Learning to Rank Using Classification and Gradient Boosting

Ping Li *
Department of Statistics
Stanford University
Stanford, CA 94305
pingli@cs.stanford.edu

Christopher J.C. Burges
Microsoft Research
Microsoft Corporation
Redmond, WA 98052
cburges@microsoft.com

Qiang Wu
Microsoft Research
Microsoft Corporation
Redmond, WA 98052
qiangwu@microsoft.com

Abstract

We cast the ranking problem as (1) multiple classification (2) multiple ordinal classification, which lead to computationally tractable learning algorithms for relevance ranking in Web search. We consider the DCG criterion (discounted cumulative gain), a standard quality measure in information retrieval. Our approach is motivated by the fact that perfect classifications naturally result in perfect DCG scores and the DCG errors are bounded by classification errors. We propose using the *Expected Relevance* to convert the class probabilities into ranking scores. The class probabilities are learned using a gradient boosting tree algorithm. Evaluations on large-scale datasets show that our approach can improve *LambdaRank* [5] and the regressions-based ranker [6], in terms of the (normalized) DCG scores.

1 Introduction

The general *ranking* problem has widespread applications including commercial search engines and recommender systems. In this study, we develop a computationally tractable learning algorithm for the general ranking problem; and we present our approach in the context of ranking in Web search.

For a given user input query, a commercial search engine returns many pages of URLs, in an order determined by the underlying proprietary ranking algorithm. The quality of the returned results are largely evaluated on the URLs displayed in the very first page. The type of ranking problem in this study is sometimes referred to as *dynamic ranking* (or simply, just *ranking*), because the URLs are dynamically ranked (in real-time) according to the specific user input query. This is different from the query-independent *static ranking* based on, for example, "page rank" [3] or "authorities and hubs" [11], which may, at least conceptually, serve as an important "feature" for dynamic ranking or to guide the generation of a list of URLs fed to the dynamic ranker.

There are two main categories of ranking algorithms. A popular scheme is based on learning the pairwise preference, for example, *RankNet* [4], *LambdaRank* [5], or *RankBoost* [7]. Both *LambdaRank* and *RankNet* used neural nets to learn the pairwise preference function.[1] *RankNet* used a cross-entropy type of loss function and *LambdaRank* directly used a modified gradient of the cross-entropy loss function. Another scheme is the regression-based ranking [6]. [6] considered the DCG measure (discounted cumulative gain) [10] and showed that the DCG errors are bounded by the regression errors. Note that the pair-based rankers [4, 5] were also evaluated in terms of DCG scores.

*Much of the work was conducted while Ping Li was an intern at Microsoft in the summer of 2006.

[1] In fact LambdaRank supports any preference function, although the reported results in [5] are for pairwise.

In this study, we also consider the DCG measure. From the definition of DCG, it appears more direct to cast the ranking problem as multiple classification as opposed to regression. In order to convert the classification results into ranking scores, we propose a simple and stable mechanism by using the *Expected Relevance*. Our evaluations on large-scale proprietary datasets demonstrate the superiority of the classification-based ranker over both the regression-based and pair-based schemes.

2 Discounted Cumulative Gain (DCG)

For an input query, the ranker returns n ordered URLs. Suppose the URLs fed to the ranker are originally ordered $\{1, 2, 3, ..., n\}$. The ranker will output a permutation mapping $\pi : \{1, 2, 3, ..., n\} \rightarrow \{1, 2, 3, ..., n\}$. We denote the inverse mapping by $\sigma_i = \sigma(i) = \pi^{-1}(i)$.

The DCG score is computed from the relevance levels of the n URLs as

$$\text{DCG} = \sum_{i=1}^{n} c_{[i]} \left(2^{y_{\sigma_i}} - 1\right) = \sum_{i=1}^{n} c_{[\pi_i]} \left(2^{y_i} - 1\right), \tag{1}$$

where $[i]$ is the rank order, and $y_i \in \{1, 2, 3, 4, 5\}$ is the *relevance* level of the ith URL in the original (pre-ranked) order. $y_i = 5$ corresponds to a "perfect" relevance and $y_i = 1$ corresponds to a "poor" relevance. For generating training datasets, human judges have manually labeled a large number of queries and URLs. In this study, we assume these labels are "gold-standard."

In the definition of DCG, $c_{[i]}$, which is a non-increasing function of i, is typically set as

$$c_{[i]} = \frac{1}{\log(1+i)}, \quad \text{if } i \leq L, \quad \text{and } c_{[i]} = 0, \text{ if } i > L, \tag{2}$$

where L is the "truncation level" and is typically set to be $L = 10$, to reflect the fact that the search quality of commercial search engines is mainly determined by the URLs displayed in the first page.

It is a common practice to normalize the DCG scores to the interval $[0,\ 1]$ for each query and report the normalized DCG score averaged over all queries. The normalized DCG scores are often called "NDCG." Suppose there are in total N_Q queries in a dataset. The individual NDCG for the jth query (NDCG_j) and the final NDCG of the whole dataset (NDCG_F) are defined as

$$\text{NDCG}_j = \frac{\text{DCG}_j}{\text{DCG}_{j,g}}, \qquad \text{NDCG}_F = \frac{1}{N_Q} \sum_{j=1}^{N_Q} \text{NDCG}_j, \tag{3}$$

where $\text{DCG}_{j,g}$ is the maximum possible (or "gold standard") DCG score of the jth query.

Note that, in Section 6, we will report NDCG_F in terms of the percentage values, i.e., 72.1 (%) instead of 0.721, to remind the readers that even an improvement of 1.0 (%) NDCG point is often considered significant, especially for commercial search engines.

3 Learning to Rank Using Classification

The definition of DCG suggests that we can cast the ranking problem naturally as multiple classification (i.e., $K = 5$ classes), because obviously perfect classifications will lead to perfect DCG scores. While the DCG criterion is non-convex and non-smooth, classification is very well-studied and many computationally tractable algorithms are available.

We should also mention that one does not really need perfect classifications in order to produce perfect DCG scores. For example, suppose within a query, the URLs are all labeled level 2 or higher by some gold-standard. If a classification algorithm always classifies the URLs one level lower (i.e., URLs labeled level 5 are classified as level 4, and so on), then we will still have the perfect DCG score even though the classification "error" is 100% for this query. This phenomenon to an extent, may provide some additional "safety cushion" for casting ranking as classification.

[6] cast ranking as regression[2] and showed that the DCG errors are bounded by regression errors. It appears to us that the regression-based approach is less direct and possibly also less accurate than our

[2] [6, Section 2.4] commented that one could cast the ranking problem as multiple classification with n classes (instead of $K = 5$ classes). Recall n is the total number of returned URLs for a given query.

classification-based proposal. For example, it is well-known that, although one can use regression for classification, it is often better to use logistic regression especially for multiple classification [8].

3.1 Bounding DCG Errors by Classification Errors

Following [6, Theorem 2], we show that the DCG errors can be bounded by classification errors.

For a permutation mapping π, the error is DCG_g - DCG_π. One simple way to obtain the perfect DCG_g is to rank the URLs directly according to the gold-standard relevance levels. That is, all URLs with relevance level $k+1$ are ranked higher than those with relevance level $\leq k$; and the URLs with the same relevance levels are arbitrarily ranked without affecting DCG_g. We denote the corresponding permutation mapping also by g.

Lemma 1 *Given n URLs, originally ordered as $\{1, 2, 3, ..., n\}$. Suppose a classifier assigns a relevance level $\hat{y}_i \in \{1, 2, 3, 4, 5\}$ to the ith URL, for all n URLs. A permutation mapping π ranks the URLs according to $\hat{y}_i$, i.e., $\pi(i) < \pi(j)$ if $\hat{y}_i > \hat{y}_j$, and, URL i and URL j are arbitrarily ranked if $\hat{y}_i = \hat{y}_j$. The corresponding DCG error is bounded by the square root of the classification error,*

$$DCG_g - DCG_\pi \leq 30\sqrt{2}\left(\sum_{i=1}^n c_{[i]}^2\right)^{1/2}\left(\sum_{i=1}^n 1_{y_i \neq \hat{y}_i}\right)^{1/2}. \tag{4}$$

Proof:

$$\begin{aligned}
DCG_\pi =& \sum_{i=1}^n c_{[\pi_i]}\left(2^{y_i}-1\right) = \sum_{i=1}^n c_{[\pi_i]}\left(2^{\hat{y}_i}-1\right) + \sum_{i=1}^n c_{[\pi_i]}\left(2^{y_i}-2^{\hat{y}_i}\right)\\
\geq& \sum_{i=1}^n c_{[g_i]}\left(2^{\hat{y}_i}-1\right) + \sum_{i=1}^n c_{[\pi_i]}\left(2^{y_i}-2^{\hat{y}_i}\right)\\
=& \sum_{i=1}^n c_{[g_i]}\left(2^{y_i}-1\right) - \sum_{i=1}^n c_{[g_i]}\left(2^{y_i}-2^{\hat{y}_i}\right) + \sum_{i=1}^n c_{[\pi_i]}\left(2^{y_i}-2^{\hat{y}_i}\right)\\
=& DCG_g + \sum_{i=1}^n \left(c_{[\pi_i]} - c_{[g_i]}\right)\left(2^{y_i}-2^{\hat{y}_i}\right).
\end{aligned}$$

Note that $\sum_{i=1}^n c_{[\pi_i]}\left(2^{\hat{y}_i}-1\right) \geq \sum_{i=1}^n c_{[g_i]}\left(2^{\hat{y}_i}-1\right)$. *Therefore,*

$$\begin{aligned}
&DCG_g - DCG_\pi \leq \sum_{i=1}^n \left(c_{[g_i]} - c_{[\pi_i]}\right)\left(2^{y_i}-2^{\hat{y}_i}\right)\\
\leq& \left(\sum_{i=1}^n \left(c_{[g_i]} - c_{[\pi_i]}\right)^2\right)^{1/2}\left(\sum_{i=1}^n \left(2^{y_i}-2^{\hat{y}_i}\right)^2\right)^{1/2} \leq \left(2\sum_{i=1}^n c_{[i]}^2\right)^{1/2} 30\left(\sum_{i=1}^n 1_{y_i\neq\hat{y}_i}\right)^{1/2}.
\end{aligned}$$

Note that $\sum_{i=1}^n c_{[\pi_i]}^2 = \sum_{i=1}^n c_{[g_i]}^2 = \sum_{i=1}^n c_{[i]}^2$, *and* $2^5 - 2^1 = 30$.

Thus, we can minimize the classification error $\sum_{i=1}^n 1_{y_i\neq\hat{y}_i}$ as a surrogate for minimizing the DCG errors. Of course, since the classification error itself is non-convex and non-smooth, we need to seek other surrogate loss functions.

3.2 Input Data for Classifications

A training dataset contains N_Q queries. The jth query corresponds to n_j URLs; each URL is manually labeled by one of the 5 relevance levels. Engineers have developed methodologies to construct "features" by combining the query and URLs, but the details are usually "trade secret."

One important aspect in designing features, at least for the convenience of using traditional machine learning algorithms, is that these features should be comparable across queries. For example, one (artificial) feature could be the number of times the query appears in the Web page, which is comparable across queries. Both pair-based rankers and regression-based rankers implicitly made this assumption, as they tried to learn a single rank function for all queries using the same set of features.

Thus, after we have generated feature vectors by combining the queries and URLs, we can create a "training data matrix" of size $N \times P$, where $N = \sum_{j=1}^{N_Q} n_j$ is the total number of "data points" (i.e., Query+URL) and P is the total number of features. This way, we can use the traditional machine learning notation $\{y_i,\ \mathbf{x}_i\}_{i=1}^N$ to denote the training dataset. Here $\mathbf{x}_i \in \mathbf{R}^P$ is the ith feature vector in P dimensions; and $y_i \in \{1, 2, 3, 4, K = 5\}$ is the class (relevance) label of the ith data point. Here we slightly overload the notation y_i but its meaning should be clear from the context.

3.3 From Classification to Ranking

Although perfect classifications lead to perfect DCG scores, in reality, we will need a mechanism to convert (imperfect) classification results into ranking scores.

One possibility is already mentioned in Lemma 1. That is, we classify each data point into one of the 5 classes and rank the data points according to the class labels (data points with the same labels are arbitrarily ranked). This suggestion, however, will lead to highly unstable ranking results.

Our proposed solution is very simple. We first learn the class probabilities by some *soft classification* algorithm and then score each data point (query+URL) according to the *Expected Relevance*.

Recall we assume a training dataset $\{y_i, \mathbf{x}_i\}_{i=1}^N$, where the class label $y_i \in \{1, 2, 3, 4, K = 5\}$. We learn the class probabilities $p_{i,k} = \mathbf{Pr}(y_i = k)$, denoted by $\hat{p}_{i,k}$, and define a scoring function as

$$S_i = \sum_{k=1}^{K} \hat{p}_{i,k} T(k), \tag{5}$$

where $T(k)$ is some monotone (increasing) function of the relevance level k. Once we have computed the scores S_i for all data points, we can then sort the data points within each query by the descending order of S_i. This approach is apparently sensible and highly stable. In fact, we experimented with both $T(k) = k$ and $T(k) = 2^k$; the performance difference in terms of the NDCG scores was negligible, although $T(k) = k$ appeared to be a better (more robust) choice than $T(k) = 2^k$. In this paper, the reported experimental results were based on $T(k) = k$.

When $T(k) = k$, the scoring function S_i is the *Expected Relevance*. Note that any monotone transformation on S_i (e.g., $2^{S_i} - 1$) will not change the ranking results. Consequently, the ranking results are not affected by any affine transformation on $T(k)$, $aT(k) + b$, $(a > 0)$, because

$$\sum_{k=1}^{K} p_{i,k} \left(a \times T(k) + b\right) = a \times \left(\sum_{k=1}^{K} p_{i,k} T(k)\right) + b, \qquad \text{since } \sum_{k=1}^{K} p_{i,k} = 1. \tag{6}$$

3.4 The Boosting Tree Algorithm for Learning Class Probabilities

For multiple classification, we consider the following common (e.g., [8, 9]) surrogate loss function

$$\sum_{i=1}^{N} \sum_{k=1}^{K} -\log(p_{i,k}) 1_{y_i = k}. \tag{7}$$

One might ask why not weight the loss function according to (e.g.,) the sample relevance levels. In fact we experimented with various weighting schemes but none of them outperformed (7).

We implemented a boosting tree algorithm for learning class probabilities $p_{i,k}$, and use basically the same implementation later for regression as well as multiple ordinal classification.

The following pseudo code in Algorithm 1 for multiple classification is taken from [9, Algorithm 6], although the presentation is slightly different.

There are three main parameters. M is the total number of boosting iterations, J is the tree size (number of terminal nodes), and ν is the shrinkage coefficient. Fortunately, as commented in [9] and verified in our experiments, the performance of the algorithm is not sensitive to these parameters, which is a significant advantage. We will comment more on these parameters in Section 6.

In Algorithm 1, Line 5 contains most of the implementation work, i.e., building the regression trees with J terminal nodes. Appendix A describes an efficient implementation for building the trees.

Algorithm 1 The boosting tree algorithm for multiple classification

0: $\tilde{y}_{i,k} = 1$, if $y_i = k$, and $\tilde{y}_{i,k} = 0$ otherwise.
1: $F_{i,k} = 0$, $k = 1$ to K, $i = 1$ to N
2: For $m = 1$ to M Do
3: For $k = 1$ to K Do
4: $p_{i,k} = \exp(F_{i,k}) / \sum_{s=1}^{K} \exp(F_{i,s})$
5: $\{R_{j,k,m}\}_{j=1}^{J} = J$-terminal node regression tree for $\{\tilde{y}_{i,k} - p_{i,k},\ \mathbf{x}_i\}_{i=1}^{N}$
6: $\beta_{j,k,m} = \frac{K-1}{K} \frac{\sum_{\mathbf{x}_i \in R_{j,k,m}} \tilde{y}_{i,k} - p_{i,k}}{\sum_{\mathbf{x}_i \in R_{j,k,m}} (1-p_{i,k}) p_{i,k}}$
7: $F_{i,k} = F_{i,k} + \nu \sum_{j=1}^{J} \beta_{j,k,m} 1_{\mathbf{x}_i \in R_{j,k,m}}$
8: End
9: End

4 Multiple Ordinal Classification to Further Improve Ranking

We can further improve our classification-based ranking scheme by taking into account the natural orders among the class labels.

A common approach for multiple ordinal classification is to learn the cumulative probabilities $\mathbf{Pr}(y_i \leq k)$ instead of the class probabilities $\mathbf{Pr}(y_i = k) = p_{i,k}$. We suggest a simple method similar to the so-called cumulative logits approach known in statistics [1, Section 7.2.1].

We first partition the training data points into two groups: $\{y_i \geq 5\}$ and $\{y_i \leq 4\}$. Now we have a binary classification problem and hence we can use exactly the same boosting tree algorithm for multiple classification. Thus we can learn $\mathbf{Pr}(y_i \leq 4)$ easily. We can similarly partition the data and learn $\mathbf{Pr}(y_i \leq 3)$, $\mathbf{Pr}(y_i \leq 2)$, and $\mathbf{Pr}(y_i \leq 1)$, separately. After we have the cumulative probabilities, we can infer the class probabilities

$$p_{i,k} = \mathbf{Pr}(y_i = k) = \mathbf{Pr}(y_i \leq k) - \mathbf{Pr}(y_i \leq k-1). \tag{8}$$

Once we have learned the class probabilities, we can again use the *Expected Relevance* to compute the ranking scores and sort the URLs.

5 Regression-based Ranking Using Boosting Tree Algorithm

With slight modifications, the boosting tree algorithm can be used for regressions. Recall the input data are $\{y_i, \mathbf{x}_i\}_{i=1}^{N}$, where $y_i \in \{1, 2, 3, 4, 5\}$. [6] suggested regressing the feature vectors $\mathbf{x}_i$ on the response values $2^{y_i} - 1$. [6] also mentioned weighting the samples according to the importance of the relevance levels, although we did not notice any performance difference in our experiments.

Algorithm 2 The boosting tree algorithm for regressions

0: $\tilde{y}_i = 2^{y_i} - 1$
1: $S_i = \frac{1}{N} \sum_{s=1}^{N} \tilde{y}_s,\ i = 1$ to N
2: For $m = 1$ to M Do
5: $\{R_{j,m}\}_{j=1}^{J} = J$-terminal node regression tree for $\{\tilde{y}_i - S_i,\ \mathbf{x}_i\}_{i=1}^{N}$
6: $\beta_{j,m} = \text{mean}_{\mathbf{x}_i \in R_{j,m}} \tilde{y}_i - S_i$
7: $S_i = S_i + \nu \sum_{j=1}^{J} \beta_{j,m} 1_{\mathbf{x}_i \in R_{j,m}}$
9: End

Algorithm 2 implements the least-square boosting tree algorithm. The pseudo code is similar to [9, Algorithm 3] by replacing the (l_1) least absolute deviation (LAD) loss with the (l_2) least square loss. In fact, we also implemented the LAD boosting tree algorithm but we found the performance was considerably worse than the least-square tree boost.

Once we have learned the values for S_i in Algorithm 2, we can use them directly as the ranking scores to order the data points within each query.

6 Experimental Results

We present the evaluations of 4 ranking algorithms (*LambdaRank* (using a two-layer, ten-hidden node neural net), regression-based, classification-based, and ordinal-classification-based) on 3 datasets, including one artificial dataset and two Web search datasets (Web-1 and Web-2). The artificial dataset and Web-1 are the same datasets used in [5, Sections 6.3.1, 6.4].

The artificial dataset was meant to remove any variance caused by the quality of features and/or relevance labels. The data were generated from random cubic polynomials, with 50 features, 50 URLs per query, and 10K/5K/10K queries for train/valid/test. We will report the NDCG scores on the 10K test queries for all 4 rankers.

The Web search data *Web-1* has 367 features, with on average 26.1 URLs per query, and 10K/5K/10K queries for train/valid/test. We will report the NDCG scores on the 10K test queries for all 4 rankers.

The Web search data *Web-2* has 355 features, with on average 100 URLs per query, and about 16K queries for training and about 10K queries for testing. We will report the NDCG scores on these (about) 10K test queries. Since *LambdaRank* needs a validation set for the stopping rule in training the neural net, we further divide the test dataset equally into two sets, one set for validation and another for testing. We train the neural net twice by switching the validation set with the test set. In the end, we report the averaged test NDCG errors for *LambaRank* for Web-2. For other 3 rankers using boosting trees, since there is no obvious over-fitting observed, we simply report the NDCG scores on the (about) 10K test queries.

6.1 The Parameters: M, J, ν

There are three main parameters in the boosting tree algorithm. M is the total number of iterations, J is the number of terminal nodes in each tree, and ν is the shrinkage factor. Our experiments verify that these parameters are not sensitive as long as they are within some "reasonable" ranges [9] .

We fix $\nu = 0.05$ ([9] suggested $\nu \leq 0.1$) for all three datasets. The number of terminal nodes, J, should be reasonably big (but not too big) when the dataset is large with a large number of features, because the tree has to be deep enough to consider higher-order interactions [9]. We let $J = 10$ for the artificial dataset and Web-1, and $J = 20$ for Web-2.

With these values of J and ν, we did not observe obvious over-fitting even for a very large number of boosting iterations M. We will report the results with $M = 1000$ for the artificial data and Web-1, and $M = 1500$ for Web-2.

6.2 The NDCG Results at Truncation Level $L = 10$

Table 1 lists the NDCG results (both the mean and standard deviation, in percentages ($\%$)) for all 3 datasets and all 4 ranking algorithms, evaluated at the truncation level $L = 10$.

For both the artificial data and Web-1, to compare with the corresponding results in [5], we report the NDCG scores on the test datasets (10K queries each). For Web-2, we report the NDCG scores on the (about) 10K test queries.

Table 1: The test NDCG scores produced by 4 rankers on 3 datasets. The average NDCG scores are presented in percentages ($\%$) with the standard deviations in the parentheses.

	Ordinal Classification	Classification	Regression	LambdaRank
Artificial ($M = 1000$, $J = 10$)	85.0 (9.5)	83.7 (9.9)	82.9 (10.2)	74.9 (12.6)
Web-1 ($M = 1000$, $J = 10$)	72.4 (24.1)	72.1 (24.1)	71.7 (24.4)	71.2 (24.5)
Web-2 ($M = 1500$, $J = 20$)	72.9 (24.1)	72.7 (24.1)	72.0 (24.5)	72.1 (24.4)

The NDCG scores indicate that the ordinal-classification-based ranker always produces the best results and the classification-based ranker is always the second best. The regression-based ranker

outperforms *LambadRank* (considerably) on the artificial data and also on the dataset Web-1. For Web-2, the regression-based ranker has similar performance as *LambdaRank*.

For the artificial data, all other 3 rankers exhibit very large improvements over *LambaRank*. This is probably due to the fact that the artificial data are generated noise-free and hence the flexible (with high capacity) rankers using boosting tree algorithms tend to fit the data very well.

For the Web search datasets, Web-1 and Web-2, we can conduct a crude t-test to assess the significance of the improvements. Using a standard deviation = 24 and a sample size = 10000, we have $1.96\frac{24}{\sqrt{10000}} = 0.47$. Thus, roughly speaking, we could consider the difference is "significant" if it is larger than 0.47 ($\%$) NDCG point.

For Web-2, Figure 1 plots the average test NDCG scores as functions of the number of boosting iterations for both the classification-based ranker and the regression-based ranker, verifying that no over-fitting is observed.

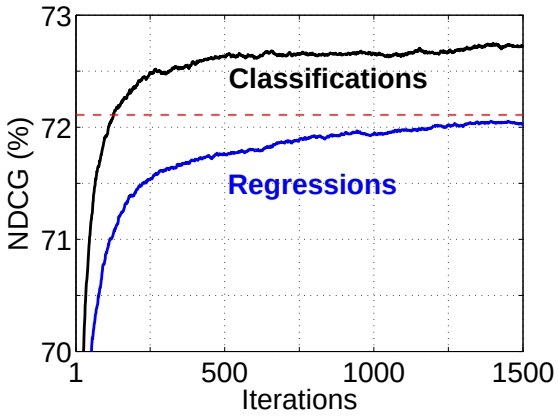


Figure 1: For Web-2, we plot the average test NDCG scores (at truncation level $L = 10$) as functions of the number of boosting iterations. The dashed horizontal line stands for the NDCG score (72.1 $\%$) produced by *LambdaRank*. We can see that after about 200 boosting iterations, the classification-based ranker starts to outperform *LambdaRank*. We do not observe obvious "over-fitting" even after 1500 iterations. The performance of the regression-based ranker approaches *LambdaRank* after 1500 iterations.

6.3 The NDCG Results at Various Truncation Levels ($L = 1$ to 10)

For the artificial dataset and Web-1, [5] also reported the NDCG scores at various truncation levels, $L = 1$ to 10. To make the comparisons more convincing, we also report similar results for the artificial dataset and Web-1, in Figure 2. For a better clarity, we plot the standard deviations separately from the averages. Figure 2 verifies that the improvements shown in Table 1 are not only true for $L = 10$ but also (essentially) true for smaller truncation levels.

7 Conclusion

The ranking problem has become an important topic in machine learning, partly due to its widespread applications in many decision-making processes especially in commercial search engines. In one aspect, the ranking problem is difficult because the measures of rank quality are usually based on sorting, which is not directly optimizable (at least not efficiently). On the other hand, one can cast ranking into various classical learning tasks such as regression and classification.

The proposed classification-based ranking scheme is motivated by the fact that perfect classifications lead to perfect DCG scores and the DCG errors are bounded by the classification errors. It appears natural that the classification-based ranker is more direct and should work better than the regression-

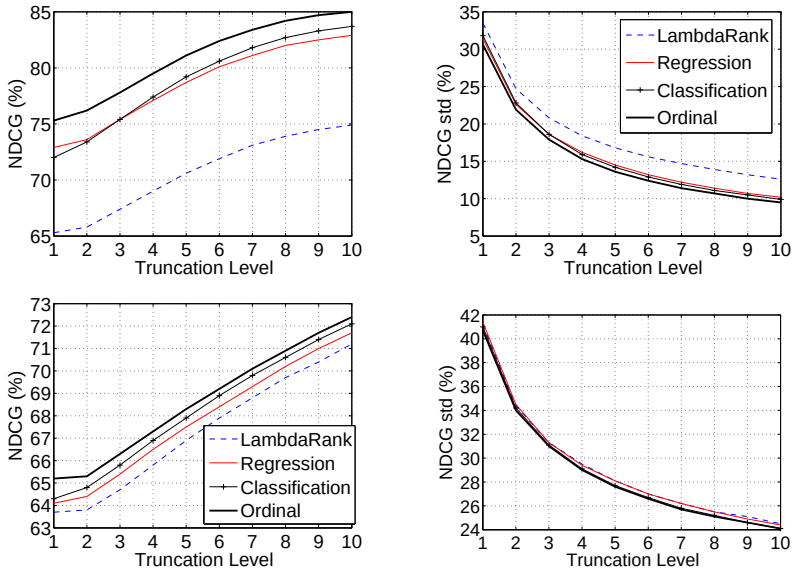


Figure 2: The NDCG scores at truncation levels $L = 1$ to 10. Upper Panels: the artificial data. Bottom Panels: Web-1. Left Panels: average NDCG. Right Panels: standard deviations.

based ranker suggested in [6]. To convert classification results into ranking, we propose a simple and stable mechanism by using the *Expected Relevance*, computed from the learned class probabilities.

To learn the class probabilities, we implement a boosting tree algorithm for multiple classification and we use the same implementation for multiple ordinal classification and regression. Since commercial proprietary datasets are usually very large, an adaptive quantization-based approach efficiently implements the boosting tree algorithm, which avoids sorting and has low memory cost.

Our experimental results have demonstrated that the proposed classification-based ranker outperforms both the regression-based ranker and the pair-based *LambdaRank*. The performance is further improved using multiple ordinal classification.

In a summary, we regard the proposed multiple and multiple ordinal classification-based ranking algorithm (retrospectively) simple, robust, and capable of producing quality ranking results.

Acknowledgment

Much of the work was conducted while Ping Li was an intern at Microsoft Research in the summer of 2006. Ping Li would like to thank Robert Ragno.

Appendix

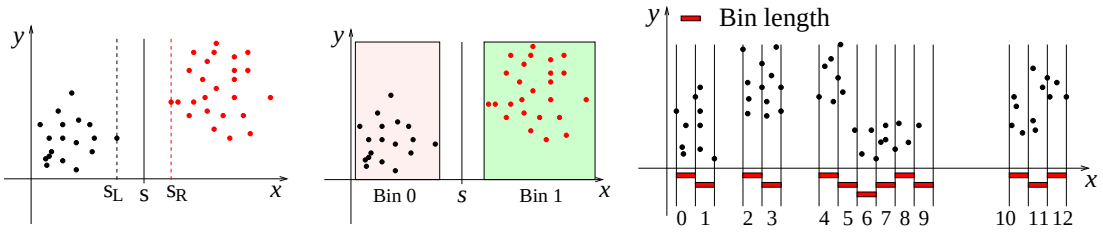
A An Efficient Implementation for Building Regression Trees

We use the standard regression tree algorithm [2], which recursively splits the training data points into two groups on the current "best" feature that will reduce the mean square errors (MSE) the most. Efficient (in both time and memory) implementation needs some care. The standard practice [9] is to pre-sort all the features. Then after every split, carefully keep track of the indexes of the data points and the sorted orders in all other features for the next split.

We suggest a simpler and more efficient approach, by taking advantage of some properties of the boosting tree algorithm. While the boosting tree algorithm is well-known to be robust and also accurate, an individual tree has limited predictive power and usually can be built quite crudely.

When splitting on one feature, Figure 3(a) says that sometimes the split point can be chosen within a certain range without affecting the accuracy (i.e., the reduced MSE due to the split). In Figure 3(b),

we bin (quantize) the data points into two (0/1) levels on the horizontal (i.e., feature) axis. Suppose we choose the quantization as shown in the Figure 3(b), then the accuracy will not be affected either.



(a) (b) (c)

Figure 3: Splitting on one feature (x) is determined by the mean square error (MSE) criterion. That is, we seek a split point s on the feature (x) such that after the splitting, the MSE (in the y axis) of the data points at the left plus the MSE at the right is reduced the most, compared to the original MSE before splitting. Panel (a) suggests that in some cases we can choose s in a range (within s_L and s_R) without affecting the reduced MSE. Panel (b) suggests that, if we bin the data on the x axis to be binary, the reduced MSE will not be affected either, if the data are binned in the way as in (b). Panel (c) pictures an adaptive binning scheme to make the accuracy loss (if any) as little as possible.

Of course, we would not know ahead of time how to bin the data to avoid losing accuracy. Therefore, we suggest an adaptive quantization scheme, pictured in Figure 3(c), to make the accuracy loss (if any) as little as possible. In the pre-processing stage, for each feature, the training data points are sorted according to the feature value; and we bin the feature values in the sorted order. We start with a very small initial bin length, e.g., 10^{-8}. As shown in Figure 3(c), we only bin the data where there are indeed data, because the boosting tree algorithm will not consider the area where there are no data anyway. We set an allowed maximum number of bins, denoted by B. If the bin length is so small that we need more than B bins, we simply increment the bin length and re-do the quantization. After the quantization, we replace the original feature value by the bin labels (0, 1, 2, ...). Note that since we start with a small bin length, the ordinal categorical features are naturally taken care of.

We consider our adaptive binning scheme simple and effective for the implementation and performance of the boosting tree algorithm.

- It simplifies the implementation. After the quantization, there is no need for sorting (and keeping track of the indexes) because we conduct "bucket sort" implicitly.
- It speeds up the computations for the tree-building step, the bottleneck of the algorithm.
- It reduces the memory cost for training. For example, if we set the maximum allowed number of bins to be $B = 2^8$, we only need one byte per data entry.
- It does not really result in loss of accuracy. We experimented with both $B = 2^8 = 256$ and $B = 2^{16} = 65536$; and we did not observe real differences in the NDCG scores.

References

[1] A. Agresti. *Categorical Data Analysis*. John Wiley & Sons, Inc., Hoboken, NJ, second edition, 2002.

[2] L. Brieman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classifi cation and Regression Trees*. Wadsworth, Belmont, CA, 1983.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. in *WWW*, 107–117, 1998.

[4] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. in *ICML*, 89–96, 2005.

[5] C. Burges, R. Ragno, and Q. Le. Learning to rank with nonsmooth cost functions. in *NIPS*, 193–200, 2007.

[6] D. Cossock and T. Zhang. Subset ranking using regression. In *COLT*, 605–619, 2006.

[7] Y. Freund, R. Iyer, R. Schapire, and Y. Singer. An effi cient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[8] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 28(2):337–407, 2000.

[9] J. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[10] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR*, 41–48, 2000.

[11] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *SODA*, 668–677, 1998.