

# Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models

Yasser Ganjisaffar  
School of Information &  
Computer Sciences  
University of California, Irvine  
Irvine, CA, USA  
yganjisa@ics.uci.edu

Rich Caruana  
Microsoft Research  
One Microsoft Way  
Redmond, WA 98052  
rcaruana@microsoft.com

Cristina Videira Lopes  
School of Information &  
Computer Sciences  
University of California, Irvine  
Irvine, CA, USA  
lopes@ics.uci.edu

## ABSTRACT

Recent studies have shown that boosting provides excellent predictive performance across a wide variety of tasks. In Learning-to-rank, boosted models such as RankBoost and LambdaMART have been shown to be among the best performing learning methods based on evaluations on public data sets. In this paper, we show how the combination of bagging as a variance reduction technique and boosting as a bias reduction technique can result in very high precision and low variance ranking models. We perform thousands of parameter tuning experiments for LambdaMART to achieve a high precision boosting model. Then we show that a bagged ensemble of such LambdaMART boosted models results in higher accuracy ranking models while also reducing variance as much as 50%. We report our results on three public learning-to-rank data sets using four metrics. Bagged LambdaMART outperforms all previously reported results on ten of the twelve comparisons, and bagged LambdaMART outperforms non-bagged LambdaMART on all twelve comparisons. For example, wrapping bagging around LambdaMART increases NDCG@1 from 0.4137 to 0.4200 on the MQ2007 data set; the best prior results in the literature for this data set is 0.4134 by RankBoost.

## Categories and Subject Descriptors

H.3.3 [Information Systems]: Information Search and Retrieval; H.4.m [Information Systems]: Miscellaneous—*Machine Learning*

## General Terms

Algorithms, Experimentation

## Keywords

Learning-to-rank, Tree Ensembles, Bagging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGIR'11, July 24–28, 2011, Beijing, China.

Copyright 2011 ACM 978-1-4503-0757-4/11/07 ...\$10.00.

## 1. INTRODUCTION

The general problem of ranking, and in particular ranking of web search results, has received significant attention. Given the large amount of training data now available it has become possible to learn effective ranking models using machine learning. Methods that learn how to combine pre-defined features for ranking are called “Learning-to-rank” methods. A large number of learning-to-rank algorithms have been proposed, such as [8, 7, 9, 21, 28, 30] (see [22] for a more complete list).

Recent studies have shown that tree models combined with ensemble techniques provide excellent predictive performance. In the recent Yahoo! learning-to-rank challenge [10], the top ranked teams all used tree ensemble methods. The winning entry in this competition used an ensemble of LambdaMART [32] models.

In this work, we study the effectiveness of bagged ensembles of ranking models to achieve higher prediction accuracy. We train multiple independent ranking models on different samples of the training data and combine the outputs of these models to get improved prediction accuracy. In our experiments, we use LambdaMART, which is a boosted ensemble of trees, as the baseline model. Thus we are forming bagged ensembles of boosted ensembles of trees. Our approach is motivated by Breiman’s Bagging [3], so we call this method *BL-MART* for *Bagged LambdaMART*. Since sub-models are independent they can be trained in parallel and the training time is not more than training time of a single LambdaMART model. In fact, because we perform sampling without replacement, each of the LambdaMART models is trained on a smaller training set and therefore the training time is less than training a single LambdaMART model on the full train set.

We summarize the main contributions of this paper as follows:

- (a) We perform thousands of parameter tuning experiments for the LambdaMART algorithm to find near optimal parameter configurations for it and also study its sensitiveness to its parameters (section 3.4).
- (b) We show that adding randomness during the training of LambdaMART models can improve the accuracy of these models. We introduce randomness by sub-sampling queries that are available to the algorithm during each of the training iterations. We also use feature sampling as another method for increasing randomness of the algorithm (section 3.3).
- (c) We study the effectiveness of bagging ensembles of Lamb-

daMART models to both increase the prediction accuracy of the ranking model and reduce the model variance.

(d) We show that bagged ensembles of overfitted base-level models (overfitted boosted tree models) result in higher prediction accuracy than bagging optimally generated base-level models and we explain why this happens.

## 2. BACKGROUND AND RELATED WORK

LambdaMART [32] is a ranking algorithm that uses Gradient boosting [15] to optimize a ranking cost function similar to the LambdaRank [7] cost function. Readers can refer to [6] for details of this algorithm. However, since it is our base model, we briefly describe some of the important concepts that are referenced in the remainder of this paper.

Gradient boosting produces an ensemble of weak models (typically regression trees) that together form a strong model. The ensemble is built in a stage-wise process by performing gradient descent in function space. The final model maps an input feature vector  $x \in R^d$  to a score  $F(x) \in R$ :

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

where each  $h_i$  is a function modeled by a single regression tree and the  $\gamma_i \in R$  is the weight associated with the  $i$ -th regression tree. Both the  $h_i$  and the  $\gamma_i$  are learned during training. A given tree  $h_i$  maps a given feature vector  $x$  to a real value by passing  $x$  down the tree, where the path (left or right) at a given node is determined by the value of a particular feature in the feature vector and the output is a fixed value associated with the leaf that is reached by following the path.

Gradient boosting usually requires regularization to avoid overfitting. In an overfitted model, the model's generalization ability degrades because of fitting too closely to the training data. Different kinds of regularization techniques can be used to reduce overfitting in boosted trees. One common regularization parameter is the number of trees in the model,  $M$ . Increasing  $M$  reduces the error on training set, but setting it too high often leads to overfitting. An optimal value of  $M$  often is selected by monitoring prediction error on a separate validation data set.

Another regularization approach is to control the complexity of the individual trees via a number of user-chosen parameters. For example, *Max Number of Leaves per tree* limits the size of individual trees thus preventing them from overfitting to the training data. Another user-set parameter for controlling tree size is the minimum number of observations allowed in leaves. This parameter is used in the tree building process by ignoring splits that lead to nodes containing fewer than this number of training set observations. This prevents adding leaves that contain statistically small samples of training data.

Another important regularization technique is shrinkage which modifies the boosting update rule as follows:

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m h_m(x), \quad 0 < \eta \leq 1,$$

where parameter  $\eta$  is called the *learning rate*. Small learning rates can dramatically improve a model's generalization ability over gradient boosting without shrinkage ( $\eta = 1$ ), however they result in more boosting iterations and therefore larger models.

## Bias-Variance Decomposition of Error

According to the bias-variance decomposition of error [16], the squared error of a single example  $x$  can be decomposed into the sum of three non-negative terms: *noise*, *bias*, and *variance*. The first term, Noise, is the irreducible error which cannot be avoided regardless of the learning algorithm. In learning-to-rank domain, it can be interpreted as the noise in relevance judgments of query-url pairs. The second term, Bias, measures how closely the average prediction of the learning algorithm (considering all possible training sets of a fixed size) matches the optimal prediction (the Bayes rate prediction). Finally, the Variance of an algorithm is how much the algorithm's prediction fluctuates over different possible training sets of a given size.

Ensemble methods such as Gradient Boosting [15] reduce bias by increasing the expressive power of the base learner and by forcing learning to attend to training cases that consistently are mispredicted. Because boosting combines the predictions of multiple trees it also can reduce variance, but boosted trees are so powerful that regularization usually is needed to prevent overfitting. Other ensemble methods such as Bagging [3] mainly reduce variance by averaging outputs of several models trained on different samples of training data. Because bagging usually does not significantly increase expressive power and often yields large reductions in variance, it is a relatively *safe* procedure that usually does not require regularization or careful parameter tuning. In fact, bagging can be used as an effective regularization method when wrapped around other high-variance learning methods that are prone to overfitting such as boosting. There has been attempts for combining bias and variance reduction techniques for classification [31, 29] and regression [5, 14, 26, 27] problems. In this work, we combine bagging and boosting for improved learning-to-rank.

Similar to us, BagBoo [23] wraps Bagging around Boosting for improved learning-to-rank. However, there are several fundamental differences that make our work different from BagBoo. The most important difference is the size of the final model. On two public data sets that BagBoo reports its results, our BL-MART models are approximately 250 times smaller and also more accurate on most metrics (section 4). BagBoo is training more than a million trees on these data sets which makes its size infeasible for real time applications. In addition, the boosting algorithm that is used in BagBoo is a pointwise method, while BL-MART uses LambdaMART for boosting which is a listwise algorithm. Pointwise methods such as [12, 21] do not exploit relative rank information between documents, instead attempting to directly create a scoring function. In contrast, listwise methods such as [7, 9, 30] use lists of ranked documents as instances during training, and learn a ranking model by minimizing a listwise loss function. In addition, we also show that bagging boosted ensembles that are mildly overfitted to their training data gives better results.

## 3. METHODS

### 3.1 Data sets

For our experiments we work with three public data sets: TD2004 and MQ2007 from LETOR data sets [24] and the recently published MSLR-WEB10K data set from Microsoft Research [1]. Table 1 summarizes the properties of these

data sets. The TD2004 and MQ2007 data sets have been used many times for evaluating new learning-to-rank algorithms. This provides a baseline for comparing our method with other state-of-the-art learning-to-rank algorithms. MSLR-WEB10K is a large data set which is more similar to commercial search data sets. Because it is larger it should provide results that are more reliable. All three data sets are pre-folded and come with evaluation scripts that allow fair comparison of different ranking algorithms.

### 3.2 Evaluation Metrics

For model comparison we use two information retrieval metrics: Normalized Discounted Cumulative Gain (NDCG) [19] and Mean Average Precision (MAP) [2]. NDCG@ $k$  is a measure for evaluating top  $k$  positions of a ranked list using multiple levels of relevance judgment. It is defined as follows,

$$NDCG@k = N^{-1} \sum_{j=1}^k g(r_j)d(j),$$

where  $N^{-1}$  is a normalization factor chosen so that a perfect ordering of the results will receive the score of one;  $r_j$  denotes the relevance level of the document ranked at the  $j$ -th position;  $g(r_j)$  is a gain function:

$$g(r_j) = 2^{r_j} - 1;$$

and  $d(j)$  denotes a discount function. The evaluation scripts that come with the three data sets use the following discount function:

$$d(j) = \begin{cases} 1 & \text{for } j = 1, 2 \\ \frac{1}{\log_2(j)} & \text{otherwise.} \end{cases}$$

We use the same scripts for fair comparison of our final models with other algorithms. However in our implementation of the LambdaMART algorithm and in all of our training and parameter tuning experiments, we optimize for the following discount function which places stronger emphasis on higher positions:

$$d(j) = \frac{1}{\log_2(1 + j)}$$

Also, based on whether we assign NDCG values of zero or one to a query where all of the documents are assigned non-relevant labels, the scale of NDCG values will change.

### 3.3 Randomization for LambdaMART

In [14], Friedman proposed a modification of the gradient boosting algorithm which was motivated by Breiman’s bagging method. He proposed that at each iteration of the algorithm, a base learner should be fit on a sub-sample of the training set drawn at random without replacement. Friedman observed a substantial improvement in gradient boosting’s accuracy with this modification. Sub-sample size is some constant fraction  $s$  of the size of the training set. When  $s = 1$ , the algorithm is deterministic. Smaller values of  $s$  introduce randomness into the algorithm and help prevent overfitting, acting as a kind of regularization. The algorithm also becomes faster, because regression trees have to be fit to smaller data sets at each iteration.

Also similar to Random Forests [4], more randomness can be introduced by sampling features that are available to the algorithm on each tree split. On each split, the algorithm

**Table 2: Values used in grid search for parameter tuning**

(a) TD2004 and MQ2007 data sets

Parameter	Values
Max Number of Leaves	2, 4, 7, 10, 15, 20, 25
Min Percentage of Obs. per Leaf	0.12, 0.25, 0.50
Learning rate	0.05, 0.1, 0.2, 0.3
Sub-sampling rate	0.3, 0.5, 1.0
Feature Sampling rate	0.1, 0.3, 0.5, 1.0

(b) MSLR-WEB10K data set

Parameter	Values
Max Number of Leaves	10, 40, 70
Min Percentage of Obs. per Leaf	0.12, 0.25, 0.50
Learning rate	0.05, 0.1, 0.2
Sub-sampling rate	0.5, 1.0
Feature Sampling rate	0.3, 0.5, 1.0

selects the best feature from a random subset of features instead of the best overall feature.

In our experiments, we add both observation sub-sampling and feature sampling as two new parameters that need to be tuned for the LambdaMART algorithm. These parameters can take values between 0 and 1, where 1 means no sampling and values less than 1 introduce sampling randomness.

### 3.4 Parameter Tuning

The original LambdaMART algorithm has three parameters that need to be tuned to achieve the best results: “Max number of leaves”, “Min percentage of observations per leaf”, and “Learning rate”. These were described in section 2. As mentioned above, to these we have added two new parameters: “Sub-sampling rate” and “Feature sampling rate”. We use grid search to test 1,008 different combination of parameters on the smaller data sets and 162 combinations on the larger data set. Table 2 shows the values we tried for each of the parameters. Since MSLR-WEB10K contains more features for each query-url pair, we need more complex trees (trees with more leaves) on this data set.

Each combination of parameters is tested on 5 folds of each data set. On each fold we use 3 different random seeds to get more accurate results. This requires  $1,008 \times 5 \times 3 = 15,120$  experiments on each of the smaller data sets and  $162 \times 5 \times 3 = 2,430$  experiments on MSLR-WEB10K. We used a MapReduce cluster of 40 nodes for these experiments. Map tasks receive experiments as input and compute validation and test NDCG for the configuration specified by that experiment. Reduce tasks perform NDCG averaging over different folds and random seeds for each parameters combination. It takes about 8 hours to run this portion of our experiments once on this cluster.

Table 3 shows the best configurations based on Validation NDCG@3 on each data set. The best performing configurations on all three data sets use feature sampling. The smaller data sets also get better results by sub-sampling of training queries on each iteration. We conjecture that sub-sampling queries helps when the training data is small because it helps avoid overfitting by not allowing trees to see all queries on each iteration of boosting. This adds diversity to the individual trees which is then reduced when boosting averages

**Table 1: Properties of data sets used for experiments**

Data set	Queries	Query-URL Pairs	Features	Relevance Labels
TD2004	75	74,146	64	{0, 1}
MQ2007	1,692	69,623	46	{0, 1, 2}
MSLR-WEB10K	10,000	1,200,192	136	{0, 1, 2, 3, 4}

**Table 3: Best combinations of parameters found after parameter tuning.**

(a) TD2004 data set

Validation NDCG@3	Max Leaves	Min Obs. Per Leaf	Learning Rate	Sub-sampling	Feature Sampling
0.5113	20	0.25	0.1	0.5	0.1
0.5105	10	0.50	0.05	0.3	0.1
0.5061	10	0.12	0.05	0.5	0.3
0.5056	10	0.50	0.1	0.3	0.1
0.5055	15	0.25	0.05	0.5	0.1

(b) MQ2007 data set

Validation NDCG@3	Max Leaves	Min Obs. Per Leaf	Learning Rate	Sub-sampling	Feature Sampling
0.5647	7	0.25	0.05	0.3	0.3
0.5643	4	0.25	0.1	1.0	0.1
0.5643	10	0.25	0.05	0.5	0.3
0.5635	7	0.50	0.05	0.5	0.5
0.5633	7	0.25	0.05	0.5	0.3

(c) MSLR-WEB10K data set

Validation NDCG@3	Max Leaves	Min Obs. Per Leaf	Learning Rate	Sub-sampling	Feature Sampling
0.4873	40	0.25	0.1	1.0	0.5
0.4872	70	0.50	0.05	1.0	0.3
0.4870	40	0.25	0.05	1.0	0.5
0.4867	40	0.50	0.1	1.0	0.3
0.4865	40	0.50	0.05	0.5	1.0

tree predictions. With very large data sets this is less critical because individual trees cannot themselves significantly overfit a large data set when tree size is limited.

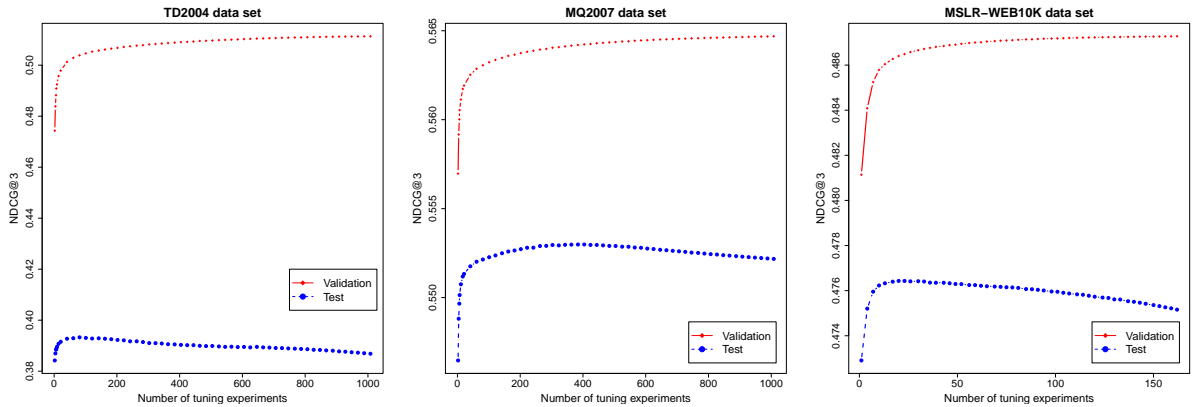
Grid search for parameter tuning is computationally expensive and becomes prohibitive as the data sets become large. Because of this it is useful to study the sensitivity of the LambdaMART algorithm to its parameters to have a better understanding of the number of training experiments needed on a new data set. We use the results of our parameter tuning experiments to study the effect of number of experiments on improvement in NDCG. For each dataset, we create a pool of configurations that we evaluated during parameter tuning experiments. Then we randomly select different numbers of these configurations. On each random selection, we pick the config with best validation NDCG@3 and then record validation and test NDCG@3 for that config. To get more accurate results, we repeat this random process 10K times and report average NDCG@3. Figure 2 shows the results.

As expected, for all three data sets, validation NDCG improves monotonically as we perform more experiments. On MSLR-WEB10K, the largest data set, there is less discrepancy between validation and test scores. The discrepancy between validation and test is largest on TD2004, the smallest data set, because the validation sets which are held aside from the training data must also be small.

Given that the parameter values we had chosen for grid search were chosen based on our experiments with LambdaMART on different data sets, we were expecting the pa-

rameter tuning experiments to reach to the pick value on test set after trying few combinations. On TD2004, the data set with the smallest validation sets, accuracy on the test set peaks after only about 100 parameter configurations, and then slowly drops. Accuracy on the validation set is still rising at 100 iterations, suggesting that hyperparameter optimization is overfitting to the validation sets. A similar effect is observed on MQ2007, but overfitting does not begin on this problem until about 400 parameter configurations have been tried. And on MSLR-WEB10K, we again observe overfitting to the validation sets after fewer than 25 configurations have been tested.

When there is randomness in the algorithm and the validation sets are not infinite, as more parameter combinations are tried search begins to find parameter combinations that look better on the validation set because of this randomness. If one is not careful, the computational power provided by MapReduce Clusters is so great that it is possible to overdo parameter tuning and find parameter combinations that work not better than the hyperparameters that would have been found by less thorough search. One way to avoid overfitting at the hyperparameter learning stage is to use a 2nd held-out validation set to detect when parameter tuning begins to overfit and early-stop the parameter optimization. Holding out a 2nd validation set will reduce the size of the primary hyperparameter tuning validation sets, making overfitting more likely. But as we have seen, even large cross-validated validation sets do not completely protect from overfitting when hyperparameter optimization is



**Figure 1: As more combinations of parameters are tested during parameter tuning of LambdaMART, NDCG@3 improves on the validation sets, but the test set curves show overfitting of the hyperparameters eventually occurs.**

exhaustive, so care must be exercised to prevent hyperparameter optimization from becoming counterproductive.

We do not directly control the best number of trees for the LambdaMART models via a user-set parameter. Instead, as iterations of boosting continue, the prediction accuracy of the model is checked on a separate validation set. Boosting continues until there has been no improvement in accuracy for 250 iterations. The algorithm then returns the number of iterations that yielded maximum accuracy on the validation set.

Figure 3 shows two sample runs on MQ2007 and MSLR-WEB10K data sets. While NDCG@3 continues to improve on the training set, it reaches its maximum on validation sets after a few hundred iterations on both data sets. It is also interesting that we do not see significant drop in NDCG on validation sets even after 2000 iterations. This suggests that the combination of regularization techniques we use is sufficient to prevent models from overfitting on these datasets.

### 3.5 BL-MART details

As mentioned before, for bagged LambdaMART we train many LambdaMART models in parallel and then aggregate their outputs for improved accuracy. In order to have more diverse models in the final ensemble, we randomly sample training data (without replacement) for training each of the LambdaMART sub-models. The randomization techniques discussed in section 3.3 also contribute additional diversity to the models.

#### 3.5.1 Combining Scores

For combining scores of the sub-models, one can simply take average of their scores. However, the scale and distribution of the scores generated from LambdaMART algorithm is dependent on the training data and on smaller training data can be very different for models trained on different sub-samples. Because of this, simply averaging LambdaMART scores may not give the best results. As an example, Figure 4 shows the distributions of validation scores generated from two models that are trained on different random samples of MQ2007 and MSLR-WEB10K training data. While on the larger data set, the output scores have very similar distributions, on the smaller data set we observe difference distributions and different scales of the

scores. Note that even on the MSLR-WEB10K data set the output score of individual queries do not necessarily have the same scales across different models. The reason that we see a final normal-like distribution on this data set is that scores of individual queries have close to uniform distributions with different scales and result in a normal-like distribution once aggregated.

Since scores of different sub-models are on different scales, we initially tried using Borda count [18] as a rank aggregation technique. Each of the LambdaMART sub-models is used for generating a ranking of the test data. For each ranking, a score of 0 is assigned to the lowest ranked result, 1 to the next-to-lowest result, and so forth; then the total score of each result is computed over all the sub-models and the ensemble orders documents by this score. However, since Borda count converts real valued scores of documents to integer value ranks and then combines these ranks, the possibility of having ties increases significantly when averaging only have a few LambdaMART sub-models. Since tied documents would have to be ordered arbitrarily, this can significantly reduce the performance of the bagged model. Our experiments showed that Borda count works well when used to aggregate ranks of many model, but performs even worse than a single model when applied to only a few sub-models.

A better approach for combining the outputs of different sub-models is to normalize the scores of documents generated by each sub-model for each query. For each query we linearly scale the scores of its corresponding documents such that the document with the highest score will have a score of 1.0 and the document with the lowest score will have a score of 0.

#### 3.5.2 Overfitting Tolerance

As mentioned in section 2, prediction error can be reduced by reducing both bias and variance. When bagging single decision or regression trees, bagging typically results in higher accuracy if applied to unpruned trees. The reason is that unpruned trees are overfitted to their training data and therefore have low bias and high variance. Since bagging is a variance reduction technique, it can reduce this variance so that the final bagged model has both low bias and low variance and thus lower total prediction error. Similarly, we may get better results by bagging boosted tree

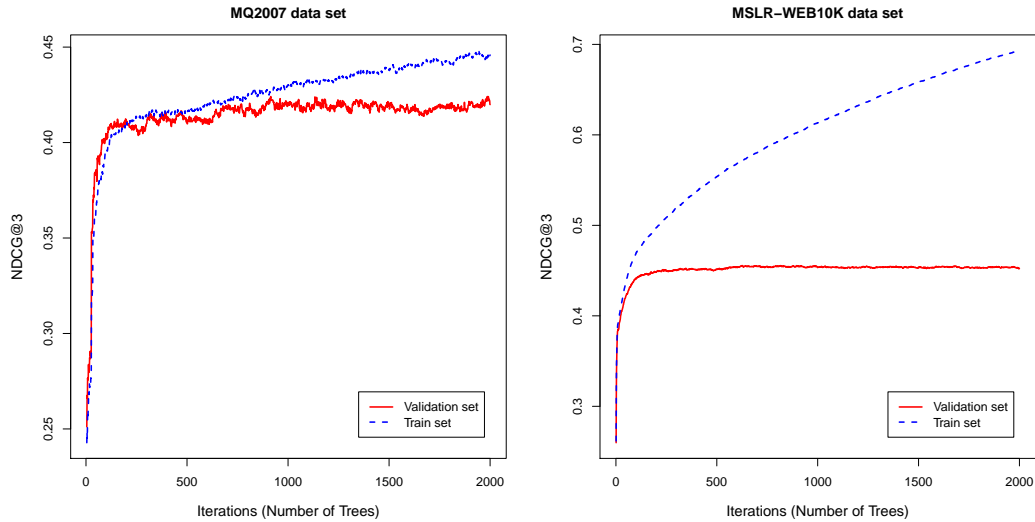


Figure 2: As more trees are added to the LamdaMART model train set accuracy begins to deviate from validation set accuracy. The appropriate number of iterations is determined by monitoring performance on the validation set.

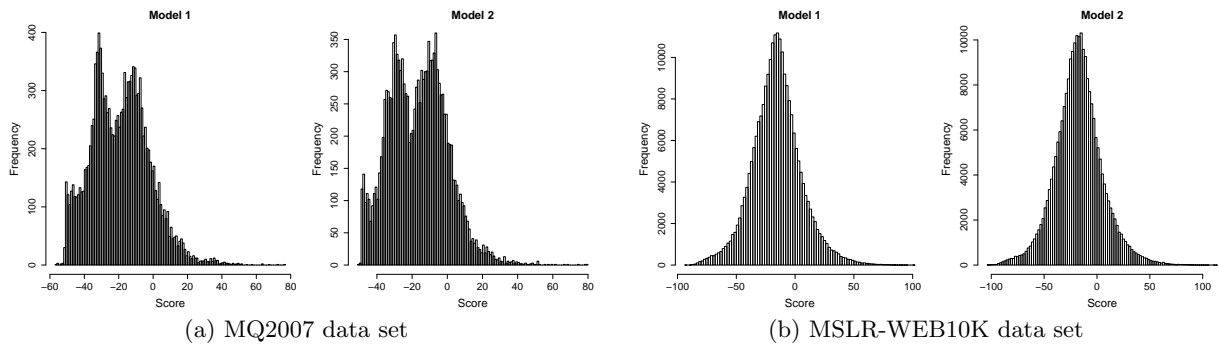


Figure 3: Distributions of validation scores generated from two models (per data set) that are trained on different random samples of training data.

models that have less bias but more variance. We test the effectiveness of this idea by adding an overfitting tolerance as a new parameter to the LamdaMART algorithm.

To generate overfitted LambdaMART models, we monitor the prediction accuracy on a separate validation set. Without overfitting, we would select the first  $N$  trees that results in the maximum prediction accuracy on the validation set. With overfitting tolerance, we add more trees after this maximum point until accuracy drop is more than a threshold value. We experimented with different threshold values and found that 2% worked well across the three data sets. As Figure 3 shows, it is possible that even after adding hundreds of additional trees, the drop in prediction accuracy on validation set is small. To prevent the size of the overfitted model from growing excessively large in this situation we allow a maximum of 250 trees to be added after the maximum point has been reached on the validation set. It is possible that allowing more overfitting would further improve the results from bagging, but at the expense of a much larger model.

## 4. RESULTS

BL-MART combines LambdaMART sub-models in order to achieve higher accuracy. There is a trade-off between increasing accuracy and model complexity: as we add more sub-models, the gain in accuracy is offset by the final model becoming too complex. It is important to balance the number of sub-models needed to achieve a reasonable gain in accuracy with the need to control complexity so that the models are still feasible to use in practice. To do this, we train BL-MART with different number of sub-models and evaluate its performance on the validation sets. We first create pools of LambdaMART models for each of the folds of the three data sets and then use these pools of models during the bagging process. We used pools of size 1000 models for TD2004 and MQ2007 data sets and 50 models for MSLR-WEB10K data set. The reason is that on the smaller data sets variance is higher and bagging needs to add more models before accuracy asymptotes.

We create different pools for overfitted and non-overfitted models. Overall, we need to train 10,000 LambdaMART models on each of the smaller data sets and 500 models on MSLR-WEB10K data set. We used a MapReduce cluster

for this purpose. Each mapper task is assigned a fold in one of the data sets. It then randomly selects 67% of the training queries from that fold and generates a LambdaMART model on this random sample. Once the model is created, it evaluates the scores of validation and test data corresponding to that fold and passes these scores to reducer tasks. Reducer tasks just dump the scores that they receive. It takes about 18 hours to generate output scores for these models on a cluster of 40 nodes.

To study the effectiveness of using overfitted models in the bagging process, we create bagged models of different sizes from overfitted and non-overfitted models and then compare their MAP. Since we use a pool of models and each bagged model is created by randomly selecting a subset of models from these pools, we repeat the random process of bagged model creation 100 times and compute average validation MAP of these bagged models to have more reliable results. Figure 5 shows the results confirming that using overfitted models results in better accuracy for the final bagged ensemble. On MSLR-WEB10K data set, we need to include more models in the bagged ensemble before overfitted models show better results.

We use the same results of Figure 5 for determining the number of models that should be included in the bagged ensemble. In order to have a balance between accuracy and model complexity, we picked 45 models on smaller data sets and 20 models on the larger data set.

To compare the performance of BL-MART model with the original LambdaMART model and other learning-to-rank algorithms, we create BL-MART models for each of the data sets. Since, we create bagged models by random selection of sub-models from our model pools, we repeat this process for 20 times and report average results. Similarly, we use 20 different random seeds for “LambdaMART with randomization” and report average results. The original LambdaMART algorithm is deterministic and we only need to run it once. Since LambdaMART code is not publicly available, we re-implemented it for our experiments.

## 4.1 Accuracy Analysis

Table 4 summarizes the results of our experiments analyzing the accuracy of BL-MART models in comparison to LambdaMART and other learning-to-rank algorithms. On TD2004 and MQ2007 all prior published results are included in the table for four metrics. For the new MSLR-WEB10K data set no other results have yet been published.

On TD2004 data set, BL-MART significantly outperforms LambdaMART on all four metrics. Also bagging overfitted models results in better performance on this data set. In comparison to other learning-to-rank algorithms, BL-MART performs best in terms of MAP and NDCG@5, but on NDCG@1 and NDCG@3 RankBoost and BagBoo are slightly better. It should be noted that we have limited the size of BL-MART to control complexity. For example, on this data set, BL-MART contains about 4,500 trees while BagBoo is using 1.1 million trees which is 250 times larger than our model.

On MQ2007 data set, BL-MART again significantly outperforms LambdaMART on all metrics and it also achieves the best results compared to other learning-to-rank algorithms which have reported their results on this data set.

On the new MSLR-WEB10K data set, BL-MART again improves LambdaMART performance on all of the metrics. However, on this data set the difference between bagging

of overfitted and non-overfitted models is not statistically significant.

Across the three problems and four metrics, BL-MART with overfitting improves accuracy an average of 2.6% when compared to LambdaMART with randomization.

While we have reported the best results (as far as we know) compared to other ranking algorithms on TD2004 and MQ2004 data sets, it should be noted that we did not tune the LambdaMART models for bagging. We tuned a single LambdaMART model and then used the same set of best parameters in the bagged model. It might be possible to get better results by bagging LambdaMART models which are generated from different set of parameters. For example, using more complex trees in sub-models might lead to overfitting which shows to be a desirable property for sub-models of a bagged ensemble. However, tuning the bagged model would have required much more experiments.

## 4.2 Variance Analysis

Variance can hurt in several ways: 1) by increasing the variance term in the bias/variance decomposition it reduces the expected accuracy of the trained models; 2) by increasing uncertainty it increases the number of experiments that must be run to determine which learning method and parameters yield the best results; and 3) it creates risk when a final model must be selected to deploy. All other things being equal (e.g. expected accuracy across multiple trials), the lower-variance learning method is preferred. For example, reducing variance by half reduces the number of experiments that must be run to pass a t-test by  $\sqrt{2}$ , and reduces the expected loss of the single deployed model compared to the expected loss of a typical model by a factor of 2. Because of this, learning methods that exhibit high variance can be difficult to work with, and learning methods that have lower variance are safer to deploy and easier to use for new feature development. Boosting often has relatively high variance. Bagging, however, is an effective variance reduction method. Thus we expect BL-MART to have significantly lower variance than LambdaMART.

In order to compare the variance of LambdaMART and BL-MART models, we use the MSLR-WEB10K data set. We randomly select 10 samples from the training data of Fold1 of this data set. Each sample contains a random selection of 67% of the training queries in this fold. We then train LambdaMART and BL-MART models on each of these samples and evaluate these models on the test data of this fold.

Table 5 shows the NDCG and MAP scores and their corresponding variance for each of the models. If we compare “LambdaMART with randomization” with “BL-MART with overfitting”, the following would be the reduction in variance for different metrics:

- NDCG@1: -67.3%
- NDCG@3: -18.8%
- Mean NDCG: -40.2%
- MAP: -57.1%

## 5. DISCUSSION

Wrapping bagging around LambdaMART boosting yields two distinct benefits: 1) it achieves accuracy comparable to

**Table 4: Evaluation results on three public learning-to-rank data sets.**  
(a) TD2004 data set

	NDCG@1	NDCG@3	NDCG@5	MAP
SVMmap [34]	0.2933	0.3035	0.3007	0.2049
RankSVM-Struct [20]	0.3467	0.3371	0.3192	0.2196
ListNet [9]	0.3600	0.3573	0.3325	0.2231
SmoothRank [11]	0.4000	0.3832	0.3555	0.2326
RankSVM [17]	0.4133	0.3467	0.3240	0.2237
AdaRank-MAP [33]	0.4133	0.3757	0.3602	0.2189
AdaRank-NDCG [33]	0.4267	0.3688	0.3514	0.1936
BoltzRank [30]	0.4767	0.3902	0.3635	0.2390
FRank [28]	0.4933	0.3875	0.3629	0.2388
RankBoost [13]	<b>0.5067</b>	<b>0.4295</b>	0.3878	0.2614
BagBoo [23]	<b>0.5067</b>	0.4080	0.3898	0.2499
LambdaMART	0.4267	0.3584	0.3266	0.2378
LambdaMART with randomization	0.4560	0.4033	0.3722	0.2513
BL-MART without overfitting	0.4947	0.4217	0.3886	0.2649
BL-MART with overfitting	0.4947	0.4270	<b>0.3948</b>	<b>0.2684</b>

(b) MQ2007 data set

	NDCG@1	NDCG@3	Mean NDCG	MAP
RankSVM-Struct [20]	0.4096	0.4063	0.4966	0.4645
ListNet [9]	0.4002	0.4091	0.4988	0.4652
AdaRank-MAP [33]	0.3821	0.3984	0.4891	0.4577
AdaRank-NDCG [33]	0.3876	0.4044	0.4914	0.4602
RankBoost [13]	0.4134	0.4072	0.5003	0.4662
CRR [25]	–	–	0.5000	0.4660
BagBoo [23]	0.4071	0.4176	–	0.4676
LambdaMART	0.4147	0.4119	0.5011	0.4660
LambdaMART with randomization	0.4137	0.4157	0.5035	0.4684
BL-MART without overfitting	0.4197	0.4217	0.5079	0.4726
BL-MART with overfitting	<b>0.4200</b>	<b>0.4224</b>	<b>0.5093</b>	<b>0.4731</b>

(c) MSLR-WEB10K data set

	NDCG@1	NDCG@3	Mean NDCG	MAP
LambdaMART	0.4580	0.4467	0.5693	0.3670
LambdaMART with randomization	0.4628	0.4487	0.5706	0.3684
BL-MART without overfitting	0.4640	0.4514	0.5720	0.3696
BL-MART with overfitting	<b>0.4642</b>	<b>0.4516</b>	<b>0.5729</b>	<b>0.3705</b>

**Table 5: Variance reduction in BL-MART models compared to LambdaMART models. The first column in each group is the mean accuracy. The second column in the variance of this accuracy measured across the trials.**

	NDCG@1		NDCG@3		Mean NDCG		MAP	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
LambdaMART	0.4484	$18 \times 10^{-6}$	0.4395	$9.1 \times 10^{-6}$	0.5640	$1.2 \times 10^{-6}$	0.3657	$0.9 \times 10^{-6}$
LambdaMART with randomization	0.4492	$22 \times 10^{-6}$	0.4421	$5.4 \times 10^{-6}$	0.5647	$1.4 \times 10^{-6}$	0.3665	$1.3 \times 10^{-6}$
BL-MART without overfitting	0.4516	$10 \times 10^{-6}$	0.4468	$7.8 \times 10^{-6}$	0.5675	$1.5 \times 10^{-6}$	0.3690	$0.9 \times 10^{-6}$
BL-MART with overfitting	0.4528	$7 \times 10^{-6}$	0.4471	$4.4 \times 10^{-6}$	0.5686	$0.8 \times 10^{-6}$	0.3703	$0.5 \times 10^{-6}$

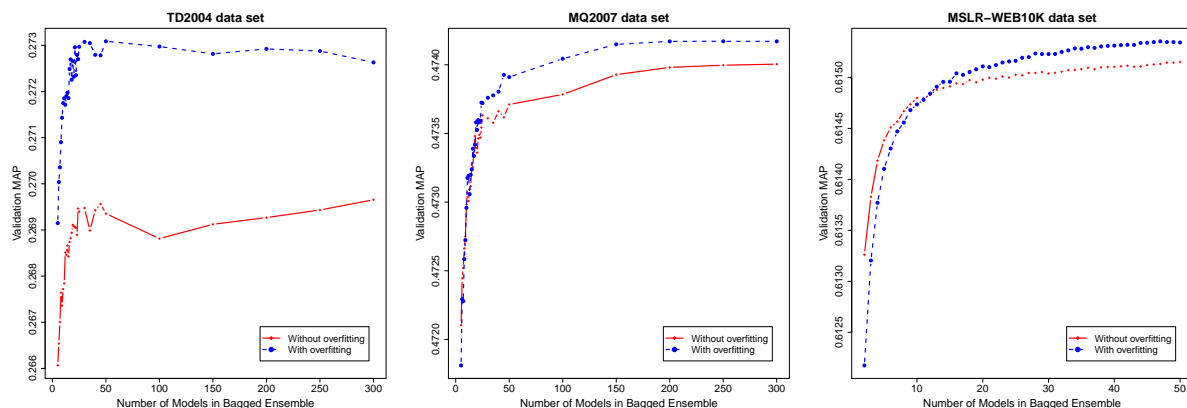


Figure 4: The effect of overfitting tolerance

and often better than the current state of the art in learning-to-rank; and 2) it achieves this very high accuracy while simultaneously reducing variance. Taken individually, each of these is a reason to be interested in BL-MART. Together, however, they represent a substantial step forward.

For those interested in delivering high accuracy search results at commercial search engines, the reduction in variance may be the more important result. In commercial search engines most gains come not from improved learning methods, but from developing new or improved features (and data) to feed into learning. Feature and data refinement requires frequent experimentation to determine if the refinement is better and should be released. Reduced learning variance makes these experiments easier and more reliable. Because bagging can easily be parallelized across multiple computers, it allows one to obtain low variance experimental results with little increase in wall clock time. That these results also will have state-of-the-art accuracy makes the method even more appealing.

The final output of our BL-MART method is a high precision and low variance bagged ensemble of models. However, in real-time applications such as search engines, it may not be feasible to use such large models. We are currently working on compression pruning techniques that allow dropping a large fraction of trees from the final model without significantly impacting the accuracy. This is possible because we find empirically that large shrinkage is necessary to prevent massive overfitting in LambdaMART models. When shrinkage is high, boosting sometimes needs to generate a sequence of nearly identical trees to do what it might have accomplished with one tree without shrinkage. Yet in other places shrinkage is critical and subsequent trees are quite different from each other. By deleting and reweighting trees *after* the full LambdaMART ensemble has been grown, we can determine post-facto which trees are redundant and contribute little to the model, and which trees are critical for the model to have high accuracy. The method is currently under development, but it looks like we can also exploit the fact that bagging boosted trees generates even more redundant trees that can safely be eliminated afterwards with little or no loss in accuracy. For example, in one set of experiments we are able to prune away 40–80% of the trees without significant loss in NDCG.

To run experiments, we developed a platform called *jforests* that not only implements LambdaMART and BL-MART,

but which supports many tree-based learning methods such as bagging, boosting, regression, classification, ranking, etc. The platform is written in such a way to make it easy to code additional tree-based methods with minimum effort. For example, AdaRank and RankBoost can each be implemented in less than a dozen lines of new code. Our code platform including the implementations of LambdaMART and BL-MART will be made publicly available. The *jforests* platform also supports running on MapReduce environments to allow parallelization of methods such as bagging.

## 6. CONCLUSIONS

We present new results for LambdaMART, a state-of-the-art learning-to-rank algorithm, on three public data sets. We show that wrapping bagging around a boosting-based ranking model can improve its performance while also significantly reducing model variance. In our experiments, bagged LambdaMART (BL-MART) increased NDCG@1, NDCG@3, Mean NDCG, and MAP on all three test problems compared to un-bagged LambdaMART. Moreover, bagging reduced variance an average of 46% across all metrics on the MSLR-WEB10K data set on which we measured variance. Most of the ideas and methods reported in this paper are general and not limited specifically to ranking. For example, our finding that overfitting boosted models that will be bagged improves accuracy can be used in other classification and regression problems to further improve performance.

## Acknowledgments

Authors would like to thank Amazon for a research grant that allowed us to use their MapReduce cluster. This work has been also partially supported by NSF grant OCI-074806.

## 7. REFERENCES

- [1] Microsoft learning to rank datasets. <http://research.microsoft.com/en-us/projects/mslr/>.
- [2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [3] L. Breiman. Bagging predictors. *Mach. Learn.*, 24:123–140, August 1996.
- [4] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. 10.1023/A:1010933404324.

- [5] L. Breiman. Using iterated bagging to debias regressions. *Mach. Learn.*, 45:261–277, December 2001.
- [6] C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. Technical report, Microsoft Research Technical Report MSR-TR-2010-82, 2010.
- [7] C. J. C. Burges, R. Ragno, and Q. V. Le. Learning to rank with nonsmooth cost functions. In *NIPS*, pages 193–200, 2006.
- [8] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.
- [9] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 129–136, New York, NY, USA, 2007. ACM.
- [10] O. Chapelle, Y. Chang, and T.-Y. Liu. The Yahoo! learning to rank challenge. <http://learningtorankchallenge.yahoo.com>, 2010.
- [11] O. Chapelle and M. Wu. Gradient descent optimization of smoothed information retrieval metrics. *Inf. Retr.*, 13:216–235, June 2010.
- [12] K. Crammer and Y. Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, pages 641–647. MIT Press, 2001.
- [13] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, 2003.
- [14] J. H. Friedman. Stochastic gradient boosting. Technical report, Technical report, Dept. Statistics, Stanford Univ., 1999.
- [15] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.
- [16] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Comput.*, 4:1–58, January 1992.
- [17] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. In A. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 115–132, Cambridge, MA, 2000. MIT Press.
- [18] T. K. Ho, J. Hull, and S. Srihari. Decision combination in multiple classifier systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(1):66–75, Jan. 1994.
- [19] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48, New York, NY, USA, 2000. ACM.
- [20] T. Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '06*, pages 217–226, New York, NY, USA, 2006. ACM.
- [21] P. Li, C. J. C. Burges, and Q. Wu. Mcrank: Learning to rank using multiple classification and gradient boosting. In *NIPS*, 2007.
- [22] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.
- [23] D. Y. Pavlov, A. Gorodilov, and C. A. Brunk. BagBoo: a scalable hybrid bagging-the-boosting model. In *Proceedings of the 19th ACM international conference on Information and knowledge management, CIKM '10*, pages 1897–1900, New York, NY, USA, 2010. ACM.
- [24] T. Qin, T.-Y. Liu, J. Xu, and H. Li. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13:346–374, 2010. 10.1007/s10791-009-9123-y.
- [25] D. Sculley. Combined regression and ranking. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '10*, pages 979–988, New York, NY, USA, 2010. ACM.
- [26] D. Sorokina, R. Caruana, and M. Riedewald. Additive groves of regression trees. In *Proceedings of the 18th European conference on Machine Learning, ECML '07*, pages 323–334, Berlin, Heidelberg, 2007. Springer-Verlag.
- [27] Y. L. Suen, P. Melville, and R. J. Mooney. Combining bias and variance reduction techniques for regression trees. In *In Proceedings of the European Conference on Machine Learning (ECML'05)*, pages 741–749, 2005.
- [28] M.-F. Tsai, T.-Y. Liu, T. Qin, H.-H. Chen, and W.-Y. Ma. FRank: a ranking method with fidelity loss. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390, New York, NY, USA, 2007. ACM.
- [29] G. Valentini and T. G. Dietterich. Low bias bagged support vector machines. In *Proceedings of the International Conference on Machine Learning, ICML '03*, pages 752–759. Morgan Kaufmann, 2003.
- [30] M. N. Volkovs and R. S. Zemel. Boltzrank: learning to maximize expected ranking gain. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09*, pages 1089–1096, New York, NY, USA, 2009. ACM.
- [31] G. I. Webb. Multiboosting: A technique for combining boosting and wagging. *Mach. Learn.*, 40:159–196, August 2000.
- [32] Q. Wu, C. Burges, K. Svore, and J. Gao. Ranking, boosting and model adaptation. Technical report, Microsoft Technical Report MSR-TR-2008-109, 2008.
- [33] J. Xu and H. Li. AdaRank: a boosting algorithm for information retrieval. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, New York, NY, USA, 2007. ACM.
- [34] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A support vector method for optimizing average precision. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR '07*, pages 271–278, New York, NY, USA, 2007. ACM.