Ranking, Boosting, and Model Adaptation

Qiang Wu, Chris J.C. Burges, Krysta M. Svore and Jianfeng Gao
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Microsoft Research Technical Report MSR-TR-2008-109

October 15, 2008

Abstract

We present a new ranking algorithm that combines the strengths of two previous methods: boosted tree classification, and LambdaRank, which has been shown to be empirically optimal for a widely used information retrieval measure. The algorithm is based on boosted regression trees, although the ideas apply to any weak learners, and it is significantly faster in both train and test phases than the state of the art, for comparable accuracy. We also show how to find the optimal linear combination for any two rankers, and we use this method to solve the line search problem exactly during boosting. In addition, we show that starting with a previously trained model, and boosting using its residuals, furnishes an effective technique for model adaptation, and we give results for a particularly pressing problem in Web Search - training rankers for markets for which only small amounts of labeled data are available, given a ranker trained on much more data from a larger market.

1 Introduction

We consider the ranking problem for Information Retrieval (IR), where the task is to order a set of results (documents, images or other data) by relevance to a query issued by a user. Ranking is a core technology that is fundamental to widespread applications such as internet search and advertising, recommender systems, and social networking systems. In this paper, we propose a new ranking algorithm that combines the strengths of two previous approaches: LambdaRank [2], and boosting. LambdaRank has been shown to be a very effective ranking algorithm for optimizing IR measures. It leverages the fact that neural net training needs only

the gradients of the cost function, not the function values themselves, and it models those gradients using the sorted positions of the documents for a given query. This bypasses two significant problems, namely that typical IR measures [10], viewed as functions of the model scores, are either flat or discontinuous everywhere [1], and that those measures require sorting by score, which itself is a non-differentiable operation. On the other hand, it was recently shown that treating the ranking problem as a simple classification problem, followed by mapping the outputs to a single score by computing the expected relevance, and using boosted trees as the classifiers ("McRank"), can work remarkably well [9]. However, McRank is inefficient in test phase (each round of boosting requires as many trees as there are classes). Yet, its success suggests that using boosted trees in an algorithm that directly optimizes the IR cost function, rather than simply treating the problem as a classification problem, may give further improvement to the accuracy / speed tradeoff. This paper presents such an algorithm.

We consider retrieval problems with five levels of relevance and we use the Normalized Discounted Cumulative Gain (NDCG) relevance measure [7], which is suitable for non-binary relevance measures and which emphasizes the top returned results. For a given query $Q_i, i = 1, ..., m$ the NDCG is defined as:

$$N_i \equiv n_i \sum_{j=1}^{T} (2^{r(j)} - 1) / \log(1 + j) \tag{1}$$

where $r(j) \in \{0, ..., 4\}$ is the integer label for the relevance level of the j^{th} URL in the sorted list, and where T is the truncation level at which the NDCG is computed. Here n_i is a normalization constant chosen so that $N_i = 1$ for a perfect ranking for truncation level T. For multiple queries, the NDCGs are simply averaged.

2 Relation to Previous Work

Recently the problem of learning to rank has attracted increasing attention in the machine learning community. As described above, a key goal is to set up a learning problem that can be solved efficiently for an underlying problem that is non-smooth, non-convex and in fact combinatoric. Yue et al. used SVMs to optimize a convex upper bound on Mean Average Precision, a widely used binary measure [13]. Le and Smola proposed using the Hungarian Marriage algorithm to optimize a convex bound on any general IR measure [8]. However, although these algorithms are fast in test phase for linear kernels, one generally needs more expressive models for the Web Search problem, and using general kernels renders such methods to be unacceptably slow. Other approaches have modified AdaBoost for NDCG

[11] and have considered ranking using the whole list of returned results as input for computing the score of a given document [4]. At the other extreme, ignoring the IR measure and treating the problem as a classification problem, using boosted trees as proposed by Li et al. [9], works remarkably well. However the resulting algorithm ("McRank") is slow (in both train and test phases) since it requires as many trees per iteration as classes (namely, five, in [9]). One might hope that simply treating the problem as a regression problem would yield the same performance speedup for similar accuracy, but [9] showed that regression does not work as well as classification for this task. Zheng et al. [14] propose a method of using gradient boosting for ranking on smooth pairwise loss functions, but most IR metrics, such as NDCG, are non-smooth and cannot be optimized directly in this framework. Prior to this work, neural nets were shown to give good results [2, 3], and in particular, a training method called LambdaRank [2] has been shown to optimize the NDCG measure [5, 12], which is a very intriguing result. The LambdaRank trick is basically to note that neural net training requires only the gradients (of the cost with respect to the model scores), and that these can be chosen heuristically, based on the rank position and label of each document, *after the sort*. The LambdaRank gradients reported in [2, 5, 12] are the gradient of the pairwise log binomial loss [3] multiplied by the NDCG gained by swapping the two documents, and then summed over pairs of documents (see Section 4); they are smooth functions of the document ranks (in that the gradients change smoothly as two adjacent documents exchange rank positions during learning); the idea is to rely on the (also smooth) RankNet cost gradient to encode the dependence on the document scores.

Boosted trees are very flexible models. For example, they handle categorical and count data better than neural nets (they can use count data directly, whereas nets require inputs with similar dynamic ranges); they give models for which the importance of each feature can be computed directly; and truncating the number of boosted trees (in the order in which they were trained) gives a simple method for trading off speed and accuracy. This tradeoff is particularly important for a Search Engine, where one is often willing to sacrifice accuracy for improved speed. The work described above raises the following question: can we combine the flexibility of boosted trees, with the empirical optimality that has been observed for LambdaRank, to construct a ranker that has the benefits of both methods? It is this question that we investigate in this paper. Following [9], we will use MART [6] as the starting point. The principal novelty of our work springs from three main ideas: first, we use the LambdaRank gradients when training each tree, so that as opposed to McRank, the number of trees per boosting iteration is just one. In addition, the use of LambdaRank gradients allows us to consider highly non-smooth IR metrics, such as DCG and NDCG. Previous work combining pairwise cost functions with MART allow for only smooth, twice-differentiable risk functions [14] and do not

take the entire results set for a given query into consideration, which is very important for complex ranking metrics such as NDCG. It is not obvious how to combine the LambdaRank gradients with MART (for example, the LambdaRank gradients depend on pairs of samples, and typically MART is used for costs that depend on individual samples); solving this is a principal contribution of our work. Second, a major problem that Search Engines face, beyond the basic ranking problem, is model adaptation: for example, using labeled data for a large, established market as a starting point to train models for markets with much smaller labeled dataset sizes. To address this problem we use the additive nature of boosted trees to replace the first tree with a previously trained model (a "submodel"); hence the name of our algorithm, "LambdaSMART", for Lambda-submodel-MART, or LambdaMART for the case with no submodel. Third, we present a new method for finding the optimal linear combination of any two rankers, for any IR measure. We begin with the latter.

3 How To Optimally Combine Two Rankers

The problems that IR measures present for optimization, as described above, can be turned to our advantage. Here we show how this property can be leveraged to find the optimal linear combination of any two rankers. For concreteness we will refer to NDCG, but the method applies to any IR measure. Our method can be used to combine, for example, rankers trained on different data sets, or trained using different algorithms; we will use it below to find optimal combinations of weak learners during boosting.

The idea is a path-following method and is illustrated in Figure 1. There, the vertical lines represent the ranges of the outputs of two different rankers, R and R', for the same single query; each point on each line is the score for a particular document, where s_i^R denotes the score for document i from Ranker R, and the scores s_i^R and $s_i^{R'}$ are convexly combined as $s_i = (1-\alpha)s_i^R + \alpha s_i^{R'}$, $\alpha \in [0,1]$. As α sweeps from 0 to 1, the score for each document follows the corresponding line moving from left to right. When $\alpha = 0$, the score is precisely Ranker R's score, and when $\alpha = 1$, the score is precisely Ranker R''s score. Due to its discrete nature, the NDCG can only change when two or more lines cross (and when the corresponding labels of the documents differ). Hence we can simply enumerate all possible values of α for which the NDCG changes by analytically computing all possible crossing points. Thus, at each crossing point, we only have to evaluate the change in NDCG caused by swapping the two documents involved in the crossing. This is an $O(n^2)$ algorithm, where n is the mean number of documents returned per query (as are many ranking algorithms). Note that the requirement that we

keep track of the NDCG as the mixing parameter α sweeps from 0 to 1 means that (1) for a given query, every pair of documents with different labels must be examined (since the NDCG will change when they swap rank positions) and (2) for a given query, every pair of documents with the same label must also be examined (since we must also keep track of every document's rank to use in subsequent computations of the NDCG). These together mean that the algorithm cannot do better than $O(n^2)$. For multiple queries, we compute all crossing points α_c for all queries, and then sort the α_c. By traversing this sorted list we can then analytically compute the change in NDCG for every crossing point across all queries, and save the value of α_c that gives the highest overall NDCG.

A boosting model in functional form may be written as $F(x) = \sum_i \alpha_i f_i(x)$, where x is the input feature vector and where the f_i are the weak learners. Usually the weight α_i is learned once f_i has been trained, using for example a Newton-Raphson step (which requires an estimate of the inverse Hessian) [6], and α_i is then left fixed. The inverse Hessian is approximated since it is too expensive to compute exactly. The method proposed here gives an $O(n^2)$ algorithm to compute α_i exactly, given the trained f_i, obviating the need for the Newton-Raphson step. Methods to avoid overfitting, such as "shrinkage" [6], can equally well be applied to the α's computed using our path following algorithm, which has the significant advantage that the α one starts with is known to be optimal for the training data. In the case of boosting models, it is more convenient to fix the weight of the current ranker R output at 1 and let α vary from 0 up to some maximal value: $s_i = s_i^R + \alpha s_i^{R'}$, where $R = \sum_{j=1}^{i-1} \alpha_i f_i(x)$ is the model up to that boosting iteration and $R' = f_i(x)$ is the new tree to be added to the model. In computing a given α, degeneracies (where several lines in Figure 1 cross at the same point) can either be computed analytically or removed by adding jitter (very small random values) to the scores. Degeneracies at the endpoints (which is commonly encountered when training trees) can be similarly handled, or can be broken by adding $\epsilon \ll 1$ times the value of a strong, floating point feature that correlates positively with relevance (such as BM25) to the model score; however we chose a more principled approach, that of computing the expectation of the NDCG, given that the ranks of the documents with a given score all have equal probability. Note that this expectation can in fact be computed efficiently with a single loop over the documents for any given query. Finally we note that, for cases where limiting the number of trees provides sufficient regularization for the data at hand (so that no shrinkage is needed), we can get improved fits for all the α_i by iteratively recomputing α_j given that all $\alpha_{k \neq j}$ are held fixed, so that at any iteration we are computing the optimal combination of two rankers. This iterative procedure is guaranteed to converge since the NDCG is monotonically non-decreasing at every step. We emphasize that our method for optimally combining ranker works for any set of rankers (although optimality is

only guaranteed for a given pair of rankers), and in particular it is not limited to boosting models; it may for example prove useful for constructing ensembles of rankers.

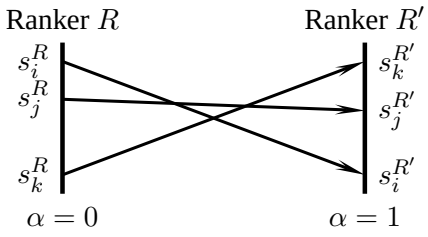


Figure 1: Optimally combining two rankers. NDCG changes only at the crossing points. The two vertical lines represent the sorted list of scores output by Ranker R and R', respectively. s_i^R indicates the score of document i ouput by ranker R.

4 The LambdaSMART Algorithm

LambdaSMART is built on MART, the details for which we refer the reader to [6]. Recall that at each boosting iteration, MART builds a regression tree to model the functional gradient of the cost function of interest, evaluated at all the training points. Our approach does the same but with the LambdaRank functional gradients, since we are interested in optimizing NDCG. Since NDCG is either flat or discontinuous everywhere, LambdaRank uses an approximation to the gradient of the cost, called λ-gradients. A particular document is given a (scalar) λ-gradient which is computed using all the pairs of documents for which that document occurs as a member of the pair, and for which the other member of the pair was generated for the same query, but has a different label. The λ-gradient consists of the product of two factors: (1) the RankNet cost [3] (a pairwise cross-entropy loss, applied to the logistic of the difference of the model scores), and (2) the NDCG gained by swapping the pair, ΔNDCG. Although the first factor is pairwise (only depending on the local information of the pair), the second factor depends on the global structure of the entire query and the metric under consideration (in our case, NDCG). It is due to these two components that LambdaRank can be applied to any IR metric (by substituting that metric for NDCG), and in fact has been shown to be empirically optimal for several such metrics [5, 12]. The λ-gradients may be written as

$$\lambda_{ij} \equiv S_{ij} \left| \Delta \text{NDCG} \frac{\partial C_{ij}}{\partial o_{ij}} \right| \tag{2}$$

where $o_{ij} \equiv s_i - s_j$ is the difference in ranking scores for a pair of documents in a query (here we are using s_i as a shorthand for $F(x_i)$), $C_{ij} \equiv C(o_{ij}) = s_j - s_i + \log(1 + e^{s_i - s_j})$ is the cross-entropy cost applied to the logistic of the difference of the scores, ΔNDCG is the NDCG gained by swapping those two documents (after sorting all documents by their current scores), and $S_{ij} \in \{-1, 1\}$ is plus one if document i is more relevant than document j (has higher label value) and minus one if document i is less relevant than document j (has lower label value) [2]. Note that $\partial C_{ij}/\partial o_{ij} = \partial C_{ij}/\partial s_i = -1/(1+e^{o_{ij}})$, and that the overall sign of λ_{ij} depends only on the labels of documents i and j, and not on their rank position. Each point then sums its λ-gradients for all pairs P in which it occurs

$$\lambda_i = \sum_{j \in P} \lambda_{ij} \tag{3}$$

LambdaRank has a physical interpretation in which the documents are point masses and the λ-gradients are forces on those point masses; the λ's generated by any given pair of documents are then equal and opposite. A positive lambda indicates a push toward the top rank position and a negative lambda indicates a push toward the lower rank positions [2].

Algorithm 1 The LambdaSMART algorithm.

1: **for** $i = 0$ to N **do**
2: $F_0(x_i) = BaseModel(x_i)$ \\$BaseModel$ may be empty or set to a submodel.
3: **end for**
4: **for** $m = 1$ to M **do**
5: **for** $i = 0$ to N **do**
6: $y_i = \lambda_i$
7: $w_i = \frac{\partial y_i}{\partial F(x_i)}$
8: **end for**
9: $\{R_{lm}\}_{l=1}^{L}$ \\Create L-terminal node tree on $\{y_i, x_i\}_{i=1}^{N}$
10: $\gamma_{lm} = \frac{\sum_{x_i \in R_{lm}} y_i}{\sum_{x_i \in R_{lm}} w_i}$ \\Find the leaf values based on approximate Newton step.
11: $F_m(x_i) = F_{m-1}(x_i) + v \sum_l \gamma_{lm} 1(x_i \in R_{lm})$
12: **end for**

Algorithm 1 summarizes the LambdaSMART algorithm; it assumes there are N total documents in our training set and that we wish to train M stages (trees). We optionally load a submodel in Step 2. This is easy to implement: one simply starts by computing the LambdaRank functional gradients of the cost function us-

ing the scores output by the submodel, and then trains the trees as described in the algorithm.

LambdaSMART training then proceeds similarly to [6]. M rounds of boosting are performed, and at each boosting iteration, a regression tree is constructed and trained on all documents for all queries. We can choose the iteration to stop at based on performance on a validation set.

Steps 6 calculates the λ-gradients for each document i, as described above. Step 7 calculates the second-order derivative using the λ-gradients (which are smooth in the scores). A regression tree with L terminal nodes is built in step 9, using Mean Squared Error to determine the best split at any node in the regression tree. The value associated with a given leaf of the trained tree is computed first as the mean of the λ gradients for the training samples that land at that leaf. Then, since each leaf corresponds to a different mean, a one-dimensional Newton-Raphson line step is computed for each leaf (Step 10). These line steps may be simply computed as the derivatives of the LambdaRank gradients with respect to the model scores s_i. Finally, in Step 11, the regression tree is added to the current boosted tree model, weighted by the shrinkage coefficient v, which is chosen to regularize the model.

LambdaSMART has three main parameters: M, the total number of boosting iterations, L, the number of leaf nodes for each regression tree, and v, the "shrinkage coefficient" - the fraction of the optimal line step taken. Using a shrinkage coefficient with value less than one is a form of regularization [6]. We selected the optimal parameters by using a validation set. Fortunately, as verified in our experiments, the performance of the algorithm is relatively insensitive to these parameters as long as they are in a reasonable range: given the training set of a few thousand queries or more $M = 500$, $L = 15$, and $v = 0.1$ usually give good performance. Smaller trees and shrinkage may be used if the training data set is smaller.

A novelty of our approach over the algorithms described in [6] is that we use a pairwise cost function, in particular for non-smooth metrics, which has been shown to give excellent performance for ranking [2, 3]. Since we are optimizing NDCG at each step, we do not need the *number-of-classes* trees per iteration that McRank needs. We could also achieve one tree per iteration by considering regression instead of classification. However, regression has been shown to cause a decrease in accuracy (see Figure 1 of [9]); our approach overcomes this drawback.

5 Experiments

We perform experiments to (1) compare the accuracy and speed of LambdaSMART and LambdaMART to LambdaRank and McRank (the latter two algorithms are state-of-the-art rankers and have been reported to outperform previous state-of-the-art rankers on the Web Search task); (2) assess the effectiveness of model adaptation by training a base model and boosting it using different data sets; and (3) provide preliminary results on whether the optimal ranker combination improves the NDCG and the learning speed over the Newton step.

5.1 The Data

The data sets include the artificial and Web-1 data sets used in [2, 3, 9], and a more recent data set, Web-2, used for the model adaptation studies. We also use four language data sets, namely Korean, English, Chinese, and Japanese, for model adaptation studies. All data sets contain samples labeled on a 5-level relevance scale. In all cases, the train/valid/test sets contain non-overlapping queries.

The artificial data set was synthetically produced to mimic a perfectly labeled data set, as in [3]. It was created from random cubic polynomials and contains 50 features. There are 50 URLs per query and 10K/5K/10K in train/valid/test sets. The Web-1 data has 367 features, with on average 26 URLs per query, and 10K/5K/10K queries for train/valid/test sets. Web-2 is constructed by sub-sampling queries whose length[1] are four or more from its superset Web-2-Super. Web-2-Super contains queries of all lengths. The Web-2 data has 4666 features. On average, there are 170 URLs per query, and 525/225/500 queries for train/valid/test sets. The Web-2-Super set contains 14893/1230/6402 queries for train/valid/test sets. In our query length model adaptation experiment, Web-2-Super serves as the background domain and Web-2 serves as the adaptation domain.

For across-domain adaptation experiments from non-Korean to Korean markets, we use Korean data for the adaptation domain, and English, Chinese, and Japanese data sets as the background domain. The Korean data has 425 features with a total of 4430 queries. The average number of URLs per query is 75. The train/valid/test sets contain 3724/372/334 queries, respectively. The English data contains 6167 queries, with on average 198 URLs per query. The Chinese data comprises 32827 queries with on average 72 URLs per query. The Japanese data comprises 45012 queries with on average 58 URLs per query.

The Web and language data sets contain features constructed from the document (including anchor text and URL information), the query, and matches between the document and the query. Although the data sets are not of the size the

[1] We use query length to mean the number of words in the query.

ranker would see at test phase, the sets used for training are of the rough order of magnitude of those used for web scale training. In particular, we show that our algorithm is fast enough at test phase to handle web scale test data, in particular due to the fewer number of required trees.

5.2 Model Parameters

Model parameters are chosen using validation sets: here we summarize the best settings found. LambdaRank is tuned by varying the number of layers, the number of hidden nodes, and the learning rate. For all data sets we use two layers unless otherwise stated, and ten hidden nodes. On the artificial data, we use a learning rate of 10^{-4}; for the Web-1 data, we use a learning rate of 10^{-5}; and for the Web-2 data, we use a learning rate of 10^{-4}. McRank and LambdaSMART are both tuned by varying the number of leaf nodes L, the shrinkage v, and the number of boosting iterations M. For McRank we set $L = 10$, $L = 0.05$ and $M = 1000$ for all datasets, as in [9]. For LambdaMART we use $M = 1000$ and $v = 0.1$ for all datasets, $L = 10$ for the artificial data, and $L = 15$ for the Web-1 data. For LambdaSMART (the model adaptation experiments) we use $M = 500$, $L = 20$, and $v = 0.1$. Although LambdaSMART is in general not sensitive to model parameters, we report the best parameters found on validation data for completeness and as a principled way to find model parameters. Our results do not imply sensitivity to model parameters.

5.3 Accuracy Results

We compare results of LambdaRank, McRank, and LambdaMART on the artificial and Web-1 data. We use LambdaMART since we found that in this setting it performs as well or better than LambdaSMART on the validation data. We report NDCG results (where queries for which all URLs have the same label have been dropped), at truncation levels 10, 3, and 1.

Table 1 lists the NDCG results on the 10K artificial test queries. The artificial data has no label noise, so less strongly regularized models such as McRank and LambdaMART learn the data well and outperform a 2-layer LambdaRank model. Both McRank and LambdaMART were run for 1000 iterations; note that McRank therefore has 5000 trees, as opposed to LambdaMART's 1000.

Table 2 lists the results of the three algorithms on the 10K Web-1 test queries. McRank and LambdaMART exhibit similar asymptotic performance here, although as we shall see in the next section, LambdaMART exhibits better speed/accuracy tradeoff behavior. The NDCG results on both data sets indicate that McRank and LambdaMART outperform LambdaRank.

Table 1: LambdaMART, McRank and LambdaRank results on the artificial data set, with 95% confidence intervals. Results are reported for NDCG at truncation levels 10, 3 and 1.

	λ-MART	**McRank**	**λ-Rank**
NDCG@10	87.9 (0.16)	83.7 (0.19)	75.4 (0.25)
NDCG@3	81.7 (0.32)	75.6 (0.36)	67.8 (0.41)
NDCG@1	79.6 (0.56)	72.2 (0.65)	65.8 (0.66)

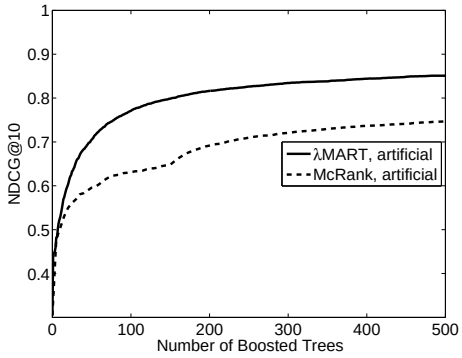
Table 2: LambdaMART, McRank and LambdaRank results on the Web-1 data set. Results are reported for NDCG at truncation levels 10, 3 and 1.

	λ-MART	**McRank**	**λ-Rank**
NDCG@10	69.3 (0.46)	69.7 (0.46)	68.6 (0.47)
NDCG@3	62.5 (0.60)	62.9 (0.60)	61.5 (0.60)
NDCG@1	61.3 (0.81)	61.6 (0.81)	60.4 (0.82)

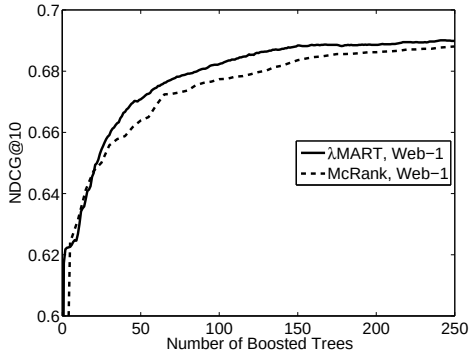
5.4 Speed vs. Accuracy Results

The most significant advantage of LambdaSMART over McRank is its improved behavior regarding the speed/accuracy tradeoff. This is crucial for real time applications such as Web Search, where typically results must be returned to the user within milliseconds of their issuing a query. Figure 2 plots accuracy (NDCG@10) versus speed (the number of boosted trees) for both λ-MART and McRank, for both the artificial and the Web-1 data. Again the validation set was used to choose the optimal settings, which were found to be $L = 20$ and $v = 0.15$ (from ranges $L = 10, 15, 20$ and $v = 0.05, 0.1, 0.15$). The graphs show the results on the test set, for systems trained with the above optimal settings. Since both methods use the same number of leaf nodes, the number of trees provides a reliable measure of speed. The faster learning exhibited by λ-MART gives a significant speed-up for a large range of accuracies: although the curves in the right panel appear close, a single point of NDCG gain is a significant increase in accuracy for Web Search. Achieving the same accuracy, but with approximately half as many trees, is a big win.

Additional speed-ups can be obtained by increasing the shrinkage parameter at



(a) Results on the artificial data.



(b) Results on the Web-1 data.

Figure 2: Speed versus accuracy results for McRank and λ-MART on the artificial and Web-1 data sets.

a small cost in accuracy or by performing early stopping by essentially reducing the number of boosting iterations. However, these methods can be applied to McRank as well, and any speed-ups gained by using them for McRank will also benefit LambdaSMART.

5.5 Model Adaptation Results

Ranking model adaptation attempts to adjust the parameters of a ranking model trained on one domain (called the background domain), for which large amounts of training data are available, to a different domain (the adaptation domain), for which only a small amount of training data is available. In Web search applications,

Table 3: Results on Web-2 test data, for LambdaRank and for LambdaSMART with the LambdaRank net as submodel. All models were trained with Web-2-Super.

	λ-Rank	**λ-SMART**
NDCG@10	70.5 (2.48)	70.7 (2.51)
NDCG@3	62.6 (3.06)	62.3 (3.04)
NDCG@1	54.0 (4.06)	55.4 (4.04)

domains can be defined by query length, languages, dates, etc.

In this section we report results on two adaptation experiments. The first uses a large set of Web data, Web-2-Super, as the background domain and uses Web-2 (data containing only queries of length 4 or more) as the adaptation domain. In this scenario, the idea is that we have very little data for long queries containing 4 or more words, but we have lots of Web data on queries of all lengths. We compare using a 2-layer LambdaRank model with 10 hidden nodes trained on Web-2-Super and a subset of 440 features, to a LambdaSMART model that takes as an initial model that LambdaRank model, but "adapts" using the Web-2 data. We train LambdaSMART with $L = 10$, $M = 263$ (chosen using the validation set) and $v = 0.1$. The results are listed in Table 3. Here, no statistically significant gain was observed. This, together with the successful adaptation experiments described below, suggests that for successful adaptation with LambdaSMART, using just a few hundred queries for the adaptation training phase is not sufficient.

The second experiment is an adaptation experiment involving data from several languages. Two-layer LambdaRank baseline rankers are first built from Korean, English, Japanese, and Chinese training data and tested on Korean test data (Table 4). These baseline rankers then serve as submodels for LambdaSMART and are "adapted" using the Korean training data, and tested on the Korean test data (Table 5). We randomly divided the Korean dataset into three non-overlapping subsets. A subset containing 3724 queries is used as training data (adaptation training data in our model adaptation experiments). The subset containing 372 queries is used as validation set, and the remaining subset with 334 queries is used as test set. For the LambdaSMART training, we used $L = 20$, $M = 500$ and $v = 0.1$. Although the Korean train data set is much smaller than the other three data sets, Table 4 shows that the ranking model trained on the Korean data set is still much better than other models trained on much larger cross-domain training data (due to the domain mismatch between training and test data). This is a typical result of cross-domain training.

Table 4: Baseline model adaptation results: LambdaRank trained on the four training sets (one for each language) but tested on Korean.

	Korean	**English**	**Japanese**	**Chinese**
NDCG@10	61.4	58.3	53.9	52.8
NDCG@3	56.9	53.3	49.1	45.8
NDCG@1	58.4	53.1	48.5	42.5

Table 5: NDCG results of the non-Korean baseline models adapted on Korean training data using LambdaSMART, and also using a linear interpolation model (Interp).

	English	**Japanese**	**Chinese**	**Interp.**
NDCG@10	64.7	63.6	64.1	61.5
NDCG@3	60.9	59.5	59.6	57.1
NDCG@1	61.2	60.2	60.8	58.8

Table 5 shows that all adaptation results are significantly better than the corresponding baseline, and that LambdaSMART is a very effective model adaption technique. We also compared our method with model interpolation. Model linear interpolation has been widely used as a baseline for model adaptation in the speech and natural language processing communities and is still considered the state-of-the-art method of model adaptation. We could also consider merging the data sets and training a model on the merged data. In our experiments, linearly interpolating models trained on background and adaptation data sets respectively achieves better results than simply training on merged datasets. We linearly interpolate the four baseline rankers, which are trained respectively on the Korean, English, Japanese, and Chinese datasets as aforementioned. The interpolation weights are learned using the Powell Search algorithm to optimize NDCG on the Korean validation data set. The results are listed in the right hand column of Table 5. They are only slightly better than the baseline results. However LambdaSMART model adaptation achieves significant NDCG gains over interpolation and over the baseline.

5.6 Optimal Combination Results

Here we present results validating the optimal combination method described in Section 3. For the model we used LambdaMART. We trained a baseline model, which uses the full Newton step to compute the combination weight for each leaf, and a model "OC" that uses the optimal combiner to compute the global combination weights (i.e. one per tree). We used the artificial data as described in [3]. The advantage of the optimal combiner is that it bypasses the (diagonalized) Newton-Raphson approximation and returns the exact answer. However, here we are replacing the per-leaf weights (each computed with its own Newton-Raphson step) with a single global (but optimal) mixing parameter. Our intent here is simply to show that using the optimal combination strategy works, and can help, despite the approximation introduced by replacing per-leaf weights by a single weight per tree; we emphasize that the optimal combination trick is likely to also prove useful elsewhere.

Figure 3 shows the results of training on the 10K queries and using the 5K validation queries to choose the optimal step size. Note that both training and test accuracy converge significantly faster using OC. This experiment used a version of OC where the combined score takes the form $s_i = s_i^R + \alpha s_i^{R'}$, where R is the model of previously trained trees and R' is the new tree to add, which is more convenient for boosting (the convex combination version requires repeatedly changing the weights of the previously trained trees). We limit α to lie in the interval [0.1,100]; the lower limit is necessary because occasionally a new tree provides almost no gain, and the optimal combiner therefore sets its weight close to zero, resulting in the training essentially stopping. In this experiment we handle the problem of ties using the probabilistic averaging method described in Section 3. This data set does not require setting the shrinkage to a value less than one, but we emphasize that using the optimal combination method does not preclude using shrinkage, or other regularization methods.

6 Discussion and Future Work

LambdaSMART inherits significant advantages from both MART and LambdaRank. It has the flexibility and the interpretability of boosted trees, and we have shown that replacing the first tree with a previously trained model significantly improves accuracy for the model adaptation problem. From LambdaRank it inherits the property of direct optimization of the IR measure at hand, and in addition produces models that have significantly better behavior regarding the speed/accuracy tradeoff. It is intriguing that the gains are so different between the artificial and real data sets. The artificial data set was chosen to have properties that are as close

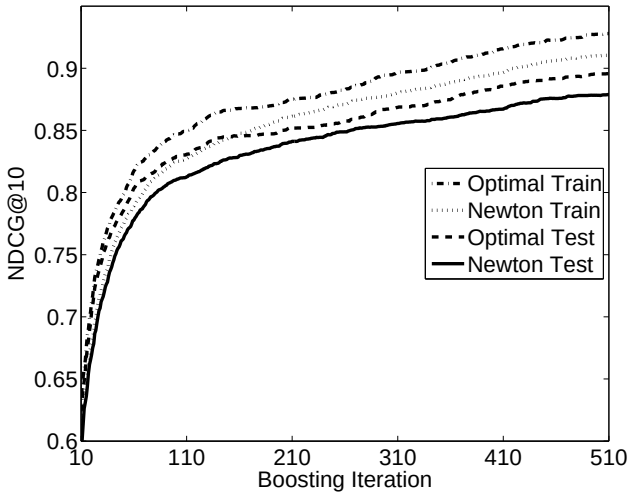


Figure 3: NDCG@10 versus boosting iteration; the curves are ordered as in the legend.

as possible to the real data (i.e. the distribution of labels, the number of features, and the number of urls per query). One significant difference is that the real data is known to be very noisy (with both label noise and feature noise) and we plan to investigate whether modifying the boosted tree methods to better handle noise gives further improvements. We also plan to investigate whether similar ideas - boosted trees trained with LambdaRank-type gradients - can be used to optimize for other commonly used IR measures. Finally, the optimal combination results suggest that finding per-leaf optimal weights may also prove useful.

References

[1] C.J.C. Burges. Ranking as Learning Structured Outputs. In C. Cortes S. Agarwal and R. Herbrich, editors, *Proc. NIPS Workshop on Learning to Rank*, 2005.

[2] C.J.C. Burges, R. Ragno, and Q.V. Le. Learning to Rank with Non-Smooth Cost Functions. In *NIPS*, 2006.

[3] C.J.C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to Rank using Gradient Descent. In *ICML 22*, Bonn, Germany, 2005.

[4] Z. Cao, T. Qin, T-Y. Liu, M-F. Tsai, and H. Li. Learning to Rank: From Pairwise Approach to Listwise Approach. In *ICML*, 2007.

[5] P. Donmez, K.M. Svore, and C.J.C. Burges. On the Optimality of LambdaRank. *In preparation*, 2008.

[6] J.H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. Technical report, Dept. Statistics, Stanford, 1999.

[7] K. Jarvelin and J. Kekalainen. IR Evaluation Methods for Retrieving Highly Relevant Documents. In *SIGIR 23*. ACM, 2000.

[8] Quoc Le and Alexander J. Smola. Direct Optimization of Ranking Measures. *CoRR*, abs/0704.3359, 2007. Informal publication.

[9] P. Li, C.J.C. Burges, and Q. Wu. Learning to Rank Using Classification and Gradient Boosting. In *NIPS*, 2007.

[10] S. Robertson and H. Zaragoza. On Rank-based Effectiveness Measures and Optimization. *Information Retrieval*, 10(3):321–339, 2007.

[11] J. Xu and H. Li. A Boosting Algorithm for Information Retrieval. In *SIGIR*, 2007.

[12] Y. Yue and C.J.C. Burges. On Using Simultaneous Perturbation Stochastic Approximation for Learning to Rank, and the Empirical Optimality of LambdaRank. Technical Report MSR-TR-2007-115, Microsoft Research, 2007.

[13] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A Support Vector Method for Optimizing Average Precision. In *ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2007.

[14] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A General Boosting Method and its Application to Learning Ranking Functions for Web Search. In *NIPS*, 2007.