Learning to Rank for Information Retrieval
DOI 10.1007/s10791-009-9112-1

Adapting Boosting for Information Retrieval Measures

Qiang Wu · Christopher J. C. Burges · Krysta M. Svore · Jianfeng Gao

Received: 21 April 2009 / Accepted: 19 August 2009

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. Our 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 significantly improved 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.

Keywords Learning to Rank · Boosting · Web Search

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.

There are two basic categories of ranking algorithms: one scheme is based on learning the pairwise preference, such as RankNet [5] and LambdaRank [4], which use neural nets to learn the pairwise preference function; the other scheme is based on relevance regression or classification such as McRank [17]. Because a perfect ranking implies perfect decisions on all pairs' preferences and an incorrect ranking implies the existence of

Microsoft Research
One Microsoft Way
Redmond, WA 98052
E-mail: {qiangwu,cburges,ksvore,jfgao}@microsoft.com

mistakenly ordered pairs, learning a ranking function is equivalent to learning a pairwise preference function. On the other hand, although predicting (with classification or regression) the relevance labels perfectly implies perfect ranking, the converse is not true. For example, if the classifier assigns class $c-1$ to each returned document whose true class is c, and the documents are ranked by class, then the ranking will be perfect even though the classification error rate is 100%. Therefore, casting ranking as learning pairwise preferences is superior to treating it as a classification or regression problem simply because it avoids solving an unnecessarily hard problem. Another advantage of the pairwise scheme is the fact that at each stage of boosting, we need train only one tree, as opposed to training a multiclass classifier, for which each stage of boosting requires as many trees as there are classes. This results in much smaller and faster models at test time, which is crucial when ranking millions of documents in real time, as is required for Web Search.

In this paper, we propose a new ranking algorithm that combines the strengths of two previous approaches: LambdaRank [4], and boosting. LambdaRank has been shown to be a very effective ranking algorithm for optimizing IR measures [8]. It is a pairwise-based approach that 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 [19], viewed as functions of the model scores, are either flat or discontinuous everywhere [3], 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 [17]. 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 [14], 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} document 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 information retrieval and machine learning communities. The superiority of learned ranking models over traditional probabilistic retrieval models has been demonstrated

on benchmark data sets. For example, Gao et al. [11] showed that a linear ranking model significantly outperforms a number of state-of-the-art language models [10, 20, 24] and the classical probabilistic retrieval model [15] on the ad hoc retrieval task using TREC test sets.

A key goal of learning to rank 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 [23]. Le and Smola proposed using the Hungarian Marriage algorithm to optimize a convex bound on any general IR measure [16]. 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 [21] and have considered ranking using the whole list of returned results as input for computing the score of a given document [6]. 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. [17], 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 [17]). One might hope that simply treating the problem as a regression problem would yield the same performance speedup for similar accuracy, but [17] showed that regression does not work as well as classification for this task. Zheng et al. [25] proposed 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 [4, 5], and in particular, a training method called LambdaRank [4] has been shown to optimize the NDCG measure [8, 22], 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 [4, 8, 22] are the gradients of the pairwise log binomial loss [5] multiplied by the NDCG gained by swapping the two documents, and then summed over pairs of documents (see Section 3); 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 smoothly 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 [17], we will use MART [9] as the starting point. The principal novelty of our work springs from three main ideas:

1. 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 allows for only smooth, twice-differentiable risk functions [25] and does 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, and contrasts with other recent work on using boosted trees with smooth costs [7]. Our work also differs from [7] in that we solve the ranking problem directly, rather than solve an intermediate regression problem.
2. 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 (for more detailed results on model adaptation and interpolation, see [13]).
3. We present a new method for finding the optimal linear combination of any two rankers, for any IR measure. This is a basic technique that has many possible applications: for example, solving the model adaptation problem by optimally combining a ranker trained on a large amount of data, with one trained on data for a small market; or, computing the optimal linear combination that is required when adding a new tree to a model during the learning phase for boosting.

The paper is organized as follows. In Section 3 we describe the LambdaSMART algorithm. In Section 4 we describe the path-following optimal combination technique. This technique can be used within the LambdaSMART algorithm to potentially find a better combination of regression trees. Experimental results are given in Section 5. Our experimental results fall into three main categories: experiments showing the speedup gained by LambdaSMART over the previous best ranker, McRank; experiments showing the significant gains that can be achieved using model adaptation, for which LambdaSMART is particularly well-suited; and experiments demonstrating the optimal combination method. We present conclusions and future work in Section 6.

3 The LambdaSMART Algorithm

LambdaSMART is built on MART (Multiple Additive Regression Trees). We refer the reader to [9] for details, although here we briefly summarize the MART algorithm for completeness. MART is a boosted tree algorithm that performs gradient descent in function space [18]. By this is meant the following: viewing the cost C as a functional of the model output (or, of the function value F), then to first order, $C = C_0 + \frac{\partial C}{\partial F}\delta F$. Thus as in ordinary gradient descent, by choosing $\delta F \propto -\frac{\partial C}{\partial F}$ for a suitable step size, the model further reduces the cost. Since the functional gradient $\frac{\partial C}{\partial F}$ can only be evaluated at the training points, the trees give a means of estimating a smooth

regression to the gradients everywhere. Each tree in MART may thus be viewed as a small step δF in function space, where the step size (which is computed using the Newton approximation) becomes the weight attached to that tree. Performance can be further improved by computing a step size for each leaf node. Each tree is computed as a standard regression tree, using least squared error to compute the best splits.

Our approach also builds regression trees to model the functional gradient of the cost function of interest, evaluated at all the training points. However we use the LambdaRank functional gradients, since we are interested in optimizing NDCG. Here we briefly summarize the ideas behind LambdaRank. Since the NDCG cost is either flat or discontinuous everywhere, LambdaRank uses an approximation to the gradient of the cost, called λ-gradients. Consider a set of documents that have been ranked, for a given query, while training the model. 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 for a given document thus depends on its position in the sorted list, and on the positions of the other documents (that have different labels) in the sorted list. Specifically, the contribution to the λ-gradient for a given document, resulting from its membership in a given pair of documents, consists of the product of two factors: (1) the RankNet cost [5] (a pairwise cross-entropy loss, applied to the logistic of the difference of the model scores), for the pair of documents, 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 on the metric under consideration (in our case, NDCG). The first factor plays the role of a smooth cost with a margin built in; that is, even documents that are correctly ordered, or that have the same rank, get a contribution from the RankNet cost, and this contribution falls off smoothly as $s_1 - s_2$ increases, where s_1 (s_2) is the score of the more (less) relevant document. Thus a key intuition behind the λ-gradient is the observation that NDCG does not treat all pairs equally; the cost depends on the global sorted order as well as on the labels. It is due to these two separate factors 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 [8, 22] (by "empirically optimal", we mean that the algorithm finds a local optimium for the cost function, which is by no means obvious, given the indirect route that LambdaRank takes in modeling the cost). This motivates our using the LambdaRank gradients as target gradients in MART. Concretely, 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 $F(x_i)$),

$$C_{ij} \equiv C(o_{ij}) = s_j - s_i + \log(1 + e^{s_i - s_j}) \tag{3}$$

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) [4]. Note that

$$\partial C_{ij}/\partial o_{ij} = \partial C_{ij}/\partial s_i = -1/(1+e^{o_{ij}}) \tag{4}$$

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:

$$\lambda_i = \sum_{j \in P} \lambda_{ij}. \tag{5}$$

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 for any given pair of documents are 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 [4].

We now combine MART and LambdaRank to form LambdaSMART, which is summarized in Algorithm 1. Here we assume that there are N total documents in our training set and that we wish to train M boosting stages (trees). The "S" in LambdaSMART refers to a submodel that one can use as the initial model (as opposed to training the first tree from scratch). 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 using the scores output by the submodel, and then trains the trees as described in the algorithm.

LambdaSMART training then proceeds similarly to [9]. 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 choose the final number of trees for the model by using a validation set.

Step 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 11). 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 14, the regression tree is added to the current boosted tree model, weighted by the "shrinkage coefficient" v [9], which is chosen to regularize the model. Choosing a fixed, global shrinkage coefficient is in fact equivalent to setting the slope of the sigmoid used in the LambdaRank gradients.

Thus LambdaSMART has three parameters: M, the total number of boosting iterations, L, the number of leaf nodes for each regression tree, and v, the "shrinkage coefficient". 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 lie within 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.

Algorithm 1 The LambdaSMART algorithm.

for $i = 0$ to N **do**
  $F_0(x_i) = BaseModel(x_i)$ \\$BaseModel$ may be empty or set to a submodel.
end for
for $m = 1$ to M **do**
  for $i = 0$ to N **do**
    $y_i = \lambda_i$ \\Calculate λ-gradient for sample i.
    $w_i = \frac{\partial y_i}{\partial F(x_i)}$ \\Calculate derivative of λ-gradient for sample i.
  end for
  $\{R_{lm}\}_{l=1}^{L}$ \\Create L-terminal node tree on $\{y_i, x_i\}_{i=1}^{N}$.
  for $l = 0$ to L **do**
    $\gamma_{lm} = \frac{\sum_{x_i \in R_{lm}} y_i}{\sum_{x_i \in R_{lm}} w_i}$ \\Find the leaf values based on approximate Newton step.
  end for
  for $i = 0$ to N **do**
    $F_m(x_i) = F_{m-1}(x_i) + v \sum_l \gamma_{lm} 1(x_i \in R_{lm})$ \\Update model based on approximate Newton step and shrinkage size.
  end for
end for

A further novelty of our approach over the algorithms described in [9] is that we use a pairwise cost function, in particular for non-smooth metrics, which has been shown to give excellent performance for ranking [4,5]. 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 [17]); our approach overcomes this drawback.

4 How To Optimally Combine Two Rankers

The problems that IR measures present for optimization, as described in Section 1, 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 of the typically used IR measures [19]. 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

$$s_i = (1 - \alpha) s_i^R + \alpha s_i^{R'}, \tag{6}$$

where $\alpha \in [0, 1]$. 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.

One can use this to compute the optimal weights for combining the weak learners in a boosting model. In functional form, any boosting model may be written

$$F(x) = \sum_i \alpha_i f_i(x), \tag{7}$$

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) [9], 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" [9], 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 $R = \sum_{j=1}^{i-1} \alpha_j f_j(x)$ 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 rankers 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. Experimental results for this algorithm, on both artificial and real web data, are given in Section 5.6. In order to explore the contributions of the various new ideas described in this paper, we used standard techniques for computing the weights assigned to the trees throughout, except in Section 5.6, where results for the optimal combiner are given.

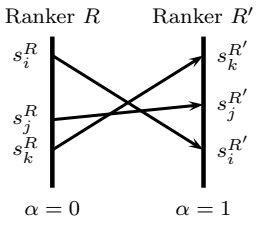


Fig. 1 Optimally combining two rankers. NDCG changes only at the crossing points. The two vertical lines represent the sorted list of scores output by Ranker R and R', respectively. s_i^R indicates the score of document i ouput by ranker R.

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 an artificial set and a Web data set, called Web-1. We perform model adaptation studies on four language data sets, namely Korean, English, Chinese, and Japanese, a names data set consisting of only person name queries, and a long query data set consisting of queries of length four or more[1]. All data sets contain samples labeled on a 5-level relevance scale and all train/valid/test sets contain non-overlapping queries. 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. Queries were sampled from search engine query logs and URLs were sampled from search engine results.

The artificial data set, generated as described in [5], was synthetically produced to mimic a perfectly labeled data set. It was created from random cubic polynomials and contains 50 features. There are 50 URLs per query and 10K/5K/10K in

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

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.

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 3724/334/372 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 names Web data set has 416 features, on average 105 URLs per query, and 5725/158/318 queries in train/valid/test sets. The long query Web data set has 416 features, on average 98 URLs per query, and 6255/176/356 queries in train/valid/test sets. In our model adaptation experiments, the names and long query data sets serve as the respective adaptation domains. The background domain is the same for name and long queries, namely Web-2. Web-2 has 416 features, on average 134 URLs per query, and 31555 queries in the train set (since we use it for model adaptation, we do not need a valid or test set).

Although the data sets are not of the size the 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.

The performance of different ranking methods is measured through NDCG evaluated against test sets. We report NDCG results (where queries for which all URLs have the same label have been dropped, since all rankers give identical NDCG on such queries), at truncation levels 10, 3, and 1. Significance test (i.e., t-test) was also employed.

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 $L = 10$, $v = 0.05$ and $M = 1000$ for all datasets, as in [17]. 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. Reported experiments do not use the optimal combiner approach, with the exception of experiments reported in Section 5.6.

Table 1 LambdaMART, McRank and LambdaRank results on the artificial and Web-1 data, with 95% confidence intervals in the parentheses. Results are reported for NDCG at 10, 3 and 1.

Artificial	**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)
Web-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)

5.3 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.

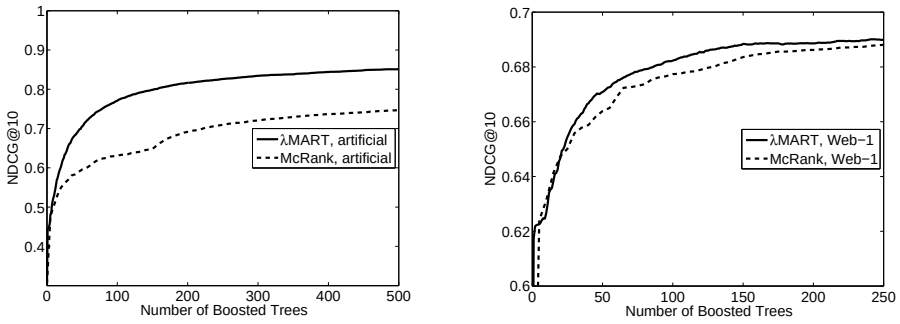
Table 1 lists the NDCG results on the 10K artificial and 10K Web-1 test queries, with 95% confidence intervals listed in the parentheses based on a statistical t-test. The artificial data has no label noise, so less strongly regularized models such as McRank and LambdaMART learn the data well and outperform 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.

On the Web-1 data, McRank and LambdaMART exhibit similar asymptotic performance, although as shown 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. We also compared our results against BM25, since BM25 alone has been used for ranking in the Information Retrieval community for several years, and we find that BM25 is consistently behind Lambda(S)MART, McRank, and LambdaRank by several NDCG points, for all datasets.

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 LambdaMART and McRank, for both the artificial and the Web-1 data. 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 LambdaMART gives a significant speed-up for a large range of accuracies: although the curves in the right panel appear close, a single percentage 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.



a) Artificial b) Web-1

Fig. 2 Speed versus accuracy results for McRank and LambdaMART.

Additional speed-ups can be obtained by increasing the shrinkage parameter at 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, domains can be defined by query length, languages, dates, etc.

Model adaptation has been well-studied in the context of statistical language models for a variety of natural langauge and speech applications. State-of-the-art adaptation techniques can be grouped into two categories: maximum *a posteriori* (MAP) estimation and discriminative training methods. MAP methods adjust the parameters of the background model so as to maximize the likelihood of the adaptation data [2]. Discriminative training methods, on the other hand, aim at using the adaptation data to directly minimize the errors on the adaptation data made by the background model [1, 12]. LambdaSMART can be viewed as a discriminative training method. In our experiments we also compare it with model interpolation, a previous state-of-the-art method of model adaptation [2].

In this section we report results on three adaptation experiments. The first uses a large set of Web data, Web-2, as the background domain and uses the long query data set (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 against several baselines: a 2-layer LambdaRank model with 15 hidden nodes and a learning rate of 10^{-5} trained on Web-2 (called the Background Ranker), a 2-layer LambdaRank

Table 2 Results on Long Query test data, for baseline models and LambdaSMART with the Background Ranker as submodel. Results are reported for NDCG at 10, 3 and 1.

Long	**Back.**	**In-dom.**	**Interp.**	**λ-SMART**
NDCG@10	47.78	48.42	48.71	48.38
NDCG@3	45.32	46.05	46.39	46.19
NDCG@1	45.23	49.10	48.00	47.80

Table 3 Results on Names Query test data, for baseline models and LambdaSMART with the Background Ranker as submodel. Results are reported for NDCG at 10, 3 and 1.

Names	**Back.**	**In-dom.**	**Interp.**	**λ-SMART**
NDCG@10	54.46	57.74	57.47	59.51
NDCG@3	49.52	52.96	52.54	54.49
NDCG@1	45.75	49.21	47.45	50.42

model with 15 hidden nodes trained on the long query train data set only (called the In-domain Ranker), and an interpolated ranker, which is a linear interpolation of the Background Ranker and the In-domain Ranker, and the interpolation weights were optimized on long query validation data. We "adapt" the Background Ranker to long queries by training off the Background Ranker with long query training data. We trained LambdaSMART with $M = 500$ trees, each with $L = 20$ leaves, and with a learning rate of $v = 0.1$. At each boosting iteration, we randomly selected 70% of training samples, instead of all training samples, to construct the regression tree. We found randomness to be crucial to the performance of the model. The results are listed in Table 2. 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 thousand queries for the adaptation training phase is not sufficient.

The second experiment is an adaptation experiment on names queries. Again, Web-2 serves as the background domain. The adaptation domain is the names query data set. All experiments report test numbers on the names test set. We again compare against several baseline rankers: a 2-layer LambdaRank model with 15 hidden nodes and a learning rate of 10^{-5} trained on Web-2 (called the Background Ranker), a 2-layer LambdaRank model with 10 hidden nodes trained on the names query train data set only (called the In-domain Ranker), and a ranker interpolated on the Background Ranker and the In-domain Ranker, where the interpolation weights were optimized on names query validation data. We "adapt" the Background Ranker to names queries by using the Background Ranker as a submodel for LambdaSMART, and training LambdaSMART on the names query train data. We again trained LambdaSMART with $M = 500$, $L = 20$ and $v = 0.1$. At each boosting iteration, we randomly selected 70% of training samples. Results are given in Table 3. In this case, the In-domain Ranker and Interpolated Ranker demonstrate similar performance. However, LambdaSMART far outperforms all baseline rankers significantly, with p-value < 0.05 for all NDCG levels, according to the paired t-test.

The third experiment is an adaptation experiment involving data from several languages (Table 4). Two-layer LambdaRank baseline rankers are first built from Korean, English, Japanese, and Chinese training data and tested on Korean test data. These

Table 4 Results for baseline model adaptation, LambdaSMART, and model interpolation (Interp.). Results are reported for NDCG at 10, 3 and 1.

Baseline	**Korean**	**English**	**Japanese**	**Chinese**
NDCG@10	62.91	58.73	60.27	57.61
NDCG@3	58.24	54.13	56.84	51.05
NDCG@1	59.27	53.71	56.40	49.66
λSMART	**English**	**Japanese**	**Chinese**	**Interp.**
NDCG@10	64.54	63.85	64.15	62.89
NDCG@3	60.57	59.66	60.95	58.70
NDCG@1	60.96	60.14	59.55	58.78

baseline rankers then serve as submodels for LambdaSMART and are "adapted" using the Korean training data, and tested on the Korean test data. We randomly divided the Korean dataset into three non-overlapping subsets. Both base and adapted models use the same feature set. 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 again used $L = 20$, $M = 500$ and $v = 0.1$. Although the Korean train data set is much smaller than the other three data sets, the first table in Table 4 shows that the ranking model trained on the Korean data set is still much better than the 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.

Results are shown in the second table in Table 4. All adaptation results are statistically significantly better (again with p-value < 0.05 for all comparisons) than the corresponding baseline. We find that LambdaSMART is a very effective model adaption technique. We also compared our method with model interpolation [2]. Model interpolation is a standard baseline for reporting model adaptation results; see [13]. 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 the second table in Table 4. They are only slightly better than the baseline results. The LambdaSMART model adaptation achieves statistically significant NDCG gains over interpolation and over the baseline.

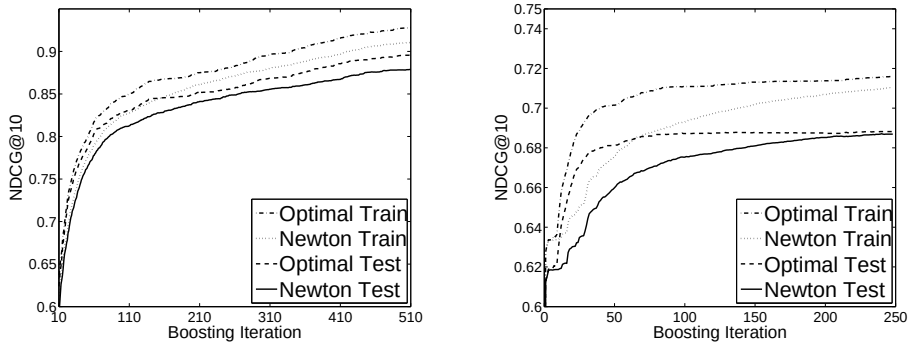
5.6 Optimal Combination Results

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 [5]. The advantage of the optimal

[2] 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.

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(a) shows the results of training on the 10K artificial 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 4. 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.



a) Artificial b) Web-1

Fig. 3 NDCG@10 versus boosting iteration; the curves are ordered as in the legends.

We performed a similar experiment on the Web-1 data. Figure 3(b) shows the results of training on the 10K Web-1 queries and using the 5K validation queries to select the optimal step size α. We trained a baseline LambdaMART model on Web-1 data using the full Newton step to compute the combination weight for each leaf. Then we trained a LambdaMART model on Web-1 data using the optimal combiner to determine the global combination weights. We compute the optimal combined score using $s_i = s_i^R + \alpha s_i^{R'}$, where R is the model of previously trained trees and R' is the new tree to add. We trained LambdaMART using shrinkage $v = 0.1$ and $L = 15$. When using the optimal combiner, we found using α in the interval [0.1,5] worked best and helped to prevent overfitting. We also experimented with smaller shrinkage, but found restricting α worked better in this case. Again, we handle the problem of ties using the

probabilistic averaging method described in Section 4. The results show we can achieve comparable performance using the optimal combiner, but with far fewer trees. We find using the optimal combiner, we require only 80 trees, whereas using the full Newton step, we require over three times as many trees, namely 250.

Figure 4 shows the values of α, chosen based on the Web-1 validation set, at each boosting iteration. Small values of α indicate the new tree provides very little gain, and thus a fractional step size is found. The fluctuation in α values across iterations indicates the optimal combiner is doing the right thing, that is it is compensating for trees of poor generalizability even when the number of trees is large. It also indicates that we can do much better than constant step size across iterations.

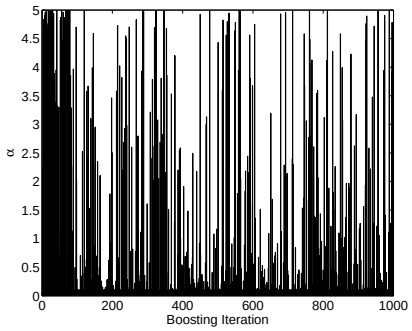


Fig. 4 α versus boosting iteration on the Web-1 data.

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 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. Bacchiani, M., Roark, B., Saraclar, M.: Language Model Adaptation with MAP estimation and the Perceptron Algorithm. In: HLT-NAACL, pp. 21–24 (2004)
2. Bellagarda, J.: An Overview of Statistical Language Model Adaptation. In: ITRW on Adaptation Methods for Speech Recognition, pp. 165–174 (2001)
3. Burges, C.: Ranking as Learning Structured Outputs. In: C.C. S. Agarwal, R. Herbrich (eds.) Proc. NIPS Workshop on Learning to Rank (2005)
4. Burges, C., Ragno, R., Le, Q.: Learning to Rank with Non-Smooth Cost Functions. In: NIPS (2006)
5. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to Rank using Gradient Descent. In: ICML. Bonn, Germany (2005)
6. Cao, Z., Qin, T., Liu, T.Y., Tsai, M.F., Li, H.: Learning to Rank: From Pairwise Approach to Listwise Approach. In: ICML (2007)
7. Chen, K., Lu, R., Wong, C., Sun, G., Heck, L., Tseng, B.: Trada: Tree based ranking function adaptation. In: ACM 17th Conference on Information and Knowledge Management (2008)
8. Donmez, P., Svore, K., Burges, C.: On the Optimality of LambdaRank. SIGIR (2008)
9. Friedman, J.: Greedy function approximation: A gradient boosting machine. Annals of Statistics **29**(5) (2001)
10. Gao, J., Nie, J.Y., Wu, G., Cao, G.: Dependence Language Models for Information Retrieval. In: SIGIR, pp. 170–177 (2004)
11. Gao, J., Qin, H., Xia, X., Nie, J.Y.: Linear Discriminative Models for Information Retrieval. In: SIGIR, pp. 290–297 (2005)
12. Gao, J., Suzuki, H., Yuan, W.: An Empirical Study on Language Model Adaptation. ACM Trans on Asian Language Information Processing **5**(3), 207–227 (2006)
13. Gao, J., Wu, Q., Burges, C., Svore, K., Su, Y., Khan, N., Shah, S., Zhou, H.: Model adaptation via model interpolation and boosting for web search ranking. In: Conference on Empirical Methods in Natural Language Processing (2009)
14. Jarvelin, K., Kekalainen, J.: IR Evaluation Methods for Retrieving Highly Relevant Documents. In: SIGIR 23. ACM (2000)
15. Jones, K., Walker, S., Robertson, S.: A Probabilistic Model of Information Retrieval: Development and Status. Tech. Rep. TR-446, Cambridge University Computer Laboratory (1998)
16. Le, Q., Smola, A.J.: Direct Optimization of Ranking Measures. CoRR **abs/0704.3359** (2007). Informal publication
17. Li, P., Burges, C., Wu, Q.: Learning to Rank Using Classification and Gradient Boosting. In: NIPS (2007)
18. Mason, L., Baxter, J., Bartlett, P., Frean, M.: Boosting algorithms as gradient descent. In: T.L. S.A. Solla, K.R. Müller (eds.) Advances in Neural Information Processing Systems 12, pp. 512–518 (2000)
19. Robertson, S., Zaragoza, H.: On Rank-based Effectiveness Measures and Optimization. Information Retrieval **10**(3), 321–339 (2007)
20. Song, F., Croft, B.: A General Language Model for Information Retrieval. In: CIKM, pp. 316–321 (1999)
21. Xu, J., Li, H.: A Boosting Algorithm for Information Retrieval. In: SIGIR (2007)
22. Yue, Y., Burges, C.: On Using Simultaneous Perturbation Stochastic Approximation for Learning to Rank, and the Empirical Optimality of LambdaRank. Tech. Rep. MSR-TR-2007-115, Microsoft Research (2007)
23. Yue, Y., Finley, T., Radlinski, F., Joachims, T.: A Support Vector Method for Optimizing Average Precision. In: SIGIR (2007)
24. Zhai, C., Lafferty, J.: Two-stage Language Models for Information Retrieval. In: SIGIR, pp. 49–56 (2002)
25. Zheng, Z., Zha, H., Zhang, T., Chapelle, O., Chen, K., Sun, G.: A General Boosting Method and its Application to Learning Ranking Functions for Web Search. In: NIPS (2007)