A fast algorithm for learning large scale preference relations

Vikas C. Raykar, Ramani Duraiswami
{vikas,ramani}@umiacs.umd.edu
Department of Computer Science
University of Maryland, CollegePark
CollegePark, MD, USA

Balaji Krishnapuram
balaji.krishnapuram@siemens.com
Computer Aided Diagnosis and Therapy Group
Siemens Medical Solutions
Malvern, PA, USA

Abstract

We consider the problem of learning the ranking function that maximizes a generalization of the Wilcoxon-Mann-Whitney statistic on training data. Relying on an ϵ-exact approximation for the error-function, we reduce the computational complexity of each iteration of a conjugate gradient algorithm for learning ranking functions from $\mathcal{O}(m^2)$, to $\mathcal{O}(m)$, where m is the size of the training data. Experiments on public benchmarks for ordinal regression and collaborative filtering show that the proposed algorithm is as accurate as the best available methods in terms of ranking accuracy, when trained on the same data, and is several orders of magnitude faster.

1 Introduction

Largely motivated by applications in search engines, information retrieval, and collaborative filtering, ranking has recently received significant attention in the statistical machine learning and information retrieval communities. In a typical formulation, we compare two instances and determine which one is *better* or *preferred*. Based on this, a set of instances can be ranked according to the desired preference relation.

The problem of learning a ranking has been formalized in many ways. We adopt the most general formulation based on directed preference graphs [4, 6]. This provides flexibility to learn different kinds of preference relations by changing the graph. We are given *training data* $\mathcal{A}$, a *directed preference graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ encoding the preference relations, and a *function class* $\mathcal{F}$ from which to choose the ranking function f.

The training data $\mathcal{A} = \bigcup_{j=1}^{S}(\mathcal{A}^j = \{x_i^j \in \mathbb{R}^d\}_{i=1}^{m_j})$ contains S classes (sets); each class $\mathcal{A}^j$ contains m_j samples, with a total of $m = \sum_{j=1}^{S} m_j$ samples in $\mathcal{A}$. Each vertex of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ corresponds to a class $\mathcal{A}^j$. The existence of a directed edge $\mathcal{E}_{ij}$ from $\mathcal{A}^i \rightarrow \mathcal{A}^j$ means that all training examples in $\mathcal{A}^j$ should be *preferred* or *ranked higher* than any training example in $\mathcal{A}^i$, *i.e.* , $\forall(x_k^i \in \mathcal{A}^i, x_l^j \in \mathcal{A}^j)$, $x_l^j \succeq x_k^i$.

The *preference relation* $x \succeq y$ means '*x is at least as good as y*'. One way of describing preference relations is by means of a ranking function. A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is a *ranking function* representing the preference relation $\succeq$ if $\forall x, y \in \mathbb{R}^d$, $x \succeq y \Leftrightarrow f(x) \geq f(y)$. The ranking function f provides a numerical score to the instances based on which they can be ordered.

Our goal is to learn a *ranking function* $f : \mathbb{R}^d \rightarrow \mathbb{R}$ such that $f(x_l^j) \geq f(x_k^i)$ for as many pairs as possible in the training data $\mathcal{A}$ and also to perform well on unseen examples. The output $f(x_k)$ can be sorted to obtain a ranking for a set of test samples $\{x_k \in \mathbb{R}^d\}$.

While a ranking function can be obtained by learning classifiers or ordinal regressors, it is more advantageous to learn the ranking function directly due to two reasons. First, in many scenarios it is more natural to obtain training data for pair-wise preference relations rather than the actual labels for individual samples. Second, the loss function used for measuring the accuracy of classification or ordinal regression—*e.g.* the 0-1 loss function—is computed for every sample individually, and then averaged over the training or the test set. In contrast, to asses the quality of the ranking for arbitrary preference graphs, we will use a generalized version of the *Wilcoxon-Mann-Whitney* (WMW) statistic [6, 12] that is averaged over *pairs* of samples:

$$\text{WMW}(f, \mathcal{A}, \mathcal{G}) = \frac{\sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \mathbf{1}_{f(x_l^j) \geq f(x_k^i)}}{\sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} 1}, \tag{1}$$

where $\mathbf{1}_{a \geq b} = 1$ if $a \geq b$, and 0 otherwise. The WMW is an estimate of the probability of correct pairwise ordering $\Pr[f(x_i) \geq f(x_j)]$, for a randomly drawn pair (x_i, x_j) such that $x_i \succeq x_j$. This is a generalization of the area under the ROC curve (for evaluating bipartite ranking), to arbitrary preference graphs between many classes of samples. For a perfect ranking WMW=1, and WMW=0.5 for a completely random choice.

Maximizing the WMW is a discrete optimization problem. Most ranking algorithms optimize a continuous relaxation instead. Although the WMW itself can be computed in $\mathcal{O}(md + m \log m)$ time, previous algorithms took $\mathcal{O}(m^2)$ time in order to evaluate the relaxed version or its gradient, seriously restricting the use of ranking formulations to large datasets; typically they could only learn from a few thousand samples.

Proposed approach: In this paper we directly maximize the relaxed version of the WMW statistic using a conjugate gradient optimization procedure. The gradient computation scales as $\mathcal{O}(dm^2)$ which is computationally intractable for large datasets. Inspired by the fast multipole methods in computational physics [8], we develop a new algorithm that computes the gradient approximately to ϵ precision in $\mathcal{O}(dm)$ time. Thus, much larger training datasets can be analyzed.

2 Previous work

Learning rankings was first treated as a classification problem on pairs of objects by Herbrich et al [9] and subsequently used on a web page ranking task by Joachims [11]. Algorithms similar to SVMs were used to learn the ranking function. Burges et al. [2], use a neural network (RankNet) to model the underlying ranking function. Similar to our approach it used a gradient descent technique to optimize a probabilistic cost function–the cross entropy. The neural net is trained on pairs of training examples using a modified backpropagation. With collaborative filtering as an application Freund et al. [5] proposed the RankBoost algorithm for combining preferences. Dekel et al. [4] present a general framework for label ranking by means of preference graphs and graph decomposition procedure. A log-linear model is learnt using a boosting algorithm.

The naive optimization strategy proposed in all the above algorithms suffer from $\mathcal{O}(m^2)$ growth in the number of comparisons. Approximation methods have recently been investigated. An efficient implementation of the RankBoost algorithm for two class problems was presented in [5]. A convex-hull based relaxation scheme was proposed in [6]. Yan and Hauptmann [17] proposed an approximate margin-based rank learning framework by bounding the pairwise risk function.

Novel Contributions: Our algorithm differs from the previous approaches in the following ways–

(1) The approaches [2,4,9,11] are computationally expensive to train due to the quadratic scaling in the number of pairwise comparisons. While our algorithm also uses pairwise comparisons the runtime is still linear. This is made possible by *fast approximate summation of erfc functions.*

(2) Unlike [17], which avoids the quadratic growth using a bound on the risk functional, and [6], which summarizes the slack variables for an entire class by a single, common scalar value, our ranking algorithm optimizes the WMW without such coarse approximations. We use *approximations only while computing the gradient* inside the optimization procedure. As a result the optimization will still converge to the optimal solution, although it will take a few more iterations.

(3) The cost function which we maximize is a *lower bound on the WMW.* Previous approaches which try to maximize the WMW [10, 16] consider only a classification problem but still incur the quadratic growth in the number of comparisons.

(4) This paper improves the computational complexity of batch optimization algorithms for analyzing large training datasets. A parallel body of literature has considered online, sequential update algorithms [3].

3 The MAP estimator

For ease of exposition we will consider the family of linear ranking functions: $\mathcal{F} = \{f_w\}$, where for any $x, w \in \mathbb{R}^d$, $f_w(x) = w^T x$; w are the weights to be learnt. A nonlinear version of the algorithm can be derived using the *kernel trick* (An SVM analog is in [9]).

Although we want to choose w to maximize the generalized WMW, for computational efficiency, we shall instead maximize a continuous surrogate via the log-likelihood $\mathcal{L}(f_w, \mathcal{A}, \mathcal{G}) = \log \Pr[\text{correct ranking}|w]$:

$$\mathcal{L}(f_w, \mathcal{A}, \mathcal{G}) = \log \prod_{\mathcal{E}_{ij}} \prod_{k=1}^{m_i} \prod_{l=1}^{m_j} \Pr\left[f_w(x_l^j) > f_w(x_k^i)|w\right].$$

In common with most papers [2, 9], we have assumed that every pair (x_l^j, x_k^i) is drawn independently, whereas only the original samples are drawn independently. We use the sigmoid function to model the pairwise probability, *i.e.*

$$\Pr\left[f_w(x_l^j) > f_w(x_k^i)|w\right] = \sigma\left[w^T(x_l^j - x_k^i)\right], \qquad (2)$$

where $\sigma(z) = (1 + \exp(-z))^{-1}$ is the sigmoid function. It has been previously used in [2] to model pairwise posterior probabilities. However they optimized the cross-entropy as the objective function. Assuming a prior $p(w) = \mathcal{N}(w|0, \lambda^{-1})$ on the weights w, the optimal *maximum a-posteriori* (MAP) estimator is of the

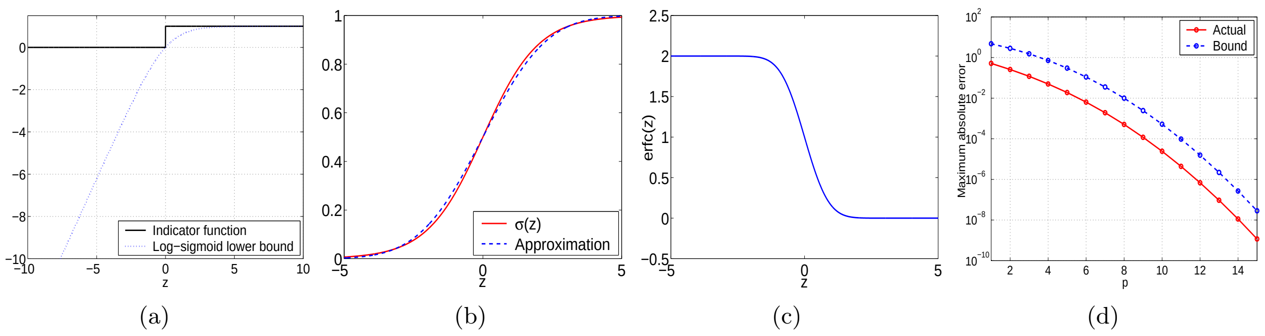


Figure 1: (a) Log-sigmoid lower bound for the 0-1 function. (b) Approximation of the sigmoid function. (c) The erfc function. (d) The maximum absolute error between the erfc and the truncated series representation as a function of the truncation number p for any $z \in [-4, 4]$. The error bound (Eq. 6) is also shown as a dotted line.

form $\widehat{w}_{\text{MAP}} = \arg\max_w L(w)$, where $L(w)$ is the penalized log-likelihood:

$$L(w) = -\frac{\lambda}{2}\|w\|^2 + \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \log \sigma \left[w^T (x_l^j - x_k^i) \right].$$

Log-Sigmoid lower bound: Since a 0-1 indicator function is lower bounded by a scaled and shifted log-sigmoid function (see Fig. 1(a))—*i.e.* $\mathbf{1}_{z>0} \geq 1 + (\log \sigma(z)/\log 2)$—comparing the formulae for the log-likelihood $\mathcal{L}(w)$ to that of the WMW we see that a suitably scaled version of $\mathcal{L}(w)$ lower bounds the WMW. Thus, maximizing the log-likelihood is equivalent to maximizing a lower bound on the WMW. The prior $p(w)$ acts as a regularizer. The log-sigmoid bound was also used in [4] in a boosting algorithm.

4 Gradient based learning

We use the Polak-Ribière variant of the nonlinear *conjugate gradients* (CG) algorithm [13] to find the w that maximizes $L(w)$. The CG method only needs the gradient $g(w)$ and does not require evaluation of either $L(w)$ or the second derivative (Hessian) matrix. The gradient vector *w.r.t.* w is:

$$g(w) = -\lambda w - \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} (x_k^i - x_l^j) \sigma \left[w^T (x_k^i - x_l^j) \right].$$

Note that the evaluation of the penalized log-likelihood or its gradient requires $\mathcal{M}^2 = \sum_{\mathcal{E}_{ij}} m_i m_j$ operations—this quadratic scaling is prohibitive for large datasets. *The main contribution of this paper is an extremely fast method to compute the gradient approximately.*

Gradient approximation: We use the approximation $\sigma(z) \approx 1 - \frac{1}{2}\text{erfc}(\frac{\sqrt{3}z}{\sqrt{2}\pi})$, where the complementary error function is defined by $\text{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} dt$ [See Figs. 1(b),1(c)]. As a result, the approximate gradient (still $\mathcal{O}(d\mathcal{M}^2)$) becomes:

$$g(w) = \nabla_w L(w) \approx -\lambda w - \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} (x_k^i - x_l^j) \left[1 - \frac{1}{2}\text{erfc}\left(\frac{\sqrt{3} w^T (x_k^i - x_l^j)}{\sqrt{2}\pi} \right) \right]. \quad (3)$$

For any x, let $z = \sqrt{3} w^T x / (\pi\sqrt{2})$. Note that z is a scalar and—for a given w—can be computed in $\mathcal{O}(dm)$ time for the entire training set. After rearranging the terms in the gradient it can be shown that (see tech report [14] for more details) the computational primitive contributing to the quadratic complexity are the two sums– $E_-^j(y) = \sum_{l=1}^{m_j} \text{erfc}(y - z_l^j)$ and $E_+^i(y) = \sum_{k=1}^{m_i} \text{erfc}(y + z_k^i)$. Note that $E_-^j(y)$ is the sum of m_j erfc functions–requires $\mathcal{O}(m_j)$ operations and to evaluate it at m_i points requires $\mathcal{O}(m_i m_j)$ operations. Below we show how it can be computed in linear $\mathcal{O}(m_i + m_j)$ time. Hence, the gradient can be computed in linear $\mathcal{O}(dSm + (S-1)m)$ time.

5 Fast summation of erfc functions

We can write $E_-^j(y)$ and $E_+^i(y)$ as the sum of N erfc functions centered at $z_i \in \mathcal{R}$, with weights $q_i \in \mathcal{R}$:

$$E(y) = \sum_{i=1}^{N} q_i \, \text{erfc}(y - z_i). \quad (4)$$

Direct computation of (4) at M points $\{y_j \in \mathcal{R}\}_{j=1}^M$ is $\mathcal{O}(MN)$. We derive an **ϵ-exact approximation** algorithm to compute this in $\mathcal{O}(M+N)$ time. For any $\epsilon > 0$, $\widehat{E}$ is an $\epsilon - exact$ approximation to E if the maximum absolute error relative to the total weight $Q_{abs} = \sum_{i=1}^N |q_i|$ is upper bounded by a specified ϵ, *i.e.* , $\max_{y_j} \left[|\hat{E}(y_j) - E(y_j)| / Q_{abs} \right] \leq \epsilon$. The constant in $\mathcal{O}(M+N)$ for our algorithm depends on the desired accuracy ϵ, which however can be *arbitrary*. At machine precision there is no difference between the direct and the fast methods. This approach relies on retaining only the first few terms of an infinite series expansion for the erfc function (to desired accuracy).

Beauliu series expansion: We use the truncated Fourier series representation derived by Beauliu [1,15]:

$$\text{erfc}(z) = 1 - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin(2nhz) + \text{error}(z). \quad (5)$$

where $|\text{error}(z)| < \left| \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} \frac{e^{-n^2h^2}}{n} \sin(2nhz) \right| + \text{erfc}\left(\frac{\pi}{2h} - |z|\right)$. Here, p is the *truncation number* and h is a real number related to the sampling interval. This series converges rapidly, especially as $z \to 0$. Fig. 1(d) shows the maximum absolute error between the actual value of erfc and the truncated series representation as a function of p. For example for any $z \in [-4, 4]$ with $p = 12$ the error is less than 10^{-6}. We will choose p and h such that the error is less than the desired ϵ. For this purpose we further bound the first term in the error as follows (see [14] for a derivation)—

$$|\text{error}(z)| < \frac{2}{\sqrt{\pi}h} \text{erfc}((2p+1)h) + \text{erfc}\left(\frac{\pi}{2h} - |z|\right) \quad (6)$$

Fast summation algorithm: A fast algorithm to compute $E(y)$ based on (5) can be written as

$$E(y) = \sum_{i=1}^{N} q_i \text{erfc}(y - z_i)$$
$$= \sum_{i=1}^{N} q_i [1 - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin\{2nh(y - z_i)\} + \text{error}].$$

Ignoring the error term for the time being, the sum $E(y)$ can be approximated as:

$$\widehat{E}(y) = Q - \frac{4}{\pi} \sum_{i=1}^{N} q_i \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin\{2nh(y - z_i)\}, \quad (7)$$

where $Q = \sum_{i=1}^{N} q_i$. The terms y and z_i are entangled in the argument of the sin function, leading to a quadratic complexity. *The crux of the algorithm is to separate them using the trigonometric identity*:

$$\begin{aligned} \sin\{2nh(y - z_i)\} = \\ \sin\{2nh(y - z_*)\} \cos\{2nh(z_i - z_*)\} \\ - \cos\{2nh(y - z_*)\} \sin\{2nh(z_i - z_*)\}. \quad (8) \end{aligned}$$

Note we have shifted all the points by z_*. Substituting the separated representation (8), exchanging the order of summation, and regrouping terms in (7),

$$\begin{aligned} \widehat{E}(y) = Q - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} A_n \sin\{2nh(y - z_*)\} \\ + \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} B_n \cos\{2nh(y - z_*)\}. \quad (9) \end{aligned}$$

$$A_n = \frac{e^{-n^2h^2}}{n} \sum_{i=1}^{N} q_i \cos\{2nh(z_i - z_*)\}, \quad \text{and}$$
$$B_n = \frac{e^{-n^2h^2}}{n} \sum_{i=1}^{N} q_i \sin\{2nh(z_i - z_*)\}. \quad (10)$$

Complexity analysis: The coefficients $\{A_n, B_n\}$ do not depend on y. Hence each A_n, B_n can be pre-computed in $\mathcal{O}(N)$ time. Since there are p such coefficients the total complexity to compute them is $\mathcal{O}(pN)$. The term Q can also be pre-computed in $\mathcal{O}(N)$ time. Once A, B, and Q are known, evaluation of $\widehat{E}(y)$ at M points requires $\mathcal{O}(pM)$ operations. Therefore, the computational complexity has reduced from the quadratic $\mathcal{O}(NM)$ to the linear $\mathcal{O}(p(N + M))$. We need space to store the points and the coefficients A and B. Hence, the storage complexity is $\mathcal{O}(N+M+p)$.

Direct inclusion/exclusion: From Eq. 6 we see that for fixed p and h, as $|z|$ increases the error increases. Therefore as $|z|$ increases, h should decrease and consequently the series converges slower leading to a large truncation number. Note that $s = (y - z_i) \in [-\infty, \infty]$. The p required to approximate erfc(s) can be quite large for large $|s|$. Luckily erfc(s) $\to 2$ as $s \to -\infty$ and erfc(s) $\to 0$ as $s \to \infty$ very quickly [See Fig. 1(c)]. Since we only want a precision ϵ, we can approximate:

$$\text{erfc}(s) \approx \begin{cases} 2 & \text{if } s < -r \\ p\text{-truncated series} & \text{if } -r \le s \le r \\ 0 & \text{if } s > r \end{cases} \quad (11)$$

The bound r and the truncation number p have to be chosen such that for any s the error is always less than ϵ, *e.g.* , for error of the order 10^{-15} we need to use the series expansion for $-6 \le s \le 6$. However, we cannot check the value of $(y - z_i)$ for all pairs of z_i and y. This would lead us back to the quadratic complexity. To avoid this, we subdivide the points into clusters.

Space sub-division: We uniformly sub-divide the domain into K intervals of length $2r_x$. The N source points are assigned into K clusters, S_k for $k = 1, \ldots, K$ with c_k being the center of each cluster. The aggregated coefficients are computed for each cluster and the total contribution from all the influential clusters is summed up. For each cluster, if $|y - c_k| \le r_y$, we

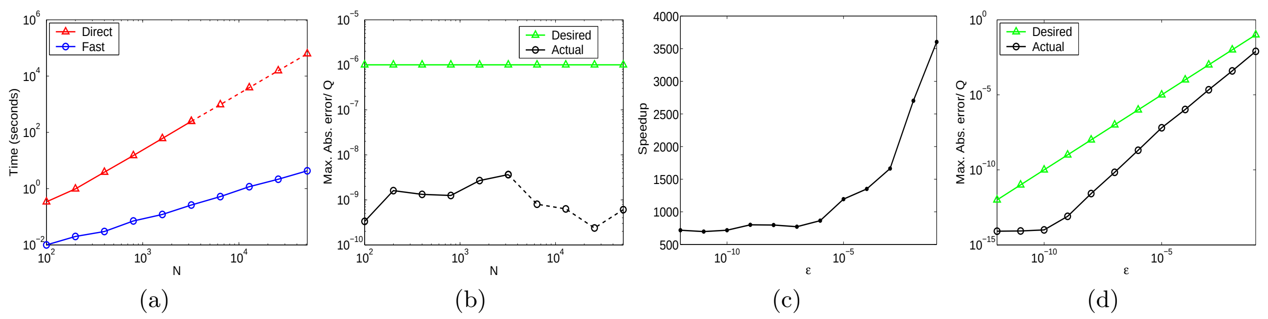


Figure 2: (a) The time in secs. and (b) max. absolute error relative to Q_{abs} for the direct and the fast methods as a function of $N(=M)$. For $N > 3,200$ the timing results for the direct evaluation were obtained by evaluating the sum at $M = 100$ points and then extrapolating (shown as dotted line). (c) The speedup achieved and (d) maximum absolute error relative to Q_{abs} for the direct and the fast methods as a function of ϵ for $N(=M) = 3,000$. Results are on a 1.6 GHz Pentium M processor with 512 MB of RAM

will use the series coefficients. If $(y - c_k) < -r_y$, we will include a contribution of $2Q_k$; if $(y - c_k) > r_y$, we ignore that cluster. The cut off radius r_y is chosen to achieve a given accuracy.

The cost to compute A,B, and Q is still $\mathcal{O}(pN)$ since each z_i belongs to only one cluster. Let l be the number of influential clusters, *i.e.*, the clusters for which $|y - c_k| \leq r_y$. Evaluating $\widehat{E}(y)$ at M points due to these l clusters is $\mathcal{O}(plM)$. Let m be the number of clusters for which $(y - c_k) < -r_y$. Evaluating $\widehat{E}(y)$ at M points due to these m clusters is $\mathcal{O}(mM)$. Hence the total time is $\mathcal{O}(pN + (pl + m)M)$. The storage complexity is $\mathcal{O}(N + M + pK)$.

Choosing parameters: Given any $\epsilon > 0$, we choose the parameters: r_x (the interval length), r_y (the cut off radius), p (the truncation number) and h, so that for any target y, $|\hat{E}(y) - E(y)| \leq Q_{abs}\epsilon$. The following choice guarantees that error$< \epsilon$: (1) $r_x = 0.1\text{erfc}^{-1}(\epsilon)$, (2) $r_y = \text{erfc}^{-1}(\epsilon){+}2r_x$, (3) $h = \pi/3\left(r + \text{erfc}^{-1}\left(\epsilon/2\right)\right)$, and (4) $p = \lceil \frac{1}{2h}\text{erfc}^{-1}\left(\frac{\sqrt{\pi}h\epsilon}{4}\right)\rceil$ (see [14] for details).

Numerical experiments: First we analyze the fast summation of erfc functions. Experiments with this primitive embedded in the MAP optimization are provided below. Fig. 2(a) and 2(b) show the running time and the maximum absolute error relative to Q_{abs} for both the direct and the fast methods as a function of $N(= M)$. The points were normally distributed with zero mean and unit variance, and q_i were set to 1. While the running time of the fast method grows linearly, that of the direct evaluation grows quadratically. The actual error is well below the permissible error, validating our bound. For $N = M = 51,200$ points, while direct evaluation takes 17.26 hours, the fast evaluation requires only 4.29 seconds with an error of 10^{-10}. Fig. 2(c) shows the tradeoff between precision and speedup. The speedup is obtained at the cost of slightly reduced accuracy.

6 Experiments: Ranking Benchmarks

Datasets: We used two artificial datasets and ten public benchmark datasets [1] in Table 1, previously used to evaluate ranking [6] and ordinal regression. RandNet and RandPoly are artificial datasets, generated as described in [2].

Evaluation Procedure: Accuracy was evaluated for each dataset in a five-fold cross validation experiment. In order to choose the regularization parameter λ, on each fold we used the training split and performed a five-fold cross validation on the training set. The ranking-accuracy is evaluated in terms of the generalized WMW statistic for a *full order graph.*

Comparisons: We compare the WMW and the runtime for the following methods.

A. RankNCG (proposed) The proposed nonlinear conjugate-gradient ranking procedure. The tolerance for the conjugate-gradient procedure was set to 10^{-3}. We compare the following two versions: (1) *RankNCG direct* which uses the exact gradient computation. (2) *RankNCG fast* which uses the fast approximate gradient computation. The accuracy ϵ for the fast gradient computation was set to 10^{-6}.

B. RankNet [2] A neural network which is trained using pairwise samples based on cross-entropy cost function. Training was done for around 500-1000 epochs. We used two versions of the RankNet: (a) *RankNet two layer* A two layer neural network with 10 hidden units; (b) *RankNet linear* A single layer neural network.

C. RankSVM [9, 11] A ranking function is learnt by

[1]Downloaded from `http://www.liacc.up.pt/~ltorgo/Regression/DataSets.html`. Since these datasets were originally for regression, we discretized the continuous target values into S equal sized bins (Table 1).

Table 1: Datasets used in the ranking experiments. N is the size of the data set. d is the number of attributes. S is the number of classes. $\mathcal{M}$ is the average total number of pairwise relations per fold of the training set.

	Dataset name	N	d	S	$\mathcal{M}$		Dataset name	N	d	S	$\mathcal{M}$
1	Diabetes	43	3	2	272	8	Airplane Companies	950	10	5	217301
2	Pyrimidines	74	28	3	1113	9	RandNet	1000	50	6	195907
3	Triazines	186	61	4	7674	10	RandPoly	1000	50	6	225131
4	Wisconsin Breast Cancer	194	33	4	8162	11	Abalone	4177	9	3	3713729
5	Machine-CPU	209	7	4	9820	12	RandNet	5000	50	6	6269910
6	Auto-MPG	392	8	3	30057	13	RandPoly	5000	50	6	5367241
7	Boston Housing	506	14	2	33693	14	California Housing	20640	9	3	82420255

training an SVM classifier[2] over pairs of examples. We used two version of the RankSVM: *RankSVM linear* using a linear kernel and *RankSVM quadratic* using a polynomial kernel $k(x, y) = (x.y + c)^2$.

D. RankBoost [5] A boosting algorithm which combines a set of *weak* ranking functions. We used weak binary rankings as the ordering information provided by the features, boosted for 50-100 cycles.

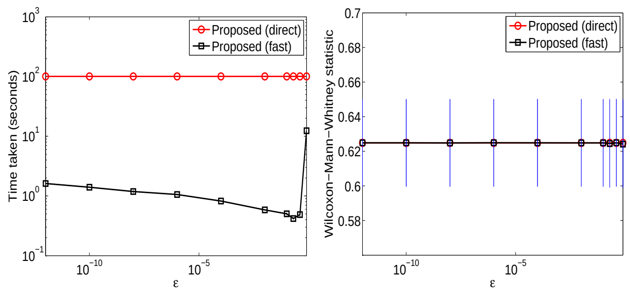
Results: The results are summarized in Table 3 and 4. The following observations can be made.

(1) **Quality of approximation** The WMW is similar for both the proposed exact method (RankNCG-direct) and the approximate method (RankNCG fast) but the run time of RankNCG-fast is one to two magnitudes lower, especially for large data sets.

(2) **Comparison with other methods** All the methods show very similar WMW scores. In terms of the training time the proposed method clearly beats all the other methods. For small datasets RankSVM linear is comparable in time to our methods. For large datasets RankBoost shows the next best run-time[3].

(3) **Ability to handle large datasets** For dataset 14 only the fast method completed execution. All the other methods (including RankBoost) either crashed due to huge memory requirements or took an incredibly large amount of time. Further, since the accuracy of learning (*i.e.* estimation) clearly depends on the ability to leverage large datasets, in real life, the proposed methods are also expected to be more accurate on large-scale ranking problems.

Gradient approximation: Fig. 3 studies the accuracy and the run-time for dataset 10 as a function of the gradient tolerance, ϵ. As ϵ increases, the time taken per-iteration (and hence overall) decreases. However, if it is too large the total time taken starts increasing (after $\epsilon = 10^{-2}$ in Fig. 3(a)). Intuitively, this is because the use of approximate derivatives slows the convergence of the conjugate gradient procedure by increasing the number of iterations required for convergence. The speedup is achieved because computing the approximate derivatives is extremely fast, thus compensating for the slower convergence. However, after a certain point the number of iterations dominates the run-time. Also ϵ has no significant effect on the WMW achieved, because the optimizer still converges to the optimal value albeit at a slower rate.



(a) (b)

Figure 3: (a) Time taken in seconds and (b) WMW statistic for the proposed method and the faster version as a function of ϵ. The CG tolerance was 10^{-3}.

7 Collaborative filtering:

We show results on a collaborative filtering task for movie recommendations on the MovieLens dataset[4] which contains approximately 1 million ratings for 3592 movies by 6040 users. Ratings are on a scale of 1 to 5. The task is to predict the movie rankings for a user based on the rankings provided by other users. We used 70% of the movies rated by each user for training and the remaining 30% for testing. The features for each movie consisted of the ranking provided by d other users. For each missing rating, we imputed a sample from a Gaussian distribution with its mean and variance estimated from the available ratings provided by the other users. Table 2 shows the time taken and the WMW score for this task for the two fastest methods. Results for the other methods are not shown due to lack of space. The WMW were not significantly different, but they took a large

[2]SVM-light: `http://svmlight.joachims.org/`

[3]Many experts consider RankBoost to be the best available algorithm for learning ranking functions.

[4]Downloaded from `http://www.grouplens.org/`.

Table 2: *Results for the EACHMOVIE dataset.* The training time in seconds and *WMW* as a function of the number of features d (averaged over 100 users).

d	RankNCG fast	RankBoost
50	0.48 [± 0.19]	6.68 [± 1.65]
	0.693 [± *0.054*]	*0.672* [± *0.056*]
100	0.44[± 0.17]	12.67 [± 2.83]
	0.707 [± *0.049*]	*0.679* [± *0.050*]
200	0.42 [± 0.17]	27.53 [± 5.99]
	0.722 [± *0.053*]	*0.685* [± *0.057*]
400	0.41 [± 0.17]	68.08[± 13.95]
	0.720 [± *0.054*]	*0.685* [± *0.051*]
800	0.45 [± 0.13]	193.18 [±39.75]
	0.721 [± *0.050*]	*0.673* [± *0.058*]
1600	0.51 [± 0.15]	613.54 [± 124.93]
	0.719 [± *0.053*]	*0.682* [± *0.058*]

amount of time to train. *The proposed method shows the best WMW and takes the least amount of time for training.*

8 Conclusions

This paper proposed an approximate ranking algorithm which directly maximizes the generalized WMW statistic. The algorithm relies on a novel, fast method for calculating a weighted sum of erfc functions for its computational efficiency. Experimental results demonstrate that despite the order(s) of magnitude speedup, the accuracy was almost identical to the exact method and other algorithms proposed in literature.

8.1 Future work

Other applications for the fast summation: The fast erfc summation method proposed could be potentially useful in neural networks, probit regression, and in Bayesian models involving sigmoids.

Nonlinear, kernelized variations: In order to retain focus, we did not discuss the non-linear version of our algorithm in detail. However, we may easily kernelize it by replacing the linear function $w^T x$ with $\sum_{i=1}^{m} \alpha_i k(x, x_i)$, where k is the kernel used. The computation of the gradient will involve calculating: (a) the weighted sum of kernel functions, and (b) the weighted sum of sigmoid (or erfc) functions. Dual-tree methods [7] and the improved fast Gauss transform [18] may be used to speedup (a). For (b) we can use the fast approximation proposed in this paper.

Independence of pairs of samples: Like most papers following [9], we have assumed that every pair (x_l^j, x_k^i) is drawn independently, even though they are really correlated . We plan to correct for this lack of independence using a statistical random-effects-model.

References

[1] N. C. Beauliu. A simple series for personal computer computation of the error function $Q(.)$. *IEEE Trans. Comm.*, 37(9):989–991, September 1989.

[2] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, 2005.

[3] K. Crammer and Y. Singer. Pranking with ranking. In *NIPS 14*, 2002.

[4] O. Dekel, C. Manning, and Y. Singer. Log-linear models for label ranking. In *NIPS 16*, 2004.

[5] Y. Freund, R. Iyer, and R. Schapire. An efficient boosting algorithm for combining preferences. *JMLR*, 4:933–969, 2003.

[6] G. Fung, R. Rosales, and B. Krishnapuram. Learning rankings via convex hull separation. In *NIPS 18*, 2006.

[7] A. G. Gray and A. W. Moore. Nonparametric density estimation: Toward computational tractability. In *SIAM Intl. Conf. Data Mining*, 2003.

[8] L. Greengard. Fast algorithms for classical physics. *Science*, 265(5174):909–914, 1994.

[9] R. Herbrich, T. Graepel, P. Bollmann-Sdorra, and K. Obermayer. Learning preference relations for information retrieval. *ICML-98 Workshop: Text Categorization and Machine Learning*, pp. 80–84, 1998.

[10] A. Herschtal and B. Raskutti. Optimising area under the ROC curve using gradient descent. In *ICML*, 2004.

[11] T. Joachims. Optimizing search engines using clickthrough data. *Proc. ACM SIGKDD*, pp. 133–142, 2002.

[12] H. B. Mann and D. R. Whitney. On a Test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, 18(1):50–60, 1947.

[13] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer-Verlag, 1999.

[14] V. C. Raykar, R. Duraiswami, and B. Krishnapuram. Fast weighted summation of erfc functions. CS-TR-4848, Dept. of Comp. Science, Univ. of Maryland CollegePark, 2007.

[15] C. Tellambura and A. Annamalai. Efficient computation of erfc(x) for large arguments. *IEEE Trans Comm.*, 48(4):529–532, April 2000.

[16] L. Yan, R. Dodier, M. Mozer, and R. Wolniewicz. Optimizing classifier performance via an approximation to the Wilcoxon-Mann-Whitney statistic. In *ICML 2003.*

[17] R. Yan and A. Hauptmann. Efficient margin-based rank learning algorithms for information retrieval. In *CIVR*, 2006.

[18] C. Yang, R. Duraiswami, and L. Davis. Efficient kernel machines using the improved fast Gauss transform. In *NIPS 17*, 2005.

Table 3: The mean training time and standard deviation in seconds (for five fold cross-validation experiment) for the datasets in Table 1. The symbol ⋆ indicates that the particular method either crashed due to limited memory requirements or took an inordinately large amount of time. All experiments were run on a 1.83GHz machine with 1.00GB of RAM. Except for RankSVM which used SVM-light, all the code was written in MATLAB.

	RankNCG direct	**RankNCG fast**	RankNet linear	RankNet two layer	RankSVM linear	RankSVM quadratic	RankBoost
1	0.11 [± 0.02]	0.06 [± 0.01]	1.79 [± 0.03]	3.32 [± 0.11]	0.09 [± 0.04]	0.10 [± 0.01]	1.70 [± 0.09]
2	0.63 [± 0.13]	0.12 [± 0.03]	7.11 [± 0.27]	13.55 [± 0.30]	0.10 [± 0.02]	0.62 [± 0.13]	1.72 [± 0.02]
3	17.63 [± 7.27]	0.70 [± 0.39]	58.14 [± 0.78]	131.41 [± 2.19]	0.55 [± 0.28]	13.96 [± 0.48]	6.70 [± 0.06]
4	13.41 [± 9.35]	0.33 [± 0.43]	48.13 [± 0.85]	97.24 [± 1.05]	0.64 [± 0.03]	23.17 [± 3.37]	1.88 [± 0.04]
5	20.38 [± 4.87]	0.97 [± 0.15]	57.99 [± 0.58]	111.14 [± 1.14]	1.14 [± 0.27]	24.46 [± 0.68]	1.24 [± 0.02]
6	28.05 [± 10.94]	0.40 [± 0.23]	175.63 [± 1.55]	333.49 [± 3.96]	0.43 [± 0.02]	37.27 [± 3.10]	1.54 [± 0.04]
7	18.92 [± 0.63]	0.16 [± 0.01]	195.14 [± 4.75]	381.28 [± 7.93]	0.36 [± 0.03]	13.93 [± 2.15]	2.32 [± 0.04]
8	332.88 [± 26.66]	3.29 [± 0.88]	1264.58 [± 3.21]	2464.84 [± 10.94]	34.32 [± 4.05]	1332.79 [± 69.47]	5.56 [± 0.37]
9	250.37 [± 21.03]	5.08 [± 0.47]	1166.23 [± 17.47]	2380.62 [± 34.53]	83.62 [± 6.30]	13628.23 [± 210.10]	13.55 [± 0.07]
10	102.48 [± 0.59]	0.78 [± 0.04]	1341.20 [± 6.91]	2733.25 [± 23.11]	1656.52 [± 99.89]	14110.48 [± 121.98]	13.99 [± 0.05]
11	1736.47 [± 191.03]	1.47 [± 0.38]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	62.91 [± 0.59]
12	6731.09 [± 312.41]	19.10 [± 1.76]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	147.04 [± 0.16]
13	2556.93 [± 15.03]	3.59 [± 0.41]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	133.42 [± 1.14]
14	⋆ [± ⋆]	46.86 [± 1.06]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]

Table 4: The corresponding generalized WMW statistic on the test set for the results shown in Table 3.

	RankNCG direct	**RankNCG fast**	RankNet linear	RankNet two layer	RankSVM linear	RankSVM quadratic	RankBoost
1	0.677 [± 0.233]	0.650 [± 0.210]	0.579 [± 0.096]	0.479 [± 0.284]	0.545 [± 0.236]	0.400 [± 0.276]	0.675 [± 0.173]
2	0.987 [± 0.019]	0.948 [± 0.077]	0.872 [± 0.088]	0.968 [± 0.038]	0.973 [± 0.048]	0.837 [± 0.142]	0.906 [± 0.144]
3	0.942 [± 0.044]	0.914 [± 0.047]	0.828 [± 0.030]	0.891 [± 0.064]	0.934 [± 0.019]	0.861 [± 0.088]	0.651 [± 0.045]
4	0.764 [± 0.028]	0.771 [± 0.046]	0.773 [± 0.046]	0.750 [± 0.035]	0.793 [± 0.018]	0.795 [± 0.035]	0.748 [± 0.056]
5	0.920 [± 0.015]	0.938 [± 0.020]	0.919 [± 0.035]	0.923 [± 0.040]	0.929 [± 0.026]	0.901 [± 0.014]	0.926 [± 0.018]
6	0.999 [± 0.002]	0.998 [± 0.002]	0.998 [± 0.003]	0.996 [± 0.003]	0.998 [± 0.002]	0.995 [± 0.008]	0.992 [± 0.004]
7	1.000 [± 0.000]	1.000 [± 0.000]	1.000 [± 0.000]	0.800 [± 0.400]	1.000 [± 0.000]	1.000 [± 0.000]	1.000 [± 0.000]
8	0.984 [± 0.004]	0.984 [± 0.003]	0.951 [± 0.004]	0.765 [± 0.245]	0.984 [± 0.004]	0.996 [± 0.001]	0.958 [± 0.003]
9	0.944 [± 0.012]	0.944 [± 0.012]	0.915 [± 0.017]	0.899 [± 0.028]	0.945 [± 0.013]	0.747 [± 0.005]	0.848 [± 0.015]
10	0.625 [± 0.025]	0.625 [± 0.025]	0.688 [± 0.032]	0.644 [± 0.054]	0.625 [± 0.026]	0.823 [± 0.008]	0.618 [± 0.024]
11	0.536 [± 0.011]	0.534 [± 0.008]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.535 [± 0.014]
12	0.917 [± 0.005]	0.917 [± 0.005]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.845 [± 0.006]
13	0.623 [± 0.008]	0.623 [± 0.008]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.607 [± 0.010]
14	⋆ [± ⋆]	0.979 [± 0.001]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]