 be inherently decoupled. When this model is deployed, $h(g_1)$ and g_2 can make inference in parallel via different services. In the T-S-N method, because of the highly-nonlinear rectifiers of NNs, it is almost impossible to decouple g_1 from the rest of the features.

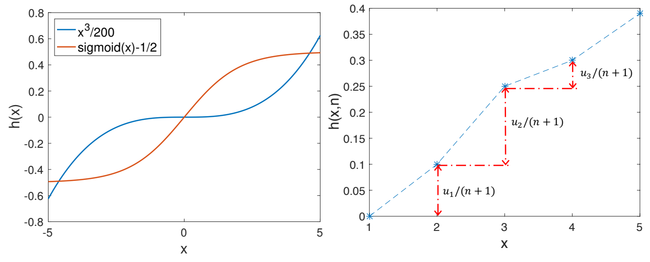

Figure 2: Monotone mapping h. Left: equation (8): odd power and sigmoid; Right: equation (11): jumping

For the h function in (7), we consider the following three types:

$$h(g_1) = \begin{cases} w_1 * g_1, & \text{linearity (lin)} \\ w_2 * g_1 + w_3 * (g_1)^3, & \text{odd power (pow)} \\ w_4 * g_1 + w_5 * \text{sigmoid}(w_6 * g_1 + b), & \text{sigmoid (sig)} \end{cases} \tag{8}$$

where $w_j \geq 0\,(1 \leq j \leq 6)$ and b are variables that are learned during the training of the NN. Note that h is fixed to be monotonically increasing to retain the rank-ordering the same as the relevance scores g_1. The motivation to choose *odd power* as an option comes from the Taylor series that can well-approximate any analytic functions in a compact space [23]. We also choose *sigmoid* as it works in a different way from odd powers, where sigmoid enlarges the small values while odd powers expand the large values (See Figure 2 left). We do not use complex parameterization for three reasons: (1) Avoid large growth of the model complexity for better generalization; (2) Keep overhead of the model serving almost unchanged; (3) In our experiments (See Section 4), even comparing these two simple non-linear choices of h with the first linear choice, there is only a slight improvement so it is not promising to try more complex forms.

3.4.3 Remarks. Comparing with standard stacking, boosted-stacking has two major benefits: 1) boosted-stacking allows DTs and the NN to do inference in parallel; 2) the final inference of boosted-stacking can be simply expressed as the sum of outputs of two models, which largely reduces the dependency between DTs and the NN. We later propose a lightweight adapter that can further remove the mutual dependency so that each base model can be deployed independently.

3.5 Boosted Stacking with Lightweight Adapter Model

It is desirable to be able to update DTs or NNs separately in reality. We propose a novel lightweight adapter model h' for the T-B-N method. h' is a scalar function over the NNs scores. We choose to implement h' as a single boosting tree. This boosting tree merely takes g_2 as the input and thus keeps decoupled from g_1. The training procedure of this tree follows LambdaMART [7] with g_1 as the base model while the gradient boosting is executed in a layer-by-layer manner [35]. This approach makes the tree more compact (less parameters) so as to keep additional inferring overhead small. We refer to this model as DTs-boosted-NN-with-Boosting-Tree (T-B-NT) and the final inference made by T-B-NT can be written as

$$\textbf{T-B-NT}: \quad f(g_1, g_2, x_{i,k}) = g_1(x_{i,k}) + h'(g_2(x_{i,k})). \tag{9}$$

Such an adapter allows us to update DTs without updating NNs. Suppose we want to quickly evaluate an updated version of DTs without retraining the NN. As the NN is trained based on the original DTs, the direct combination of new DTs and the NN via T-B-N (7) may not be good. However, since this adapter h' can be easily retrained, it provides a soft combiner of new DTs and the original NN. Such a procedure is analogous to the "coordinate descent" optimization technique. Our experimental results in Section 4 show that this adapter can further improve the performance over T-B-N.

3.6 Boosted Stacking with Ranks

The standard stacking strategies adopted by Yahoo! Search [43] and DLCM [2] leverage the ranks instead of relevance scores obtained by the first ranker to train the second ranker. This scenario occurs either when the relevant score obtained by the first ranker is not stable or the first ranker only provides the ranking order of candidate documents. For a more comprehensive study, we also consider the case that the NN boosts the ranks provided by DTs, termed R-B-N. In R-B-N, without loss of generality, we suppose $g_1(x_{i,k}) = n_i + 1 - \text{rank}_{i,k}$ where $\text{rank}_{i,k} \in \{1, 2, ..., n_i\}$ is the rank of document k for query i predicted by DTs. We adopt the similar idea that is used for T-B-N. We first rescale the output of DTs, $h(g_1(x_{i,k}))$, and then add it the original output of NNs. So R-B-N follows the same inference formula as T-B-N:

$$\textbf{R-B-N}: \quad f(g_1, g_2, x_{i,k}) = h(g_1(x_{i,k}), n_i) + g_2(x_{i,k}). \tag{10}$$

As for the intrinsic discreteness of g_1 here, we set the parameterized h to be

$$h(g_1(x_{i,k}), n_i) = \begin{cases} u * g_1(x_{i,k})/(n_i + 1), & \text{linearity (lin)} \\ \sum_{1 \le j < g_1(x_{i,k})} u_j/(n_i + 1), & \text{jumping (jum)} \end{cases} \tag{11}$$

where $u_j \ge 0$ $(1 \le j \le \max_i n_i)$ and u are learned during the training of the NN. The first type of h is simply a linear mapping of the the ranks after normalization. The second type, termed *jumping*, is to learn the full parameterization of h. As $u_k/(n_i + 1)$ can be viewed as the nonnegative jump from $h(k - 1, n_i)$ to $h(k, n_i)$, all monotone mappings h over the discrete set of ranks $\{1, 2, ...\}$ can be parameterized according to this formula, which is illustrated in the right figure of Figure 2. Jumping keeps the monotonicity of h with respect to g_1, and actually lets the NN recover the missing relevance information underlying the ranks output by DTs.

3.7 Discussion

In this subsection, we analyze the properties of different combination methods from the perspective of model deployment. Particularly, we compare them in 3 dimensions: parallel training, parallel inference, and dependency between two models. Our results are summarized in Table 2. For easy reference, we also include their ranking performance according to the experiments in Section 4.

Parallel Training: LIN allows for training DTs and the NN independently and thus holds a good parallel training property. All

Table 2: The properties and ranking performance of models. "✓": good; "–": neutral; "✗": not good.

Models	Para. Train	Para. Infer	Dependency	Performance
LIN	✓	✓	✓	✗
T-S-N	✗	✗	✗	✓
N-S-T	✗	✗	✗	–
N-B-T	✗	✓	–	–
T-B-N	✗	✓	–	✓
T-B-NT	✗	✓	✓	✓
R-B-N	✗	✓	–	–

the other combinations require the two base models to be trained in a certain order.

Parallel Inference: Different from training, the real-time requirements of model inference is more important as it directly affects the latency of online serving. LIN and all boosted-stacking methods allow DTs and the NN to make inference in parallel, because the scores of two base models can be computed independently and merged via a simple transformation and sum. The two standard stacking methods do not have these properties as shown in equations (4) and (5).

Dependency: Dependency between two models determines whether these two base models can be developed and maintained separately. Particularly, practitioners may hope that adjusting each base model may explicitly affect the overall performance without retraining the other model. For this purpose, LIN is the best as the DTs and NN do not have mutual dependency. The boosted-stacking methods are also good, as the final score is a simple sum of the outputs of two base models after simple monotone transformations. Some dependency between the two base models may be introduced in boosting training, but the adapter in T-B-NT can further decrease such dependency. The two standard stacking methods, T-S-N and N-S-T, are the worst as the second base models deeply couple the outputs of the first base models with other features.

4 EVALUATION

In this section, we conduct a series of experiments over two commercial personal search systems — Gmail search and Google Drive search, to evaluate different combination strategies proposed in Section 3. We describe our experimental setup and report the experimental results. We also provide some in-depth analysis to understand the behaviors of different strategies.

4.1 Experimental Design

4.1.1 Data Sets and Metric. Our experiments use the click-through data from Gmail and Google Drive, which we refer to as Email and File for simplicity in the remainder of this section. In both services, we regard the clicks of users as the labels (i.e. $y_{i,k} \in \{0, 1\}$). We discard all the queries without clicks. We also account for the selection bias of clicks via our previously proposed approach [39].

We collect Email/File search logs for a consecutive period of time, resulting in hundreds of millions of queries with clicks for each system. Each query is associated with about 5 documents on average, which is a direct result of the search user interface. Among all the queries, 80% are used for training and 10% are used for validation and parameter tuning. The remaining 10% are held-out for testing. We use Mean Reciprocal Rank (MRR) and Negative Average Click Position (NACP) as the testing metrics, which are specifically defined as

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}, \ \text{NACP} = -\frac{1}{|Q|}\sum_{i=1}^{|Q|}\text{rank}_i$$

where rank_i refers to the rank of the first clicked document for query i inferred by a ranker.

4.1.2 Base Models for Testing. We briefly introduce the specific base models that we use for testing. We use the standard forms of these two models and expect the same conclusions to apply to a wider range of base models. Our DT-based model is based on LambdaMART [41] with MRR as the metric to compute the lambda gradient [14]. The final number of leaves of all trees is 1000. The neural network is a standard feed-forward network with three hidden fully connected layers. The number of neurons in these layers vary from 128 to 512 and the rectifiers are all ReLUs. All the parameters are chosen via proper tuning on the condition that each base model is trained and evaluated independently over the datasets. We tried different types of loss functions including the softmax cross-entropy loss for listwise comparison, the logistic loss for pairwise comparison [5], and the lambda loss [8]. Although the performance slightly varies among different types, they all achieve similar conclusions when we compare different combination strategies. Due to space constraints, we only show the results based on the cross-entropy loss. Specifically, the softmax cross-entropy loss can be viewed as a smooth version of logarithmic reciprocal rank/negative logarithmic rank, which can be derived as

$$-\ell\left(\{y_{i,k}, f(x_{i,k})\}_{k=1}^{n_i}\right) = \log\frac{1}{\sum_{k\in[n_i]}\exp[f(x_{i,k}) - f(x_{i,k^*})]} \tag{12}$$

$$\approx \log\frac{1}{\sum_{k\in[n_i]}\mathbf{1}(f(x_{i,k}) \geq f(x_{i,k^*}))} = \log\frac{1}{\text{rank}_i} = -\log\text{rank}_i.$$

where k^* is the index of the document that is clicked i.e., $y_{i,k^*} = 1$ and $\mathbf{1}(\cdot)$ is an indicator function. This may be the reason why the cross-entropy loss is a proper choice for our click-through data with MRR and NACP as the metrics.

The training features can be roughly categorized into two types: sparse and dense features. Sparse features consist of frequent query and document character n-grams that are encrypted for privacy reasons. This results in a vocabulary of about 3 million tokens. Because of encryption, we cannot use generic word embeddings. Hence, due to scalability concerns, the sparse features are not fed into DTs and only go through the NN with an embedding layer of 600 linear neurons. Dense features contain some categorical values like document types, binary features like being starred or not and floats like document age result into a vector with more than 20 dimensions for each query-document pair. Dense features are fed into both models.

4.2 Main Evaluation Results and Analysis

In this subsection, we extensively compare different types of combining methods proposed in Section 3 (See Table 1).

For comparison, we choose one of the base models — the NN as the baseline (trained independently) and show the relative improvement of different ensemble learning methods. We also provide the performance of sole DTs for comparison. Cheng et al. [10] proposed a wide&deep NN (W&D for brevity) which improved the performance of NN by adding a wide and shallow network that processed dense features extracted from complex feature engineering. Our boosted-stacking strategy seems to share some similar idea as the scores/ranks predicted by DTs play a similar role to learn interactions amongst dense features. Hence, we also add the W&D for comparison. In W&D, based on the NN, we additionally duplicate dense features and further concatenate them and their transformations obtained via a cross layer [38] (with about 500 dimensions in total) to the original last hidden layer.

The overall performance is shown in Table 3. Please note that in large commercial search systems such as Gmail search and Google Drive search, an approach with an improvement like 0.2% improvement over a large-scale evaluation data set is fairly significant [39]. For both datasets, we observe the following 5 types of consistent trends:

(1) Sparse features are extremely informative since the NN solely achieves much better performance than DTs.
(2) Ensembles of DTs and the NN outperform each base model and W&D.
(3) Among all methods, T-S-N, T-B-N, and T-B-NT achieve the first-level performance. Among these three schemes, T-B-NT works the best.
(4) Among the three T-B-N methods, more complex monotone mappings h give better prediction, although the improvement is not significant. This improvement may be compensated by the adapter h', if we compare the three results for T-B-NT over the Email dataset.
(5) By comparing T-B-N-lin and T-B-N-jum, T-B-N-jum consistently works better.

Based on these observations, we may draw five major conclusions.

(1) Ensemble models of DTs and NNs are particularly useful for ranking tasks. Moreover, an ensemble model of DTs and NN performs significantly better than W&D. The underlying reason may come from the difference in the training procedures of DTs and NNs. DTs iteratively make partitions of the feature space while NNs leverage gradient descent that gradually adjusts the boundaries of partitions in some greedy manner, which makes DTs in the ensemble model perform substantially different from the wide network of W&D.
(2) Boosted-stacking is the optimal strategy to combine DTs and NNs. Besides its ease-of-deployment and low latency inference, boosted-stacking also achieves comparable or even better performance than the standard stacking, and much better performance than LIN. Among the different mappings h for strategies T-B-N and T-B-NT, complex monotone mappings h only yields slightly better performance. Considering Occam's razor principle, we suggest to choose h as simple as a weighted linear mapping.
(3) Comparing T-B-N with T-B-NT, the final adapter helps with the performance to some extent. Given the final adapter is

Data	Models	DTs	W&D	LIN	T-S-N	N-S-T	N-B-T	T-B-N			R-B-N		T-B-NT		
								lin	pow	sig	lin	jum	lin	pow	sig
Email	△MRR(%)	-3.73	0.01	0.10	0.27*	0.14*	0.13*	0.26*	0.27*	0.27*	0.09*	0.13*	**0.31*** †	0.29*†	0.30*†
	△NACP(%)	-6.93	0.01	0.13	0.43*	0.17	0.16	0.43*	0.43*	0.43*	0.09	0.17	**0.48***†	0.47*	**0.48***†
File	△ MRR(%)	-1.50	0.02	0.14	0.37*	0.18*	0.16	0.35*	0.40*†	0.39*	0.13	0.17	0.40*†	**0.46***†	0.41*†
	△NACP(%)	-3.25	0.06	0.31	0.86*	0.50*	0.39	0.86*	0.94*†	0.94*†	0.31	0.44*	0.94*†	**1.00***†	0.94*†

Table 3: Comparison of Different methods on Email and File. The △ in front of MRR and NACP indicates relative improvement over a well-trained NN. The values for △ MRR, △ NACP are in percentile scales. * and † denote statistically significant improvements (at the $p < 0.05$ level using a two-tailed t-test) over the optimally linear combination LIN and the standard stacking model N-D-T, respectively. The boldfaced values refer to the best results achieved.

also useful of independent model updates, we suggest to use T-B-NT. Note that adding the adapter may increase the latency of inference but as it only transforms one feature g_2, the additional complexity is typically very low.

(4) By comparing T-B-N with N-B-T, we see using the NN to boost DTs is better than using DTs to boost the NN. This is because the NN effectively leverages sparse features to understand on the subspace that DTs cannot learn well.

(5) If the base ranker only provides ranks, i.e., R-B-N-lin and R-B-N-jum, we suggest using the complex mapping "jumping" as more improvements may be achieved. By comparing R-B-N and T-B-N, we claim that the output scores of DTs are much more informative than simple ranks, which should be used to train the boosting NN when available.

As aforementioned, the study targeted at Ads CTR [27] also considered combining NNs and DTs. However, some opposite conclusions were drawn because of the difference in the settings. NNs therein were not directly fed with sparse features but with summarized dense scores obtained via logistic regression. This manner degraded the representation power of NNs so even the DTs-only model could beat NNs. Moreover, our suggested strategies including T-B-N and T-B-NT were not considered in [27]. In next subsection, we further look into the experimental results to better understand why using NNs as boosters provides the right direction.

4.3 Understanding Behavior of Base models

4.3.1 Behavior of the Neural Network in Boosted-Stacking. In this subsection, we investigate why boosted-stacking strategies work better than others.

Intuitively, by taking the combination $h(g_1) + g_2$ to compute the loss, boosted-stacking strategies reserve the power of the NN for the queries that the decision tree fails to handle well. We further quantify the insight above. We denote rank[DT] (or rank[NN]) as the rank of the clicked document among all documents associated with a given query according to scoring functions obtained by DTs (or the NN respectively). Figure 3 draws the distributions of rank[NN] given rank[DT] of different combining strategies over two datasets. For LIN, as DTs and the NN are trained independently, two models behave similarly: if rank[DT]= j it is most likely that rank[NN]= j for any j. However, when using boosted-stacking strategies, the behavior of the NN tends to compensate DTs. That is, for easy queries (rank[DT]= 1), rank[NN] is still most likely to be 1, but the percentages of rank[NN]= 1 are much less than that of LIN (Email: $79\% \rightarrow 60\%, 68\%$, File: $90\% \rightarrow 75\%, 87\%$ for T-B-N, R-B-N); For difficult queries (rank[DT]= 5), the majority

Data	Models	Evaluating only the neural network (g_2)				
		T-B-N			R-B-N	
		lin	pow	sig	lin	jum
Email	△ MRR (%)	-3.76	-3.96	-4.44	-1.77	-1.85
	△ NACP (%)	-5.37	-5.67	-6.28	-2.42	-2.51
File	△ MRR (%)	-5.59	-5.69	-5.78	-0.42	-0.56
	△ NACP (%)	-11.19	-11.44	-11.63	-2.13	-2.31

Table 4: Performance of NNs via boosted-stacking training. All values are significantly below 0 at the $p < 0.05$ level using a two-tailed t-test.

of rank[NN] obtained via boosting becomes rank 1. As Table 4 shows, because boosted-stacking forces the NN to focus on hard queries, the performance of the NN itself can deteriorate a lot while the overall performance in Table 3 remains good. We further compare among boosted-stacking strategies. Figure 3 shows T-B-N can compensate better than R-B-N. This observation is consistent with the conclusion summarized from Table 3, which implies the scores output by DTs are more informative than the ranks. Moreover, the more complex h's tend to give slightly stronger compensation but the difference is not much.

The standard stacking strategy, T-S-N, which takes the output of DTs as an input feature of the NN, performs such compensation in a similar but implicit way. T-S-N may further learn the complex interactions between the output of DTs and other features, which the boosted-stacking strategy T-B-N cannot. But our experimental results show that the gain from this part is limited.

As a remark, our strategies to use NN for boosting also shares some insight from the prestigious ResNet [17] that carried out ground-breaking improvements in extracting image representations. Both models force the NN to focus on some smaller spaces of the original problem by introducing plus operations among the layers (Figure 1). As experiments showed in both cases, such tricks alleviate some learning difficulty of NNs. The main difference between our models and ResNet is where such focused improvements come from (DTs vs. the NN itself).

4.3.2 Behavior of the adapter in the model T-B-NT. As described in Section 3, this adapter (boosting tree) can essentially be viewed as a transformation h' which tweaks the output of the NN so that $g_1 + h'(g_2)$ may achieve better ranking performance. According to Table. 3, by comparing T-B-N and T-B-NT, we have seen that this operation can result in slight but consistent improvement. To understand the underlying reason, let us compare the final training step of T-B-N and T-B-NT. For T-B-N, the final "plus" layer (see Figure 1) can be viewed as a one-layer NN with two dense features

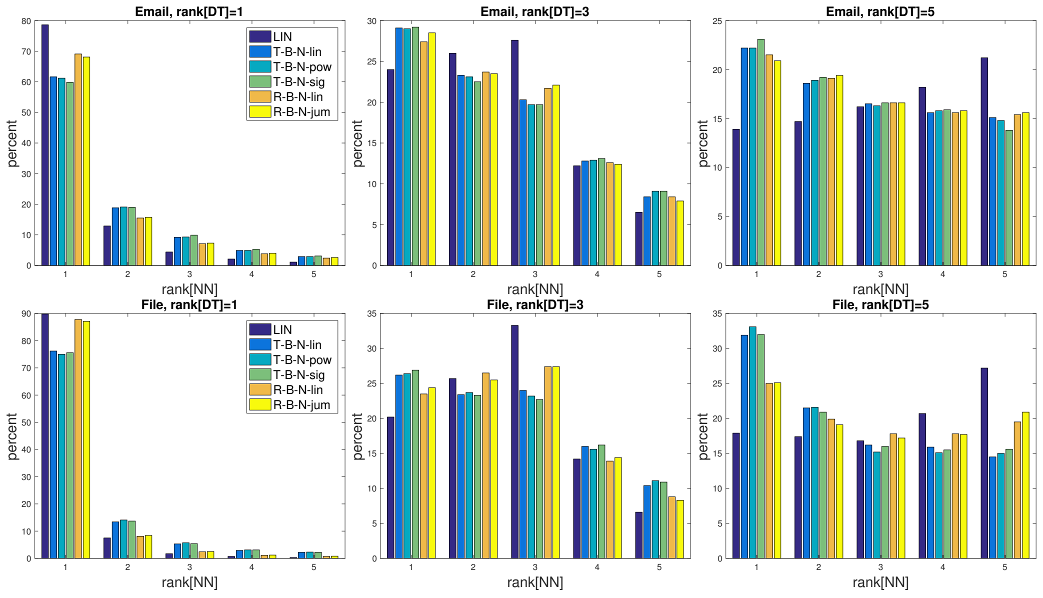

Figure 3: Distributions of rank[NN] given rank[DT]= j, $j \in \{1, 3, 5\}$ of different ensemble learning stategies over Email and File datasets. rank[DT] (or rank[NN]) denotes the rank of the clicked document among all documents associated with a given query according to scoring functions obtained by DTs (or the NN respectively).

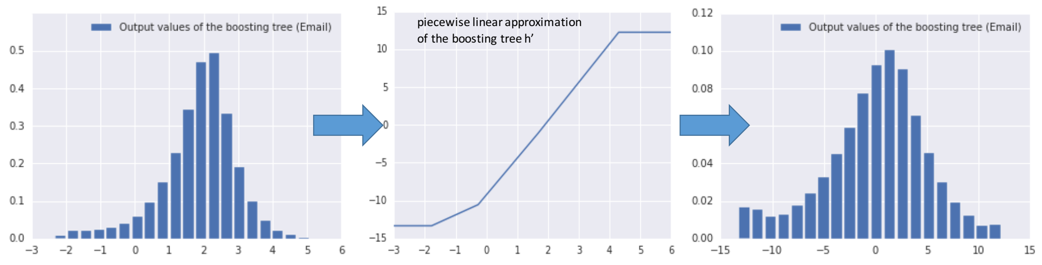

Figure 4: The functionality of the adapter h' in T-B-NT trained over the Email dataset

$h(g_1)$ and g_2 to minimize the cross-entropy loss (12), while, for T-B-NT, the adapter h' is learned via a DT-based approach LambdaMART [7]. So the superiority of T-B-NT over T-B-N essentially comes from the better expressiveness of LambdaMART versus NNs when dealing with dense features for learning-to-rank metrics. To see the functionality of the adapter h', Figure 4 shows a piecewise linear approximation of h' and how it adjusts the distribution of g_2. Essentially, it helps to shrink the extreme values obtained via a trained NN (See $h'(x)$ when $x < -0.3$ and $x > 4.2$) so that the final ensemble works more properly for the ranking task.

4.3.3 Faster training speed of boosted-stacking. Although experiments show that T-S-N achieves comparable performance with T-B-N, the training speed of NN in standard stacking should be slower than that in boosted-stacking. As the effect of scores from DTs on the final performance of T-B-N is more direct, training NN in T-B-N is easier and thus demonstrates a better rate of convergence. We demonstrate the idea by evaluating the MRRs of two models after being trained over certain numbers of epochs. We use the same learning rate to train these two models and also normalize their MRRs with the optimal MRRs they achieve correspondingly. The results are shown in Figure 5.

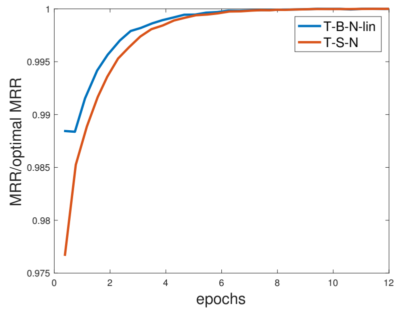

Figure 5: MRR/optimal MRR vs training epochs (Email).

5 CONCLUSION

In this work, we studied how to combine DTs and NNs for learning-to-rank problems with both dense and sparse features. Inspired by the idea of boosting and stacking for ensemble learning, we proposed 12 combining strategies and compared them in terms of ranking performance and ease-of-deployment. Through extensive experiments over two of the world's largest personal search engines, we observed that an ensemble of DTs and NNs can significantly improve ranking performance. We also found that using NNs to compensate DTs through a boosted-stacking strategy offers both competitive performance and engineering flexibility, and provided some in-depth analysis to understand the compensating behavior of the boosting approaches.

There are a few of interesting directions for future study. (1) Our methods were evaluated in the context of large-scale personal search with clicked data. It would be interesting to evaluate them on other eligible scenarios where both dense and sparse features can be used, and where large-scale data is available. (2) Since NNs are most likely used to extract representations of raw texts, it would be interesting to investigate the underlying semantics of those embeddings when NNs work as boosters. (3) We adopted a boosting tree as the adapter to merge the output of two models. We would like to explore if other methods are better in learning an adapter.

REFERENCES

[1] Eugene Agichtein, Eric Brill, and Susan Dumais. 2006. Improving web search ranking by incorporating user behavior information. In *Proceedings of the 29th annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 19–26.

[2] Qingyao Ai, Keping Bi, Jiafeng Guo, and W Bruce Croft. 2018. Learning a Deep Listwise Context Model for Ranking Refinement. In *Proceedings of the 41st annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 135–144.

[3] Michael Bendersky, Xuanhui Wang, Donald Metzler, and Marc Najork. 2017. Learning from user interactions in personal search via attribute parameterization. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 791–799.

[4] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

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

[6] Christopher Burges, Krysta Svore, Paul Bennett, Andrzej Pastusiak, and Qiang Wu. 2011. Learning to rank using an ensemble of lambda-gradient models. In *Proceedings of the Learning to Rank Challenge*. 25–35.

[7] Christopher JC Burges. 2010. From RankNet to LambdaRank to LambdaMART: An overview. *Learning* 11, 23-581 (2010), 81.

[8] Christopher J Burges, Robert Ragno, and Quoc V Le. 2007. Learning to rank with nonsmooth cost functions. In *Advances in Neural Information Processing Systems*. 193–200.

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

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM, 7–10.

[11] W Bruce Croft, Donald Metzler, and Trevor Strohman. 2010. *Search engines: Information retrieval in practice*. Vol. 283. Addison-Wesley Reading.

[12] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schutza. 2008. Introduction to information retrieval. *An Introduction To Information Retrieval* 151, 177 (2008), 5.

[13] Mostafa Dehghani, Hamed Zamani, Aliaksei Severyn, Jaap Kamps, and W Bruce Croft. 2017. Neural ranking models with weak supervision. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 65–74.

[14] Pinar Donmez, Krysta M Svore, and Christopher JC Burges. 2009. On the local optimality of LambdaRank. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 460–467.

[15] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. 2003. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research* 4, Nov (2003), 933–969.

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

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[18] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*. ACM, 2333–2338.

[19] Muhammad Ibrahim and Mark Carman. 2016. Comparing pointwise and listwise objective functions for random-forest-based learning-to-rank. *ACM Transactions on Information Systems (TOIS)* 34, 4 (2016), 20.

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

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

[22] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. 2017. Unbiased learning-to-rank with biased feedback. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 781–789.

[23] Steven G Krantz and Harold R Parks. 2002. *A primer of real analytic functions*. Springer Science & Business Media.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[25] Francesco Lettich, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2018. Parallel Traversal of Large Ensembles of Decision Trees. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[26] Ping Li, Qiang Wu, and Christopher J Burges. 2008. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in Neural Information Processing Systems*. 897–904.

[27] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. 2017. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 689–698.

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

[29] Tie-Yan Liu, Jun Xu, Tao Qin, Wenying Xiong, and Hang Li. 2007. Letor: Benchmark dataset for research on learning to rank for information retrieval. In *Proceedings of SIGIR 2007 workshop on learning to rank for information retrieval*, Vol. 310. ACM Amsterdam, The Netherlands.

[30] Yoelle Maarek. 2017. Mail Search: It's Getting Personal!. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '17)*. ACM, New York, NY, USA, 3–3. https://doi.org/10.1145/3077136.3080642

[31] Llew Mason, Jonathan Baxter, Peter L Bartlett, and Marcus R Frean. 2000. Boosting algorithms as gradient descent. In *Advances in Neural Information Processing Systems*. 512–518.

[32] Bhaskar Mitra and Nick Craswell. 2017. Neural Models for Information Retrieval. *arXiv preprint arXiv:1705.01509* (2017).

[33] David Opitz and Richard Maclin. 1999. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research* 11 (1999), 169–198.

[34] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[35] Natalia Ponomareva, Thomas Colthurst, Gilbert Hendry, Salem Haykal, and Soroush Radpour. 2017. Compact multi-class boosted trees. In *2017 IEEE International Conference on Big Data*. IEEE, 47–56.

[36] Jay M Ponte and W Bruce Croft. 1998. A language modeling approach to information retrieval. In *Proceedings of the 21st annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 275–281.

[37] Si Si, Huan Zhang, Sathiya Keerthi, Druv Mahajan, Inderjit Dhillon, and Cho-Jui Hsieh. 2017. Gradient boosted decision trees for high dimensional sparse output. In *International Conference on Machine Learning*.

[38] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. ACM, 12.

[39] Xuanhui Wang, Michael Bendersky, Donald Metzler, and Marc Najork. 2016. Learning to rank with selection bias in personal search. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 115–124.

[40] Xuanhui Wang, Nadav Golbandi, Michael Bendersky, Donald Metzler, and Marc Najork. 2018. Position bias estimation for unbiased learning to rank in personal search. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 610–618.

[41] 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.

[42] Jun Xu and Hang Li. 2007. AdaRank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 391–398.

[43] Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly, Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, et al. 2016. Ranking relevance in yahoo search. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 323–332.

[44] Hamed Zamani, Michael Bendersky, Xuanhui Wang, and Mingyang Zhang. 2017. Situational context for ranking in personal search. In *Proceedings of the 26th International Conference on World Wide Web*. 1531–1540.